Parallel Branch and Cut

T.K. Ralphs*

Revised January 17, 2006

Abstract

We discuss the main issues that arise in parallelizing the well-known branch-and-cut algorithm for solving mixed-integer linear programs. Designing an efficient parallelization scheme requires careful analysis of various tradeoffs involving the degree of synchronization, the degree of centralized storage of information, and the degree to which information discovered during the algorithm is shared between processors. We first present a methodological framework within which these tradeoffs can be analyzed and then show how the framework applies to the design of two software packages that take opposing approaches to achieving scalablility. Finally, we present computational results obtained solving three different problem classes in parallel with increasing numbers of processors. The results illustrate the degree to which various sources of parallel overhead affect scalability and demonstrate that properties of the problem class itself can dictate the effectiveness of a particular methodology.

1 Introduction

In this paper, we discuss parallelization of the branch-and-cut algorithm for solving general mixed-integer linear programs (MILPs). Branch and cut is a specialization of the more general class of algorithms known as *branch and bound* and is currently the most effective and commonly used approach for solving difficult MILPs. Virtually all modern software packages for solving MILPs use a variant of the branch-and-cut approach. Despite vast improvements in implementation over the past two decades and recent quantum leaps in computing power, however, many MILPs arising in practice remain difficult to solve by branch and cut. The difficulty stems mainly from limitations in memory and processing power, so a natural approach to overcoming this difficulty is to consider the use of parallel computing platforms, which can deliver a pooled supply of computing power and memory that is virtually limitless.

Branch and cut is a divide-and-conquer algorithm that partitions the original solution space into a number of smaller subsets and then solves the resulting smaller MILPs in a recursive fashion. Such an approach appears easy to parallelize, but this appearance is deceiving. Although it is easy in principle to divide the original solution space into subsets, it is difficult to do this in such a way that the amount of effort required to solve each of

*Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA 18015, tkralphs@lehigh.edu, http://www.lehigh.edu/~tkr2

the resulting smaller MILPs is approximately equal. If the work is not divided equally, then some processors will become idle long before the solution process has been completed, resulting in inefficiency. Even if this challenge is overcome, it might still be the case that the total amount of work required to solve the subproblems far exceeds the amount of work required to solve the original problem on a single processor.

The challenge to be faced in parallelizing any algorithm is to use additional resources, i.e., processors, as efficiently as possible. This consists not only of maximizing the number of processors that have work to do at any given time, but also ensuring that the work they are doing is "useful." In other words, the total amount of work performed in a parallel algorithm for solving a given problem should be as close as possible to the total amount of work performed using the best sequential algorithm. These two goals together mean that if we have p processors available and the time required to solve a given instance on one processor is S, we would ideally like to be able to solve the instance on p processors in time S/p. This is the yardstick against which parallel performance is generally measured.

The purpose of this paper is not to present a single approach to parallelization of branch and cut, but to introduce the reader to a range of issues that arise in such parallelization. Defining a single "best" approach is not possible in general—an approach suited for one class of problems may fail miserably on another, as we demonstrate in Section 5. Achieving good parallel performance requires, first and foremost, efficient mechanisms for sharing information among the available processors. In fact, the way in which processors share information is the primary determinant of parallel efficiency. Because sharing information is usually costly, there is a fundamental tradeoff between the cost of this sharing and the loss of efficiency that can result from *not* sharing. It is this tradeoff that we examine in the remainder of the paper.

The paper is organized as follows. In Section 2, we review necessary background, including the branch-and-cut algorithm and basic concepts in parallel computing. In Section 3, we discuss in broad terms the issues involved in parallelizing branch and cut, including the types of information that must be shared and what specific tradeoffs must be considered in designing an efficient algorithm. In Section 4, we describe the implementational details of two software packages that can be used for solving MILPs in parallel, SYMPHONY and ALPS. The two packages take very different approaches to parallelization and we use them to illustrate the tradeoffs discussed in Section 3. In Section 5, we analyze computational results obtained using SYMPHONY to solve instances from a number of representative problem classes. Finally, in Section 6, we conclude and summarize the material presented in the rest of the paper.

2 Background

2.1 Mixed-Integer Linear Programming

2.1.1 Definitions

A mixed-integer linear program is the problem of optimizing a linear objective function over a polyhedral feasible region with the additional constraint that some of the variables are

required to take on integer values. More formally, a MILP is a problem of the form

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

where $\mathcal{P} = \{x \in \mathbb{R}^n \mid Ax = b, x \geq 0\}$ is a polyhedron defined by *constraint matrix* $A \in \mathbb{Q}^{m \times n}$ and *right-hand side vector* $b \in \mathbb{Q}^m$, and $c \in \mathbb{R}^n$ is the *objective function vector*. For the remainder of the paper, we will use this standard notation to refer to the data associated with a given MILP. Note that we have assumed without loss of generality that the variables indexed 1 through p are the *integer variables*, that is, those required to take on values in $\mathbb{Z}$. The variables indexed $p+1$ through n are then the *continuous variables*. The case in which all variables are continuous ($p = 0$) is called a *linear program* (LP). Associated with each MILP is an LP, called the *LP relaxation*, with feasible region $\mathcal{P}$, obtained by relaxing the integrality restrictions. By convention, we set $z_{IP} = \infty$ if $\mathcal{P} \cap (\mathbb{Z}^p \times \mathbb{R}^{n-p}) = \emptyset$.

2.1.2 Branch and Bound

Overview. Branch and bound is the basic algorithmic approach taken by virtually all modern MILP solvers. The algorithm uses a divide and conquer strategy to partition the feasible set $\mathcal{F} = \mathcal{P} \cap (\mathbb{Z}^p \times \mathbb{R}^{n-p})$ into subsets and then optimizes over each resulting subset in a recursive fashion. The goal is to determine a least cost member of $\mathcal{F}$ (or prove $\mathcal{F} = \emptyset$), so we first attempt to find a "good" solution $\bar{x} \in \mathcal{F}$ (called the *incumbent*) by a heuristic procedure or otherwise. If we succeed, then $\bar{z} = c^\top \bar{x}$ serves as an initial upper bound on z_{IP}. If no such solution is found, then we set $\bar{z} = \infty$.

The *processing* or *bounding* operation is to solve a relaxation of the original problem, yielding a lower bound on the value of an optimal solution.[1] If solving the relaxation yields a member of $\mathcal{F}$, then such member is also optimal for the MILP itself and we are done. Otherwise, we identify k disjoint polyhedral subsets of $\mathcal{P}$, $\mathcal{P}_1, \ldots, \mathcal{P}_k$, such that $\cup_{i=1}^k \mathcal{P}_i \cap (\mathbb{Z}^p \times \mathbb{R}^{n-p}) = \mathcal{F}$. This is called the *branching* or *partitioning* operation. Each of these subsets defines a new MILP with the same objective function as the original, called a *subproblem*. Based on this partitioning of $\mathcal{F}$, we have

$$\min_{x \in \mathcal{F}} c^\top x = \min_{i \in 1..k} \left(\min_{x \in \mathcal{P}_i \cap (\mathbb{Z}^p \times \mathbb{R}^{n-p})} c^\top x \right), \tag{2}$$

so we have reduced solution of the original MILP to solution of a family of smaller MILPs. The subproblems associated with $\mathcal{P}_1, \ldots, \mathcal{P}_k$ are called the *children* of the original MILP, which is itself called the *root subproblem*, as well as the *parent* of each of its children.

After partitioning the root subproblem, we initialize $\mathcal{C}$, the set of *candidate subproblems* (those that await processing or partitioning) with the children of the root subproblem and associate with each of these an initial lower bound computed during the partitioning procedure. The next step is to select a candidate subproblem i (with feasible region $\mathcal{P}_i \cap (\mathbb{Z}^p \times \mathbb{R}^{n-p})$), remove it from $\mathcal{C}$, and process it. Processing results in a new lower bound $\underline{z}_i$ and (possibly) a relaxed solution $\hat{x}^i \in \mathbb{R}^n$. There are three possible outcomes:

1. If $\underline{z}_i \geq \bar{z}$, then the subproblem cannot have a solution with value strictly better than $\bar{x}$ and we may discard, or *fathom*, it. This includes the case where $\mathcal{P}_i \cap (\mathbb{Z}^p \times \mathbb{R}^{n-p}) = \emptyset$.

[1] We assume the relaxation is bounded, or else the original MILP is itself unbounded.

2. If $\hat{x}^i \in \mathcal{F}$ and $\underline{z}_i = c^\top \hat{x}^i < \bar{z}$, then $\hat{x}^i$ becomes the new incumbent. We set $\bar{z} \leftarrow c^\top \hat{x}^i$, $\bar{x} \leftarrow \hat{x}^i$, and again fathom the subproblem.

3. If none of the above three conditions hold, then the subproblem becomes a candidate for the partitioning operation.

If a subproblem becomes a candidate for partitioning, it can either be partitioned immediately, or placed back in the candidate list. Once partitioned, the children of a subproblem are added to the candidate list and the subproblem itself is discarded. The overall algorithm consists of continuing to select subproblems from the candidate list in a prescribed order (called the *search order*) and processing or partitioning them, as appropriate, until $\mathcal{C}$ is empty, at which point the current incumbent must be the optimal solution. If no incumbent exists at termination, then $\mathcal{F} = \emptyset$.

It is common to associate the set of subproblems with a tree, called the *search tree*, in which each node corresponds to a subproblem and is connected to both its children and its parent. We therefore use the term *search tree node*, or simply *node*, interchangeably with the term subproblem and refer to the original MILP as the *root node* or *root* of this tree.

Implementation. From the above description, it can be seen that any branch-and-bound algorithm consists of four essential elements:

- *Upper bounding method*: A method for determining an initial incumbent and corresponding upper bound $\bar{z}$ (optional).
- *Lower bounding method*: A method for processing a subproblem.
- *Branching method*: A method for partitioning a subproblem.
- *Search strategy*: A method for determining the search order.

By implementing these elements in various ways, one can derive a wide range of specialized versions of branch and bound.

The branch and bound procedure can be seen as an iterative method for improving the difference between the current upper bound (the objective function value of the current incumbent) and the current lower bound (the minimum of the lower bounds associated with the candidate subproblems). The difference between these two bounds is called the *optimality gap*. Typically, the goal of both the bounding and the branching operations is to improve the lower bound, while the search strategy can be focused on improving either the upper or the lower bound. Bounding and branching methods are generally developed with a particular application or problem class in mind, but search strategies can be discussed in a more generic fashion. The possible strategies are numerous, but we summarize the most common approaches below.

Many search strategies employ a fixed rule for selecting the next subproblem to process. A common such method is *best-first search*, which chooses a candidate node with smallest lower bound. Because of the fathoming rule employed in branch and bound, a best-first search strategy ensures that no subproblem with a lower bound above the optimal solution

value can ever be chosen for processing. Therefore, the best-first strategy tends to minimize the number of subproblems processed and to improve the lower bound quickly. However, this comes at the price of sacrificing incremental improvements to the upper bound, since the upper bound will generally become finite only when an optimal solution has been located. The lack of a good upper bound can also hurt overall efficiency by reducing the effectiveness of procedures both for fathoming and for tightening of variable bounds based on the size of the optimality gap.

At the other extreme, *depth-first search* chooses the next candidate to be a node at maximum depth in the tree, i.e., a node whose path to the root node in the search tree is longest. Depth-first is one of a class of strategies, called *diving strategies*, that may retain one or more children of the current subproblem for processing even when there exist candidates nodes with smaller lower bounds. In contrast to best-first search, which will produce few suboptimal solutions, diving strategies may produce many suboptimal solutions, typically early in the search process. This allows the upper bound to be improved quickly in the early phases of the algorithm, which can be advantageous if early termination is necessitated. Diving strategies also have the advantage that the change in the relaxation being solved from subproblem to subproblem may be very slight, so the relaxations may be solved more quickly than in best-first search.

Neither best-first search nor depth-first search attempt to select nodes that have a high probability of leading to improved feasible solutions. Estimate-based methods are improvements in this regard. The *best-projection* method [22, 46] measures the overall "quality" of a node by combining its initial lower bound with the degree of infeasibility of a relaxed solution obtained either while processing the parent or during branching. Alternatively, the *best-estimate* [6] method combines a node's lower bound, degree of infeasibility, and an estimate of the value of an optimal solution to the subproblem.

Since we have two goals in node selection—finding improved feasible solutions (i.e., improving the upper bound) and proving that the current incumbent is itself a "good" solution (i.e., improving the lower bound)—it is natural to develop node selection strategies that switch from one goal to the other during the course of the algorithm. This results in a two-phase search strategy. In the first phase, we try to determine "good" feasible solutions, while in the second phase, we try to prove this goodness. Perhaps the simplest two-phase algorithm is to perform depth-first search until a feasible solution is found, then switch to best-first search.

Hybrid methods also combine two or more node selection methods, but in a different manner than in two-phase methods. In a typical hybrid method, the search tree is explored using a diving strategy until the lower bound of the child subproblem being explored rises above a prescribed level in comparison to the overall lower or upper bound. At this point, a new subproblem is selected by a different criterion (e.g., best-first or best-estimate), and the diving process is repeated. For an in-depth discussion of search strategies for mixed-integer linear programming, see the paper of Linderoth and Savelsbergh [42].

2.1.3 Branch and Cut

When the relaxation used in the processing step is an LP relaxation, we obtain a general class of algorithms known as *LP-based branch and bound*. For many problem classes, the bound

yielded by the initial LP relaxation is not strong enough to allow efficient solution of difficult instances, but we can improve the bound by dynamically generating valid inequalities that can than be added to the LP relaxation to strengthen it. Padberg and Rinaldi called this technique *branch and cut* [51].

