SCALABLE ALGORITHMS FOR PARALLEL TREE SEARCH

by

Yan Xu

Presented to the Graduate and Research Committee

of Lehigh University

in Candidacy for the Degree of

Doctor of Philosophy

in

Industrial and Systems Engineering

Lehigh University

October 2007

Approved and recommended for acceptance as a dissertation in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Date

Dr. Theodore K. Ralphs
Dissertation Advisor

Accepted Date

Committee:

Dr. Theodore K. Ralphs, Chairman

Dr. Laszlo Ladányi

Dr. Jeffrey T. Linderoth

Dr. George R. Wilson

Acknowledgments

First of all, I must express my sincere appreciation to my advisor Theodore Ralphs for his wonderful guidance that made this work possible. I would also like to thank Laszlo Ladányi, Jeffrey Linderoth and George Wilson for sharing their brilliant ideas on many subjects with me, and also for their consistent support for completing this work. Thanks are due to Matthew Saltzman for letting me use the cluster at Clemson University and for contributions to CHiPPS.

So many people made my life easier at Lehigh. Here are some names come to my mind: Rita Frey, Kathy Rambo, Minzhou Jing, Rui Kuang, Qing Ye, Peiling Wu, Jianbiao Pan, Tianhao Wu, Jinglong Ban, Qunhong Tang, Shangyuan Luo, Wei Wang, Menal Guzelsoy, Ashutosh Mahajan, Svetlana Oshkai, Mu Qing, and Ying Rong. Thank you all for the advices and accommodations that you generously provided to me.

I have wonderful co-workers at SAS, who tolerated my frequently vacations at Lehigh. Special thanks to Alexander Andrianov, Manoj Chari, Hao Cheng, Matthew Galati, Jennie Hu, Trevor Kearney, Radhika Kulkarni, and Benhao Wang for the understanding and support.

Although my parents are in China, I feel they are here with me everyday. My wife Kun Yang disagreed with me on quite a few things, fortunately, she firmly supported

me to complete this research. It is almost impossible to forget to mention my daughter Emily Xu, who always competes with me to play games on the laptop that I used to write this thesis. You bring so much happiness to me. I love you all very much.

Contents

List of Tables

List of Figures

Abstract

Tree search algorithms play an important role in research areas such as constraint statisfaction, game theory, artificial intelligence, and mathematical programming. Typical tree search algorithms include backtracking search, game tree search and branch and bound. Such tree search algorithms can be naturally parallelized, but it is often non-trivial to achieve scalability, especially for so-called "data-intensive" applications.

The scalability of parallel algorithms is essentially determined by the amount by which parallel overhead increases as the number of processors increases. The amount of parallel overhead, on the other hand, is largely determined by the effectiveness of the mechanism for sharing knowledge during the course of the search. Knowledge sharing has a cost, in the form of increased communication overhead, but it can help to improve the overall efficiency of the algorithm by reducing other types of overhead such as redundant work. For certain applications, the amount of knowledge generated during the search can be quite large. We must properly address the question of what knowledge to share and when to share it.

To explore our ideas about how to implement scalable tree search algorithms, we developed a software framework, the COIN-OR High Performance Parallel Search (CHiPPS) Framework, which can be used to implement parallel tree search applications. The

framework currently includes a hierarchy of libraries. The Abstract Library for Parallel Search (ALPS) is the search-handling layer and implements a basic tree search algorithm. Because of its general approach, ALPS supports the implementation of a wide variety of algorithms and applications by creating application-specific derived classes implementing the algorithmic components required to specify a tree search. The Branch, Constrain, and Price Software (BiCePS) is the data-handling layer and provides support for the implementation of relaxation-based branch and bound. The BiCePs Linear Integer Solver (BLIS) is a concretization of BiCePS specialized to problems with linear constraints and linear objective function. BLIS solves general mixed integer linear programs, and also provides a base framework for implementing specialized applications. During the course of this work, we developed two applications. The first one is built on the top of ALPS and specialized to solve the knapsack problem; the other is based on BLIS and used to solve the Vehicle Routing Problem (VRP). These two applications were used to demonstrate the flexibility and effectiveness of CHiPPS.

We conducted a wide range experiments to test the overall scalability of CHiPPS. In these experiments, we solved knapsack instances, VRP instances, and generic mixed integer linear programs (MILPs). Our results show that scalability is relatively easy to achieve for the knapsack instances. We were able to obtain good scalability even when using several hundreds or thousands of processors. We failed to achieve good scalability for the VRP instances due to the fact that the number of nodes increased significantly as the number of processors increased. For generic MILPs, overall scalability is quite instance dependent. We observed speedup close to linear for some instances, but had poor results for others.

Finally, we performed a number of experiments to test the effectiveness of specific

methods implemented in CHiPPS to improve scalability and efficiency. The results of our experiments confirm that effective knowledge sharing is the key to improving parallel scalability. An asynchronous implementation, along with effective load balancing, is the most essential component of a scalable algorithm. The Master-hub-worker programming paradigm provides extra improvement for large-scale parallel computing. Other factors, such as task granularity, task management, and termination detection, also require careful consideration. Furthermore, the differencing scheme that we use to handle data-intensive applications can significantly reduce memory usage without slowing down the search.

Chapter 1

Introduction

In areas like discrete optimization, artificial intelligence, and constraint programming, tree search algorithms are among the key techniques used to solve real-world problems. The literature contains numerous examples of successful application of tree search algorithms. For instance, Feldmann, *et al.* [37] developed a parallel game tree search program called ZUGZWANG that was the first parallel game tree search software successfully used to play chess on a massively parallel computer and was vice world champion at the computer chess championships in 1992. Applegate, Bixby, Chvátal, and Cook used a parallel implementation of *branch and cut*, a type of tree search algorithm (see [50, 77, 57]), to solve a *Traveling Salesman Problem* (TSP) instance with 85,900 cities in 2006. The number of variables in the standard formulation for TSP is approximately the *square* of the number of cities. Thus, this instance has roughly *7 billion variables*.

There are several reasons for the impressive progress in the scale of problems that can be solved by tree search. The first is the dramatic increase in available computing power over the last two decades, both in terms of processor speed and memory size. The

second is significant advancements in theory. New theoretical research has boosted the development of faster algorithms, and many techniques once declared impractical have been "re-discovered" as faster computers have made efficient implementation possible. The third reason is the use of parallel computing. Developing a parallel program is not as difficult today as it once was. A number of tools, like OpenMP[25], MPI [49], and PVM [44], provide users a handy way to write parallel programs. Furthermore, the use of parallel computing has become very popular. Many desktop PCs now have multiple processors, and affordable parallel computer systems like Beowulf clusters appear to be an excellent alternative to expensive supercomputers.

With the rapid advancement of computational technologies and new algorithms, one of the main challenges faced by researchers is the effort required to write efficient software. To effectively handle discrete optimization problems, we may need to incorporate problem-dependent methods (most notably for dynamic generation of variables and valid inequalities) that typically require the time-consuming development of custom implementations. It is not uncommon today to find parallel computers with hundreds or even thousands of processors. Scalability is still not easy to obtain in many applications, however as computing environment have become increasingly distributed, developing scalable parallel algorithms has becomes more and more difficult.

In the following sections, we introduce some definitions and background related to tree search, parallel computing and parallel tree search. Also, we review previous work in the area of parallel tree search.

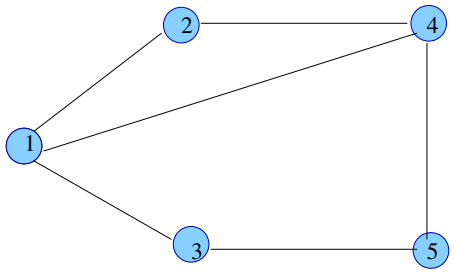

Figure 1.1: A Sample Graph

1.1 Definitions

This section presents the necessary background on tree search. Here, we introduce the definitions and notation relevant for describing general tree search problems and algorithms.

1.1.1 Graphs

We first present some definitions and notation related to graph theory since they are extensively used in tree search. Detailed introduction to these two topics can be found in [4, 75, 26, 62, 92].

Definition 1.1 *A* graph *or* undirected graph $G = (N, E)$ *consists of a set* N *of* vertices *or* nodes *and a set* E *of edges whose elements are pairs of distinct nodes. Figure 1.1 shows a graph with five nodes and six edges, where* $N = \{1, 2, 3, 4, 5\}$ *and* $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{3, 5\}, \{4, 5\}\}$.

Definition 1.2 *A* complete graph *is a graph where every pair of vertices is connected by an edge.*

Definition 1.3 *The* degree of a node *is the number of edges incident to the node.*

Definition 1.4 *A graph* $G' = (N', E')$ *is a* subgraph *of* $G = (N, E)$ *if* $N' \subseteq N$, $E' \subseteq N' \times N'$ *and* $E' \subseteq E$.

Definition 1.5 *A* walk *is an alternating sequence of nodes and edges, with each edge being incident to the nodes immediately preceding and succeeding it in the sequence.*

Definition 1.6 *A* path *is a walk without any repetition of nodes.*

Definition 1.7 *A* cycle *is a path in which the initial node is also the terminal node.*

Definition 1.8 *Two nodes* i *and* j *are* connected *if there is at least one walk from node* i *to node* j.

Definition 1.9 *A* connected graph *is a graph in which every pair of nodes is connected.*

Definition 1.10 *A* tree *is a connected graph that contains no cycle.*

Definition 1.11 *A* subtree *is a connected subgraph of a tree.*

Definition 1.12 *A* rooted tree *is a tree with a specially designated node, called the* root *of the tree.*

Definition 1.13 *A* leaf node *of a tree is a node that has degree of* 1. *When a tree has only one node, then the root of the tree is a also a leaf node.*

1.1.2 Tree Search

Following are definitions and notation related to tree search (The knapsack example in section 1.3.1 shows how to use these definitions and notation.)

Definition 1.14 *A problem is a set of decisions to be made subject to specified constraints and objectives.*

Definition 1.15 *A* state *describes the status of the decisions to be made in a problem, e.g., which ones have been fixed or had their options narrowed and which ones are still open.*

Definition 1.16 *A* successor function *is used to change states, i.e., fix decisions or narrow the set of possibilities. Given a particular state x, a successor function return a set of $< action, successor >$ ordered pairs, where each* action *is one of the possible activities in state x and each* successor *is a state that can be reached from state x by applying the action.*

Definition 1.17 *A* state space *of a problem is the set of all states reachable from the initial state. The initial state and successor functions implicitly define the state space of a problem.*

Definition 1.18 *A* path *in a state space is a sequence of states connected by a sequence of actions.*

Definition 1.19 *A* goal test function *determines whether a given state is a goal state.*

Definition 1.20 *A* path cost function *assigns a numeric cost to each path.*

Definition 1.21 *A* solution *to a problem is a sequence of actions that map the initial state to a goal state. A solution may have a path cost, which is the numeric cost of the path in the state space generated by the sequence of actions applied to reach that state.*

Definition 1.22 *An* optimal solution *is a solution that has lowest path cost among all solutions.*

Definition 1.23 *The* feasible region *of a problem is the set of all solutions.*

Definition 1.24 Expanding *or* branching *is the process of applying a successor function to a state to generate a new set of states.*

Definition 1.25 Processing *is the procedure of computing the path cost of a state, checking if the state a goal state, and expanding the state.*

Definition 1.26 Tree search *is the process of finding a solution or optimal solution that map an initial state to a goal state. Tree search involve the iterative steps of choosing, processing, and expanding states until either there are no more states to be expanded or certain termination criteria are satisfied. Tree search can be divided into two categories:* feasibility search *that aims at finding a feasible solution and* optimality search *that aims at finding an optimal solution.*

1.2 Tree Search Algorithms

1.2.1 General Framework

The process of searching for solutions can be visualized by constructing *a search graph*. The nodes in the search graph correspond to the states in the state space, and the edges in the search graph correspond to the actions in the state space. There is a close relationship between the search graph and the state space, but it is important to note the difference between nodes and states. A node is a bookkeeping data structure used to represent a

state, while a state corresponds to a representation of the status of the decisions to be made. Two different nodes can contain the same state if that state can be generated via two different paths, although this is to be avoided if at all possible.

A node's description generally has the following components [92]:

- *State*: The state in the state space to which the node corresponds;
- *Parent*: The node in the search tree that generated this node;
- *Action*: The action that was applied to the parent to generate the node;
- *Path cost*: The cost of the path from the root to the node; and
- *Depth*: The number of steps along the path from the root.

For some problems, like the knapsack problem, repeated states will not be encountered if the search is executed properly. However, there are problems for which repeated states are unavoidable, like the sliding-blocks puzzles [92]. If we can avoid repeated states, then the generated search graph is a search tree.

Figure 1.2 illustrates a typical search tree. The root of the search tree represents the initial state and corresponds to the original problem instance. For each node (except the root), a *subproblem* can be formulated based on the state information contained in the node. This subproblem generally has a smaller state space than the original problem. For a given node j in a rooted tree, there is unique path in the search tree from the root to that node. Intermediate nodes on this path from the root are called its *ancestors*, with the node directly preceding j in the path called its *parent* or *predecessor*. Conversely, if i is the parent of j then j is called the *child* or *successor* of i. Furthermore, j is called a *descendant* of all the nodes that precede it on the path from the root.

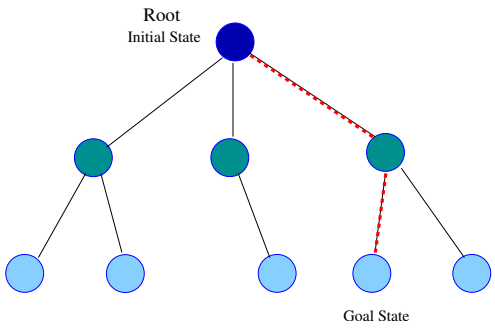

Figure 1.2: A Search Tree

A variety of algorithms have been proposed and developed for tree search. Tree search algorithms are among the most important search techniques to handle difficult real-world problems. Due to their special structure, tree search algorithms can be naturally parallelized, which makes them a very attractive area research in parallel computing. The main elements of tree search algorithms include the following.

- *Processing method*: A method for computing path cost and testing whether the current node contains a goal state.
- *Successor function*: A method for creating a new set of nodes from a given node by expanding the state defined by that node.
- *Search strategy*: A method for determining which node should be processed next.
- *Pruning rule*: A method for determining when it is possible to discard nodes whose successors cannot produce solutions better than those found already or who cannot produce a solution at all.

To implement tree search, we simply keep the current list of leaf nodes of the search tree in a set from which the search strategy selects the next node to be expanded. This

is conceptually straightforward, but it can be computationally expensive if one must consider each node in turn in order to choose the "best" one. Therefore, we usually assign each node a numeric priority and store the nodes in a priority queue that can be updated dynamically. In this scheme, the priority of a node is determined based on the search strategy. For example, if the priority of a node is its *depth*, then this scheme is the well-known *depth-first search*. If the priority is *path cost*, then this scheme represents *best-first search*. Algorithm 1.1 describes a generic tree search algorithm.

Algorithm 1.1 A Generic Tree Search Algorithm

```
1: Add Root r to a priority queue Q.
2: while Q is not empty do
3:    Take the node i with the highest priority out of Q.
4:    Process the node i.
5:    Apply pruning rules.
6:    if Node i cannot be pruned then
7:       Create successors of node i based on the successor function, and add the successors to Q.
8:    else
9:       Prune node i.
10:   end if
11: end while
```

Generally, the performance of a search algorithm can be evaluated by one of the following criteria [92].

- *Completeness*: Is the algorithm guaranteed to find a solution when there is one?
- *Optimality*: Is the algorithm guaranteed to find the optimal solution?
- *Time complexity*: How long does it take to find a solution or an optimal solution?
- *Space complexity*: How much memory is needed to perform the search?

Some algorithms are guaranteed to find a solution while others are not. Completeness is a desirable feature of an algorithm, but, in practice, it is not always possible to find a solution because of problem difficulty or resource limitation for tree search algorithms. Russell and Norvig [92] state that the following three quantities determine the time and space complexities of a search algorithm:

- the maximum number of successors of any node, i.e., *branching factor*;
- the depth of the shallowest goal node; and
- the maximum length of any path in the state space.

Time complexity can be measured by the number of nodes generated, while space complexity can be measured by the maximum number of nodes stored in memory during the search. For many problems, the number of states in the state space is extremely large. To find a solution, a search algorithm might need to generate a large number of nodes and take a long period of time. In most case, it is important that the search graph be acyclic in order to guarantee termination of the search.

1.2.2 Branch and Bound

Brand and bound is a type of tree search that prunes parts of the state space by using *bounds* on the value of an optimal solution to the subproblems examined during the search. Branch and bound can be viewed as an iterative scheme for reducing the gap between upper and lower bound on the optimal objective value. Besides the essential elements of tree search algorithms, branch-and-bound algorithm have two additional elements.