More formally, an *inequality* is a pair (a, a_0) consisting of a *coefficient vector* $a \in \mathbb{R}^n$ and a *right-hand side* $a_0 \in \mathbb{R}$. Any member of the half-space $\{x \in \mathbb{R}^n \mid a^\top x \leq a_0\}$ is said to *satisfy* the inequality and all other points are said to *violate* it. An inequality is *valid* for a given MILP if all members of the feasible set $\mathcal{F}$ satisfy it. A valid inequality (a, a_0) is called *improving* for the MILP if

$$\min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{P}, ax \leq a_0\} > \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{P}\}.$$

A necessary and sufficient condition for an inequality to be improving is that it be violated by all optimal solutions to the LP relaxation, so violation of the *fractional solution* $\hat{x} \in \mathbb{R}^n$ generated by solving the LP relaxation is a necessary condition. Even if a given valid inequality violated by $\hat{x}$, also called a *cut*, is not improving, adding it to the current LP relaxation may still result in the generation of a new fractional solution and, in turn, additional candidate inequalities.

An important observation is that an inequality (a, a_0) is valid for $\mathcal{F}$ if and only if it is valid for the associated polyhedron $\text{conv}(\mathcal{F})$. Valid inequalities that are necessary to the description of $\text{conv}(\mathcal{F})$ are called *facet-defining inequalities* (see [49] for a precise definition). Because they provide the closest possible approximation of $\text{conv}(\mathcal{F})$, facet-defining inequalities are typically very effective at improving the lower bound. They are, however, difficult to generate in general. For an arbitrary vector $\hat{x} \in \mathbb{R}^n$ and polyhedron $\mathcal{R} \subseteq \mathbb{R}^n$, the problem of either generating a facet-defining inequality (a, a_0) violated by $\hat{x}$ or proving that $\hat{x} \in \mathcal{R}$ is called the *facet identification problem*. The facet identification problem for a given polyhedron is polynomially equivalent to optimization over the same polyhedron [28], so generating a facet-defining inequality violated by an arbitrary vector is in general as hard as solving the MILP itself. The problem of generating a valid inequality violated by a given fractional solution, whether facet-defining or not, is called the *separation problem*.

A high-level description of the iterative bounding procedure used in branch and cut is shown in Figure 1. Generally, the loop consists of an alternation between solution of the current LP relaxation and the generation of valid inequalities violated by the relaxed solution. Because the number of violated valid inequalities generated in each iteration can be quite large, they are first added to a local queue. Following the generation step, a limited number of violated inequalities are taken from the queue and added to the LP relaxation. It is important to note that not only are valid inequalities added to the relaxation each iteration, but both valid inequalities and variables are considered for deletion as well. For valid inequalities, this deletion is based on the values of each constraint's corresponding slack and dual variables. For variables, removal occurs when both the lower and upper bound for a variable are fixed to the same value through a procedure that compares each variable's reduced cost to the current optimality gap (for a description of this procedure, called *reduced cost fixing*, see [50]).

Efficient management of the LP relaxation is critical to the efficiency of branch and cut, since both the memory required to store the search tree and the time required to process

a node are dependent on the number of constraints and variables that are "active" in each subproblem. In practice, there are a number of extra steps that can be taken, such as logical preprocessing and execution of primal heuristics, to accelerate the overall performance of the algorithm, but these are ancillary to the topic of this paper. More details regarding the management of the LP relaxation in a typical MILP solver are provided in [57].

If the procedure in Figure 1 fails to fathom the subproblem, then we are forced to branch. In branch and cut, the branching method should generally have three properties. First, the feasible region of the parent problem should be partitioned in such a way that the resulting subproblems are also MILPs. This means that the subproblems are usually defined by imposing additional linear inequalities. Second, the union of the feasible regions of the subproblems should contain at least one optimal solution to the parent problem. Finally, since the primary goal of branching is to improve the overall lower bound, the current fractional solution should not be contained in any of the members of the partition. Otherwise, the overall lower bound will not be improved.

Given a fractional solution $\hat{x} \in \mathbb{R}^n$ to the LP relaxation, an obvious way to fulfil the above requirements is to choose an index $j \leq p$ such that $\hat{x}_j \notin \mathbb{Z}$ (a *fractional variable*) and create two subproblems, one by imposing an upper bound of $\lfloor \hat{x}_j \rfloor$ on variable j and a second by imposing a lower bound of $\lceil \hat{x}_j \rceil$. This is a valid partitioning, since any feasible solution must satisfy one of these two linear inequalities. Furthermore, $\hat{x}$ is not feasible for either of the resulting subproblems. This partitioning procedure is known as *branching on a variable*. More generally, one can branch on other disjunctions. For any vector $a \in \mathbb{Z}^n$ whose last $n - p$ entries are zero, we must have $a^\top x \in \mathbb{Z}$ for all $x \in \mathcal{F}$. Thus, if $a\hat{x} \notin \mathbb{Z}$, a can be used to produce a disjunction by imposing the inequality $a^\top x \leq \lfloor a^\top \hat{x} \rfloor$ in one subproblem and the inequality $a^\top x \geq \lceil a^\top \hat{x} \rceil$ in the other subproblem. This is known as *branching on a hyperplane*. Typically, branching on hyperplanes is a problem-specific method that exploits special structure, but it can be made generic by keeping a pool of inequalities that are slack in the current relaxation as branching candidates [36].

When branching on variables, there are usually many fractional variables, so we must have a method for deciding which one to choose. A primary goal of branching is to improve the lower bound of the resulting relaxations. The most straightforward branching methods choose a branching variable based solely on the set of fractional variables and do not use any auxiliary information. Branching on the variable with the fractional part closest to 0.5, the first variable (by index) that is fractional, or the last variable (by index) that is fractional are examples of such procedures. These rules tend to be too myopic to be effective, so many solvers use more sophisticated approaches. Such approaches fall into two general categories: *forward-looking methods* and *backward-looking methods*. Methods in each category attempt to choose the best partitioning by predicting, for a given candidate partitioning, how much the lower bound will actually be improved. Forward-looking methods generate this prediction based solely on locally generated information obtained by "pre-solving" candidate subproblems. The most common forward-looking branching method is *strong branching*, in which the solver explicitly performs a limited number of dual simplex pivots on the LP relaxations in each of the children resulting from branching on a given variable in order to estimate the change in bound that would occur from that choice of branching. Backward-looking methods take into account the results of previous partitionings to predict the effect of future ones. The most popular such methods depend on the computation of *pseudo-costs* [6, 23], which are calculated based on a history of the effect of branching on

Input: A subproblem defined by $\mathcal{P}_i \in \mathcal{C}$, an initial set of additional valid inequalities defining an auxiliary polyhedron $\mathcal{R}$ (possibly generated during processing of the parent subproblem), and the global upper bound $\bar{z}$.

Output: Either (1) a lower bound $\underline{z}_i$ on the optimal value of the subproblem, or (2) an indication that the subproblem can be fathomed.

1. Form the initial LP relaxation

$$\min_{x \in \mathcal{P}_i \cap \mathcal{R}} c^\top x, \tag{3}$$

   $\mathcal{R}$ is the polyhedron representing additional valid inequalities.

2. Solve the current LP relaxation

$$\underline{z}_i = \min_{x \in \mathcal{P}_i \cap \mathcal{R}} c^\top x \tag{4}$$

   and let $\hat{x}$ be the resulting fractional solution.

3. If $\underline{z}_i \geq \bar{z}$, then subproblem i can be fathomed. STOP.

4. If $\hat{x} \in \mathcal{P}_i \cap (\mathbb{Z}^p \times \mathbb{R}^{n-p})$, then subproblem i can be fathomed. If $\underline{z}_i < \bar{z}$, then set $\bar{x} \rightarrow \hat{x}$ and $\bar{z} \rightarrow \underline{z}_i$. STOP.

5. Generate a set valid inequalities violated by $\hat{x}$ and add them to the local queue $\mathcal{L}$.

6. Remove variables that can be fixed from the relaxation.

7. Remove ineffective valid inequalities from the description of $\mathcal{R}$.

8. If $\mathcal{L} = \emptyset$, then subproblem i is a candidate for partitioning. STOP and output the lower bound $\underline{z}_i$.

9. Otherwise, add valid inequalities from $\mathcal{L}$ to the description of $\mathcal{R}$ and go to Step 2.

Figure 1: The node processing loop in the branch-and-cut algorithm

each variable. Of course, as one might expect, there are also hybrids that combine these two basic approaches [1].

2.2 Parallel Computing Concepts

2.2.1 Architecture

The architecture of the parallel platform (defined as a specific combination of software and hardware) on which an algorithm will be deployed can be a significant factor in its design. In particular, the topology of the communications network determines which pairs of processors can communicate directly with each other [13]. Although many specialized network topologies have been developed and studied, recent trends have favored the use of commodity hardware to construct so-called *Beowulf clusters* [10]. In this paper, a simplified parallel architecture similar to that of a typical Beowulf cluster is assumed. The main properties of such an architecture are listed below.

- The cluster is comprised of homogeneous *processing nodes*, each with a single central processing unit (or *processor*).
- The processing nodes are connected by a dedicated high-speed communication network that allows every processing node to communicate directly with every other processing node.
- There is no shared access memory (memory that can be accessed directly by multiple processors). Only local memory is available to each processor. No assumption is made regarding the local memory hierarchy.
- Communication between processing nodes is via a *message-passing* protocol. This means that information can only be passed from one processing node to another as a string of bits with an associated *message tag* indicating the structure of the information contained in the message. The two most common message-passing protocols are PVM [24] and MPI [27].

This cluster architecture and the algorithms we discuss are *asynchronous* by nature, meaning that the processors do not have a common clock by which to synchronize their calculations. Synchronization can only be accomplished through explicit communication. In our simplified model, the two main resources required for computation are memory and processing power, each of which can only be increased by adding *processing nodes* to the cluster. A processing node consists of both a processor and associated memory, but we will assume that each processing node has sufficient memory for required calculations and will refer to processing nodes simply as "processors."

2.2.2 Scalability

As mentioned earlier, the primary goal of parallel computing is to take advantage of increased processing power to solve problems faster. The *scalability* of a parallel platform is the degree to which it is capable of efficiently utilizing increased computing resources

(usually processors). We focus here on the effect of the algorithm design and therefore compare the speed with which we can solve a particular problem instance using a given parallel algorithm to that with which we could solve it on a single processor. The *sequential running time* (S) is used as the basis for comparison and is usually taken to be the wall clock running time of the best available sequential algorithm. The *parallel running time* (T_p) is the wall clock running time of the parallel algorithm running on p processors. The *speedup* (S_p) is simply the ratio S/T_p. Finally, the efficiency (E_p) is the ratio S_p/p of speedup to number of processors. Note that all these quantities depend on p.

A parallel algorithm is considered scalable if it results in an efficiency close to one as the number of processors is increased. When a parallel algorithm maintains an efficiency greater than or equal to one, we say that it has achieved *linear speedup*. In theory, a parallel algorithm cannot have an efficiency strictly greater than one (this is called *superlinear speedup*), but in practice, such a situation can sometimes occur (see [15] and [37] for a treatment of this phenomenon). When an algorithm has an efficiency less than one, the *parallel overhead* is $O_p = T_p(1 - E_p)$.

Conceptually, execution of a parallel algorithm can be divided into three phases. The *ramp-up phase* is the period during which work is initially partitioned and allocated to the available processors. This phase is loosely defined to last until all processors have been assigned at least one task (the exact definition can vary, depending on the application). The division of work that occurs during the ramp-up phase may be accomplished on a single processor or may itself be parallelized to the extent possible. The second phase is the *primary phase*, during which the algorithm operates in steady state. This is followed by the *ramp-down* phase, during which termination procedures are executed and final results are tabulated and reported. The ramp-down phase is loosely defined to start when the number of available tasks first falls below the number of available processors.

The division of the algorithm into phases is to highlight the fact that certain portions of every algorithm (e.g., the ramp-up and ramp-down phases) are inherently sequential. Amdahl was the first to point this out and called this part of the running time the *sequential fraction* [2]. The inherently sequential portions of the algorithm can be significant contributors to parallel overhead. Determining the initial pool of tasks and allocating them to processors is an inherently sequential task, at least in part, and the efficiency with which this can be done is usually an important factor in determining the amount of parallel overhead. Because of this, if the problem size is kept constant, efficiency generally drops as the number of processors increases. If the number of processors is kept constant, however, then efficiency generally *increases* as problem size increases [34, 26, 29]. This led Kumar and Rao to suggest a measure of scalability called the *iso-efficiency function* [35], which measures the rate at which the problem size has to be increased with respect to the number of processors in order to maintain a fixed efficiency.

2.2.3 Knowledge Management

As we mentioned in the introduction, achieving good parallel performance involves the design of a scheme for sharing information between processors as the algorithm progresses. Consider the following four main components of parallel overhead:

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

The first three sources of overhead are costs incurred in order to share information among the processors, whereas the last one is essentially the cost incurred by *not* sharing *enough* information. This highlights a fundamental tradeoff—to achieve high efficiency, we must limit the impact of the first three sources of overhead without increasing the impact of the fourth source.

We refer to information generated during the execution of the algorithm as *knowledge*. Trienekens and de Bruin introduced the notion that the efficiency of a parallel algorithm is inherently dependent on the strategy by which this knowledge is stored and shared among the processors [66]. From this viewpoint, a parallel algorithm can be thought of roughly as a mechanism for coordinating a set of autonomous agents that are either *knowledge generators* (KGs) (responsible for producing new knowledge), *knowledge pools* (KPs) (responsible for storing previously generated knowledge), or both. Specifying such a coordination mechanism consists of specifying what knowledge is to be generated, how it is to be generated, and what is to be done with it after it is generated (stored, shared, discarded, or used for subsequent local computations).

In order to effectively manage and organize the potentially huge amount of information to be generated, knowledge is typically categorized by type, with each KP containing only knowledge of a particular type. In addition, KPs may be either *local* (only accessible locally) or *global* (available to share with other processors). Within each pool, knowledge is organized to make retrieval and management as easy as possible. Whenever the amount of knowledge generated could exceed the amount of storage capacity, each knowledge object can be assigned a numerical priority that reflects its perceived importance to the overall computation. This allows the KPs to be purged periodically to remove low priority items.

2.2.4 Task Management

Just as KPs can be divided by the type of knowledge they store, KGs may be divided either by the type of knowledge they generate or the method by which they generate it. In other words, processors may be assigned to perform only a particular task or set of tasks. If a single processor is assigned to perform multiple tasks simultaneously, a prioritization and time-sharing mechanism must be implemented to manage the computational efforts of the processor.

The *granularity* of an algorithm is the size of the smallest task that can be assigned to a processor. Choosing the proper granularity can be important to efficiency. Too fine a

granularity can lead to excessive communication overhead, while too coarse a granularity can lead to excessive idle time and the performance of redundant work. We have assumed an asynchronous architecture, in which each processor is responsible for autonomously orchestrating local algorithm execution by prioritizing local computational tasks, managing locally generated knowledge, and communicating with other processors. Because each processor is autonomous, care must be taken to design the entire system so that *deadlocks*, in which a set of processors are all mutually waiting on one another for information, do not occur.

2.3 Previous Work

The branch and bound algorithm described in Section 2.1.2 was first suggested by Land and Doig in 1960 [38]. 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 MILPs began to be developed. Most of the software packages currently available implement some version of the branch-and-cut algorithm we have described. Available noncommercial generic MILP solvers include bonsaiG [30], CBC [20], GLPK [44], lp_solve [7], MINTO [48], and SYMPHONY [52, 59]. Commercial MIP solvers include ILOG's CPLEX and Dash's XPRESS. Generic frameworks that allow the user to take advantage of special structure by implementing specialized functionality, such as problem-specific cut generation, include SYMPHONY, COIN/BCP [60], ABACUS [32], and CBC. CONCORDE [4, 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 and parallel branch and cut have also been developed. The previously mentioned SYMPHONY, COIN/BCP, and CONCORDE all have parallel execution modes and can be run on networks of workstations. Other related software includes frameworks for implementing parallel branch and bound such as PUBB [64], BoB [5], PPBB-Lib [68], and PICO [16]. PARINO [40] and FATCOP [11] are parallel generic MILP solvers.

3 Parallelizing Branch and Cut

3.1 Knowledge Management

In branch and cut, an optimal solution to a given problem instance can be thought of as a type of knowledge, which is typically the sole end product of the algorithm. Producing such a solution, however, requires production of a great deal of auxiliary knowledge during the course of the algorithm. This knowledge is typically discarded at termination, but may be retained in some cases for the purpose of providing a proof of optimality or performing sensitivity analysis. Below, we discuss the various types of knowledge that can be produced during branch and cut and the issues involved in sharing and storing each type.

Bounds. The bounds that must be shared in branch and cut consist of the single global upper bound and the lower bounds associated with the subproblems that are candidates for

processing. Knowledge of these bounds is important mainly for the avoidance of redundant work. In branch and cut, the primary source of redundant work is the processing of nodes whose initial lower bound exceeds the optimal solution value. In theory, the processing of such nodes is avoidable with the proper search strategy, but in practice, such redundant work may occur even in the sequential case if a search strategy other than pure best-first is employed.

Although the final output of the algorithm is a single optimal solution, suboptimal solutions may be produced during the course of the algorithm. The primary importance of such solutions is that they may be the new incumbent at the time of their production and hence may provide a new global upper bound. It is important that new global upper bounds be broadcast as quickly as possible to other processors, as this knowledge allows nodes whose lower bounds exceed this new upper bound to be fathomed, thus avoiding the performance of redundant work. Dissemination of upper bounds generally incurs low overhead and does not pose a serious scalability issue.

Knowledge of lower bounds is also important in avoiding the performance of redundant work. The distribution of lower bounds among the candidate subproblems is used to determine if work should continue on nodes that are locally available or if new nodes should be requested from a remote pool. This is part of the overall process of redistributing candidate nodes during the algorithm, called *load balancing* (see Section 3.3.2). Making knowledge of lower bounds available may be difficult to accomplish because the number of candidate nodes available globally can be extremely large and they may not be stored centrally.

Node Descriptions. Two computational tasks that arise naturally in branch and cut are the processing of a subproblem in the search tree (bounding) and the partitioning of a subproblem into a number of new subproblems (branching). These two tasks are generally, though not always, the smallest units of work assigned to a processor. For efficiency, the branching operation is frequently accomplished immediately following the processing operation.

In order to process a node, it is necessary to have a detailed description of it. The description of a search tree node consists primarily of descriptions of the valid inequalities and variables that are active in the node and a complete description of the current *basis* (assuming a simplex-based LP solver) or other information either inherited from the parent or computed during branching 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 [69] or [50]). Along with the set of active constraints and variables, we must also store the branching hyperplane(s) that led to generation of the node. Processing a node results in the generation of new knowledge in the form of bounds (described above) and (possibly) valid inequalities. Valid inequalities discovered during the processing of a node may also be shared, as described in the next section. If a node fails to be fathomed, then it becomes a candidate for branching.

The branching operation results in the generation of new node descriptions, which may be stored locally (in a *node pool*) or shared with other processors. An important question that arises is how to ensure that each processor assigned the task of processing search tree nodes has a constant supply of *high-priority* nodes in its local pool, where the priority of a node is determined by the search strategy being employed. We further discuss this and other topics related to load balancing in Section 3.3.2 below.

Cuts. One of the advantages of branch and cut over generic LP-based branch and bound is that the inequalities generated at each node of the search tree may be valid and useful in the processing of search tree nodes in other parts of the tree. Valid inequalities are usually categorized as either *globally valid* (valid for the convex hull of solutions to the original MILP and hence for all other subproblems as well), or *locally valid* (valid only for the convex hull of solutions to a given subproblem). Because some classes of valid inequalities are difficult to generate, inequalities that prove effective in the current subproblem may be shared through the use of *cut pools* that contain lists of such inequalities for use during the processing of subsequent subproblems. The cut pools can thus be utilized as an auxiliary method of generating violated valid inequalities during the processing operation.

Because the number of cuts generated during execution of the branch-and-cut algorithm can be very large, careful management of these cuts is crucial. This includes not only periodic purging of duplicate, dominated, and "ineffective" cuts from the pool(s), but also the use of efficient data structures, called *representations*, for storing the cuts in a form that is independent of any LP relaxation or specific search tree node. A cut's representation is a compact description that contains information about how to add it to a particular LP relaxation and allows for the efficient calculation of the degree of violation with respect to a given fractional solution. Adding a cut to a given LP relaxation consists of constructing the row to be added to the constraint matrix and determining the corresponding right-hand side value, taking into account the current set of active variables. The representation is used not only to store each cut, but also to pass it from one processor to another when necessary. Prudent maintenance of the cut pools can provide a global picture of which cuts are the "most important," leading to significant improvements in the effectiveness of the algorithm.

In some implementations of branch and cut, it may be necessary or desirable to identify cuts or variables by assigning them unique global indices. The set of variables is static, so a priori assignment of global indices to each variable is easy. The set of potential valid inequalities, on the other hand, is not generally known or may be too large to index. In such a case, global indices can be issued from a central bank to avoid conflict, or can be generated by hashing the representation. The first approach is simpler, but creates another potential bottleneck operations. This bottleneck may be reduced by issuing large blocks of indices to individual processors for future allocation to generated cuts.

Branching Information. If a backward-looking branching method, such as one based on pseudo-costs, is used, then the sharing of historical information regarding the effect of branching can be important to the implementation of the branching scheme. The information that needs to be shared and how it is shared depends on the specific scheme used. For an in-depth treatment of the issues surrounding pseudo-cost sharing in parallel, see [40] and [16].

3.2 Task Management

In branch and cut, there are a number of distinct tasks to be performed and these tasks can be assigned to processors in a number of ways. The main tasks to be performed are:

- *Node processing*: From the description of a candidate node, the processing procedure produces either an improved bound or a feasible solution to the original MILP.
- *Node partitioning*: From the description of a processed node, the partitioning procedure is used to select a method of branching and subsequently produce a set of children to be added to the candidate list.
- *Cut generation*: From a solution to a given LP relaxation produced during node processing, the cut generation procedure produces a violated valid inequality (either locally or globally valid).
- *Pool maintenance*: Some processors may be assigned the task of managing either node or cut pools.
- *Load balancing*: One or more processors may be assigned the task of collecting information about the distribution of candidate subproblems globally and coordinating their redistribution when necessary (see Section 3.3.2).

The way in which these tasks are grouped and assigned to processors partly determines the parallelization scheme of the algorithm and its scalability. In Section 4, we discuss the scheme for assigning tasks to processors and coordinating them in two different software packages and analyze the effectiveness of each. Next, we discuss the main scalability issues surrounding the parallelization scheme.

3.3 Scalability Issues

3.3.1 Ramp-up Phase

In branch and cut, the ramp-up phase is usually defined to last roughly until there are enough candidate nodes available to occupy all available processors. The problem of reducing idle time during the ramp-up phase has long been recognized as a challenging one [25, 9, 16]. For instances in which the processing time of a single node is large relative to the overall solution time, idle time during the ramp-up phase can be one of the biggest contributors to parallel overhead. This is due to the amount of time required to generate a pool of candidate nodes sufficient to occupy all processors. There are two obvious strategies available for reducing idle time during this initial phase—accelerating the production of the initial pool of candidate nodes and occupying processors with auxiliary tasks that may help accelerate execution during the primary phase.

To occupy otherwise idle processors, two auxiliary tasks that can be undertaken during the ramp-up phase are calculation of an initial upper bound and various types of problem preprocessing. Determination of the initial upper bound involves execution of one or more heuristic solution generation procedures, which can themselves be parallelized. Preprocessing tasks can include the execution of traditional integer preprocessing algorithms, the processing of the root node, the computation of initial pseudo-costs, and the tightening of bounds on variable values by comparison of reduced cost to the optimality gap. These tasks may also be parallelized.

Although processors can be effectively occupied with auxiliary tasks such as those described above, the contribution of these auxiliary tasks to reductions in running time may

not be large enough to justify their execution if the assigned processors could be occupied with the processing of candidate nodes. It is therefore still advantageous to keep the ramp-up phase as short as possible. Unfortunately, effective techniques for accelerating the generation of the initial pool of candidate nodes have proven elusive. Following processing of the root node, it is clear that generation of the remaining pool can be parallelized to a limited extent by distributing nodes as they are produced rather than waiting until a full complement is available. This will obviously help, but has limited effect if the processing of the first few nodes in the tree is time-consuming with respect to the overall solution time. Another obvious approach is to reduce the processing time of each node in the ramp-up phase, either by limiting cut generation or by branching more quickly than one would otherwise.

Forrest *et al.* noted that effective branching is most important in the early phases of the algorithm when it has the biggest impact on the eventual size of the search tree [21]. In a sequential algorithm, additional time spent making branching decisions, e.g., exploring additional strong branching candidates, does result in an overall reduction in tree size, which can translate into a reduction in solution time (up to a point). From this perspective, one would be tempted to devote more time to branching during the ramp-up phase, not less. This creates another challenging tradeoff between limiting ramp-up time and making good branching decision early. This tradeoff is an important one, but one that has also proved difficult to analyze. Informal experiments aimed at limiting ramp-up time by forcing early termination of the node processing loop shown in Figure 1 and thus allowing the branching step to be invoked more quickly failed to produce positive results. Likewise, efforts at limiting the time devoted to performing the branching operation (e.g., considering fewer strong branching candidates, for instance) have also failed. A promising approach that has not yet been tested is to devote *more* effort to branching near the top of the tree, but to parallelize the branching procedure itself. With a strong branching approach, this could easily be done by distributing the candidates for pre-solving to multiple processors. It might also be possible to employ a branching rule that generates more than the usual two children. Such a rule would presumably produce candidate nodes more quickly, but might also increase the size of the search tree. To our knowledge, little or no investigation of such branching rules has been undertaken.

3.3.2 Primary Phase

During the primary phase, the main issue that arises is how to most effectively generate and share the two main types of knowledge that arise in branch and cut—node descriptions and valid inequalities. Node generation and sharing involves effective strategies for searching, load balancing, and branching, while cut generation and sharing involves effective strategies for managing the extremely large number of valid inequalities that may be generated during the course of the algorithm. In each case, there is a difficult balance to be struck between the efficiencies gained by centralization of knowledge, which leads to a much clearer global picture, and decentralization of knowledge, which allows the tasks associated with knowledge management to be distributed among a larger number of processors.

Search Strategy. The choice of search strategy has a number of important implications in a parallel implementation that go beyond those already discussed in Section 2.1.2. Be-

cause their implementation often involves the existence of multiple node pools from which candidates can be drawn, parallel algorithms require us to view search strategies in an entirely new light. The lack of global information about available candidate subproblems, for instance, renders the implementation of a pure best-first search strategy in parallel impractical and inefficient. Not only would it be difficult and time-consuming to identify the candidate subproblem with the smallest lower bound globally at a given point in time, but the cost associated with moving such subproblems from their current locations to available processors would be prohibitive.

In general, executing a given search strategy in parallel exactly as it would be executed sequentially is difficult because of the additional movement of nodes between processors that may be required. Because of this movement of nodes, parallel search strategies are inextricably linked with methods for load balancing, discussed below. As in sequential branch and bound, there is a tradeoff between strategies that attempt to minimize the overall size of the search tree and strategies that emphasize the generation of feasible solutions. With the additional expense that may be incurred retrieving nodes from remote pools, however, parallel search strategies require consideration of a third dimension to this tradeoff—that between choosing nodes available locally for processing and choosing those that must be retrieved from a remote pool. Parallel search strategies may give a higher priority to nodes available locally when deciding what node to process next.