- *Lower bounding method*: A method for determining a lower bound on the optimal value to a given instance.

- *Upper bounding method*: A method for obtaining a upper bound on the optimal value to a given problem.

Most of the terminologies used in branch-and-bound are the same as those in general tree search, but there are a few differences: *branching* in branch and bound is the same as *expanding* in tree search, and the *branching method* in branch and bound is called *successor function* in tree search,

In 1960, Land and Doig [66] first proposed the idea of using a branch-and-bound algorithm for solving integer programs. Since then, branch and bound has became the primary method for finding optimal solutions of MILPs. The majority of branch-and-bound algorithms for MILPS are based on linear programming, meaning that subproblems are relaxed to LPs by dropping the integrality restriction of variables, and those LPs are solved to provide *lower bounds* on the optimal value of the original MILP. The method is based on the following properties of LP relaxations:

- if an optimal solution to the LP relaxation of a subproblem satisfies the integrality restrictions, then the solution is also an optimal solution to the subproblem and the subproblem can be pruned;

- if the LP is infeasible, then the subproblem is also infeasible and can be pruned; and

- if the subproblem is not pruned, it can be divided into several new subproblems.

The objective values of feasible solutions to the MILP provide *upper bounds* on the optimal value of the MILP. The difference between the smallest upper bound and largest

lower bound is called the *optimality gap*. Next, we briefly describe an LP-based branch-and-bound algorithm for MILPs.

Let z_u be the smallest upper bound found so far, x^u be the corresponding solution, N^i be the subproblem formulated at node i, and N^0 be the problem formulated at the root (N^0 is the original MILP). For subproblem N^i, let z_i be a lower bound on the value that a solution can have and let x^i be such a solution, if it exists. Let $\mathcal{L}$ represent the set of subproblems waiting to be processed. Algorithm 1.2 outlines an LP-based branch-and-bound algorithm (See Chapter 3 for more details).

Algorithm 1.2 LP-based branch-and-bound Algorithm

1: **Initialize.**
$\mathcal{L} = \{N^0\}$. $z_u = \infty$. $x^u = \emptyset$.
2: **Terminate?**
If $\mathcal{L} = \emptyset$ or optimality gap is within certain tolerance, then, the solution x^u is optimal. Terminate the search.
3: **Select.**
Choose a subproblem N^i from $\mathcal{L}$.
4: **Process.**
Solve the linear programming relaxation of N^i. If the problem is infeasible, got to step 2, else let z_i be its objective value and x^i be its solution.
5: **Prune.**
If $z_i \geq z_u$, go to step 2. Otherwise, if x_i satisfies the integrality restrictions, let $z_u = z_i$, $x_u = x_i$, delete from $\mathcal{L}$ all problem j with $z_j \geq z_u$, and go to step 2.
6: **Branch.**
Partition the feasible region of N_i into q polyhedral subsets $N^{i1}, \ldots, N^{ik}$. For each $i = 1, \ldots, q$, let $z_{ik} = z_i$ and add subproblems N^{ik} to $\mathcal{L}$, and go to step 2.

With different implementations of the elements of branch and bound, various branch-and-bound algorithms can be obtained. If we add valid inequalities to the LP relaxations during node processing, then we have *branch and cut*. If we add variables to the LP relaxations during node processing, then we have *branch and price*. If we add both constraints and variables to the LP relaxations when processing nodes, then we have

branch, cut, and price. In this study, we are mainly interested in branch and cut, since it is the most effective method for solving generic MILPs. In Chapter 3, we describe the details of the basic elements of branch and cut.

1.3 Classes of Tree Search Problems

In this section, we introduce two majors classes of problems (mathematical programming and constraint satisfaction problems) that can be solved by tree search.

1.3.1 Mathematical Programming Problems

Definition. *Mathematical programming* refers to the study of problems in which one seeks to minimize or maximize a real function by systematically choosing the values of real or integer variables from within an allowed feasible region. [8]. For mathematical programming problems, we want to search for an *optimal solution* that minimizes (or maximizes) the objective function, and often can be solved by tree search algorithms like branch and bound (see Section 1.2.2).

A mathematical programming problem can be formulated as

$$\begin{aligned} \min \quad & f(x) \\ s.t. \quad & g_j(x) \leq 0 \ , \ j = 1 \ldots m, \end{aligned} \tag{1.1}$$

where function $f(x)$ is called an *objective function* or *cost function* and the vector x represents a vector of variables whose value are to be determined when solving the problem. The functions $g_j(x),\ j = 1 \ldots m$, determines the constraints that define the feasible region.

If all $x \in A$ are restricted to discrete values, such as integers, then the problem is a *discrete optimization problem*. If the objective function f is linear and the set of constraints $g_j(x),\ j = 1 \ldots m$ are linear, we have a *linear program* (LP). For an LP, if some or all variables are restricted to integer values, then the LP is a *mixed integer linear program* (MILP). If all variables are restricted to integer values, we have a *(linear) integer program* (IP).

Many classes of mathematical programming problems can be classified as discrete optimization problems. Examples include MILPs (like the knapsack problem), permutation assignment problems (such as the traveling salesman problem), and set covering and node packing problems.

Example: The Knapsack Problem. As an example to show how to formulate mathematical programming problems as tree search problems, we how introduce the knapsack problem. Given a set of items, each with a weight and a value, the knapsack problem is the problem of determining the number of each item to include in a knapsack so that the total weight is not greater than the capacity of the knapsack and the total value is as large as possible [72]. The 0-1 knapsack problem restricts the number of each item to zero or one. If the number of items is n. then the 0-1 Knapsack problem can be formulated as follows.

- *States*: Each state represents a subset A of items that must be placed in the knapsack and a subset B of the items that must not be placed in the knapsack. Note that we must have $A \cap B = \emptyset$.

- *Initial state*: No item has either been included in or excluded, i.e., $A = \emptyset$ and $B = \emptyset$.

- *Successor function*: This takes a given state and produces two new states: one state results from including an item i in A and the other state results from adding the item in B. Note $i \notin A \cup B$.

- *Goal test*: Are we at a state where the capacity of the knapsack is not exceeded by the items in A, and all items are either in A and B?

- *Path cost*: The total value of items that are in A.

In this formulation, the number of states in the state space is on the order of 2^n. As n increases, the number of states increases exponentially. Searching for an optimal solution for knapsack problem can be very difficult. Unfortunately, this is true for most interesting problems. How to search effectively is a very active research area, and also one of the main subjects of this research.

1.3.2 Constraint Satisfaction Problems

Constraint satisfaction is the process of finding a solution to a set of constraints that express the allowed values for the variables [56]. *Constraint satisfaction problems* (CSPs) are problems in which one needs to find solutions that satisfy a set of constraints. In contrast to optimization problems, the goal of a CSP is to find a feasible solution. A CSP can be formally defined as a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a domain of values, and C is a set of constraints. Every constraint is in turn a pair $\langle t, R \rangle$, where t is a tuple of variables and R is a set of tuples of values, all these tuples having the same number of elements. As a result R is a relation. An evaluation of the variables is a function from variables to domain $v : X \rightarrow D$. Such an evaluation satisfies a constraint $\langle (x_1, \ldots, x_n), R \rangle$ if $(v(x_1), \ldots, v(x_n)) \in R$. A solution is an evaluation that

satisfies all constraints.

Examples of CSPs include the eight queens puzzle, the map coloring problem, sudoku, and boolean satisfiability (SAT). Such problems are generally solved via tree search, usually some form of depth-first search or local search [10]. Constraint propagation is also a common method applied to CSPs. Constraint propagation enforces a form of local consistency, with respect to the consistency of a group of variables and/or constraints. Typical local consistency includes arc consistency, hyper-arc consistency, and path consistency. Detailed description of constraint satisfaction problems can be found in [10, 43, 107, 56]

1.4 Parallel Computing

The demand for greater computing power is essentially limitless. No matter what computing power current systems have, there will always be applications that require still more. The speed of a single-processor computer is physically limited by the speed of light and the rate of heat dissipation. Although the speed of single-processor computers has increased greatly during recent years, use of a single processor is still not the best way to obtain massive computing power because of the technical difficulty and cost of improving the power of single-processor computers. One practical alternative is to use multiple processors instead of just using one "super" processor. This idea is what we call *parallel computing*. In parallel computing, the problem to be solved is first partitioned into smaller parts. Then, each part is analyzed on separate processors simultaneously. To develop parallel algorithms, it is important to identify the existing various forms of parallelism and then express them in programs that can be run on parallel platforms.

There are many important issues in parallel computing. Here, we focus on the aspects that we believe are important in the context of our research.

1.4.1 Definitions

In this section, we introduce some definitions and notations related to parallel computing. Detailed introduction to parallel computing can be found in [5, 20, 35, 42, 44, 63, 71, 112].

Definition 1.27 *A* processor *is a silicon integrated circuit that may contain either a single core or multiple cores. A core mainly consists of a physical microprocessor, cache, cache controller, and paths to the system front-side bus. Processors interpret computer program instructions and process data, and are the "brains" of computers. As to personal computers, the terms processor and CPU are used interchangeably.*

Definition 1.28 *A* process *is a set of instructions that, when executed, causes the computer to behave in a predetermined manner.*

Definition 1.29 *A* task *is a part of a set of instructions that solves a problem or accomplishes an assignment. Tasks may refer to analysis, communication, computation, or scheduling.*

Definition 1.30 Wall-clock time*, also called* real time *or* wall time*, is the elapsed time from beginning to end of the program as determined by a chronometer such as a wristwatch or wall clock. When several programs are running simultaneously on a computer, the wall time for each program is determined separately.*

Definition 1.31 CPU time *is the amount of time that the CPU is actually executing instructions of a program. During the execution of most programs, the CPU mainly sits idle while the computer fetches data from the keyboard or disk, or sends data to an output device. The CPU time of an executing program, therefore, is generally much less than the wall-clock time of the program.*

Definition 1.32 Parallel overhead *is the extra work associated with parallel version compared to its sequential code, mostly the extra time and memory space requirements from synchronization, data communications, parallel environment creation and cancellation, etc.*

Definition 1.33 Granularity *is a qualitative measure of the ratio of computation to communication:*

$$Granularity = \frac{Computation\,Time}{Communication\,Time}. \tag{1.2}$$

Coarse granularity *means that there are relatively large amounts of computational work are done between communication events; while* fine granularity *means that there are relatively small amounts of computational work done between communication events.*

1.4.2 Parallel Computers

To do parallel computing, we need a *parallel computer* on which we can execute our parallel program. A parallel computer is a large collection of processing elements that can communicate and cooperate to solve large problems quickly [5]. It can be a specially designed system containing multiple processors or a number of independent computers interconnected through certain network technologies.

Depending on how their processors, memory, and interconnects are laid out, parallel computers can be classified in many different ways. Generally speaking, parallel computers can be grouped into two major categories based on how memory is allocated [23].

- *Shared-memory computer*: Each processor can access any memory module. Processors communicate through variables stored in a shared address space. Communication among processors is easy. The drawback is that it is difficult for all processors to access all of the shared memory quickly if the number of processor is large. A typical example of a shared-memory computer is a *symmetric multiprocessor* (SMP).

- *Distributed-memory computer*: Each processor has its own local memory and can only access locations in its own memory directly. Processors communicate with each other over a physical network. Distributed-memory computers physically scale better than shared-memory computers. Programming on distributed-memory computers, however, requires explicit message-passing calls, which cause communication overhead. Types of distributed-memory computers include *massively parallel processors* (MPP), *clusters*, and *computational grids*.

Some researchers have also proposed the concepts of *distributed shared-memory* and *shared virtual memory* systems, which gives the illusion of shared-memory when memory is actually distributed. Interested readers are referred to [68, 111] for details.

Symmetric Multiprocessors

Symmetric multiprocessors (SMPs) generally have 2 to 64 processors [23] and are controlled by a single central operating system. The processors share computing resources

such as the bus, memory and I/O systems, and communicate through shared-memory. Some examples of popular SMPs are the IBM RS/6000 R30, the Silicon Graphics Power Challenge, and the Sun Microsystems SPARCcenter 2000. Programming on SMPs is attractive, since sharing data is relatively easy. There exist special parallel programming languages like High Performance Fortran (HPF) that are well-suited to deployment on SMPs. HPF was invented in 1993 to provide a portable syntax for expressing data-parallel computations in Fortran. With HPF, users can write parallel programs in a way that is similar to writing serial programs. Threads [22] and OpenMP [25] are also widely-used protocols for programming on SMPs. The main problem with SMPs is that the hardware scalability is generally poor due to physical limitations. Also, if one processor fails, the entire SMP must be shut down.

Massively Parallel Processors

A massively parallel processor (MPP) is usually a large parallel machine in which the processors do not share computing resources. It typically has several hundred processing elements (nodes) interconnected through a high-speed interconnection network/switch. The nodes of an MPP can have various hardware configurations, but they generally consist of a main memory and one or more processors. Special nodes, in addition, can allow peripherals, such as disks, printers, or a backup system to be connected. Every node runs a separate copy of the operating system. In order to utilize MPPs effectively, jobs must be broken down into pieces that can be processed simultaneously. It is relatively easy to recognize jobs that are well-suited for execution on MPPs. For instance, certain simulations and mathematical problems can be split apart and each part can be processed independently.

MPPs usually have a dedicated, high-bandwidth network that scales well with the number of processors so that the communication performance is good. MPPs also have global system view, which means that parallel programs run on several nodes as a single entity, rather than as an arbitrary collection of processes. However, MPPs have several weakness. First of all, they are expensive to manufacture due to low volume. Second, it generally takes quite a long time to build an MPP and put it to market. Moreover, MPPs require many specialized components, which can cause maintenance problems. In spite of these disadvantages, MPPs still play an important role in parallel computing. By the time of November, 2006, 21.60% of the world's top 500 fastest supercomputer were MPPs [101].

Clusters

In the last decade, many parallel programming tools, such as the message-passing protocols MPI and PVM, and the parallel programming language HPF, were standardized. Also, communication network technology was improved dramatically. Huge numbers of PCs and workstations are now connected through various networks. Today, computers in most companies and schools are networked together. Moreover, the performance-cost ratio of workstations has improved much faster than that of supercomputers. It is claimed that the performance-cost ratio of workstations is increasing at a rate of 80% per year, while that of supercomputers is only increasing $20 - 30\%$ per year [7]. All these factors have helped to bring the idea of *cluster computing* into reality.

A *cluster* is a collection of PCs, workstations, or SMPs that are interconnected via some network technology. This includes *networks of workstations* (NOWs) [11] and *clusters of workstations* (COWs) [109]. A cluster works as an integrated collection of

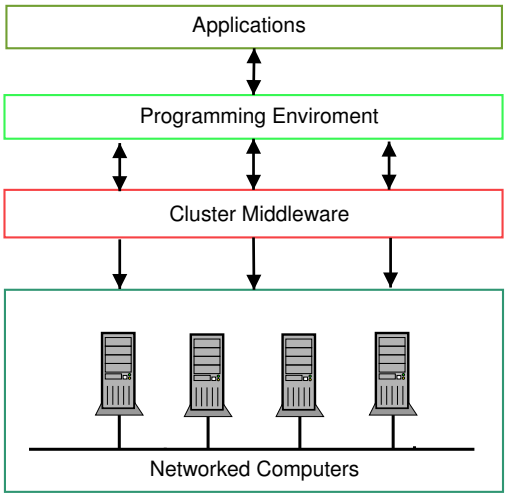

Figure 1.3: Generic Cluster Architecture

resources. As Figure 1.3 shows, a typical cluster consists of multiple high-performance computers, networks/switches, cluster middleware (single system image, etc.), a programming environment (compiler, parallel programming libraries, etc.), and applications.

Clusters have a number of advantages, including that the cost is low, existing software can be used, and new processors can easily be added to the system. It is not surprising to observe that parallel computing is moving away from expensive, specialized supercomputers to cheaper, commoditized clusters. The number of clusters in the world's top 500 fastest computers has increased from 28 to 361 in six years (from November June 2001 to November 2006) [101].

Representative types of clusters are the Berkeley NOW [11] and the Beowulf clusters [103]. A Berkeley NOW consists of a number of commodity workstations and switch-based network components. NOWs are frequently used to harvest unused cycles on workstations on existing networks. Programming in this environment requires

algorithms that are extremely tolerant of hardware failure and large communication latency. A Beowulf cluster is a collection of PCs interconnected by commodity networking technology running any one of the open source Unix-like operating system [104]. A Beowulf cluster is distinguished from a NOW by several subtle but significant characteristics. First, the nodes in the cluster are dedicated only to the cluster. This helps ease load balancing problems, because the performance of individual nodes is not subject to external interference. Also, the network load is determined only by the applications being run on the cluster because the interconnection network is isolated from the external network. The communication latency is not as long as in NOWs and all the nodes in a cluster are under the control of a single system administrator, while those in a NOW are usually not.