The specification of a search strategy in parallel thus has two distinct elements—a local strategy and a global strategy. The local strategy specifies what nodes are preferred among those available locally, whereas the global strategy, tied closely to the load balancing scheme, specifies how nodes should be shifted between processors in order to pursue a given global strategy. We next present an overview of load balancing strategies. In Section 4, we describe the implementation of two very different search and load balancing strategies used in two software packages for solving MILPs in parallel.

Load Balancing. The tradeoff between centralization and decentralization of knowledge is most evident in the mechanism for sharing node descriptions among the processors. This is perhaps the most challenging aspect of implementing parallel branch and cut during the primary phase and has been studied by a number of authors [19, 31, 33, 39, 63, 65]. Effective load balancing methods reduce both idle time associated with task starvation and the performance of redundant work. In branch and cut, the goal is to ensure that "high-priority" nodes, i.e., those favored by the search procedure (typically nodes with small lower bounds), are available locally on all processors that require them. This means moving node descriptions from processors with an excess of high-priority nodes to those with a deficit of such nodes. Without such movement of nodes during the algorithm, the node pools of some processors would degrade in quality or become empty, while the node pools of other processors would contain too many high-priority nodes to be processed locally.

Before nodes can be shared, the first challenge is to recognize that an imbalance exists. If the size of a local pool becomes too small, this is easy to identify, but determining if the *quality* of the pool is too low may be more difficult, since this requires knowledge of the quality of the node pools residing at other processors. Once a deficit has been identified, it must be repaired through an exchange process that moves nodes from pools with excesses to pools with deficits.

Any scheme for accomplishing the load balancing described above involves some degree of centralization of knowledge in the form of either bounds or node descriptions themselves. Such centralization can either be undertaken through the maintenance of permanent centralized knowledge pools or through a periodic knowledge gathering and redistribution process. In Section 4, we discuss two software packages, one of which takes the first approach and one of which takes the second.

Cut Sharing. Cut sharing, while not as challenging or as critical as the sharing of node descriptions, can be important for applications where effective classes of structured valid inequalities are known, but are difficult to generate. The well-known Traveling Salesman Problem (TSP) and the related Vehicle Routing Problem, discussed in Section 5, are good examples of such problems. A number of interesting and difficult questions arise in designing strategies for cut management. The first of these is how many cut pools to maintain and whether to restrict the contents of these pools by type, by validity (i.e., by subtree in which the cuts are valid), or by some other criterion. Other questions to be answered include:

- Which locally generated cuts should be shared globally?
- When should a remote request for cuts be made and to which pool or pools should it be sent (if there is more than one)?
- Which cuts should be returned in response to such a request?
- Which cuts should be retained retain and which discarded when memory becomes a limitation?

Unfortunately, the answers to most of these questions depend strongly on the application. We will discuss one implementation of cut sharing in Section 4.1. For an in-depth discussion of a wide range of options for sharing cuts, see [40].

3.3.3 Ramp-down Phase

In the ramp-down phase, we may again face the problem of not having enough nodes globally to occupy all available processors, as in the ramp-up phase. This problem is not as well recognized or studied in the literature as the problem of reducing ramp-up time, but it can also pose a serious scalability issue in some cases, as we demonstrate in Section 5. The methods available for addressing the problem are similar to those employed during the ramp-up phase, but further study of this phase of the computation is merited.

3.3.4 Properties of Instances

In addition to properties of the algorithm, properties of the instance or instances to be solved can also have a dramatic effect on scalability. If such properties can be predicted or discovered in advance, modification of the algorithm may lead to improved efficiency. In the case of a MILP instance about which nothing is initially known, the situation is more difficult and efficiency can suffer, even with a well-designed algorithm.

The main properties that are relevant when considering solution of a particular problem class are:

- the length of time it takes to process a search tree node relative to total solution time;
- the length of time it takes to process nodes shallow in the tree relative to the length of time it takes to process nodes at deeper levels; and
- the effectiveness of the initial upper bounding method.

In the first two cases, if the length of time it takes to process a search tree node is short relative to solution time, then the number of nodes in the search tree may be very large and this could result in an excessive amount of time spent load balancing. On the other hand, if the processing time per node is long relative to solution time, especially near the top of the tree, then the ramp-up phase may rob the overall algorithm of efficiency.

The effect of the initial upper bounding method is somewhat less clear. If the initial upper bound is optimal (or close to optimal), then the goal of the algorithm becomes proving optimality of the known solution. In this case, the search order (and hence load balancing) is much less important, and scalability is generally easier to achieve. On the other hand, if the initial upper bound is not near optimal, the solution time depends to a much greater extent on when an optimal or near-optimal solution is discovered during the search process. One may be fortunate and discover the optimal solution relatively early in the process when searching in parallel, while not discovering the solution until late in the sequential search. Such early discovery of the optimal solution can reduce the size of the search tree and make node processing more efficient by allowing the bounds on variables to be tightened based on the size of the reduced costs and the optimality gap.

4 Algorithms

In this section, we briefly review the implementation of two frameworks we have developed based on the concepts just described. The first is SYMPHONY, which is written in C and takes a very centralized approach to knowledge management. The second is ALPS, a C++ framework, which takes a completely decentralized approach to knowledge management.

4.1 SYMPHONY

SYMPHONY is both a "black-box" MILP solver with an associated callable library and a highly customizable framework for implementing branch-and-cut algorithms tailored to specific applications [52, 59]. SYMPHONY evolved from the COMPSys framework of Ralphs and Ladányi [36, 55] and is now part of the Computational Infrastructure for Operations Research (COIN-OR) [12]. The source code for packaged releases is available for download and is licensed under the open-source Common Public License (CPL) [58]. The source code for the current development version is available from the COIN-OR source code repository [12]. SYMPHONY is fully documented, with a complete set of examples and specialized solvers included with the distribution. The solver contains several advanced features not available in other MILP codes, such as the ability to solve multi-criteria MILPs, the ability to warm

start the solution procedure, and the ability to perform basic sensitivity analyses. The basic algorithm can be modified and customized solvers built by the user through the use of numerous parameters and callback functions.

4.1.1 Knowledge Management

SYMPHONY employs a highly centralized knowledge management scheme. In particular, the algorithm maintains a single central node pool from which nodes are distributed for processing. The use of a central node pool means that accurate global information about the tree is always available, but only from the central server. This simplifies certain aspects of the parallel implementation dramatically, but also limits scalability, as the central pool inevitably becomes a bottleneck when a large number of processors are dedicated to node processing. Cuts can be stored either in a single central pool or in multiple pools, each dedicated to a particular subtree. Using a single cut pool creates another potential bottleneck, but the use of multiple pools may decrease effectiveness by limiting the extent to which cuts are shared throughout the search tree. We will discuss these tradeoffs in more detail in Section 4.1.3 below.

4.1.2 Task Management

SYMPHONY is implemented in C using a modular design that enables easy and highly configurable parallelization. There are five independent modules, each responsible for a distinct set of tasks. The *master* module is responsible for overall execution, including input and output, and the creation of all other modules. Of the other four modules, the *node processing* (NP) and *cut generation* (CG) modules are responsible for generation of new knowledge (node descriptions, feasible solutions, and valid inequalities), while the *tree management* (TM) and *cut management* modules are responsible for storage and management of previously generated knowledge (node descriptions and valid inequalities). The current incumbent is stored and broadcast by the master module.

SYMPHONY can be built and run in three different modes—sequential, parallel with shared memory, and parallel with distributed memory. We concentrate here on the distributed memory implementation. As a distributed memory code, the modules can be combined in a number of different ways at compile-time in order to produce executables capable of performing multiple functions. The most typical configuration is to combine the master, tree management, and cut management modules into a single executable responsible for all knowledge storage functions, while combining the node processing and cut generation modules into a single executable responsible for all knowledge generation processes. This configuration is very efficient for small numbers of processors, but makes the central knowledge storage process an even more significant computational bottleneck. For larger numbers of processors, locating the cut manager(s) on separate processors can provide some relief from this. Separating the master and tree manager modules or the node processing and cut generation modules provides little advantage in most cases. Below, we provide some details of the purpose and implementation of each module.

The Master Module. The master module performs problem initialization and I/O, stores problem data, stores and reports the results of solution procedure calls, and provides the user interface. Data stored in the master module is persistent and is maintained between solution procedure calls. Such data include information for warm starting of the solution procedure and lists of cuts generated by previous solution procedure calls. Retention of these data can facilitate the solution of sequences of slightly modified instances (such as those that arise in decomposition algorithms, among others).

The master module is not heavily tasked once the computation has begun, but functions independently in order to monitor the status of the solution procedure. The specific 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 (see the description of the TM module below).
- Create the tree manager module and initialize the algorithm by sending start data (either a description of the root node or warm starting data) to the TM module.
- Collect the output during the solution procedure and pass it to the output device.
- Process requests for problem data from remote processors.
- Receive new solutions and store them.
- Ensure that all other modules are still functioning.
- Store and report results at algorithm termination.
- Store any persistent data, such as warm starting information, needed for future solution procedure calls.

The Tree Management Module. Each time SYMPHONY's solution procedure is invoked, a new TM module is created to control the overall execution of the algorithm. The main task of the TM module is to act as the central node pool, maintaining the search tree and distributing descriptions of the nodes to be processed to the NP modules. After each node is processed, the responsible NP module sends the results to the TM module and queries it for the next next node to be processed, which is either a child of the node whose processing was just completed (and hence available locally) or a new node from the list of candidates, depending on the search strategy being employed. Specific functions performed by the TM module are:

- Receive start data and initialize the list of candidates for processing (typically just the root node).
- Handle requests from NP modules to determine the next subproblem for processing.

- Receive the results of node processing and partitioning, create the child nodes (if necessary), and add them to the list of candidate subproblems.
- 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.

Because of the single-pool approach taken by SYMPHONY, the search tree can be stored very efficiently. The set of active inequalities and the description of the basis tend not to change much from parent to child, so all of these data can be 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. This allows efficient generation of the next node to be processed.

The size of the node descriptions themselves is limited 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 normally consists 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 main importance of the core is that its description can be stored statically at each of the processors and need not be part of each individual node description. This saves on both node set-up costs and communication costs, as well as making storage of the search tree more efficient.

The Cut Management Module. The concept of a cut pool was first suggested by Padberg and Rinaldi [51], based on the observation that inequalities generated while processing a given node in the search tree may potentially be useful at other nodes. Since generation of cuts is sometimes a relatively expensive operation, the cut management module can maintain a list of the "best" or "strongest" cuts found in the tree thus far for use in processing future subproblems. The cut management modules are thus knowledge pools that can be queried by the NP modules for violated valid inequalities to be added to the current LP relaxation. As mentioned previously, multiple cut management modules can be used, in which case each one services a separate subtree. More explicitly, the functions of the cut management 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.

The Node Processing Module. The NP modules are responsible both for processing and partitioning of search tree nodes. Each node is processed completely and partitioned before the results are sent to the TM module—there is no option for partial processing

or processing without partitioning. These operations are, of course, central to the performance of the algorithm and comprise a large part of the "useful" work. Search tree nodes are processed in an iterative loop similar to that described in Figure 1. First, the initial LP relaxation is solved, then cuts are generated based on that solution, the relaxation is augmented, and the cycle repeats. This continues until either no more new cuts can be generated or the bound improvement in each round becomes too small, 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 ranges. Functions performed by the NP module are:

- Inform the TM module when a new subproblem is needed.
- Receive a subproblem and process it, in conjunction with the cut management module.
- 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 TM module.
- Perform the fathoming operation.

The Cut Generation Module. The CG module performs only one function—that of generating valid inequalities violated by the current LP solution and sending them back to the requesting NP module. The current implementation allows for only one dedicated CG module per NP module. The functions performed by the cut generator module are:

- Receive an LP solution and attempt to separate it from the convex hull of all solutions.
- Send generated valid inequalities back to the NP module.
- When finished processing a solution vector, inform the NP module not to expect any more cuts in case it is still waiting.

4.1.3 Scalability

The main computational task in SYMPHONY consists of the processing and (possible) partitioning of a single subproblem. After processing and partitioning a subproblem, the NP module must return the results to the TM module and query it to determine its next task. SYMPHONY uses global indices to identify cuts, so the NP modules must also query the TM module for the assignment of these indices, as well as querying the cut management modules(s) for violated valid inequalities. Because of these synchronization steps and the centralized approach to knowledge storage, load balancing is not an issue in SYMPHONY. The NP modules have no local node pools, so work distribution is simply a matter of sending the candidate node with the highest priority to each NP module at the time of its request or instructing it to continue working on one of the children of the node it just finished processing.

As with most implementations of parallel branch and bound, ramp-up time is a major contributor to parallel overhead. If desired, SYMPHONY can employ a "quick branching"

strategy similar to the one described earlier, 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 strategy is pursued until all processors have useful work to do, after which the usual algorithm is resumed. This strategy has shown little success, however. No other strategies for decreasing ramp-up time or otherwise employing processors during this phase are available in SYMPHONY.

The fact that all NP modules must frequently query the single TM module also leads to scalability issues during the primary phase. Almost all of the overhead in SYMPHONY during this phase is either communications overhead (time spent packing, sending, receiving, and unpacking messages) or time spent idle waiting for an answer to a query sent to a knowledge pool. The majority of this idle time is associated with communication between the TM and NP modules regarding node processing. Additional idle time may be incurred waiting for the cut pool to answer its queries. When either of these knowledge storage modules become overtasked, idle time can increase dramatically. For this reason, SYMPHONY cannot typically achieve good scalability beyond 32 processors, as we discuss in Section 5.

4.2 ALPS

The *Abstract Library for Parallel Search* (ALPS) [61, 70] is a C++ framework for implementing customized versions of parallel *tree search*. Tree search is an algorithmic paradigm in which the nodes of a directed, acyclic graph are systematically searched in order to locate one or more *goal nodes*. A wide variety of specialized algorithms, including branch and cut, can be classified as tree search algorithms. Because of its more general approach, ALPS supports the implementation of a wider variety of algorithms and applications than SYMPHONY. Like SYMPHONY, ALPS is part of the COIN-OR repository [12]. The source code is licensed under the open-source Common Public License (CPL) and is available from the COIN-OR source code repository [12]. The library of C++ classes that constitutes ALPS can be derived to implement specialized classes that define various tree search algorithms. Two libraries built on top of ALPS, called the *Branch, Constrain, and Prices Software* (BiCePS) and the BiCePS Linear Integer Solver (BLIS), implement the specialized methods required for branch and cut. Two prototype solvers built with ALPS, one for solving the well-known knapsack problem and one for solving generic MILPs, are also available for download.

4.2.1 Knowledge Management

In contrast to SYMPHONY, ALPS takes an almost completely decentralized approach to knowledge management. In [62], building on ideas in [67], we proposed a tree search methodology driven by the concepts of knowledge discovery and sharing discussed in Section 2.2.3. This methodology is the basis for the class structure of ALPS. A central notion in ALPS is that all information generated during execution of the search is treated as knowledge and is represented by C++ objects derived from a single common base class described below.

The most fundamental knowledge objects generated during the search are the descriptions of the search tree nodes themselves, which are organized locally into subtrees. To avoid the bottlenecks associated with central storage of the entire candidate list, every processor

responsible for node processing hosts its own local node pool from which the node processing task can draw new candidates. The local node pools collectively contain the global list of candidate nodes, which are shared through the load balancing procedure described in Section 4.2.3. To further avoid the introduction of bottlenecks, load balancing is performed using a three-level scheme we call the *master-hub-worker* paradigm, also discussed in Section 4.2.3 below.

The `AlpsKnowledge` class is the virtual base class for any type of information that must be shared or moved from one processor to another. `AlpsEncoded` is an associated class that contains an encoded or packed form of an `AlpsKnowledge` object, which consists of a bit array containing the data needed to replicate the object. This representation takes less memory than the object itself and is appropriate both for storage of knowledge and for transmission of knowledge between processors. The packed form is also independent of type, which allows ALPS to deal effectively with user-defined knowledge types. To avoid the assignment of global indices, ALPS uses hashing of the packed form to identify duplicate objects. ALPS has the following four native knowledge types:

- `AlpsSolution`: Contains the description of a goal state or solution to the problem being solved.
- `AlpsTreeNode`: Contains the data and methods associated with a node in the search graph, including a node description (of type `AlpsNodeDesc`) and the definitions of the process and branch methods.
- `AlpsModel`: Contains the data describing the original problem.
- `AlpsSubTree`: Contains the description of a subtree, which is a hierarchy of `AlpsTreeNode` objects, along with the methods needed for performing a tree search.

The first three of these classes are virtual and must be defined by the user in the context of the problem being solved. The last class is generic and application-independent.

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

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

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

4.2.2 Task Management

In ALPS, each processor hosts a single, multi-tasking executable controlled by a *knowledge broker* (KB). The KB is tasked with routing all knowledge to and from the processor and determining the priority of each task assigned to the processor. A crude version of threading allows the single executable to performs multiple tasks, which can include hosting multiple knowledge pools, processing candidate nodes, or generating application-specific knowledge, e.g., valid inequalities. Each specific type of knowledge, represented by a C++ class derived from `AlpsKnowledge`, must be registered at the inception of the algorithm so that the KB knows how to route it when it arrives and where to send requests for knowledge from other KBs. The KB associated with a particular KP may field two types of requests on its behalf: (1) new knowledge to be inserted into the KP, or (2) a request for relevant knowledge to be extracted from the KP, where "relevant" is defined for each category of knowledge with respect to data provided by the requesting process. A KP may also choose to "push" certain knowledge to another KP, even though no specific request has been made.

Derivatives of the `AlpsKnowledgeBroker` class implement the KB and encapsulate the desired communication protocol. Switching from a parallel application to a sequential one is simply a matter of constructing a different KB object. Currently, the protocols supported are a serial layer, implemented in `AlpsKnowledgeBrokerSerial`, and an MPI [27] layer, implemented in `AlpsKnowledgeBrokerMPI`.

4.2.3 Scalability

In contrast to SYMPHONY, the basic computational task in ALPS is to process all the nodes of a subtree with a given root node. Each worker is capable of processing an entire subtree autonomously and has access to all of the methods needed to manage a sequential tree search. The potential for increased granularity reduces idle time due to task starvation, but, without proper load balancing, may increase the performance of redundant work. Because the processing of a subtree can be an extremely lengthy and unpredictable procedure, the task can be interrupted at any time for the purpose of load balancing and may even be preempted if higher-priority work is made available. By storing subtrees as a complete unit, it is possible to use a data structure based on the concept of differencing introduced earlier in Section 4.1.2. This may help to minimize memory requirements, which could potentially be increased by the decentralized node storage scheme.

To overcome the drawbacks of the master-worker approach employed by SYMPHONY, ALPS employs a *master-hub-worker* paradigm, in which a layer of "middle management" is inserted between the master process and the worker processes. In this scheme, a *cluster* consists of a *hub* and a fixed number of *workers*. Within a cluster, the hub manages the workers and supervises load balancing, while the master ensures the load is balanced globally. As the number of processors is increased, clusters can be added in order to keep the load of the hubs and workers relatively constant. The workload of the master process can be managed by controlling the frequency of global balancing operations. This scheme is similar to one implemented by Eckstein et al. in the PICO framework [16], except that PICO does not have the concept of a master. The decentralized approach maintains many of the advantages of global decision making while reducing overhead and moving much of the burden for load balancing and search management from the master to the hubs.

This burden is then further shifted from the hubs to the workers by increasing the task granularity, as described below.

Because all processes are completely autonomous in ALPS, the biggest scalability issues are idle time during ramp-up and effective load balancing. In ALPS, each node has an associated priority that indicates the node's relative "quality," i.e., the probability that the node or one of its successors is a goal node. In assessing the distribution of work to the processors, we consider both *quantity* and *quality*. ALPS employs a three-tiered load balancing scheme, consisting of *static*, *intra-cluster dynamic*, and *inter-cluster dynamic* load balancing.

Static load balancing, or *mapping*, takes place during the ramp-up phase. The main task is to generate the initial pool of candidate nodes and distribute them to the workers to initialize their local node pools. ALPS uses a *two-level root initialization* scheme, a generalization of the *root initialization* scheme of [31]. During static load balancing, the master creates and distributes a user-specified number of nodes to the hubs. The hubs in turn create and distribute a user-specified number of successors to their workers, after which the workers initialize their subtree pools and begin. Time spent performing static load balancing is the main source of ramp-up, which can be significant when node processing times are large.

Inside a cluster, the hub manages dynamic load balancing by periodically receiving workload reports from cluster members. If it is found that the qualities are unbalanced, the hub asks workers with a surplus of high-priority nodes to share them with workers that have fewer such nodes. Intra-cluster load balancing can also be initiated when an individual worker reports to the hub that its workload is below a given threshold. Upon receiving the request, the hub asks its most loaded worker to donate nodes to the requesting worker.

The master is responsible for balancing the workload among hubs, which periodically report their workload information to the master. The master has an approximate global view of the system load and the load of each cluster at all times. If either the quantity or quality of work is unbalanced among the clusters, the master identifies pairs of *donors* and *receivers*. Donors are clusters whose workloads are greater than the average workload of all clusters by a given factor. Receivers are clusters whose workloads are smaller than the average workload by a given factor. Donors and receivers are paired and each donor sends nodes to its paired receiver.

A unique aspect of the load balancing scheme in ALPS is that it takes account of the differencing scheme for storing subtrees. In order to allow efficient storage of search tree nodes using differencing, we try at all times to ensure that search tree nodes are shared in a way such that those sent and stored together locally constitute connected subtrees of the search tree. To accomplish this, groups of candidate nodes that constitute the leaves of a given subtree are shared as a single unit, rather than being shared as individual nodes. Each subtree is assigned a priority, defined as the average of the priorities of a given number of its best nodes. During load balancing, the donor chooses the best subtree in its subtree pool and sends it to the receiver. If a donor does not have any subtrees to share, it splits the subtree that it is currently exploring into two parts and sends one of them to the receiver. In this way, differencing can still be used effectively, even without centralized storage of the search tree.

5 Computation

To give the reader a better appreciation for the performance of the schemes we have just described and for the specific components of overhead that are most prevalent, we now present the results of computational studies involving the solution of MILP instances from three different problem classes using the currently available beta version of SYMPHONY 5.1 and associated solvers. The LP relaxations were solved with the most recent version of the COIN-OR LP solver [12], also available under the CPL. The problem classes here have been chosen because they illustrate the wide range of properties that problem instances can have and the scalability issues that arise because of them. All tests were performed on a Beowulf cluster with 60 1.8 GHz 64-bit AMD Opteron processors, each with 1G local memory. In Section 5.1, we analyze the solution of generic MILP instances, a wide ranging class containing instances whose properties are generally difficult to predict in advance. We report results from instances that exhibited good scalability, as well as a few that did not. In Sections 5.2 and 5.3, we discuss the solution of two classical combinatorial optimization problems, the Vehicle Routing Problem (VRP) and the Set Partitioning Problem (SPP). In the case of the VRP, node processing times are short and scalability is relatively easy to achieve. In the case of the SPP, node processing times can be extremely long and achieving scalability is much more difficult.

For all runs, SYMPHONY was configured so that the master and TM modules ran as a single process, with multiple combined NP and CG modules performing the node processing and cut generations tasks. The number of NP/CG modules used ranged from 1 to 32. Except where noted, no global cut pools were used. For each set of runs, the number of NP/CG modules used in the computation is identified. In parallel branch and cut, the size of the search tree is subject to a good deal of random fluctuation due to the asynchronous nature of the search. To reduce the effect of these fluctuations on the analysis, we performed three identical runs of each experiment. The numbers that appear in the tables of results are averages over these runs. We also report the timing information on a "per node" basis, which separates the effect of fluctuations in the number of nodes from fluctuations in the time needed to process a node, resulting in a much more consistent view of the trends as the number of processors increases.

The results in the tables that appear here break the main sources of parallel overhead down into specific identifiable components that were significant to SYMPHONY's performance. In the tables, the column headers have the following interpretations:

- *Tree Size* is the number of nodes in the search tree. Observing the change in the size of the search tree as the number of processors is increased provides a rough measure of the amount of redundant work. Ideally, the total number of nodes explored stays constant as the number of processors is increased.

- *Ramp-up* and *Ramp-down* are the total accumulated idle time during periods before and after the primary phase when the queue did not contain enough nodes to keep all processors busy.

- *Node Pack* is the time spent by the TM module generating node descriptions. This can be significant because explicit descriptions need to be constructed from the differenced form in which they are stored. Note that this is technically not parallel overhead,

since it must be incurred even in the sequential algorithm. However, it does represent computation required to support the differencing mechanism, which is needed in part because of SYMPHONY's centralized node scheme.

- *Idle Nodes* is the idle time spent by the NP module waiting for a new node description to be sent from the TM module.
- *Idle Cuts* is the idle time spent by the NP module waiting for valid inequalities to be sent from the cut management module (if applicable).
- *Idle Index* is the idle time spent by the NP module waiting for the TM module to assign indices to valid inequalities still active at the termination of processing of a search tree node or those being sent to the cut management module.
- *Idle Diving* is the idle time spent by the NP module waiting for instructions from the TM module regarding whether to continue processing one of the available children of the current node or wait for a new node from the global candidate list.
- *CPU sec* is the total CPU time used by all modules. Note that this does not include idle time and hence can be significantly different from wallclock time, described below.
- *Wallclock* is the amount of real time used by the solution procedure from start to finish. By multiplying wallclock running time by the number of processors and subtracting CPU time, one can obtain a rough estimate of the idle time incurred by all processes as a whole.
- *Eff* is the parallel efficiency and is equal to p time the wall clock running time with 1 NP/CG module divided by the wall clock running time with p NP/CG modules. In other words, it is approximately the percentage of time spent by all processors of the parallel algorithm doing "useful work," where the amount of useful work to be done is measured by the running time for the sequential algorithm. Note that the statistic reported here is not precisely the parallel efficiency described in Section 2.2.2, since the experiments with 1 NP/CG module were done in parallel as well (the single NP/CG module was running in parallel with the TM module) and since we use the number of NP/CG modules (not the number of processors) as the baseline. However, the resulting numbers give a clear picture of the trends in efficiency and this should not cause confusion.

5.1 Generic MILP

The test problems discussed below were selected from MIPLIB3 [8], MIPLIB2003 [45], and a suite of instances available from the Computational Optimization Research at Lehigh (COR@L) Web site [43]. For these tests, SYMPHONY was used as a black box solver, with valid inequalities generated using subroutines from the Cut Generator Library, also part of the COIN-OR repository [12]. Strong branching was used to make branching decisions and the search strategy was a hybrid diving strategy in which one of the children of a given node was retained as long as its bound was within a given percentage of the best available.

Table 1 in the Appendix shows the results of the first set of experiments, in which SYMPHONY was run with default settings and no a priori upper bound. Detailed results

are shown for each instance in the test set for the runs with a single NP/CG module and summary results only for all other runs. Table 2 shows the detailed results for the run with 32 NP/CG modules. For most of these instances, the time needed to process a search tree node is small in comparison with the overall running time, which tends to lead to good scalability. The results reflect this to a large extent, but as expected, overhead increases across the board as the number of NP/CG modules is increased. The increase from 16 to 32 NP/CG modules results in a much more significant amount of parallel overhead and a corresponding drop in efficiency. It is evident from these results that SYMPHONY will probably not scale well beyond approximately 32 NP/CG modules for instances with properties similar to these.