Computational Grids

Many large organizations today are sitting on an enormous quantity of unused computing power. Mainframes are typically idle 40% of the time, while Unix servers are typically serving something less than 10% of the time. Most desktop PCs are idle for 95% of a typical day [59]. On the other hand, there is always a demand for greater computing power to solve larger-scale problems or perform more realistic simulations. The construction of *computational grids* allow users to utilize the idle cycles of the potentially hundreds of thousands of connected computers and provide the desired computing power.

A *computational grid* is a collection of distributed, possibly heterogeneous resources that can be used as an ensemble to execute large-scale applications. Computational grids are also called “metacomputers”. The term computational grid comes from the analogy

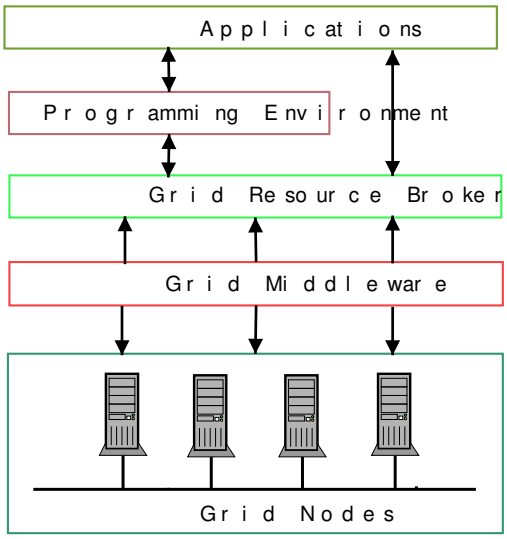

Figure 1.4: A Generic Grid Architecture

with the electric power grid.

Figure 1.4 shows the generic architecture of a grid. A computational grid basically has the following components.

- *Grid nodes (computers).* The grid nodes can join or leave the grid whenever they choose to.

- *Network and communication software.* Grid nodes are physically connected via certain networking technologies. Communication software facilitate the actual communication between computers, as well as between computers and people.

- *Grid middleware.* This software offers services such as remote process management, allocation of resources, storage access, and security.

- *Grid resource broker.* This software is responsible for resource discovery, resource selection, finding of software, data and hardware resources, and initiating

computation.

- *Programming environment*. The environment includes compilers, parallel programming libraries, and editors.

A number of computational grid infrastructures are available or under development, examples are

- *Condor*: a high-throughput scheduler [105],
- *Globus*: an integrated toolkit based on a set of existing components [41], and
- *Legion*: a single, coherent virtual machine model constructed from new and existing components [48].

1.4.3 Parallel Programming Tools

A parallel programing tool specifies what type of operations are available without relying on specific details of the hardware and software. In theory, any programming tool can be used on any modern parallel machine. However, the effectiveness of a specific tool depends on the gap between the tool and the machine [49]. Sometimes multiple tools can be used together to develop an algorithm. We will briefly describe some common tools. For more information, please refer to [49, 24].

Early Tools

During the early period (1970s) of parallel computing, parallel programs were written with very basic primitives, most of which were based on ideas in operating system design. When processes communicate via shared-memory, the tools often used are

semaphores, *monitors*, and *conditional critical regions*. These tools provide different approaches for mutually exclusive access to resources. When processors communicate via message-passing, the popular tools are *socket* and *remote procedure call*. A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. The server program is associated with a specific hardware port on the machine where it runs by a socket, so any client program with a socket associated with that same port can communicate with the server program in the network. Socket is one of the most popular forms of inter-process communication. Remote procedure call is another powerful technique for developing distributed, client-server based applications. It is based on extending the notion of conventional local procedure calling. The called process need not exist in the same address space as the calling process. The two processes may be on the same system, or they may be on different systems with a network connecting them. By using remote procedure call, programmers avoid the details of the interface with the network when developing distributed applications.

Threads

A *thread* is a sequence of instructions that can run independently. A thread can directly share data with other threads. Threads can be distinguished by the value of their *program counters* and *stack pointers*. Threads share a single *address space* and a set of global variables. Threads allow a program to split itself into two or more simultaneously running tasks, and provide a practical way to write parallel programs on a shared-memory computer. Threads are not, however, straightforward to use, since they requires programming at a level lower than many programmer would prefer. The POSIX Standard

[22] specifies the most widely used thread model called pthreads. The Java language also has a thread programming model via its `Thread` and `ThreadGroup` classes.

OpenMP

OpenMP is a standard application programming interface (API) that supports multi-platform shared-memory parallel programming. OpenMP provides a set of directives, library routines, and environment variables that are used to control the parallelization and runtime characteristics of a program. OpenMP is based on threads. The languages supported by OpenMP are Fortran, C, and C++.

OpenMP was designed to exploit certain characteristics of shared-memory computers, which have the ability to directly access memory throughout the system. Computers that do not have a shared-memory architecture may provide hardware or software layers that present the appearance of a shared-memory system, but often at the cost of higher latencies and some limitations.

Message Passing

Message-passing protocols assume that processes have only local memory and can only communicate with each other by sending and receiving messages. When data is transferred from one process to another, message-passing requires both processes to participate. Message-passing was initially defined for client/server applications running across a network. Today, most hardware architectures support message-passing, and people generally use message-passing to program on distributed-memory computers.

Message-passing facilitates the development of parallel programs. Without knowing many low level programming techniques, programmers can write parallel programs

that run on parallel computers. Message-passing has become a popular style of parallel programming. However, message-passing generally is still difficult to use in some cases. Also, message-passing does not support incremental parallelization of an existing sequential program. A number of message-passing protocols can be used in developing message-passing programs. MPI [49] and PVM [44] are among the most popular ones.

Parallelizing Compilers

A parallelizing compiler can automatically transform a given sequential program into a form that runs efficiently on a parallel computer. Parallelizing compilers have been successfully applied on shared-memory computers. There is still much ongoing research in this area. For instance, the Polaris [21] compiler is designed to automatically parallelize Fortran Programs so that they can run on shared-memory computers. The PARADIGM [16] compiler is designed to automatically parallelize sequential programs and compile them for efficient execution on distributed-memory machines.

Parallel Languages

There are a number of languages that can be used to write parallel programs. However, most of them are not attractive to programmers, partially due to complicated syntax. Recently, High Performance Fortran (HPF) [71] has attracted some attention. HPF extends ISO/ANSI Fortran 90 to support applications that follow the data parallel programming paradigm. Some applications written in HPF perform well, but others do not, due to the limitation of the HPF language itself or the compiler implementation. Also, HPF focuses on data parallelism, which limits its appeal. HPF is still struggling to gain wide acceptance among parallel application developers and hardware vendors.

1.4.4 Parallel Algorithm Paradigms

A *paradigm* is a class of algorithms that have the same basic control structure [54]. Paradigms are useful in designing parallel algorithms since they make it easy for programmers to construct the basic flow of algorithms. Many types of parallel computing paradigms have been proposed [24, 54, 83, 112]. This section briefly describes the most commonly used ones.

Single-Program Multiple-Data (SPMD)

In the SPMD paradigm, each process executes the same code but using different data sets. Figure 1.5 illustrates the basic structure of this paradigm. SPMD is also called *data parallelism*, *domain decomposition*, or *geometric parallelism*. SPMD is a popular paradigm because of its simplicity. This paradigm can be very efficient if the data are well distributed or there is a good load balancing scheme. It is also a good choice problems with many huge data sets like weather forecasting. Many problems have underlying structures that allows the use of SPMD.

Data Pipelining

This paradigm is based on functional decomposition. Programs are divided into parts that can be executed concurrently by different processes (see Figure 1.6). The processes are organized in a pipeline. Each process is a stage of the pipeline and works on a particular part of the program. Data pipelining has a very structured and clear communication pattern: only adjacent stages have interaction. Data pipelining is a good choice for image processing or data reduction problems. Its efficiency depends primarily on the effective decomposition of the program's functions.

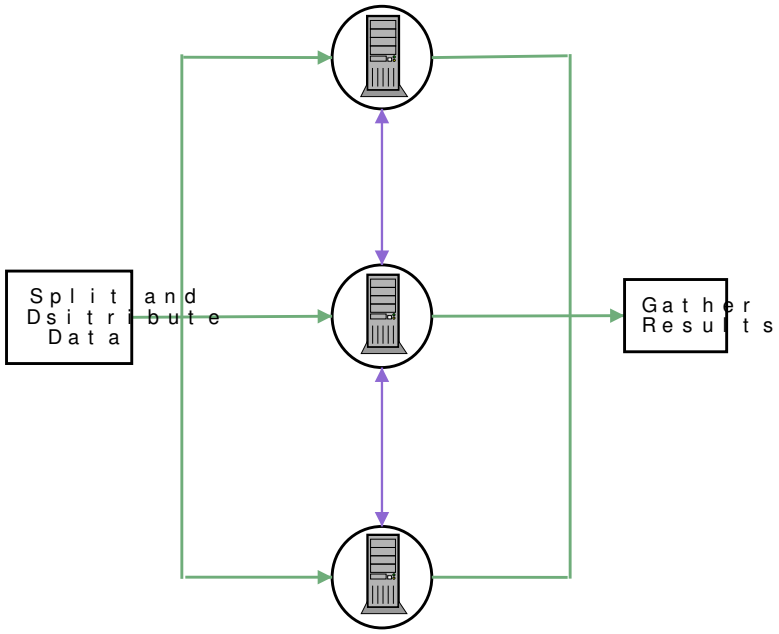

Figure 1.5: SPMD Paradigm

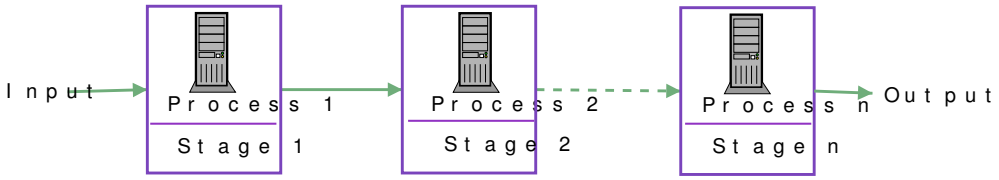

Figure 1.6: Data Pipelining Paradigm

Task-Farming (Master-Worker)

In this paradigm, the pool of processes consists of a master and a number of workers (see Figure 1.7). The master process decomposes the problem-solving task into smaller tasks and allocates them to a farm of worker processes. Workers do the actual computation. After the computation has been done, the master gathers the results. Usually, there are many more tasks than workers. To balance the workload of workers, the task-farming approach generally requires a load-balancing scheme, which can adjust the workload of

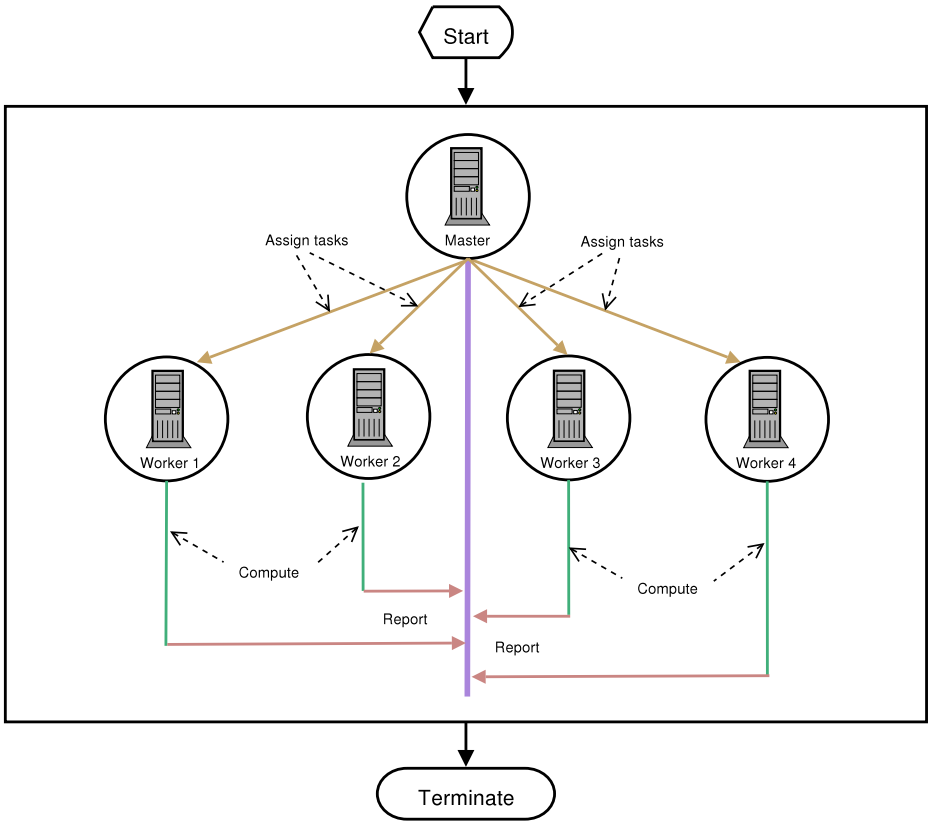

Figure 1.7: Master-Worker Paradigm

each process so that processors are not idle. This paradigm is an excellent choice for inherently recursive algorithms such as many types of tree search algorithms. Although the master process can become a communication bottleneck if a large number of workers are used.

1.4.5 Scalability Analysis

Scalability

Scalability is defined as how well a parallel system (parallel algorithm and parallel computer) takes advantage of increased computing resources. Amdahl's law [6] and Gustafson's law [53] are often used to analyze scalability of an algorithm. Let s be the time for executing the part of a program that cannot be parallelized, and p be the time for executing the part of a program that can be parallelized on a *single* processor. For algebraic convenience, let $s + p = 1$ so that s and p can be viewed as the fractions of the total computation of the serial part and parallel part. Suppose the number of processes used is n. Amdahl's law assumes that the sequential part is independent of problem size and that the sequential part can be perfectly separated from the the parallel part. Thus the maximum possible speedup according to Amdahl's law is

$$S(n) = \frac{s+p}{s+p/n} = \frac{1}{s+(1-s)/n}. \tag{1.3}$$

According to Amdahl's law, the speedup is limited to $1/s$ no matter how large a number of processes are used. Amdahl's law overlooks the fact that larger parallel computers can solve larger problems and the serial portion of execution tends to decrease as the problem size increases. Thus, Amdahl's law is generally not accurate in practice.

Gustafson's law assumes that the serial section of the code does not increase as the problem size increases. Thus, the maximum speedup factor is

$$S(n) = \frac{s+np}{s+p} = n + (1-n)s = s + (1-s)n. \tag{1.4}$$

Gustafson's law correct the problems with Amdahl's law and is more reasonable in analyzing the speedup factor of an algorithm, but even its assumptions are not very reasonable in practice.

Isoefficiency Analysis

Isoefficiency analysis [47, 63] is also used in characterizing the scalability of parallel systems. The key in isoefficiency analysis is to determine an *isoefficiency function* that describes the required increase of the problem size to achieve a constant efficiency with increasing computer resources. A slowly increasing isoeffficiency function implies that small increments in the problem size are sufficient to use an increasing number of processors efficiently; thus, the system is highly scalable. The isoefficiency functions does not exist for some parallel systems if their efficiency cannot be kept constant as the number of processors increases, no matter how fast the problem size increases. Using isoefficiency analysis, one can test the performance of a parallel program on a few processors, and then predict its performance on a larger number of processors. It also helps to study system behavior when some hardware parameters, such as the speeds of processor and communication, change. Isoefficiency analysis is more realistic in characterizing scalability than other measures like efficiency, although it is not always easy to perform the analysis.

Parallel Overhead

The amount of parallel overhead essentially determines the efficiency of a parallel algorithm. Therefore, efficiency demands that parallel overhead should be reduced as much as possible and should also be a key focus in the parallel algorithm design. The overhead

of parallel algorithms can be classified generally into the following categories.

- *Communication*: Time spent in transferring knowledge from one process to another, i.e., packing the information into the send buffer and unpacking it at the other end.
- *Handshaking/Synchronization*: Time spent idle while waiting for information requested from others or waiting for others to complete a task.
- *Redundant Work*: Time spent in performing the work that would not appear in the serial algorithm.
- *Ramp-up/Ramp-down*: Time at the beginning/end of the algorithm during which there is not enough useful work for all processes to be kept busy.

Sharing the right knowledge at the right time can significantly reduce parallel overhead. However, knowledge sharing itself can be a major cause of parallel overhead. This is the basic tradeoff that must be analyzed in parallel algorithm design.