Examining each component of overhead in detail, we see that both ramp-up and ramp-down time grow significantly as the number of NP/CG modules is increased. This overhead is predictable, but difficult to eradicate. Time spent by the tree manager constructing node descriptions (*Node Pack*) remains relatively constant, as expected, and is not a scalability issue. The three columns representing idle time spent by the NP/CG modules waiting for various queries to be answered by the TM module are the most significant and addressable sources of inefficiency for SYMPHONY. The contention associated with the distribution of new node descriptions (*Idle Node*) is the most significant, but would be difficult to address without completely abandoning SYMPHONY's master-worker architecture. The idle time spent waiting for global indices to be assigned to cuts (*Idle Index*) and for decisions regarding whether to retain one of the children of the current node for processing (*Idle Dive*), however, might be reduced with a redesign of SYMPHONY's methodology. Pre-assigning blocks of indices to each NP/CG module could help alleviate the first of these bottlenecks, while more autonomy with respect to deciding whether or not to dive could help alleviate the second. These improvements would significantly improve SYMPHONY's scalability, but contention at the TM module could not be eliminated entirely without moving away from SYMPHONY's master-worker architecture.

To assess the degree of performance of redundant work, we examine the trends in the total number of nodes in the search tree. For these experiments, as the number of NP/CG modules is increased, the number of nodes in the search tree actually drops slightly, accounting for the superlinear speedup observed with 4 and 8 NP/CG modules. This drop is presumably due to the fact that feasible solutions are difficult to find for some instances and are discovered earlier in the search process during the parallel runs, resulting in smaller search trees. Overall, there is no evidence of additional redundant work being performed in the presence of parallelism.

To test the effect of having a good a priori upper bound, we performed a similar set of experiments in which the optimal solution value was provided a priori, so that the goal was simply to prove optimality of a known solution. The results (with a slightly smaller test set) are shown in Table 3. In this case, it is advantageous to follow an unconditional diving strategy in which the child of the current node is always preferred when there is one. This eliminates redundant work, but the results still exhibit a very slight increase in the number of search nodes as the number of NP/CG modules is increased. This hurts the parallel efficiency, but the provision of an a priori upper bound still improves solution times across the board in comparison to those in Table 1. Note that the total amount of overhead, especially the ramp down and the idle time spent waiting for new node descriptions to be sent from the TM module are very significantly reduced for these runs over the runs with

default settings. This is because the depth-first strategy employed requires far fewer new candidate nodes to be sent from the TM module to the NP/CG modules. Despite this, the calculated efficiencies are similar because the basis of comparison in each case is itself a parallel run with a single NP/CG module. Because the baseline run with the depth-first strategy exhibits a much lower level of overhead to begin with than the baseline run with the default strategy, the relative efficiencies calculated are similar. Despite this, the depth-first approach is clearly superior in terms of the absolute level of overhead if an priori bound close to optimal is known.

To illustrate how variations in individual instances can effect scalability, Tables 4–6 show the scalability of three instances not included in the larger test set. Instance `pk1` has a large search tree and relatively short node processing times, but still exhibits poor scalability, with all categories of overhead higher than expected. The large amount of ramp-down time is particularly intriguing for this instance. Instance `p2756` exhibits very significant increases in the size of its search tree (indicating the performance of redundant work) as the number of NP/CG modules increases, resulting in increased overall running times beyond two NP/CG modules. Finally, instance `nw04` is a set partitioning instance for which node processing times are extremely lengthy. Although initially exhibiting superlinear speedup (presumably due to earlier location of the optimal solution) and a drop in search tree size, the efficiency is eventually wiped out by significant increases in ramp-up time.

5.2 Vehicle Routing Problem

We next consider solution of instances of the well-known Vehicle Routing Problem introduced by Dantzig and Ramser [14]. In the VRP, a fleet of k vehicles with uniform capacity C must service known customer demands for a single commodity from a common depot at minimum cost. Let $V = \{1, \ldots, |V|\}$ index the set of customers and let the depot have index 0. Associated with each customer $i \in V$ is a demand d_i. The cost of travel from customer i to j is denoted c_{ij}. We assume that $c_{ij} = c_{ji} > 0 \ \forall i, j \in V \cup \{0\}$, $i \neq j$ and that $c_{ii} = 0 \ \forall i \in V \cup \{0\}$. By constructing an associated complete undirected graph G with vertex set $V \cup \{0\}$ and edge set E, we can formulate the VRP as an integer program.

The instances in the test set were selected from a test set maintained by the author [53] and were solved with an application written by the author using SYMPHONY. Both the instances and the solver are available for download [54]. The solver has previously been described in [56] and [55]. The VRP is an ideal candidate for parallelization, since node processing times are consistently small relative to overall solution times and good a priori upper bounds are easy to generate. Table 7 in the Appendix shows the results of the first set of experiments in which the solver was run with default settings using an upper bound determined heuristically (not necessarily optimal) before starting the solution procedure. Detailed results are shown for each instance in the test set for the runs with 1 NP/CG module and summary results only for all other runs. Table 8 shows detailed results for the run with 32 NP/CG modules. In terms of efficiency, the results are similar to those for the generic MILP instances in Section 5.1, but the levels of overhead are much smaller. As in the case of the generic MILP instances, because the baseline levels of overhead are small to begin with, the efficiencies are similar in each case.

Table 9 shows results of a slightly smaller set of instances with no a priori upper bounds

given. Here, the running times are longer and the search trees are bigger, but in terms of efficiency, the results are similar to those in Table 7. Finally, Table 10 shows the results with the same set of instances using a global cut pool. The use of the global pool decreases the size of the search tree, but introduces another source of idle time during node processing—time spent waiting for the global pool to return a set of violated valid inequalities. Overall, the additional overhead is worthwhile and is offset by a decrease in the number of search tree nodes. It is interesting to note, however, that the use of the global cut pool does cause a loss in efficiency. We conjecture that this is because the effect of the pool is lessened as the number of NP/CG modules is increased since effective inequalities have less time be shared after being discovered and therefore have a smaller effect. The difference in running times is relatively large for the runs with 1 NP/CG module, but there is almost no difference in the running times for the 32 NP/CG module runs, resulting in the observed relative loss of efficiency.

5.3 The Set Partitioning Problem

Finally, we consider the well-known Set Partitioning Problem. Given a ground set S of m objects and a collection $\mathcal{G} = \{S_1, \ldots, S_n\}$ of subsets of S with associated costs c_j, $1 \leq j \leq n$, the SPP is to select a subset of $\mathcal{G}$ of minimum (or maximum) total cost such that the selected members are disjoint and their union is equal to S. In other words, the problem is to choose a minimum (or maximum) cost partitioning of the ground set from among the members of $\mathcal{G}$. In contrast to the VRP, set partitioning problems are exceptionally difficult to solve in parallel because the time required to process a node can be extremely long. This is both because the LP relaxations are extremely difficult to solve and because it takes a significant amount of preprocessing, as well as a number of rounds of cut generation, to process each node. For this reason, most instances that can be solved in a reasonable time produce very small trees. This means that for most instances, the ramp-up phase dominates the total running time.

For our tests, we used the SPP solver originally implemented by Esö and described in [17]. The solver, which includes its own custom cut generator and other application-specific methodology, was updated to work with the current version of SYMPHONY by the author and is available for download [18]. The instances reported on here are from a test set also described in [17] and compiled from a number of different sources, including major airlines. In preliminary testing, most of the instances exhibited very poor scalability—it was not difficult to find instances for which the root node itself took several hours to process. The results in the tables are for five instances selected from those in [17] that exhibit the most reasonable scalability with SYMPHONY "out of the box." The small cardinality of this test set should serve to emphasize that these instances are the exception to the rule.

As in the previous section, Table 11 shows summary results obtained when solving these five instances with different numbers of NP/CG modules. Detailed results are shown only for the case with 1 NP/CG module. Table 12 shows results for the runs with 32 NP/CG modules. As could be expected, the results show that the node processing times are an order of magnitude larger than for the instances in Sections 5.1 and 5.2, so overhead is dominated very substantially by ramp-up and ramp-down time. Even for these relatively well-behaved instances, the efficiency that can be obtained with a straightforward implementation such as we have described here is very low. In [41] and [17], it was shown that achieving high

efficiency when solving the SPP can be achieved only if other parts of the algorithm, such as cut generation, execution of primal heuristics, and preprocessing, are also parallelized.

6 Conclusions and Future Work

In this paper, we have introduced the basic concepts and methodology required for parallelization of the branch-and-cut algorithm. Parallelizing branch and cut in a scalable fashion is a challenge that involves careful analysis of the costs and benefits of synchronization and the sharing of knowledge during the algorithm. Because this analysis can yield different results for different problem classes, it is not possible in general to develop a single ideal approach that will be effective across all problem classes. The synchronous approach taken by SYMPHONY is very effective for small numbers of processors, but is not scalable beyond about 32 processors, even under ideal conditions. Large-scale parallelism requires an asynchronous approach, such as that taken by ALPS, that avoids creating bottlenecks in communication. In future work, we plan to continue developing ALPS, with the goal of correcting SYMPHONY's deficiencies and achieving the scalability needed to run on much larger computational platforms.

Appendix: Tables of Results

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Dive	CPU sec	Wallclock	Eff
23588	591	0.00	0.00	0.02	0.09	0.02	0.09	52.56	53.52	
aligninq	761	0.00	0.00	0.07	0.43	0.02	0.09	225.42	229.14	
bell3a	24957	0.00	0.00	1.48	4.16	0.22	4.08	171.96	191.45	
blend2	2105	0.00	0.00	0.13	0.56	0.26	0.37	84.84	89.46	
enigma	3046	0.00	0.00	0.03	0.08	0.19	0.56	35.05	38.21	
fixnet6	1438	0.00	0.00	0.18	1.38	0.13	0.24	106.55	111.87	
gesa2	4634	0.00	0.00	0.97	6.79	0.74	1.06	950.50	1001.08	
geas2_o	662	0.00	0.00	0.12	0.93	0.09	0.12	128.66	135.04	
l152lav	1326	0.00	0.00	0.19	1.32	0.26	0.21	322.27	329.44	
misc07	12606	0.00	0.00	1.24	3.83	0.76	1.77	1186.99	1241.22	
mod008	1681	0.00	0.00	0.25	2.40	0.70	0.23	2124.19	2189.79	
pg	1868	0.00	0.00	0.07	0.40	0.30	0.45	402.43	418.29	
pp08a	46385	0.00	0.00	44.76	69.44	8.15	44.31	2186.69	2360.81	
pp08aCUTS	68093	0.00	0.00	108.32	137.13	11.96	114.16	4023.62	4373.30	
rgn	1284	0.00	0.00	0.05	0.42	0.08	0.15	10.92	12.00	
roy	346	0.00	0.00	0.01	0.08	0.06	0.05	12.53	13.18	
stein27	1589	0.00	0.00	0.05	0.48	0.31	0.14	57.38	60.06	
stein45	12108	0.00	0.00	1.44	8.82	4.37	1.68	2281.73	2324.85	
vpm1	10012	0.00	0.00	0.65	4.33	0.86	1.19	245.64	257.89	
vpm2	11444	0.00	0.00	1.51	7.60	1.39	1.56	459.96	477.37	
bienst1	13372	0.00	0.00	1.88	7.11	1.53	2.14	3180.44	3229.25	
p0282	538	0.00	0.00	0.03	0.28	0.09	0.09	30.37	36.70	
1 NP	220846	0.00	0.00	163.45	258.09	32.49	174.74	18280.70	19173.94	1.00
Per Node		0.0000	0.0000	0.0007	0.0012	0.0001	0.0008	0.0828	0.0868	
2 NP′s	224266	18.41	0.01	166.45	271.57	35.99	177.14	18357.87	9697.61	0.99
Per Node		0.0001	0.0000	0.0007	0.0012	0.0002	0.0008	0.0819	0.0865	
4 NP′s	222446	60.29	2.46	161.70	306.21	45.02	178.13	17822.92	4737.28	1.02
Per Node		0.0003	0.0000	0.0007	0.0014	0.0002	0.0008	0.0801	0.0852	
8 NP′s	213954	163.26	127.87	139.33	378.45	59.06	171.20	17433.65	2395.28	1.01
Per Node		0.0008	0.0006	0.0007	0.0018	0.0003	0.0008	0.0815	0.0896	
16 NP′s	215597	393.70	605.28	128.37	617.01	123.56	265.65	17127.50	1277.09	0.94
Per Node		0.0018	0.0028	0.0006	0.0029	0.0006	0.0012	0.0794	0.0948	
32 NP′s	212672	911.73	2282.13	148.22	2506.58	693.08	1035.35	16723.89	794.87	0.75
Per Node		0.0043	0.0107	0.0007	0.0118	0.0033	0.0049	0.0786	0.1196	

Table 1: Solving generic MILPs with SYMPHONY: Default settings and no a priori upper bound (summary)

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Dive	CPU sec	Wallclock
23588	582	27.60	0.00	0.03	0.07	0.02	0.09	56.29	2.76
aligninq	1139	98.19	0.00	0.14	0.67	0.02	0.13	362.38	14.90
bell3a	24128	2.46	10.39	2.43	33.77	1.13	16.10	172.69	7.54
blend2	2104	6.03	0.25	0.13	0.72	0.81	1.32	81.27	3.25
enigma	292	3.09	0.00	0.00	0.07	0.01	0.06	2.64	0.23
fixnet6	1705	12.59	0.00	0.23	1.27	0.19	0.34	119.65	4.44
gesa2	5324	148.49	0.21	1.33	8.50	1.12	2.27	1008.88	38.41
geas2_o	1123	44.27	0.00	0.18	1.04	0.11	0.23	184.86	8.06
l152lav	891	78.02	0.00	0.14	0.66	0.18	0.12	204.12	9.20
misc07	8989	33.60	0.78	0.96	4.84	0.95	2.15	809.08	27.42
mod008	1711	1.97	1294.42	0.29	50.88	1.12	0.64	1043.72	125.00
pg	1231	306.90	7.44	0.06	0.89	0.56	0.64	286.77	22.86
pp08a	41223	25.72	230.78	38.40	593.43	174.32	280.19	1825.54	92.44
pp08aCUTS	62529	22.22	707.14	93.64	1753.50	490.87	709.92	3766.79	214.47
rgn	1357	5.98	0.66	0.07	0.66	0.17	0.32	11.41	0.69
roy	555	4.26	11.69	0.02	0.97	0.08	0.09	17.35	1.51
stein27	1576	7.33	0.01	0.07	0.51	0.36	0.18	58.11	2.24
stein45	11972	28.68	3.92	2.39	13.27	7.87	4.93	2218.94	72.16
vpm1	14123	11.55	3.42	1.44	9.06	1.80	2.70	337.21	11.88
vpm2	13797	15.23	3.01	3.77	19.99	4.83	6.59	571.51	19.96
bienst1	14183	15.93	7.99	2.31	10.91	6.38	5.86	3522.96	112.94
p0282	2132	11.63	0.02	0.16	0.91	0.21	0.46	61.71	2.50
32 NP′s	212672	911.73	2282.13	148.22	2506.58	693.08	1035.35	16723.89	794.87
Per Node		0.0043	0.0107	0.0007	0.0118	0.0033	0.0049	0.0786	0.1196

Table 2: Solving generic MILPs with SYMPHONY: Default settings and no a priori upper bound (32 NPs)

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Dive	CPU sec	Wallclock	Eff
23588	259	0.00	0.00	0.00	0.01	0.01	0.02	21.86	22.25	
aligninq	1892	0.00	0.00	0.08	0.51	0.04	0.15	443.80	496.96	
bell3a	23048	0.00	0.00	0.44	1.35	0.52	2.98	174.86	191.40	
blend2	1099	0.00	0.00	0.01	0.01	0.04	0.06	30.78	32.12	
enigma	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	
fixnet6	529	0.00	0.00	0.02	0.12	0.03	0.06	29.66	30.97	
gesa2	1707	0.00	0.00	0.10	0.76	0.18	0.17	396.93	419.19	
geas2_o	2410	0.00	0.00	0.18	0.15	0.07	0.12	362.61	377.38	
l152lav	1553	0.00	0.00	0.09	0.54	0.22	0.16	289.20	295.78	
misc07	7795	0.00	0.00	0.22	0.78	0.37	0.87	648.57	665.83	
mod008	1979	0.00	0.00	0.04	0.25	0.20	0.20	98.27	150.59	
pg	793	0.00	0.00	0.10	0.58	0.06	0.11	202.40	208.07	
pp08a	26090	0.00	0.00	2.05	11.29	3.12	4.29	1099.47	1151.15	
rgn	1466	0.00	0.00	0.02	0.18	0.08	0.15	11.59	12.81	
roy	588	0.00	0.00	0.01	0.05	0.05	0.05	17.92	18.46	
stein27	1630	0.00	0.00	0.03	0.18	0.24	0.12	51.12	53.34	
stein45	12179	0.00	0.00	0.65	3.66	3.43	1.22	2071.94	2102.30	
bienst1	9721	0.00	0.00	0.79	0.21	0.18	0.24	1801.06	1824.56	
p0282	613	0.00	0.00	0.01	0.11	0.06	0.06	29.37	30.10	
1 NP	95352	0.00	0.00	4.84	20.74	8.91	11.03	7781.41	8083.28	1.00
Per Node		0.0000	0.0000	0.0001	0.0002	0.0001	0.0001	0.0816	0.0848	
2 NPs	96569	19.95	0.00	3.99	15.88	8.29	10.23	8011.45	4127.76	0.98
Per Node		0.0002	0.0000	0.0000	0.0002	0.0001	0.0001	0.0830	0.0855	
4 NPs	96006	72.39	0.00	4.11	21.79	9.75	11.74	7794.86	2212.54	0.91
Per Node		0.0008	0.0000	0.0000	0.0002	0.0001	0.0001	0.0812	0.0922	
8 NPs	97104	189.71	1.35	3.99	23.24	10.19	12.38	7752.63	1173.96	0.86
Per Node		0.0020	0.0000	0.0000	0.0002	0.0001	0.0001	0.0798	0.0967	
16 NPs	98960	470.38	18.91	4.37	27.61	11.35	13.67	7953.20	638.36	0.79
Per Node		0.0048	0.0002	0.0000	0.0003	0.0001	0.0001	0.0804	0.1032	
32 NPs	98223	1080.33	301.06	4.86	70.31	15.18	25.38	7793.92	349.24	0.72
Per Node		0.0110	0.0031	0.0000	0.0007	0.0002	0.0003	0.0793	0.1138	

Table 3: Proving optimality of generic MILPs with SYMPHONY (unconditional diving)

NPs	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Dive	CPU sec	Wallclock	Eff
1	239313	0.00	0.00	258.20	279.60	16.20	378.15	3775.52	4533.70	1.00
2	239461	0.01	0.36	256.94	302.26	28.24	404.23	3776.76	2296.48	0.99
4	238633	0.05	13.24	240.17	358.89	54.90	431.11	3751.31	1170.62	0.97
8	238588	0.19	197.95	236.39	930.87	163.85	652.97	3747.77	697.49	0.81
16	240238	0.58	966.25	248.13	3429.10	782.94	1933.32	3798.91	626.75	0.45
32	238795	1.53	2881.87	231.07	8725.32	1681.38	3701.61	3761.61	561.19	0.25

Table 4: Scalability of `pk1` with SYMPHONY (unconditional diving)

NPs	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Dive	CPU sec	Wallclock	Eff
1	353	0.00	0.00	0.17	1.02	0.04	0.05	100.41	130.37	1.00
2	619	0.42	0.00	0.28	1.66	0.07	0.11	161.51	84.29	0.77
4	11244	1.65	0.00	6.76	28.13	0.97	2.08	2195.41	571.95	0.05
8	19502	2.93	0.03	14.51	57.67	1.35	3.95	3906.41	507.37	0.03

Table 5: Scalability of `p2756` with SYMPHONY (default settings)

NPs	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Dive	CPU sec	Wallclock	Eff
1	198	0.00	0.00	1.28	4.94	0.00	0.04	1075.76	1138.38	1.00
2	85	27.29	0.00	0.33	1.21	0.00	0.02	439.92	246.35	2.32
4	84	101.21	61.65	0.17	107.40	0.00	0.01	463.87	181.29	1.58
8	87	941.67	0.00	0.20	220.33	0.00	0.02	473.78	181.56	0.78
16	86	2283.04	0.00	0.17	225.41	0.00	0.02	468.53	182.60	0.39
32	87	4954.92	0.00	0.20	220.20	0.06	0.06	474.64	185.75	0.19

Table 6: Scalability of `nw04` with SYMPHONY (default settings)

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Diving	CPU sec	Wallclock	Eff
hk48 − n48 − k4	110	0.00	0.00	0.00	0.03	0.01	0.01	16.77	17.35	
att − n48 − k4	384	0.00	0.00	0.01	0.05	0.04	0.05	36.61	37.77	
E − n51 − k5	41	0.00	0.00	0.00	0.01	0.01	0.01	14.86	15.49	
A − n39 − k5	1307	0.00	0.00	0.04	0.23	0.18	0.15	242.33	251.04	
A − n39 − k6	328	0.00	0.00	0.01	0.07	0.04	0.03	31.95	32.66	
A − n45 − k6	631	0.00	0.00	0.02	0.13	0.08	0.07	175.12	181.64	
A − n46 − k7	43	0.00	0.00	0.00	0.01	0.01	0.00	15.03	15.71	
B − n34 − k5	795	0.00	0.00	0.01	0.10	0.08	0.09	29.99	31.08	
B − n43 − k6	288	0.00	0.00	0.01	0.05	0.04	0.03	38.24	39.67	
B − n45 − k5	133	0.00	0.00	0.00	0.02	0.02	0.02	15.69	16.21	
B − n51 − k7	367	0.00	0.00	0.01	0.12	0.05	0.05	53.50	55.37	
B − n64 − k9	152	0.00	0.00	0.00	0.03	0.02	0.02	44.00	46.12	
A − n53 − k7	3056	0.00	0.00	0.17	1.16	0.39	0.33	1295.06	1362.33	
A − n37 − k6	5873	0.00	0.00	0.21	1.36	0.73	0.67	832.46	857.40	
A − n44 − k6	5963	0.00	0.00	0.31	1.95	0.80	0.67	1677.44	1741.78	
B − n45 − k6	2039	0.00	0.00	0.07	0.46	0.28	0.23	379.23	394.49	
B − n57 − k7	3205	0.00	0.00	0.17	1.11	0.42	0.41	976.52	1036.09	
1 NP	24715	0.00	0.00	1.03	6.89	3.20	2.86	5874.80	6132.22	1.00
Per Node		0.0000	0.0000	0.0000	0.0003	0.0001	0.0001	0.2377	0.2481	
2 NP′s	24680	19.48	0.00	1.01	6.65	3.18	2.91	5832.25	3054.99	1.01
Per Node		0.0008	0.0000	0.0000	0.0003	0.0001	0.0001	0.2363	0.2476	
4 NP′s	23930	74.77	0.00	0.96	7.03	3.05	2.73	5620.94	1488.74	1.03
Per Node		0.0031	0.0000	0.0000	0.0003	0.0001	0.0001	0.2349	0.2488	
8 NP′s	24366	221.67	3.56	0.97	8.41	3.36	3.00	5701.00	774.99	0.99
Per Node		0.0091	0.0001	0.0000	0.0003	0.0001	0.0001	0.2340	0.2544	
16 NP′s	25668	572.93	11.83	0.98	11.52	3.75	3.51	6090.37	437.66	0.87
Per Node		0.0223	0.0005	0.0000	0.0004	0.0001	0.0001	0.2373	0.2728	
32 NP′s	25516	1416.55	60.98	3.02	113.66	121.30	110.45	5998.93	257.71	0.74
Per Node		0.0555	0.0024	0.0001	0.0045	0.0048	0.0043	0.2351	0.3232	

Table 7: Solving VRP instances with SYMPHONY: Default settings and heuristic upper bound (summary)

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Diving	CPU sec	Wallclock
hk48 − n48 − k4	178	59.11	2.78	0.01	0.46	1.20	0.91	26.11	3.06
att − n48 − k4	349	34.03	0.04	0.01	0.75	1.51	1.29	32.45	2.53
E − n51 − k5	37	74.11	0.00	0.00	0.05	0.05	0.06	13.82	3.24
A − n39 − k5	1228	99.86	0.04	0.06	4.11	6.57	5.99	232.97	11.36
A − n39 − k6	328	35.52	9.34	0.01	0.99	1.23	0.98	32.02	2.75
A − n45 − k6	727	105.07	0.08	0.06	2.71	3.38	2.64	215.34	10.96
A − n46 − k7	36	157.81	0.00	0.00	1.19	0.04	0.03	13.93	5.53
B − n34 − k5	859	10.77	0.01	0.02	1.34	1.69	2.34	32.36	1.68
B − n43 − k6	296	65.73	6.82	0.02	3.32	1.16	0.94	37.50	3.68
B − n45 − k5	179	72.23	7.21	0.01	1.75	0.50	0.48	20.82	3.72
B − n51 − k7	365	41.01	12.07	0.02	2.05	1.66	1.39	52.84	3.79
B − n64 − k9	136	158.05	0.00	0.04	3.51	0.85	0.70	36.11	6.59
A − n53 − k7	3012	178.18	0.53	0.45	14.87	12.61	11.52	1278.10	49.22
A − n37 − k6	5999	54.78	12.96	0.51	23.60	27.99	24.62	849.45	32.03
A − n44 − k6	5916	115.84	1.03	0.79	28.99	30.29	27.20	1667.73	60.78
B − n45 − k6	1991	74.93	0.32	0.26	8.53	11.16	9.86	372.13	16.08
B − n57 − k7	3877	79.51	7.76	0.73	15.44	19.40	19.49	1085.24	40.73
32 NP′s	25516	1416.55	60.98	3.02	113.66	121.30	110.45	5998.93	257.71
Per Node		0.0555	0.0024	0.0001	0.0045	0.0048	0.0043	0.2351	0.3232

Table 8: Solving VRP instances with SYMPHONY: Default settings and heuristic upper bound (32 NPs)