1.4.6 Performance Measures

In general, there are two types of performance metrics, i.e., *resource performance* metrics and *system performance* metrics. Resource performance metrics are used to measure the performance of specific resources and include measures such as processor speed, communication latency, and network bandwidth. The common resource performance metrics include the following.

- *FLOPS*: Floating point operations per second is a common benchmark measure for rating the speed of processors.

- *IPS*: Instructions per second is used to measure processor speed.
- *BYTES*: bytes per second is used to measure communication latency or network bandwidth.

Floating point operations include any operations involving decimal numbers. Such operations take much longer than operations involving only integers. Some people believe that FLOPS, IPS and BTYES are not relevant measures because they fail to take into account factors such as the condition under which the processor is running (e.g., heavy or light loads) and exactly which operations are included as floating-point operations.

System performance metrics measure the performance of the combination of a parallel algorithm and the computer on which the program runs. The performance of resources, as well as the design of the algorithm itself, combine to determine the performance of an algorithm. The most commonly used system performance metrics are *Speedup Factor* and *Efficiency*.

Definition 1.34 Speedup *is a measure of relative performance between a multiprocessor system and a single processor system. It is defined as*

$$S(n) = \frac{Time\ using\ one\ processor}{Time\ using\ a\ parallel\ computer\ with\ n\ processors} = \frac{t_s}{t_n}, \tag{1.5}$$

where t_s is the execution time using one processor and t_n is the execution time using a multiprocessor computer with n processors.

In theory, the maximum speedup with n processors referred to as *linear speedup* is n. When $S(n) > n$, this is called *super-linear speedup*. Supperlinear speedup is unusual

and occurs due to a suboptimal sequential algorithm or some unique feature of the architecture and algorithm that favors parallel execution. It is also possible for a parallel algorithm to take longer than its sequential counterpart (so called *speedup anomalies*) [30] .

Definition 1.35 Efficiency *is defined as*

$$E = \frac{Time\ using\ one\ processor}{Time\ using\ a\ parallel\ computer \times number\ of\ processors} = \frac{t_s}{t_n \times n}. \quad (1.6)$$

If the efficiency of a parallel algorithm can be maintained at a desired level when the number of processors increases, provided that the problem size also increase, we call it a *scalable* parallel algorithm.

Other Measures for Tree Search

Speedup and efficiency are the standard metrics for measuring performance. However, there are situations in which these standard metrics are not applicable. These situations include:

- *heuristic search*: a search whose goal is to quickly find a "good" solution. Heuristic search generally does not guarantee to find a solution if one exists; and
- *incomplete search*: a search that terminates before searching through the whole search space due to resource limits, such as time or memory.

For heuristic or incomplete search, a serial algorithm and its parallel counterpart might behave differently in terms of the state space searched and the number of nodes expanded. Speedup and efficiency alone cannot accurately measure the performance

of the parallel algorithm. For instance, this is the case if both the serial and parallel programs terminate because they exceeded the time limit and the parallel program finds a better solution than the serial one. Although the parallel program does not have any meaningful speedup, it does find a better solution. We need some other metrics to properly evaluate the performance of parallel algorithms in these situations. The following are a number of metrics can be considered to use when speedup and efficiency are not applicable.

- *Quality of solution*: This metric looks at the quality of the best solutions found by the serial and parallel algorithms. There are two main issues with this metric. First, it is difficult to quantify the comparison of quality. Second, it is not clear what to do if neither the serial or parallel program finds a solution. In this case, solution quality information is not complete for comparison.

- *Amount of work done*: In some cases, it is enough to consider the total number of nodes generated in the search graph. This metric considers all contributions to parallel overhead. If no redundant work is being done, it is a good choice. This metric overlooks the information about quality of solutions.

- *Amount of work before the best solution*: This metric considers the number of nodes generated before the best solution is found. This metric considers solutions and is reasonable if the quality of best solutions found by serial and parallel programs are same.

- *Time per node*: This metric measures the average time spent in processing a node. This method considers parallel overhead, but does not consider solution quality.

1.4.7 Termination Detection

It is important to terminate an algorithm as soon as it finishes the search and output results. For parallel programs that have a centralized control scheme, it is not difficult to determine that the computation has come to an end. However, when computation is distributed and the algorithm is asynchronous, detecting termination is sometimes difficult. Without timely, global, and centralized knowledge, it is not easy to detect that the algorithm has already finished. In theory, an algorithm should terminate when the following condition holds [20]:

- all processes satisfy application-specific local termination conditions, and
- all messages sent have been received

A number of methods have been proposed for detecting termination for asynchronous distributed computation. For example, Mattern [73] proposed several methods like the *four counter* method, *sceptic* algorithm, and *channel counting* method. Mattern's termination-detection methods are all based on the idea of messaging counting, which states that a distributed system is defined as being terminated if all messages sent are also received. The four counter method has been proven to work well in practice [34]. There are also other methods described in [20, 111]. Mattern proved that the system is idle and program can exit if the following three conditions are satisfied:

- the system appears to have no work left,
- the number of message sent is equal the number of message received, and
- the numbers of message sent and received before termination check are equal the numbers of message sent and received after termination check.

1.5 Scalability Issues

As we have seen in Section 1.2, the basic structure of a tree search algorithm is extremely simple. Tree search algorithms are also excellent candidates for parallel processing because there is a good match between the structure of tree search algorithms and master-worker programming paradigms. Many specific implementations, and general frameworks have been developed for tree search algorithms. Algorithm 1.3 shows a naive master-worker type of parallel tree search algorithm.

Algorithm 1.3 A Naive Parallel Tree Search Algorithm

1: One process (The master) generates a number of nodes and distribute them to other processes (workers).
2: Each worker adds the nodes from the master into its priority queue.
3: Workers search for solutions according to the generic Tree Search Algorithm described in Algorithm 1.1.
4: The search terminates when all workers have no node left for processing.

This naive tree search algorithm is formally correct, but is not scalable because it will likely be very inefficient. Hence, although tree search algorithms looks straightforward, there are actually quite a number of issues to be studied and challenges to be overcome. We discuss these issues in the following sections.

1.5.1 Levels of Parallelism

There are three primary levels of parallelism that we can employ when parallelizing tree search algorithms. Ranked from coarse to fine based on granularity, they are *tree level*, *node level* and *operation level*.

At the tree level, several trees can be searched in parallel and the knowledge generated when building one tree can be used for the construction of other trees. In other

words, several tree searches are looking for solutions in the same state space at the same time. Each tree search algorithm can take a different approach, e.g., different successor function, search strategy, etc. For example, Pekny [80] developed a parallel tree search algorithms for solving the Traveling Salesman Problem, where the trees being built differed only in the successor functions.

At the node level, a single tree can be searched in parallel by processing multiple nodes simultaneously. There may be a master process to coordinate the search so that nodes are not processed more than once. This is the most widely used type of parallelism for tree search algorithms. This level of parallelism is appropriate for distributed computers and the master-worker paradigm, since the processing of a tree is perfectly separable into independent node processing tasks.

At the operation level, the parallelism may be introduced by performing the processing of a single node in parallel. This level of parallelism is more appropriate for a shared-memory computer.

1.5.2 Parallel Algorithm Phases

Conceptually, a parallel algorithm consists of three phases. The *ramp-up phase* is the period during which work is initially partitioned and allocated to the available processors. In our current setting, this phase can be defined loosely as lasting until all processors have been assigned at least one task. 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. Defining when the ramp-down phase begins is slightly problematic, but we will define it here as the earliest time at which one of the processors becomes permanently

out of work. The division of the algorithm into phases is to highlight the fact that ramp-up and ramp-down portions of the algorithm cannot be fully parallelized simply because the granularity cannot be made fine enough. For certain algorithms, such as branch and bound, the ramp-up and ramp-down periods can take long time. Thus, good scalability becomes difficult to achieve.

1.5.3 Synchronization and Handshaking

Inevitably, there are situations in which a process must wait until one or more other processes have reached a particular reference point before proceeding. This action is called *synchronization*, and usually happens when processes need to proceed simultaneously from a known state. Synchronization often happens when a serial section of work must be done by all processes. There are many applications that require synchronization. Fox [42] reported that 70% of the first set of applications that he studied used some synchronization. Synchronization can be achieved either by using a *barrier* or by some kind of counting scheme [111]. A barrier is a mechanism that prevents processes from continuing past a specified point in a parallel program until certain others processes reach this point. The problem with synchronization is that some processes might reach the synchronization point much more quickly than others and will waste time in waiting for other processes to reach the same state.

Asynchronous algorithms do not have any synchronization points and reduce overhead by reducing the time that processes spend idle. An asynchronous execution mode is appropriate for algorithms running on NOWs and computational grids, where synchronization is hard to achieve. A major issue with asynchronous algorithms is that there is typically no process that has accurate information about the overall state of the search.

This can cause difficulties in effectively balancing workload and detecting termination. Load balancing and termination detecting schemes should have the ability to deal with such inaccurate information issue.

1.5.4 Knowledge Sharing

Knowledge is the information, such as solutions and path costs, generated during the course of a search or input by users. Knowledge can be used to guide the search, e.g., by determining the order in which to process available states. *Global knowledge* is knowledge that is valid for the whole state space. *Local knowledge* is knowledge that is only valid for a specific part of the state space.

With knowledge about the progress of the search, the processes participating in the search are able to make better decisions. When useful knowledge is shared among processes, it is possible to avoid part or all of the performance of redundant work, because the progress of a parallel search can be executed in much the same fashion as its serial counterpart. If all processes have complete knowledge, then in principle no redundant work will be performed.

There are several challenges associated with knowledge sharing, however. First, knowledge sharing may change the shape of the search tree dynamically, which makes load balancing difficult. The workload of each process might change more frequently and drastically in the presence of knowledge sharing. For instance, many of the nodes that a process has in its queue may suddenly be pruned if another process finds a good solution whose value is better than the path costs of those nodes. Knowledge sharing requires load balancing schemes that are able to respond to these kinds of change quickly.

Secondly, knowledge sharing has significant impact on parallel overhead. Communication overhead and handshaking are essentially the cost of sharing knowledge, while redundant work is the cost of *not* sharing knowledge. This highlights the fundamental trade-off inherent in knowledge sharing: it reduces the performance of redundant work, but comes at a price. The goal is to strike the proper balance in this trade-off. Trienekens and Bruin [106] give a detailed description about of the issues involved in knowledge generation and sharing.

1.5.5 Implementation Paradigms

The master-worker is the most widely used paradigm for implementing tree search because a search tree can be easily partitioned into a number of subtrees. In fact, there are not many studies in parallel tree search that use other paradigms other than master-worker. When designing the control strategy of the algorithm, we can either let the master have a central node pool storing all the generated nodes or let workers also have their own local node pools. However, a major problem with the *master-worker* paradigm is that the tree manager can become overburdened with requests for knowledge of various type. Moreover, most of these requests are *synchronous*, which means that the requesting process is idle while waiting for a reply. As the number of processes that participate in the search increases, this problem becomes more serious. To overcome this, some new ideas have been proposed, such as the multiple master-worker paradigm used by PEBBL and PUBB [34, 99].

1.5.6 Task Granularity

Generally, a node is treated as a basic work unit in tree search algorithms due to the fact that the processing of a node is a separable task. Most currently available parallel tree search packages, such as SYMPHONY [88] and PEBBL [34], treat a node as a work unit. As we know, increasing the granularity of task is a useful way to cut down on communication overhead, though it may increase redundant work. Also, large granularity helps to reduce the amount of work that might need to be transferred and also decrease the frequency to sharing workload. Hence, it is important to consider increasing task granularity.

1.5.7 Static Load Balancing

Static load balancing is also referred to as *mapping* and is a method for initially distributing tasks to processes. As we know, a tree search generally starts with a single root node. Hence, the first task is to create enough candidate nodes to ensure all processors are busy with useful work. Static load balancing should allocate tasks as evenly as possible, given available knowledge about potential workload, so that resource utilization is maximized. In some cases, a well-designed static load balancing scheme can be very effective [95].

Henrich [55] describes and compares four different initialization methods: *root initialization*, *enumerative initialization*, *selective initialization* and *direct initialization*. Each method has its advantages and disadvantages.

When applying the *root initialization* method, one process expands the root of the search tree and creates a required number of nodes. The descendants of the root are then distributed to other processes according to certain rules. Root initialization is the most

common approach due to the fact that

- it is easy to implement; and
- it is effective when the number of nodes created during initialization is large.

A major shortcoming of root initialization is that many of the processes are *idle* while waiting to receive their allocation of nodes.

Enumerative initialization broadcasts the root node to all processes, which then expand the root according to the sequential algorithm. When the number of leaf nodes on each process is at least the number of processes, processes can stop expanding. The i^{th} process then keeps the i^{th} node and deletes the rest. In this method, all processes are working from the very beginning and no communication is required. On the other hand, there is redundant work because each process is initially doing an identical task. This method has been successfully implemented in PEBBL [34].

Selective initialization starts with broadcasting the root to each process. Each process then generates one single path from the root. The method require little communication, but requires a sophisticated scheme to ensure processes work on distinct paths.

Direct initialization does not build up the search tree explicitly. Instead, each process directly creates a node from a certain depth of the search tree. The number of nodes at this depth should be no less than the number of processes. This method require little computation and communication, but it works only if the structure of the search tree is known in advance.

1.5.8 Dynamic Load Balancing

Although static load balancing is effective in certain cases, the uneven processor utilization, processor speed, memory size, and the change of tree shape due to the pruning of subtrees can make the workloads on processes gradually become unbalanced, especially in distributed computing environments. This necessitates the need for dynamic load balancing, which involves reallocating workload among the processes during the execution of a parallel program. Dynamic load balancing needs to consider both *quality* and *quantity* of work when redistributing workload. Following are the definitions of quality and quantify of work:

Definition 1.36 *The* quality of work *is a numeric value to measure the possibility that the work (node or a set of nodes) contains good solutions.*

Definition 1.37 *The* quantity of work *is a numeric value to measure the amount of work. It can be the number of nodes to be processed. The unit of the workload can be a node or a subtree.*

A number of methods have been proposed to dynamically balance workloads [64, 67, 76, 94, 97, 100]. A parallel program may use several schemes to dynamically balance the workload. Kumar, *et al.* [64] studied several dynamical load balancing schemes: *asynchronous round robin*, *nearest neighbor*, and *random polling*.

In an asynchronous round robin scheme, each process maintains an independent variable *target*, which is the identification of the process to ask for work, i.e., whenever a process needs more work, it sends a request to the process identified by the value of the target. The value of the target is incremented each time the process requests work. Assuming P is the number of processes, the value of target on process i is initially

set to (i+1) modulo P. A process can request work independently of other processes. However, it is possible that many requests are sent to processes that do not have enough work to share.

The nearest neighbor scheme assigns each process a set of neighbors. Once a process needs more work, it sends a request to its immediate neighbors. This scheme ensures locality of communication for requests and work transfer. A disadvantage is that the workload may take a long time to be balanced globally.

In the random polling scheme, a process sends request to a randomly selected process when it need work. The possibility of selection of any process is the same. Although it is very simple, random polling is quite effective in some applications [64].

1.6 Previous Frameworks and Applications

During the last two decades, a number of software package, for implementing tree search algorithms have been developed. However, to our knowledge, the only other framework for parallel general tree search algorithms is the Parallel Implicit Graph Search Library (PIGSeL), which was developed by Peter Sanders [96, 93]. The library PIGSeL includes several layers:

- *machine layer*: specifies the hardware, the operating system, the compiler and its parallel libraries that the library has to adapt to;
- *machine interface layer*: defines the messaging and I/O functions that specific to the machine;
- *load balancing layer*: contains all the components required for parallelizing the

search including the load balancing algorithm and the components for handling solutions and tree pruning procedures;

- *sequential search layer*: implements a generic search module; and
- *application interface layer*: provides a number of interfaces for developing applications.

At the beginning of the search, every process expands the root node locally. When the number of leaf nodes on each process is at least the number of processes, the i^{th} process keeps the i^{th} node and delete the rest. Then, each process works on the retained node. If a process is out of work, it randomly chooses another process to ask for work. Sanders developed two applications, one for the *Golomb ruler* problem, and one for the *knapsack* problem and tested them on a cluster with 1024 processors. Near linear speedup was achieved for the Golomb ruler problem and good speedup was achieved for the knapsack problem. The overall parallel execution time for 1024 processors was 1410 times smaller than the sequential time when solving knapsack instances. Sanders claimed that the good speedups were due to the load balancing scheme, bottleneck-free implementation and termination detection algorithm.

There have also been some study of the design and implementation of parallel constraint satisfaction algorithms. Platzner and Rinner [82] developed a parallel algorithm for CSP based on partitioning the search space of a CSP into independent subspaces. A backtracking algorithm was used to search for solutions in parallel in these subspaces. They used the master-worker paradigm and implemented the parallel CSP algorithm on an MPP system. The experiments showed reasonable speedup up to 8 processors. Feeley

et al. [36] designed a parallel CSP algorithm to determine the three dimensional structure of nucleic acids. The algorithm was based on a standard backtracking algorithm. Initially, the master process created nodes for the slaves. Then, the slaves searched for solutions in parallel. A dynamic load balancing method was used to balance the workload. They found the speedup was dependent on the data set. For one data set, nearly linear speedup was obtained up to 64 processors, but for another data set, the speedup was 16 when using 64 processors.

A number of researchers have studied parallel game tree search. Feldmann *et al.* [37] developed a distributed algorithm for searching game trees with massively parallel systems. They implemented a distributed chess program called *ZUGZWANG*, which was introduced in Chapter 1. In ZUGZWANG, one process starts searching the game tree by expanding the root node. An idle processor can ask for work from any other processor. After a process has finished the evaluation of its subproblem, it returns the results to its master. Feldmann *et al.* claimed that *ZUGZWANG* achieved a speedup of 344 when using 1024 processors. Hopp and Sanders [58] described an approach to parallelize game tree search on SIMD machines. They used multiple masters to manage workers. At the beginning of the search, every process expands the root node and keeps a designated portion of search spaces once there is enough work for all processes. The workload among processes are dynamically balanced. They achieved speedups of 5850 on a MasPar MP-1 with $16,000$ processors.

For parallel branch and bound, there are a number of frameworks that are available for developing specific applications, such as BoB [18], COIN/CBC[70], PARINO [69],

PEBBL [34, 33], PPBB-Lib [108], and SYMPHONY [88]. Here, we introduce SYMPHONY and PEBBL because they both significantly influenced our research. SYMPHONY is an open source solver for mixed integer linear programs (we discuss mixed integer linear programs in more details in Chapter 3). The core solution methodology of SYMPHONY is a highly customizable branch-and-bound algorithm that can be executed sequentially or in parallel [90]. SYMPHONY handles inter-process communication via PVM [44] and employs the master-worker paradigm. The master maintains a single central node pool from which nodes are distributed to each worker. This simplifies certain aspects of the parallel implementation, such as load balancing, but the scalability of this approach is somewhat limited because the central pool can becomes a bottleneck when a large number of workers are dedicated to node processing. Eckstein, *et al.* developed PEBBL [34, 33], a package to construct serial and parallel branch-and-bound algorithms. PEBBL is designed to run in a generic message-passing environment. In PEBBL, processes are grouped into clusters. Each cluster consists of a hub and a set of workers. PEBBL does not have an explicit *master* process to manage the search. The main features of PEBBL include dynamic load balancing, on-processor multitasking and checkpoints.

As discussed previously, little research about frameworks for developing general scalable tree search algorithms has been undertaken. Also, there are few studies we know of that emphasize scalability for *data-intensive* applications. in which describing the state of the search requires a large amount of data. Data-intensive applications can cause memory problems and incur increased communication overhead if not implemented carefully. There are few studies concerning how to store data compactly and how to effectively handle data management for this type of search algorithm. SYMPHONY

uses a scheme called *differencing*, which stores partial information under certain circumstances. In what follows, we study this scheme in more detail.

1.7 Research Objectives and Thesis Outline

The overall objectives of this research are as follows.

- To study the factors that have a major impact on scalability. These factors include the implementation paradigm, load balancing strategy, asynchronicity, task granularity, knowledge sharing mechanism, etc.
- To study the methods for handling data-intensive applications effectively. We analyze a differencing scheme in which we only store the difference between the descriptions of a child node and its parent. Also, we study methods for efficiently handling duplicated and redundant knowledge.
- To develop a framework that supports implementation of scalable parallel tree search algorithms. We make no assumption about the algorithm to be implemented only that it is based on tree search.
- To use the framework to implement a parallel MILP solver and to study its scalability.

In Chapter 1, we have introduced the background of tree search algorithms and parallel computing, then we discuss the key issues in parallelizing tree search and review the current state-of-art of parallel tree search. In Chapter 2, we present the main ideas and techniques used to develop scalable parallel tree search algorithms. We discuss the

master-hub-worker programming paradigm, load balancing schemes, termination detection, and task management. We also describe the design and implementation of the ALPS library, which handles the parallel tree search. In Chapter 3, we focus on the *branch-and-cut* algorithms that are typically used to solve MILPs. We first discuss how we handle data-intensive application, then present ways to develop scalable branch-and-cut algorithms when the relaxation scheme is based on linear programming (LP). We present the design and implementation of the BiCePS library, which implements the data-handling routines for a wide variety of relaxation-based branch-and-bound algorithms, and the BLIS library, which is a specialization of the BiCePS library in which the relaxation scheme is based on the solution of LP. In Chapter 4, we present computational results, including the overall scalability analysis resulting from our efforts to solve the knapsack problem, generic MILPs, and the vehicle routing problem. We also tested the effectiveness of the methods that were used to improve scalability. In Chapter 5, we conclude with a discussion of proposed future research.

Chapter 2

A Framework for Parallel Tree Search

In the first part of our work, we have formalized and extended the concept of *knowledge-based tree search*. In knowledge-based tree search, all information generated during the tree search is represented as *knowledge*, which is stored in *knowledge pool*s and shared during the search. We have developed a framework, the Abstract Library for Parallel Search (ALPS), to implement our ideas involving knowledge-based tree search. ALPS uses knowledge to guide the progress of tree search and to reduce parallel overhead and improve scalability.

ALPS is the search-handling layer of the COIN-OR High-Performance Parallel Search (CHiPPS) framework. We focus on the ALPS library in this chapter and describe the other two libraries that are part of CHiPPS in Chapter 3. In ALPS, there is no assumption about the algorithm that the user wishes to implement, except that it is based on a tree search. ALPS is designed for deployment in a distributed computing environment and assumes processors can communicate with each other through network technologies. In the following sections, we elaborate on the design and implementation of our knowledge-based parallel tree search framework, and the techniques that are used to

improve scalability.

2.1 Implementation Paradigm

ALPS is designed to be deployed on distributed architectures, such as clusters or grids, and is targeted at large-scale computing platforms where the number of processors participating in the search can be very large. To effectively share knowledge and handle the search, ALPS proposes a new programming paradigm called the *master-hub-worker* paradigm.

2.1.1 The Master-Hub-Worker Paradigm

Compared to the traditional master-worker paradigm, the *Master-Hub-Worker* Paradigm adds a level of coordination (hub level) between the master and its workers. Figure 2.1 shows the basic structure of the master-hub-worker paradigm. A similar idea has been used in PEBBL [34] and PUBB [99], each of which has multiple master-worker clusters.

In the master-hub-worker paradigm, the master process manages a set of hubs, and each hub is responsible for managing a set of workers. A hub and the set of workers that it manages form a *cluster*. The master is responsible for the entire search, while each cluster is responsible for searching solutions corresponding to a particular part of the state space. The master generally does not communicate directly with workers. Each cluster works roughly in a traditional *master-worker* fashion. By limiting the number of workers in each cluster, we prevent the hub from becoming a communication bottleneck. As the number of processors increases, we simply add more clusters to compensate.

The master-hub-worker paradigm is a decentralized paradigm. It moves some of

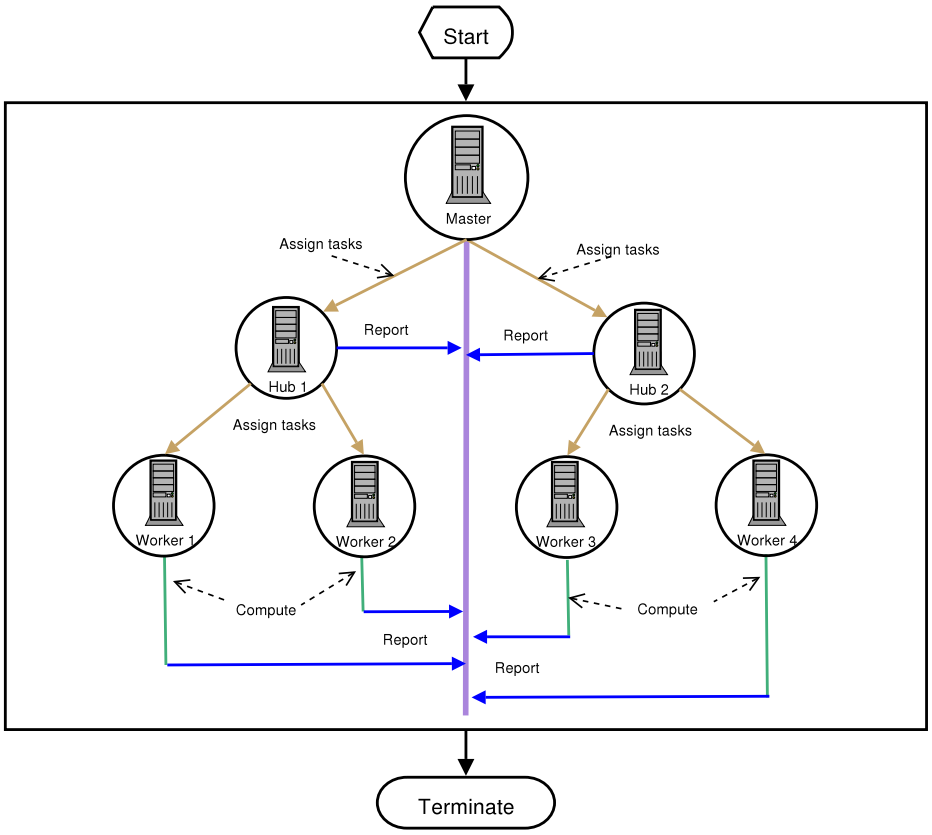

Figure 2.1: Master-Hub-Worker Paradigm

the communicational and computational burden from the master process to the hubs. The communicational and computational burden to the hubs themselves can be further reduced by increasing the task granularity in a manner that we describe later.

2.1.2 Process Clustering

To implement the master-hub-worker paradigm, ALPS groups the processes into different *clusters*. Each cluster has one *hub* and at least one *worker*. To some extent, a cluster functions as a master-worker system. The hub allocates tasks to its workers and controls the search process, while the workers follow the hub's instruction and do the

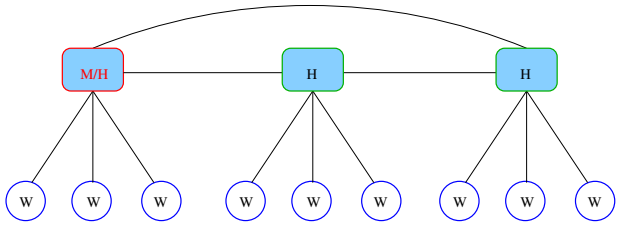

Figure 2.2: Processes Clustering

actual work. What makes a cluster different from a stand-alone master-slave system is that the hub does not have complete control over its workers, since it must also follow the instructions of the master during the search. Figure 2.2 illustrates the structure of the process clustering.

To group processes, ALPS first needs to decide the size of a cluster. Given the value of runtime parameters `processNum` and `hubNum`, ALPS can calculate the cluster size S. ALPS determines S by finding the minimal S satisfying following inequality

$$\texttt{hubNum} \times S \geq \texttt{processNum}. \tag{2.1}$$

The processes ranked from 0 to $S-1$ are assigned to the first cluster, and in general form, the process ranked from kS to $(k+1)S$ are in cluster k, $k = 1, \ldots, \texttt{hubNum} - 2$. The last cluster has processes ranked from $(\texttt{hubNum} - 1)S$ to $\texttt{processNum} - 1$. The size of last cluster might be less than S.

It should be pointed out that the parameter `hubNum` is only a suggested value, which means that ALPS can override the value of `hubNum` when it is better to do that. For instance, if the actual size of the last cluster is 1, ALPS will reduce `hubNum` by one and recalculate the size of each cluster. It repeats this procedure until the size of last cluster is greater than one.

According to its rank in a cluster, one of the processes is designated as a hub and is marked as `AlpsProcessHub`. The rest of the processes in that cluster are workers and marked as `AlpsProcessWorker`. The hub of one of the clusters is designated as the master and marked as `AlpsProcessMaster`. To save resources, the master also functions as a hub.

2.2 Knowledge Management

We propose a novel knowledge management system to efficiently handle knowledge acquisition, storage and distribution. The system consists of three components: *knowledge objects*, *knowledge pools* and *knowledge brokers*.

2.2.1 Knowledge Objects

As defined earlier, knowledge is the information generated during the search or input by users. Knowledge is the key components of knowledge-based tree search. In this section, we discuss the basic types of knowledge, how to add new types of knowledge, and how to encode and decode knowledge.

In ALPS, a tree search either takes as input or generates four basic types of knowledge: *model*, *node*, *solution* and *subtree*. A *model* contains the data to describe the problem being solved. A *node* represents a state in the state space. A *solution* represents a goal state. A *subtree* contains the description of a hierarchy of nodes with a common root. Subtrees can be stored efficiently using differencing (see Chapter 3), which is the main reason that we make subtree a separate knowledge type. Specific tree search algorithms might have other types of knowledge.

Users can create new types of knowledge and ask ALPS to manage them. New types of knowledge must first be registered so that ALPS knows how to handle them. Each knowledge type has a compact encoded format, represented as a `string`, so that it can be easily shared. There is a default encoding function for knowledge in ALPS, which packs the member data of a knowledge object into a message buffer, which is sent to other processes by the knowledge broker. This default implementation can be used if the memory of data members is continuously allocated. If a knowledge object has data members whose memory is not continuously allocated, such as pointers, `std::vector` or `std::map`, then the default encoding function cannot be used. In this case, users have to define a customized encoding function for this knowledge type.

The encoded representation of a knowledge object can also be hashed to generate an "index" useful for identifying duplicate knowledge objects. In ALPS' default implementation, the hash value is the sum of the decimal numbers associated with the `characters` in the `string` obtained by encoding. By comparing the hash values of various knowledge objects, duplicated knowledge can easily be identified.

ALPS use the decoding function to reconstruct a knowledge object after it has been encoded. ALPS does not provide default decoding functions, since it has no idea about the representation scheme for particular knowledge types. Each type of knowledge type must define its own decoding function for parallel execution.

2.2.2 Knowledge Pools

A knowledge pool (KP) functions simply as a repository for a specific type of knowledge. For instance, a node pool stores a set of nodes. The generated knowledge can be stored locally or sent to a knowledge pool residing on another process. A knowledge

pool might also request knowledge from other knowledge pools. Within a local process, each type of knowledge may be stored in one or several associated knowledge pools. Some knowledge pools might only store global knowledge, while other only store local knowledge.

2.2.3 Knowledge Brokers

A *knowledge broker* (KB) is an agent responsible for sending, receiving, and routing all knowledge associated with a local process. Each process has one knowledge broker that manages a set of knowledge pools. In order to maintain platform independence, knowledge pools do not communicate directly with each other. Instead, requests and responses are passed through the broker. A KB encapsulates all the necessary platform-dependent communication subroutines. For instance, if the communication is via MPI [49], then the KB has interface functions based on MPI. Different communication protocols require different types of knowledge brokers. Although the implementation of different knowledge brokers might be different, they all provide the same API.

A knowledge broker associated with a knowledge pool may issue two types of requests: (1) a request to insert a new knowledge object into a knowledge pool, and (2) a request to retrieve a knowledge object from a knowledge pool. A knowledge pool may also choose to "push" certain knowledge to another knowledge pool, even though no specific request has been made. It is even possible for one knowledge pool to request that another knowledge pool send knowledge to a third knowledge pool

Figure 2.3 shows the architecture of the knowledge management system that has

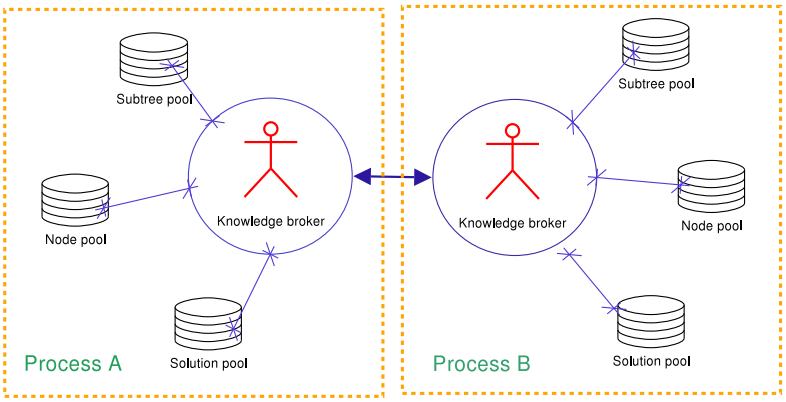

Figure 2.3: Knowledge Management System

been implemented in ALPS. All knowledge is collected and stored in associated knowledge pools. This makes classifying and prioritizing knowledge much easier. Also, duplicate and invalid knowledge can be identified quickly. With the concept of knowledge brokers, supporting different computing platform becomes easier. The flexibility of ALPS' knowledge management system is very powerful in today's rapidly changing computing environment.