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Diving	CPU sec	Wallclock	Eff
hk48 − n48 − k4	110	0.00	0.00	0.00	0.02	0.01	0.01	17.88	18.46	
att − n48 − k4	292	0.00	0.00	0.00	0.05	0.03	0.04	29.37	30.34	
E − n51 − k5	38	0.00	0.00	0.00	0.01	0.01	0.00	13.29	13.88	
A − n39 − k5	1139	0.00	0.00	0.03	0.13	0.14	0.12	218.66	226.46	
A − n39 − k6	273	0.00	0.00	0.00	0.04	0.03	0.03	26.88	27.53	
A − n45 − k6	807	0.00	0.00	0.02	0.14	0.13	0.10	247.97	257.05	
A − n46 − k7	83	0.00	0.00	0.00	0.01	0.02	0.01	45.01	47.51	
B − n34 − k5	1101	0.00	0.00	0.02	0.10	0.10	0.11	41.66	43.15	
B − n43 − k6	197	0.00	0.00	0.01	0.04	0.03	0.02	27.52	28.54	
B − n45 − k5	126	0.00	0.00	0.00	0.02	0.02	0.02	17.92	18.48	
B − n51 − k7	327	0.00	0.00	0.01	0.09	0.05	0.05	47.01	48.75	
B − n64 − k9	137	0.00	0.00	0.01	0.03	0.02	0.02	39.41	40.83	
A − n53 − k7	1682	0.00	0.00	0.08	0.63	0.21	0.17	698.52	735.62	
A − n44 − k6	5233	0.00	0.00	0.20	1.13	0.79	0.73	1471.47	1523.40	
B − n45 − k6	5413	0.00	0.00	0.18	0.95	0.82	0.81	1115.43	1157.26	
1 NP	16958	0.00	0.00	0.56	3.40	2.40	2.26	4058.00	4217.26	1.00
Per Node		0.0000	0.0000	0.0000	0.0002	0.0001	0.0001	0.2393	0.2487	
2 NP′s	16918	16.61	0.00	0.55	3.73	2.54	2.36	4051.79	2115.68	1.00
Per Node		0.0010	0.0000	0.0000	0.0002	0.0001	0.0001	0.2395	0.2501	
4 NP′s	16933	68.54	0.00	0.54	4.02	2.41	2.25	4030.29	1066.63	0.99
Per Node		0.0040	0.0000	0.0000	0.0002	0.0001	0.0001	0.2380	0.2520	
8 NP′s	16891	199.20	1.02	0.54	4.27	2.56	2.40	4031.21	551.15	0.96
Per Node		0.0118	0.0001	0.0000	0.0003	0.0002	0.0001	0.2387	0.2610	
16 NP′s	18715	513.93	29.95	0.58	7.29	2.97	2.84	4520.36	330.86	0.80
Per Node		0.0275	0.0016	0.0000	0.0004	0.0002	0.0002	0.2415	0.2829	
32 NP′s	16160	1249.31	75.52	1.42	60.44	81.97	78.37	3781.30	176.54	0.75
Per Node		0.0773	0.0047	0.0001	0.0037	0.0051	0.0048	0.2340	0.3496	

Table 9: Solving VRP instances with SYMPHONY: Default settings and no a priori upper bound (summary)

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Cuts	Idle Index	Idle Diving	CPU sec	Wallclock	Eff
hk48 − n48 − k4	101	0.00	0.00	0.00	0.02	0.04	0.01	0.01	16.56	17.25	
att − n48 − k4	384	0.00	0.00	0.01	0.07	0.13	0.06	0.05	37.76	39.11	
E − n51 − k5	53	0.00	0.00	0.00	0.01	0.09	0.01	0.01	18.64	19.46	
A − n39 − k5	390	0.00	0.00	0.01	0.07	0.58	0.07	0.04	74.36	77.22	
A − n39 − k6	315	0.00	0.00	0.01	0.05	0.15	0.05	0.03	29.70	30.48	
A − n45 − k6	851	0.00	0.00	0.02	0.15	1.22	0.15	0.10	236.26	244.64	
A − n46 − k7	24	0.00	0.00	0.00	0.01	0.08	0.00	0.00	12.90	13.64	
B − n34 − k5	1032	0.00	0.00	0.02	0.15	0.45	0.14	0.11	36.92	38.61	
B − n43 − k6	746	0.00	0.00	0.02	0.14	0.42	0.12	0.08	80.08	83.17	
B − n45 − k5	45	0.00	0.00	0.00	0.01	0.02	0.01	0.01	8.27	8.59	
B − n51 − k7	897	0.00	0.00	0.03	0.23	0.47	0.14	0.10	125.30	130.27	
B − n64 − k9	99	0.00	0.00	0.00	0.04	0.05	0.02	0.01	23.93	24.90	
A − n53 − k7	1358	0.00	0.00	0.06	0.50	4.21	0.21	0.14	515.92	539.48	
A − n37 − k6	5115	0.00	0.00	0.18	1.16	4.11	0.86	0.59	721.11	741.37	
A − n44 − k6	3633	0.00	0.00	0.14	1.09	4.97	0.59	0.39	889.40	920.67	
B − n45 − k6	3829	0.00	0.00	0.13	0.86	2.90	0.71	0.48	734.27	760.16	
B − n57 − k7	2345	0.00	0.00	0.09	0.65	1.94	0.35	0.32	508.84	536.20	
1 NP	21217	0.00	0.00	0.70	5.22	21.82	3.49	2.48	4070.21	4225.21	1.00
Per Node		0.0000	0.0000	0.0000	0.0002	0.0010	0.0002	0.0001	0.1918	0.1991	
2 NP′s	23446	19.28	0.00	0.75	5.28	23.80	3.72	2.81	4622.35	2411.40	0.87
Per Node		0.0008	0.0000	0.0000	0.0002	0.0010	0.0002	0.0001	0.1971	0.2057	
4 NP′s	23424	74.96	2.18	0.77	6.83	26.20	3.79	2.69	4611.47	1217.96	0.86
Per Node		0.0032	0.0001	0.0000	0.0003	0.0011	0.0002	0.0001	0.1969	0.2080	
8 NP′s	22062	233.71	3.17	0.70	8.36	26.21	3.74	2.64	4463.30	611.69	0.86
Per Node		0.0106	0.0001	0.0000	0.0004	0.0012	0.0002	0.0001	0.2023	0.2218	
16 NP′s	22756	597.00	31.30	0.71	10.59	30.20	4.17	3.08	4626.46	343.32	0.76
Per Node		0.0262	0.0014	0.0000	0.0005	0.0013	0.0002	0.0001	0.2033	0.2414	
32 NP′s	24669	1455.51	29.88	2.27	93.70	71.22	112.91	110.86	5108.75	229.53	0.75
Per Node		0.0590	0.0012	0.0001	0.0038	0.0029	0.0046	0.0045	0.2071	0.2977	

Table 10: Solving VRP instances with SYMPHONY: Default settings with heuristic upper bounds and global cut pool (summary)

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Dive	CPU sec	Wallclock	Eff
aa01	758	0.00	0.00	0.63	7.03	0.10	0.09	23626.58	24216.30	
aa04	1153	0.00	0.00	0.48	6.21	0.10	0.15	12746.12	13063.71	
kl02	184	0.00	0.00	0.25	2.58	0.01	0.03	1255.83	1408.35	
v0415	191	0.00	0.00	0.06	0.49	0.03	0.03	94.39	140.57	
v1622	200	0.00	0.00	0.09	0.79	0.01	0.02	143.06	215.30	
1 NP	2486	0.00	0.00	1.51	17.11	0.24	0.32	37865.98	39044.22	1.00
Per Node		0.0000	0.0000	0.0006	0.0069	0.0001	0.0001	15.2317	15.7056	
2 NPs	2159	419.68	0.00	1.32	13.67	0.21	0.27	31932.62	16809.16	1.16
Per Node		0.1944	0.0000	0.0006	0.0063	0.0001	0.0001	14.7905	15.5712	
4 NPs	2097	1575.82	0.00	1.23	10.32	0.17	0.25	29667.74	8198.46	1.20
Per Node		0.7515	0.0000	0.0006	0.0049	0.0001	0.0001	14.1477	15.6385	
8 NPs	2601	4524.93	1087.81	1.48	773.35	0.23	0.42	37868.39	5724.06	0.85
Per Node		1.7397	0.4182	0.0006	0.2973	0.0001	0.0002	14.5592	17.6057	
16 NPs	2478	11368.97	8125.87	1.87	1626.52	0.26	0.71	35603.94	3702.59	0.66
Per Node		4.5880	3.2792	0.0008	0.6564	0.0001	0.0003	14.3680	23.9070	
32 NPs	1971	30144.80	34072.61	1.27	3328.57	0.18	0.59	28660.28	3142.59	0.38
Per Node		15.2903	17.2826	0.0006	1.6883	0.0001	0.0003	14.5373	51.0083	

Table 11: Solving SPP instances with SYMPHONY: Default settings and no a priori upper bound (summary)

Instance	Tree Size	Ramp Up	Ramp Down	Node Pack	Idle Node	Idle Index	Idle Dive	CPU sec	Wallclock
aa01	566	18191.06	31069.37	0.51	2910.02	0.08	0.21	16525.35	2208.65
aa04	1044	7174.27	3003.24	0.44	74.22	0.08	0.16	11119.85	722.99
kl02	143	4118.31	0.00	0.23	208.02	0.01	0.11	865.38	181.49
v0415	218	661.16	0.00	0.10	136.31	0.02	0.10	149.71	29.46
v1622	222	487.67	0.00	0.12	11.29	0.00	0.08	173.08	24.61
32 NPs	1971	30144.80	34072.61	1.27	3328.57	0.18	0.59	28660.28	3142.59
Per Node		15.2903	17.2826	0.0006	1.6883	0.0001	0.0003	14.5373	51.0083

Table 12: Solving SPP instances with SYMPHONY: Default settings and no a priori upper bound (32 NPs)

References

[1] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2004.

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

[3] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. CONCORDE TSP solver. http://www.tsp.gatech.edu/concorde.html.

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

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

[6] M. Bénichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1:76–94, 1971.

[7] M. Berkelaar. lp_solve 5.1, 2004. Available from `http://groups.yahoo.com/group/lp_solve/`.

[8] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, 1998.

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

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

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

[12] COIN-OR: Computational Infrastructure for Operations Research, 2004. `http://www.coin-or.org`.

[13] M. Cosnard and D. Trystram. *Parallel Algorithms and Arcitectures*. International Thomson Computer Press, Boston, MA, USA, 1995.

[14] G.B. Danzig and R.H. Ramser. The truck dispatching problem. *Management Science*, 6:80–91, 1959.

[15] A. de Bruin, G. A. P. Kindervater, and H. W. J. M. Trienekens. Asynchronous parallel branch and bound and anomolies. Report EUR-CS-95-05, Erasmus University, Rotterdam, 1995.

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

[17] M. Esö. *Parallel Branch and Cut for Set Partitioning*. PhD thesis, Department of Operations Research and Industrial Engineering, Cornell University, 1999.

[18] M. Esö and T.K. Ralphs. Symphony set partitioning problem solver.

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

[20] J. Forrest. CBC, 2004. Available from `http://www.coin-or.org/`.

[21] J. J. H. Forrest, J. P. H. Hirst, and J. A. Tomlin. Practical solution of large scale mixed integer programming pr oblems with UMPIRE. *Management Science*, 20:736–773, 1974.

[22] J. J. H. Forrest, J. P. H. Hirst, and J. A. Tomlin. Practical solution of large scale mixed integer programming problems with UMPIRE. *Management Science*, 20:736–773, 1974.

[23] J. M. Gauthier and G. Ribière. Experiments in mixed-integer linear programming using pseudocosts. *Mathematical Programming*, 12:26–47, 1977.

[24] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. The MIT Press, Cambridge, MA, 1994.

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

[26] A.Y. Grama and V. Kumar. Parallel search algorithms for discrete optimization problems. *ORSA Journal on Computing*, 7:365–385, 1995.

[27] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, Cambridge, MA, USA, 2nd edition, 1999.

[28] Grötschel, M. and Lovász, L. and Schrijver, A. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[29] J. L. Gustafson. Reevaluating Amdahl's Law. *Commications of the ACM*, 31:532–533, 1988.

[30] L. Hafer. bonsaiG 2.8, 2004. Available from `http://www.cs.sfu.ca/~lou/BonsaiG/dwnldreq.html`.

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

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

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

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

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

[36] L. Ladányi. *Parallel Branch and Cut and Its Application to the Traveling Salesman Problem*. PhD thesis, Cornell University, May 1996.

[37] T.H. Lai and S. Sahni. Anomalies in parallel branch and bound algorithms. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 183–190, 1983.

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

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

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

[41] J. T. Linderoth. *Topics in Parallel Integer Optimization*. PhD thesis, Georgia Institute of Technology, 1998.

[42] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies in mixed integer programming. *INFORMS Journal on Computing*, 11:173–187, 1999.

[43] J.T. Linderoth. MIP instances, 2004. Available from `http://coral.ie.lehigh.edu/mip-instances`.

[44] A. Makhorin. GLPK 4.2, 2004. Available from `http://www.gnu.org/software/glpk/glpk.html`.

[45] A. Martin, T. Achterberg, and T. Koch. MIPLIB 2003. Avaiable from `http://miplib.zib.de`.

[46] G. Mitra. Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming*, 4:155–170, 1973.

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

[48] G.L. Nemhauser, M.W.P. Savelsbergh, and G.S. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.

[49] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.

[50] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley & Sons, Inc., USA, 1st edition, 1988.

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

[52] T. K. Ralphs. SYMPHONY Version 5.0 user's manual. Technical Report 04T-011, Lehigh University Industrial and Systems Engineering, 2004.

[53] T.K. Ralphs. Library of vehicle routing problem instances.

[54] T.K. Ralphs. Symphony vehicle routing problem solver.

[55] T.K. Ralphs. *Parallel Branch and Cut for Vehicle Routing*. PhD thesis, Cornell University, 1995.

[56] T.K. Ralphs. Parallel branch and cut for capacitated vehicle routing. *Parallel Computing*, 29:607–629, 2003.

[57] T.K. Ralphs. SYMPHONY Version 4.0 User's Manual. Technical Report 03T-006, Lehigh University Industrial and Systems Engineering, 2003.

[58] T.K. Ralphs. SYMPHONY 5.0, 2004. Available from `http://www.branchandcut.org/SYMPHONY/`.

[59] T.K. Ralphs and M. Guzelsoy. The SYMPHONY callable library for mixed-integer linear programming. In *Proceedings of the Ninth INFORMS Computing Society Conference*, pages 61–76, 2005.

[60] T.K. Ralphs and L. Ladányi. *COIN/BCP User's Manual*, 2001. Available from `http://www.coin-or.org`.

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

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

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

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

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

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

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

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

[69] V. Chvátal. *Linear Programming*. W.H. Freeman and Company, 1983.

[70] Y. Xu, T.K. Ralphs, L. Ladányi, and M.J. Saltzman. ALPS: A framework for implementing parallel search algorithms. In *Proceedings of the Ninth INFORMS Computing Society Conference*, pages 319–334, 2005.