2.3 Knowledge Sharing

For a parallel tree search, knowledge generated by one process may not be known by other processes and may be useful to the global search. It is important to make this kind of knowledge easily available to all processes so that the utility of the generated knowledge is maximized. In its default implementation, ALPS shares the four basic types of knowledge. Next, we briefly discuss the mechanism by which each of them is shared.

Model. At the beginning of search, the master either reads in the model from a data file or creates the model based on user input. It broadcasts the model to all other KBs. This is the only time during execution when original model data is shared.

Nodes. Nodes are shared in distinctly different ways during the ramp-up and primary phases of the algorithm. During the ramp-up phase, nodes are shared individually according to the specific static load balancing scheme being employed. During the primary phase, nodes are no longer shared individually, but as elements of subtrees. How subtrees are shared is determined by the dynamic load balancing scheme and is discussed next.

Subtrees. During the search, ALPS redistribute the workload among processes by dynamically sharing subtrees. The reason for sharing subtrees rather than individual nodes is to enable the possible use of an efficient method for encoding groups of nodes that can be represented as a single subtrees (see Section 3.4.1). ALPS has several dynamic load balancing schemes, described in Section 2.5.2.

Solutions. Solutions generated during the search are shared as quickly as possible with all other KBs by the use of a binary communication tree. When a process k finds a solution, ALPS forms all process into a binary tree with process k as the root. Process k sends the solution to its two children, and then its children send the solution to its children's children, and so on. in this way, the solution is broadcast quickly and efficiently, so that all KBs will have upper bound information that is as up-to-date as possible.

Other Knowledge. For other knowledge generated during search (like the pseudocost and valid inequalities that we discuss in Chapter 3), ALPS periodically checks if there are objects of those types to be shared. If there are, ALPS shares them in a way similar to the way solutions are shared. To share generated knowledge, the user simply defines where to retrieve the knowledge to be shared and where to store the received knowledge. ALPS takes care of the communication and distribution tasks.

2.4 Task Management

The knowledge broker associated with each process is responsible for managing the tasks involved in a parallel tree search. The KBs schedule the execution of various tasks and need to balance the time spent in *communication tasks* (those involving message passing, like load balancing) and the time spent in *computation tasks* (those involving actual searching). The task management system of ALPS is built on the concept of *task* (or *thread*) and *scheduler*. A similar idea has been used in PEBBL[34, 33]. Each task exists in one of the three states:

- *ready*: the task is waiting for a message to trigger it to run;
- *running*: the task is running; and
- *blocked*: the task is not allowed to run at this time.

2.4.1 Multi-level Task Management System

The master, hubs, and workers play different roles in a parallel tree search, and they have their own sets of tasks to achieve different functionality. ALPS divides those tasks

into three levels: *master level*, *hub level* and *worker level*. The master only manages the tasks that belong to master level. Similarly, the hubs and workers only manage the tasks that belong to their levels.

Master Task Management

The master monitors the overall progress of the search. It periodically balances the workload among clusters and decides whether it should perform a termination check. The KB of the master needs to schedule the following tasks.

- *Balance Hubs*: The master balances the workload among hubs. The load balancing scheme used has been described before.
- *Termination Checking*: A process checks whether the termination conditions are satisfied.
- *Update System State*: The master receives the state information for the cluster from the message buffer and updates the system state.
- *Unpack and Set Solution*: A process unpacks a solution and then updates solution information.
- *Receive a Subtree*: A process unpacks and reconstructs a subtree that was sent by another process.

Figure 2.4 shows the task management system of the master. First, the master listens and processes messages for a defined period of time (determined automatically by the search or adjusted manually by users). The messages may contains the state information of a cluster, a feasible solution, or a shared subtree. Then, the master checks whether the

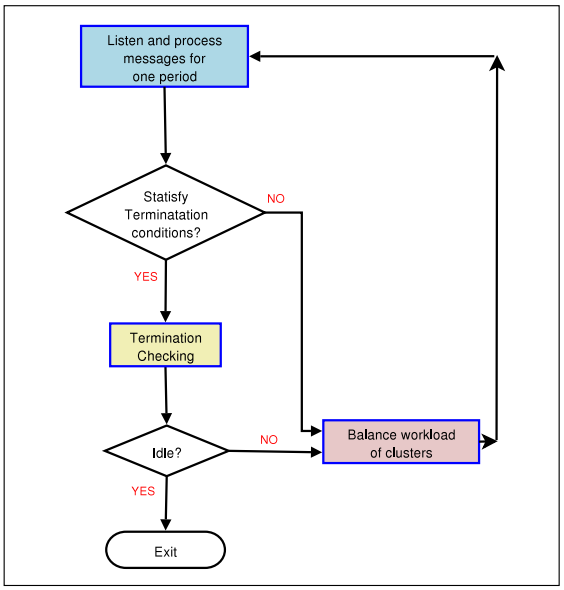

Figure 2.4: Master Task Management

termination conditions (see section 2.4.2) are satisfied. If those conditions are satisfied, the master starts the terminate check and checks whether the system is idle. If the system is idle, the master asks all processes to terminate the search and collects final search results. If the termination conditions are not satisfied or the system is not idle, the master balances the workload among hubs, and continues to listen and process messages for one period. This procedure repeats until the terminations conditions are satisfied or the system is idle.

Hub Task Management

The hubs manage a set of workers and periodically balance the workload among workers. The KBs of the hubs need to schedule the following tasks.

- *Balance Workers*: The hub balances the workload of its workers. It first checks

whether there are workers that have no work. If there are, it will do a quantity balance. If not, it will do a quality balance.

- *Termination Checking*: The hub checks whether the termination conditions are satisfied.

- *Do One Unit of Work*: The hub explores a subtree for a certain number of nodes or a certain period of time. The same subtree will be explored next time if it still has unexplored nodes. The hub performs the task only if the hub is also functioning as a worker. By default, hubs do not perform this task, but there is a parameter for users to make hubs perform this task.

- *Report Cluster State*: The hub reports its status (workload and message counts) to the master.

- *Hub Update Cluster State*: After receiving the state of a worker, the hub updates its cluster's status.

- *Unpack and Set Solution*: The hub unpacks a solution and the identification of the process that found the solution. It then updates solution information.

- *Receive a Subtree*: The hub unpacks and reconstructs a subtree that was sent by another process.

- *Share Work*: After receiving the master's request to donate work, the hub finds its most loaded worker and asks it to donate a subtree to another hub, whose information (process id and quantity of work) is packed in the message buffer.

- *Allocate Donation*: The hub allocates the donated subtree to the worker in its cluster that has the smallest workload.

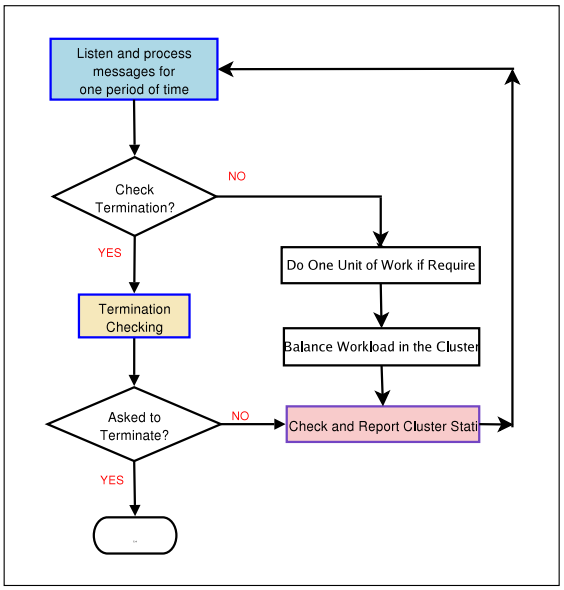

Figure 2.5: Hub Task Management

Figure 2.5 shows the task management system of a hub. First, the hub listens and processes messages for one period of time (automatically determined by the search or manually adjusted by users). The messages may contains the state information of a worker, a feasible solution, a shared subtree, or work request for a worker. If there is a termination check request from the master, the hub asks its workers to report their latest state, and sends this state information to the master. If the master asks the hub to terminate, the hub in turn asks its workers to terminate. If there is no termination check request, the hub explores a subtree for a certain number of nodes or a certain period of time. After exploring the subtree, the hub balances the workload among its workers, reports the state of the cluster to the master, and continues to listen and process messages for one period. This procedure repeats until the hub is asked to terminate by the master.

Worker Task Management

Workers search for solutions and periodically report their state to their associated hubs. The KBs of the workers need schedule following tasks:

- *Donate Work*: The worker donates a subtree to another worker.
- *Worker Report State*: The worker reports its state to its hub.
- *Ask for Work*: The worker ask its hub for more work.
- *Do One Unit of Work*: The worker explores a subtree for a certain number of nodes or a certain period of time. The same subtree will be explored next time if it still have unexplored nodes.
- *Termination Checking*: The worker checks whether the termination conditions are satisfied.
- *Unpack and Set Solution*: The worker unpacks a solution and then updates solution information.
- *Receive a Subtree*: The worker unpacks and reconstructs a subtree that was sent by another process.

Figure 2.6 shows the task management system of a worker. First, the worker listens and processes messages until there are no more left. The messages may contains a feasible solution, a shared subtree, or a request to share a subtrees. If there is a termination check request from the master, the worker reports its state to its hub. If it is asked to terminate by its hub, the worker terminate search; otherwise, it continues to listen and process messages. If there is no termination check request, the worker checks whether it has

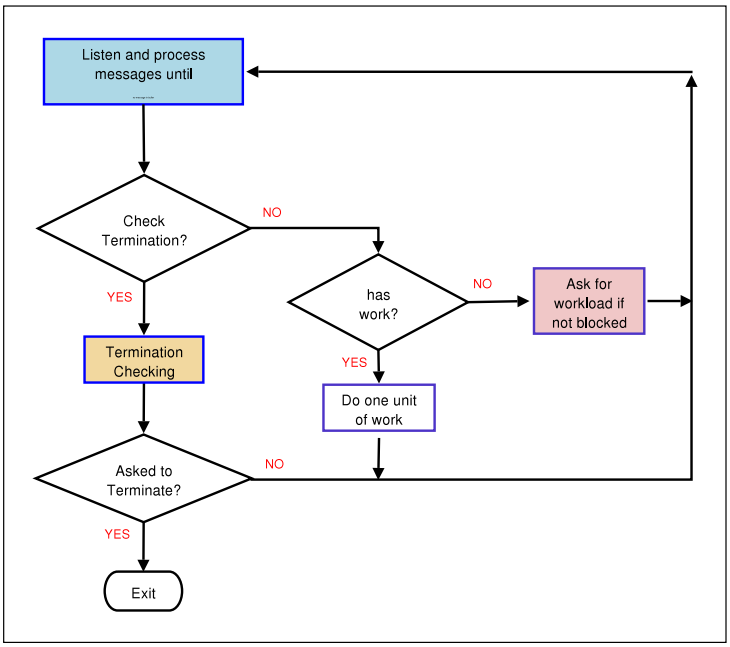

Figure 2.6: Worker Task Management

work. If it has no work, it asks its hub for more work and continues to listen and process messages. If it has work, the worker explores a subtree for a certain number of nodes or a certain period of time, and continues to listen and process messages. This procedure repeats until the worker is asked to terminate by its hub.

2.4.2 Key Tasks

In this section, we discuss several important tasks in detail so that we can give a clear picture of what the specific tasks are and how they are managed in ALPS.

Node Processing

Within a subtree, the nodes that are candidates for processing and expansion are stored in a node pool as a priority queue. ALPS has the notion of a *quality threshold*. Any node whose quality falls below this threshold can be *pruned* without actually processing it.

ALPS assigns each node a status. There are five possible stati indicating what state the node is in.

- *Candidate*: Indicates the node is a candidate for processing, i.e., it is a leaf node in the search tree.
- *Evaluated*: Indicates the node has been processed, but expansion has not been performed. Note that evaluation may change the quality of the node.
- *Pregnant*: Indicates an action has already been chosen to expand the node and the children can be produced when instructed.
- *Branched*: Indicates that expansion has been performed, i.e., the children of the node have been produced and added to the appropriate node pool.
- *Pruned*: Indicates the node has been pruned. This means that (1) the quality of the node fell below the quality threshold, (2) it was determined that the node is *infeasible*, i.e., does not contain any solutions, (3) the node has expanded and all its children have been created. In most cases, pruned nodes are deleted immediately, and hence, there are typically no nodes with this status in the tree.

In terms of these stati, processing a node involves converting its status from *candidate* to *evaluated*, *pregnant* or *pruned*. A node whose processing results in the status *evaluated* is put back into the queue with its new quality to be branched upon when next selected

for processing. Within the context of ALPS, expansion simply means creating the children of the current node and determining their initial quality measures. After expansion, the node's status is set to *branched* and the stati of each of the node's descendants is set to *candidate*.

Node Selection

ALPS provides the following five built-in search strategies to select the node to be processed next: *Best-first search*, *Best-estimate search*, *Breadth-first search*, *Depth-first search*, and *Hybrid search*. ALPS' best-first search, breadth-first search, and depth-first search are standard strategies, which are described in textbooks such as [92]. In best-estimate search, ALPS choose the node with the best solution estimate as the next node to explore. The default value of the estimate is the same as the path cost, which means that, by default, best-estimate search is the same as the best-first search. The user can provide a more accurate solution estimate scheme to assign an estimate to each node if desired. In the hybrid search strategy, ALPS selects one of the children of the currently processed node as the next node until the currently processed node is pruned, then it selects a sibling (if it is not pruned) as the next one to process. It stops diving if all the siblings of the currently processed node are pruned or the quality of the children or siblings is worse than the best available by a certain threshold. The hybrid strategy is the default search strategy of ALPS. Two of the provided search strategies (breadth-first and depth-first) do not require user-defined quality measures. The best-first, best-estimate, hybrid search strategies require user-defined quality measures.

ALPS does not have a central pool to store the nodes to be processed. Each process has its own subtree pool and each subtree has its own node pool. When searching

in parallel, each KB can only guarantee that the requirements of a search strategy are satisfied for the part of search tree that is explored by the KB. ALPS does not have search strategies that can be applied to the whose tree mainly because of its totally decentralized design.

Termination Detection

In ALPS, we use Mattern's four counter method for termination detection. Figures 2.7, 2.8, and 2.9 list the main steps that the master, hubs, and workers take during termination checking. Figure 2.10 shows the message flow during the termination checking.

The user can set a maximum number of nodes that ALPS can process and a maximum wallclock time that ALPS can spend in the search. If either of these limits is reached, the master forces the system to terminate. It sends a force termination message to all hubs, and then hubs send the same message to the workers. After receiving the message, the workers delete all their work. The system then does the termination checking in the normal way.

2.5 Load Balancing

2.5.1 Static Load Balancing

ALPS has two static load balancing schemes: *two-level root initialization* and *spiral initialization*. Based on the characteristics of the problem to be solved, users can choose either one to initialize the search.

1. The master will activate the termination check task if it finds that the following conditions hold (we say it is *pseudo-idle*):
   (a) there are no messages left in its message buffer,
   (b) all hubs have reported their status (workload and counts of messages sent and received.) at least once,
   (c) the system workload *appears* to be zero, and
   (d) the message sent count equals the message received count.
2. The master first sends a message with tag `AlpsMsgAskPause` to the hubs and asks them to do termination checking, and then waits for the clusters to report new stati.
3. After obtaining the stati of all clusters, the master sums up the quantities of work, and the numbers of message sent and received. If the three conditions of the four counter method are satisfied, then the system is *truly idle* and the master informs hubs to terminate. Otherwise, the master sends a message to the hubs to ask them to continue to work. The master blocks the termination check task and does a round of load balancing.

Figure 2.7: Termination Checking of the Master

1. After receiving a message with tag `AlpsMsgAskPause`, the hub activates its termination check thread and informs its workers to do termination checking.
2. The hub receives and sums up the status information sent by its workers.
3. The hub sends the updated cluster status to the master.
4. The hub waits to be notified of whether to terminate or continue by the master. If the instruction is to terminate, then it sets the `terminate` flag to `true`, otherwise, it sets the `terminate` flag to `false` and blocks the termination check thread. It also forwards the instruction to its workers.

Figure 2.8: Termination Checking of the hubs

1. Once receiving a message with tag `AlpsMsgAskPause`, the worker activates its termination check thread
2. If all messages have been processed, the worker reports its current status to its hub.
3. The worker then waits for instruction from its hub to decide whether to terminate. If the instruction is to terminate, the worker sets the `terminate` flag to `true`, otherwise, it sets the `terminate` flag to `false` and blocks the termination check task.

Figure 2.9: Termination Checking of the worker

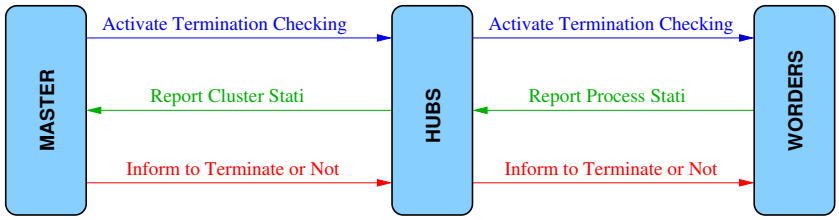

Figure 2.10: Message Flow in Termination Checking

Two-level Root Initialization

Two-level root initialization is based on the idea of root initialization [55]. Figure 2.11 shows the main steps of this scheme. Unlike root initialization, two-level root initialization allows more than one process to help in initializing the search. By doing so, we are able to reduce the ramp-up time. The implementation of two-level root initialization is straightforward. Also, experiments show that if the number of nodes created and allocated to each worker is large enough, the workload will be reasonably balanced, at least for certain classes of problems.

ALPS uses a formula to determine the proper number of nodes to be generated during initialization. The formula is a function of node processing time and the number of launched processes. The shorter the node processing time is, the larger the number of nodes to be generated. In cases where processing a node takes significant time, the two-level root initialization scheme might take a long time, even if it generates few nodes.

1. The master expands the root and creates a set of new nodes. The master then selects a new node and expand it again. This procedure continues until the number of leaf nodes is not less than a pre-specified number. In this step, the master is essentially conducting a serial tree search. For easy problems, the master might fail to generate the pre-specified number of nodes and complete the search by itself.
2. The master distributes the leaf nodes to the hubs in a round-robin fashion.
3. After receiving the nodes sent by the master, the hubs generate a pre-specified number of nodes for theirs workers just as the master did for them.
4. The hubs send the leaf nodes to their workers in a round-robin fashion.
5. The workers receive nodes and form subtrees from each received node.

Figure 2.11: Two-level Root Initialization

Spiral Initialization

Spiral initialization is designed for circumstances in which processing a node takes a long time or where creating a large number of nodes during ramp up is difficult. Figure 2.12 shows how spiral initialization works. Spiral initialization is similar to the initialization strategies used by some packages with the master-worker paradigm, such as SYMPHONY. The advantage of spiral initialization is that as many KBs as possible can help to partition the state space initially by expanding nodes. The initialization is generally faster than the two-level root initialization. The major objective of spiral initialization is to ensure all KBs have work to do as quickly as possible. Once every KB

1. Given the root, the master expands it and creates a set of new nodes.
2. The master distributes the new nodes to other KBs in a round-robin fashion. The master records which processes have been sent a node.
3. After receiving a node sent by the master, a KB expands the received node. If this results in multiple nodes, it informs the master that it can donate nodes; otherwise, it tells the master that it can not.
4. The master asks the processes having extra nodes to donate a node to the processes that have not yet been sent a node. If all KBs (except the master) have been sent at least one node, the master tells other KBs that the static load balancing is completed.

Figure 2.12: Spiral Initialization

has nodes to work on, spiral initialization stops. It does not take into account the quality of work at each KB. Therefore, spiral initialization requires dynamic load balancing to redistribute work more evenly as the search proceeds.

2.5.2 Dynamic Load Balancing

Dynamic load balancing includes both *quantity balancing* and *quality balancing*. Quantity balancing ensures that all worker have work to do, while quality balancing ensures that all workers are doing *useful* work. During the search, the master periodically balances the workload in each cluster. This is done by maintaining skeleton information about the full search tree in the master process. This skeleton information only includes

what is necessary to make load-balancing decisions, primarily the quality of subtrees available for processing on each cluster. Each hub is responsible for balancing the load among its workers. In dynamic load balancing, subtrees are shared instead of nodes. We must be more careful about how we perform load balancing in order to keep subtrees together (this is important to allow implementation of our differencing scheme, described later). The dynamic load balancing scheme of ALPS consisting of

- *Intra-cluster dynamic load balancing*, and
- *Inter-cluster dynamic load balancing*

These two dynamic load balancing schemes work together to ensure that the workload is balanced among the processes that participate in the tree search.

Intra-cluster Dynamic Load Balancing

Intra-cluster dynamic load balancing is used to dynamically balance the workload among processes that are in the same cluster. We propose two schemes to perform intra-cluster dynamic load balancing. One is the *receiver-initiated* scheme and the other is the *hub-initiated* scheme. In these load-balancing schemes, ALPS assumes that a worker does not know the workload information of other workers, while the hub roughly knows the workload information of its workers because they periodically report their workload quality and quantity to it. The hub is in the position to do both quality and quantity balancing, but the workers are only able to initiate quantity balancing.

In the receiver-initiated balancing operation, a worker that needs more work requests its hub to find another worker that can share some of its work. The goal of the receiver-initiated scheme is only to balance the *quantity* of workload among the workers. A high-level description of the receiver-initiated scheme is shown in Figure 2.13.

1. Once the quantity of the workload of a worker is below a predetermined threshold, this worker (receiver) sends a message to inform the hub of this situation.

2. Upon receiving the message, the hub knows the receiver needs more work and tries to choose a worker (donor) to donate work. If the hub finds that no worker can donate, it sends an empty message to the receiver. Otherwise, it sends a message to the donor and asks it to donate a subtree to the receiver.

3. After the donor receives the message from the hub, it checks its workload. If it has no extra workload, it sends an empty message to the receiver to let the receiver know that it cannot donate a subtree. If it only has one subtree, it will split the subtree into two parts and send one part to the receiver. If it has more that one subtree, it sends the subtree with best quality to the receiver.

4. The receiver either receives an empty message or a subtree. If it receives an empty message, it knows there is not enough work in this cluster and it stops asking for workload temporarily; otherwise, the receiver unpacks the message, reconstructs the subtree, and puts it in its subtree pool.

Figure 2.13: Receiver-initiated Dynamic Load Balancing

When a donor has several subtrees, it donates its the subtree with the best quality. When a donor just has one subtree, it splits the subtree into two parts. The way in which ALPS splits a subtree is as follows

1. Check how many nodes are in the subtree. If the subtree contains only one node, then the splitting procedure fails.

2. Find the node N_j with the best quality.

3. Backtrack from node N_j, and stop at node N_k if

   - the number nodes in the subtree with root at N_k is roughly the half of the total number nodes in the original subtree, or
   - the number nodes in the subtree with root at N_k reaches a prescribed maximum. This maximum is determined by the message buffer size and the estimated size of a node.

After splitting the subtree, the donor sends the new subtree, whose root is N_k, to the receiver. Sharing high quality subtrees reduces the frequency of performing dynamic load balancing.

In the hub-initiated intra-cluster dynamic load balancing, the hub identifies workers that need extra work and the workers that have extra work. This scheme balances the work in terms of both *quantity* and *quality*. Figure 2.14 shows the main steps in the hub-initiated scheme. Figure 2.15 shows the algorithm flow of this scheme.

Inter-cluster Dynamic Load Balancing

The workload can also become unbalanced between clusters, though this should happen much more slowly. During the tree search, the hubs periodically report their workload

1. Periodically, the hub checks the quality and quantify of work among its workers. The hub chooses receivers whose workload quality is less than the average workload quality by a pre-specified factor and donors whose workload quality is greater than the average workload quality by a pre-specified factor.

2. Let the number of donors be d and the number of receivers be r. The hub then pair the first $n = min\{d,\ r\}$ donors and receiver. If n is 0, the hub terminates from the load balancing procedure.

3. The hub sends a message to each donor. The message has a receiver's identification and workload information.

4. Once a donor has received the message, it first checks its workload. If it has no work to share, it sends an empty message to the receiver and lets the receiver know that it cannot donate a subtree. If it has only one subtree, it splits the subtree into two parts and sends one part to the receiver. If it has more than one subtree, it sends the subtree with best quality to the receiver.

5. After receiving the reply from its donor, the receiver unpacks the message, reconstructs the subtree, and stores it in its subtree pool if the message is not empty.

Figure 2.14: Hub-initiated Dynamic Load Balancing

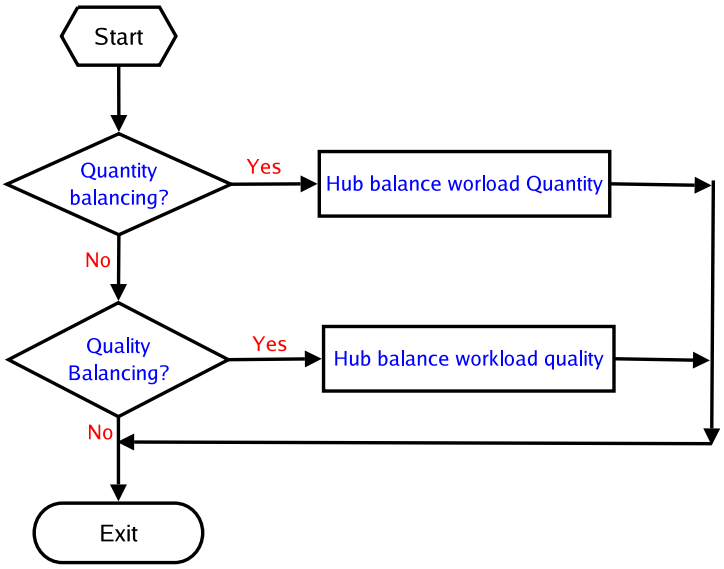

Figure 2.15: Algorithm Flow of The hub-initiated Scheme

information to the master. The master has a rough idea of the system workload and the workload of each cluster. Periodically, the master balances the workload among clusters in terms of both quality and quantity. The inter-cluster load balancing scheme is similar to the hub-initiated intra-cluster load balancing scheme. The main steps of this scheme are shown in Figure 2.16.

In the inter-cluster load balancing scheme, the master determines when and how to balance both the quality and quantity of workload between clusters. The hubs choose which workers should donate or receive work. The master and hubs need to know the system workload information in order to effectively balance the work.

1. Periodically, the master checks the workload among each cluster. First, the master chooses the clusters whose quantities of work are less than a certain pre-specified level as receivers. If the master cannot find any receivers, then it chooses clusters whose quality of work is less than a pre-specified level. If the master still cannot find receivers, it quits from the load balancing procedure. Otherwise, the master select the clusters whose workload is greater than the pre-specified level as donors.

2. Let the number of donors be d and the number of receivers be r. The master then pair the first $n = min\{d,\ r\}$ donors and receivers. If n is 0, the master terminates the load balancing procedure.

3. The master sends a message to each donor hub to ask it to share its work. The message has a receiver's identification and workload information.

4. Once the donor hub receives the message, the hub tries to identify its most loaded worker. If it is successful, the hub asks the worker to sends a subtree the receiver hub; otherwise the hub sends an empty message to the receiver hub.

5. After receiving the message from the donor, the receiver hub forwards the message to its most lightly loaded worker, which reconstructs the subtree if the message is not empty.

Figure 2.16: Inter-cluster Dynamic Load Balancing

2.6 Class Hierarchy of ALPS

Figure 2.17 shows the class hierarchy of ALPS. The class names with prefix `Alps` belongs to ALPS library, while the class names with prefix `User` are the class that the user need derive when developing an application. Figure 2.17 shows that ALPS consists of three parts.

- **Knowlege**:
  - `AlpsSolution`,
  - `AlpSubTree`,
  - `AlpTreeNode`, and
  - `AlpModel`.
- **Knowlege pools**:
  - `AlpSolutionPool`,
  - `AlpSubTreePool`, and
  - `AlpNodePool`.
- **Knowege brokers**:
  - `AlpKnowledgeBrokerSerial`, and
  - `AlpKnowledgeBrokerMPI`.

As mentioned previously, ALPS has four basic types of knowledge: *model*, *node*, *subtree*, and *solution*. The `AlpsModel` class stores a logical or mathematical representation of the problem to solve. The `AlpsTreeNode` class has the bookkeeping information for a node, such as the index, depth, quality, parent index, etc. Two member

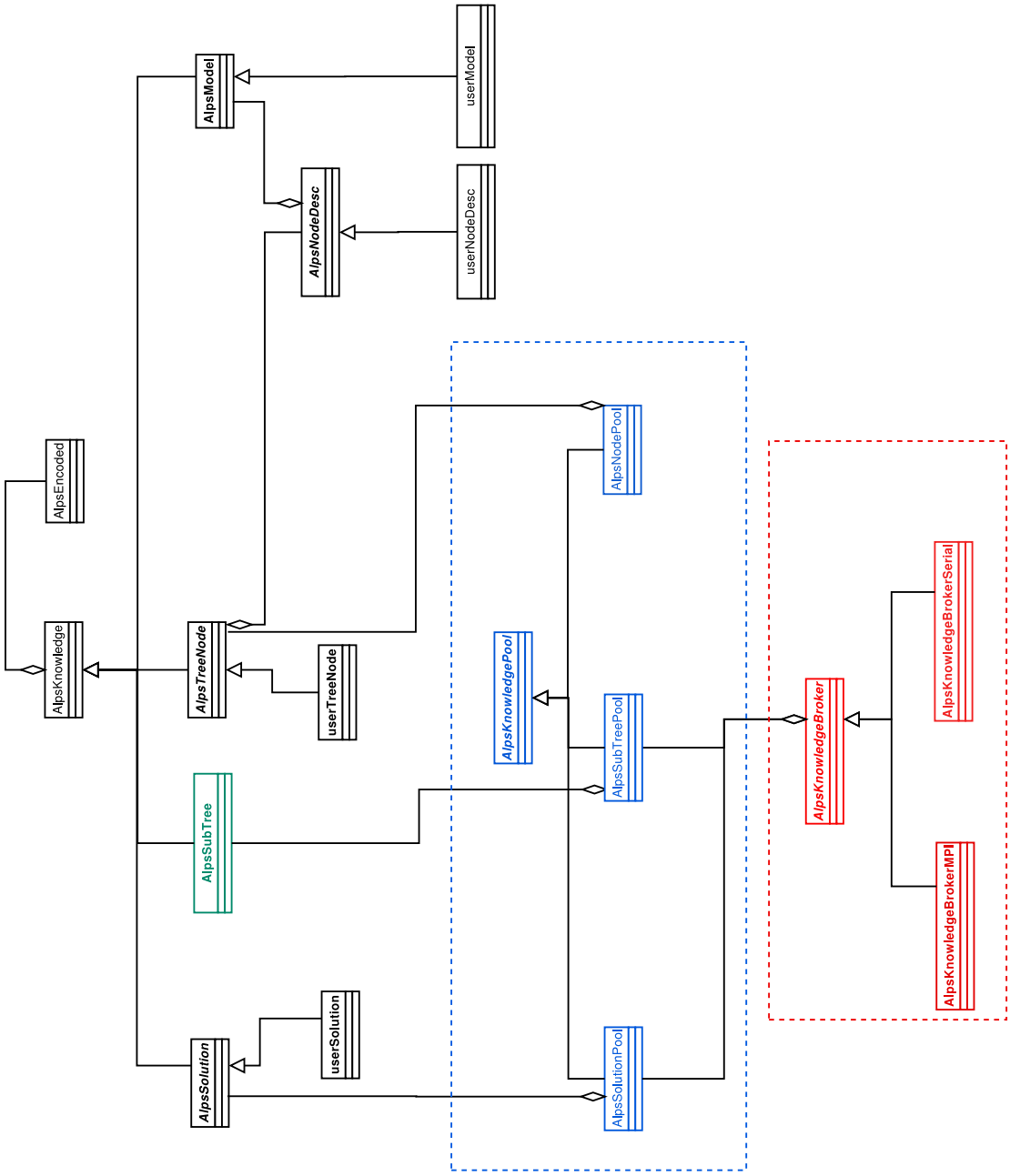

Figure 2.17: Class Hierarchy of ALPS

functions, `process()` and `branch()`, are virtual and required to be defined by users so that ALPS knows how to process and how to create the successors of a node. The `AlpsSolution` class is used to describe a solution. The base class has no member data because ALPS does not know what a solution looks like for a particular application. The `AlpsSubTree` class holds node pools that store the nodes that comprise the subtree. It defines the functions that implement exploration of a subtree.

The `AlpsNodePool`, `AlpsSolutionPool`, and `AlpsSubTreePool` classes define the knowledge pools supported by ALPS. These pools are managed either based on priority queues or store knowledge in a sorted fashion. ALPS provides two types of knowledge brokers to manage knowledge pools and handle inter-process communication for parallel execution.

- *AlpsKnowledgeBrokerSerial* is for single-process execution, in which knowledge is passed via memory directly.
- *AlpsKnowledgeBrokerMPI* provides facilities for communication over a network using the message passing interface MPI.

2.7 Developing Applications

2.7.1 Procedure

To develop applications based on ALPS, the user needs to follow two steps:

- derive problem-specific subclasses from ALPS' base classes and
- write a `main()` function.

In deriving these classes, the user needs provide problem specific information such as how to define an instance: how to describe, process, and expand a node; and how to represent a solution. The base classes that the user must derive from are

- `AlpsModel`,
- `AlpsTreeNode`,
- `AlpsNodeDesc`, and
- `AlpsSolution`.

If the application has custom parameters, the user can also derive a subclass based on the class `AlpsParameterSet`. The user-derived subclasses contain problem-specific data and define a set of functions that are needed to solve the problem. If the user wants to use other communication protocols besides MPI, then a new knowledge broker based on ALPS's base knowledge broker class is needed. Next, we use the knapsack problem solver KNAP as an example of developing applications based on ALPS.

2.7.2 Example: The KNAP Application

As described in Chapter 1 , the *knapsack problem* is to find the maximum total value without exceeding the capacity constraint. It has the following mathematical formulation,

$$\max\{\sum_{i=1}^{m} p_i x_i : \sum_{i=1}^{m} s_i x_i \leq c, x_i \geq 0 \text{ integer}, i = 1, 2, \ldots, m\}, \tag{2.2}$$

where

$p_i =$ the unit value of item i;

$x_i =$ the number of item i in the knapsack;

$s_i =$ the unit size of item i;

$c =$ the capacity of the knapsack.

If additional constraints $x_i \in \{0, 1\},\ i = 1, 2, \ldots, m$ are added, then it is the 0-1 knapsack problem. The knapsack problem is a classical discrete optimization problem and has a wide range of applications. It is also the most elementary form of MILP, in the sense that there is only one constraint. The knapsack problem belongs to the complexity class $\mathcal{NP}-$hard. Therefore, in principle, any MILP can be transformed into a knapsack problem. The knapsack problem is one of the most important optimization problem.

Two basic approaches to finding solutions to the knapsack problem are *dynamic programming* and *branch-and-bound*. Dynamic programming is a technique often used to solve sequential, or multi-stage, decision problems. Dynamic programming recursively decomposes certain hard problems into smaller pieces that make it easier to find solutions. The book by Bertsekas [19] provides an extensive description about the theory and application of dynamic programming. Branch and bound, introduced in in Chapter 1, is another popular way to solve the knapsack problem.

In the rest of this section, we describe KNAP, an application built using ALPS for solving the 0-1 knapsack problem. The algorithm used by KNAP is a *branch-and-bound* tree search algorithm. Following is a brief description of the main elements of the KNAP application.

- *Processing method*: Computes the path cost as the sum of the values of items that are in the knapsack. It also tests whether the current solution is optimal by checking if the optimality gap is zero or the number of leaf nodes is zero.

- *Successor function*: Expands a node into two successors by deciding whether to put the given item into the knapsack or not.

- *Pruning rule*: Decides whether a node can be pruned based on whether it can produce a solution whose path cost is lower than that of the current best solution.

Based on the procedure for developing applications, we derived four subclasses (`KnapModel`, `KnapTreeNode`, `KnapNodeDesc`, and `KnapSolution`) from ALPS' base classes. Those subclasses provide problem specific-data and functions that are required to solving knapsack problem.

The Model Subclass

The member data of the user's model class are simply the data needed to describe the problem to be solved. Subclasses (`KnapModel` is derived from `AlpsModel`. Its main data members include the following:

- `int capacity_`: specifies the capacity of the knapsack and

- `std::vector< std::pair<int, int> > items_`: stores the sizes and profits of the items.

`KnapModel` has various access and modification member functions. Since the mathematical description of the model must be broadcast to all processes for parallel execution, `KnapModel` also defines `encode()` and `decode()` functions to convert the model data into a `string` and re-construct a model from a string.

The Tree Node Subclass

The tree node subclass `KnapTreeNode` contains member data to specify the position of a node in the search tree. It has additional member data `branchedOn` to indicate which item is involved in the branching decision. `KnapTreeNode` overrides some functions in ALPS' base tree node class, including

- `evaluate()`: Computes the path cost and checks if the current state is a goal state;
- `branch()`: Defines how to expand a node; and
- `createNewTreeNode()`: Creates the children of a node after expanding.

`KnapTreeNode` also defines functions `encode()` and `decode()` for running in parallel.

The Node Description Subclass

KNAP's node description subclass `KnapNodeDesc` stores the actual information about the state in the search space that is contained in the tree node. The data members of the class include the following.

- `KnapVarStatus* varStatus_`: Keeps track of which variables (items) have been fixed by branching and which are still free. Type `KnapVarStatus` can takes values *not decided*, *fixed to be in the knapsack* or *fixed not to be in the knapsack*.
- `int usedCapacity_`: The total size of the items fixed to be in the knapsack.

- `int usedValue_`: The total value of the items fixed to be in the knapsack.

`KnapNodeDesc` provides a set of functions to access and modify the description of a node, and defines `encode()` and `decode()`.

The Solution Subclass

A solution is a sequence of actions that map the initial state to a goal state. In the user's solution class, there must be member data that can store the actions, and the member functions to access and modify member data. `KnapSolution` has the following data members:

- `int* solution_`: indicates whether put each item in the knapsack (1) or leave it out of the knapsack (0) and
- `int value_`: the total value of the items that are in knapsack.

`KnapSolution` provides a set of functions to access and display the solution, and defines `encode()` and `decode()`.

The `main()` Function

A `main()` function for KNAP is shown in figure 2.18 and is based on the template provided by ALPS. The `main()` function is quite simple. To run parallel code, the user needs to declare a parallel broker, such as `AlpsKnowledgeBrokerMPI`; to run serial code, the user needs to declare a `AlpsKnowledgeBrokerSerial` broker. If desired, the registration of knowledge types and root formulation steps can be moved into other functions so that `main()` function can be further simplified. Then, the broker searches for solutions and prints search results.

```
int main(int argc, char* argv[])
{
    KnapModel model;

#if defined(SERIAL)
    AlpsKnowledgeBrokerSerial broker(argc, argv, model);
#elif defined(PARALLEL_MPI)
    AlpsKnowledgeBrokerMPI broker(argc, argv, model);
#endif

    broker.registerClass("MODEL", new KnapModel);
    broker.registerClass("SOLUTION", new KnapSolution);
    broker.registerClass("NODE", new KnapTreeNode);

    broker.search();
    broker.printResult();
    return 0;
}
```

Figure 2.18: `Main()` Function of KNAP

Chapter 3

A Framework for Parallel Integer Programming

In this chapter, we discuss the tree search algorithms that have been successfully used to solve difficult mixed integer linear programs (MILPs). Also, we study methods to improve scalability when solving MILPs in parallel.

3.1 Branch and Cut

Branch and cut is an extension of the basic branch-and-bound algorithm described in Section 1.2.2. During the *processing* step in branch and cut, if solving the linear programming relaxation of the given subproblem results in a solution that does not satisfy the integrality constraints, we try to find linear inequalities valid for all integer solutions of the subproblem but violated by the solution to the current linear programming relaxation. If we find such inequalities, we add them to the formulation and resolve the linear program. This procedure continues until no additional inequalities can be found

or a pre-specified limit on the number of passes is reached. With the exception of the processing step, branch and cut works exactly like branch and bound.

Algorithm 3.1 LP-based branch-and-Cut Algorithm

1: **Initialize.**
$\mathcal{L} = \{N^0\}$. $z_u = \infty$. $x^u = \emptyset$.
2: **Terminate?**
If $\mathcal{L} = \emptyset$ or optimality gap is within certain tolerance, then the solution x^u is optimal. Terminate the search.
3: **Select.**
Choose a subproblem N^i from $\mathcal{L}$.
4: **Process.**
Solve the linear programming relaxation of N^i. If the problem is infeasible, go to step 2, else let z_i be its objective value and x^i be its solution. If x^i does not satisfy the integrality restrictions, try to find constraints that violated by x^i. If any are found, add the constraints to the description of N^i and go to step 3.
5: **Prune.**
If $z_i \geq z_u$, go to step 2. Otherwise, if x_i satisfies the integrality restrictions, let $z_u = z_i$, $x_u = x_i$, delete from $\mathcal{L}$ all problem j with $z_j \geq z_u$, and go to step 2.
6: **Branch.**
Partition the feasible region of N_i into q subsets $N^{i1}, \ldots, N^{ik}$. For each $i = 1, \ldots, q$, let $z_{ik} = z_i$ and add subproblems N^{ik} to $\mathcal{L}$, go to step 2.

Algorithm 3.1 describes an LP-based branch-and-cut algorithm. There are a number of decisions to be made in branch and cut. These decisions include which node to be selected for processing next, how to divide the feasible region, what kinds of constraints should be added to formulation, and how to improve upper bounds on the value of an optimal solution. We now try to answer these questions.

3.1.1 Branching Methods

The most common way to expand a node is to branch on disjunctions defined by hyperplanes, so that the resulting subproblems are also MILPs. Typically, the hyperplanes

are unit vectors, in which case they divide the feasible range for a single variable. This is called *branching on a variable* or *variable branching*. Given a fractional solution $\hat{x}$ of the subproblem formulated at the node and a set of integer variables G that have a fractional value in the current LP solution $\hat{x}$, we can branch on a variable $j \in G$ and create two subproblems, one by adding the trivial inequality $x_j \leq \lfloor \hat{x}_j \rfloor$ and the other by adding the trivial inequality $x_j \geq \lceil \hat{x}_j \rceil$. Often, the cardinality of the set G is greater than 1, so we need a method to choose which variable to be branched on. Besides branching on variables, there are more complicated branching methods that branch on other types of disjunctions and are sometimes used for specific instances. We introduce the most commonly used branching methods below.

Infeasibility Branching

A simple way to choose the branching variable is to look at degree of violation of integrality for each integer variable. When choosing the variable that is farthest from being integer valued, the method is called *maximum infeasibility* branching. When choosing the variable that is closest to being integer value, the method is called *minimum infeasibility* branching. Achterberg, *et al.* [2] found that choosing branching variables based on maximum infeasibility is no better than simply choosing at random. Generally, the performance of infeasibility branching is not very good because it ignores the fact that variables are often not equally important. Since the information used by infeasibility branching is limited to the LP solution, no special handling is needed for parallel search.

Strong Branching

Strong branching was introduced in CONCORDE [9], a software package for solving the traveling salesman problem. The main idea of strong branching is to estimate the effect of branching on each fractional variable before actually selecting one to branch on. To estimate the objective change that would occur by branching on a given variable, the solver tentatively branches on each candidate variable and evaluates the resulting LP relaxations of the children. Applegate, *et al.* [9] showed that strong branching is an effective method for solving the traveling salesman problem. Estimating the importance of a candidate in this way is relatively expensive, so in strong branching one generally does not calculate estimates for all candidates. There are two ways to reduce the computational overhead associated with the estimation: either (1) select a limited number of candidates (usually based on fractionality of integer variables) or (2) do not solve the LP relaxations to optimality. *Strong branching* is well-suited for difficult combinatorial problems. For parallel branch and bound, there is generally no special handling required.

Pseudocost Branching

Pseudocost branching is a less expensive way to estimate the effect of branching on integer variables [14]. For each variable, we associate two quantities, P_j^- (*down pseudocost*) and P_j^+ (*up pseudocost*), which are the estimates of the increase of the objective value after fixing variable j to its floor and ceiling values respectively. One way to compute the pseudocost of variable j is to use the actual change in objective value changes when variable j is chosen as the branching variable. For a given subproblem, let $f_j = \hat{x}_j - \lfloor \hat{x}_j \rfloor$, z_{LP} be the objective value of a subproblem, z_{LP}^- be the objective

value of the child when fixing variable j to its floor, and z_{LP}^{-} be the objective value of the child when fixing variable j to its ceiling. Then, the up pseudocost is

$$P_j^{-} = \frac{z_{LP}^{+} - z_{LP}}{1 - f_j}, \tag{3.1}$$

and the down pseudocost is

$$P_j^{-} = \frac{z_{LP}^{-} - z_{LP}}{f_j}. \tag{3.2}$$

Each time variable j is chosen as a branching variable, the pseudocosts of variable j can be updated to include this latest information. A common way of doing this is by averaging all observations. Based on up and down pseudocosts, we can assign variable j a score

$$P_j = \mu \max\{P_j^{+}(1 - f_j), P_j^{-} f_j\} + (1 - \mu) \min\{P_j^{+}(1 - f_j), P_j^{-} f_j\}, \tag{3.3}$$

where μ is a non-negative scalar. Achterberg *et al.* suggest using $\mu = 1/6$ [1]. *Pseudocost branching* chooses the variables with the largest score as the branching variable. The advantage of pseudocost branching is that it tries to branch on important variables first in order to improve the lower bound quickly. Most solvers use pseudocost branching or a variant as the default branching method. However, the estimation of pseudocosts is based on historic information, which might not be accurate for future search. Also, at the beginning of the search, there is no information available to initialize pseudocosts. Therefore, we need a method to initialize them. Some methods use the objective coefficients, while others use strong branching to initialized pseudcosts.

Reliability Branching

Achterberg *et al.* [2] proposed a branching method called *reliability branching*, in which strong branching is used not only on variables with uninitialized pseudocosts, but also on variable with *unreliable* pseudocosts. Let η_j^- be the number of times that variable j has been branched down (creating a child by setting the upper bound of the variable to the floor of the LP solution value of this variable), η_j^+ be the number of times that variable j has been branched up (creating a child by setting the lower bound of the variable to the ceiling of the LP solution value of this variable), and η_{rel} be the *reliability* parameter. Variable j is said to be *unreliable* if $\min\{\eta_j^-, \eta_j^+\} < \eta_{rel}$. After comparing the performance of maximum infeasibility branching, strong branching, pseudocost branching, and reliability branching, Achterberg *et al.* conclude that reliability branching is the best. For parallel search, pseudocosts can be shared among processes as discussed previously.

SOS Branching

Special Ordered Sets (SOS) were originally used to efficiently handle multiple choice and separable non-convex problems [17] . A *Special Ordered Set of Type I* (SOS1) is a set of variables (integral or continuous) from which no more than one member may be non-zero. A *Special Ordered Set of Type II* (SOS2) is a set of variables from which no more than two members may be non-zero, and for which the two variables with non-zero values must be adjacent if there are two. The *order relationship* of SOS is specified by the *weight* of each member variable. The weights can simply be sequence numbers or entries in some row of the constraint matrix, which is called the *reference row*. *SOS branching* is a special branching method that uses SOS information to create

subproblems. Here, we only introduce a classic case in which the problem formulation has a constraint like

$$\begin{aligned} \textstyle\sum_{i=1}^{n} x_i &= 1, \\ x_i &\in \{0,1\}. \end{aligned} \tag{3.4}$$

We can choose an interval (x_r, x_{r+1}), where $1 \leq r \leq n-1$ and branch on the set of variables by forcing $\sum_{i=1}^{r} x_i = 0$ in one branch and $\sum_{i=r+1}^{n} x_i = 0$ in the other branch. A common way to decide the interval (x_r, x_{r+1}) is to first compute the *weighted average* of the SOS

$$\bar{w} = \frac{\sum_{i=1}^{n} w_i x_i}{\sum_{i=1}^{n} x_i}, \tag{3.5}$$

then choose x_r, x_{r+1} such that

$$w_r \leq \bar{w} \leq w_{r+1}. \tag{3.6}$$

In the presence of this problem structure, the branch-and-bound tree is more *balanced* when branching on an SOS instead of a variable. However, if there is no logical order among the variables, SOS branching does not usually help [31].

General Hyperplane Branching

Branching on variables may not be effective in certain cases. Cornu*é*jols, *et al.* [27] designed a set of Market Split instances that are very difficult to solve if branching on variables, but easy to solve when branching on constraints. Research on generalized branching methods has increased recently. Most of this research has focused on branching on general disjunctions obtained from hyperplanes. Karamanov and Cornu*é*jols [61]

show that branching on the disjunctions defining mixed integer Gomory cuts at an optimal basis of the linear programming relaxation is more effective than branching on variables for many test instances. Mehrotra and Li [74] study the problem of finding a good branching hyperplane. They propose a heuristic method of generating branching hyperplanes and show good results on solving dense difficult Market Split instances.

3.1.2 Search Strategy

For branch-and-bound algorithms, we would like to select a node that can help with improvement of both the upper and lower bounds. To reduce solution time, it is also important to select a node whose setup costs is not very expensive. Overall, node quality, memory requirements and setup cost are the main criteria in selecting a node. The most common search strategies include *depth-first*, *best-first*, *best-estimate* and *hybrid* methods. Below, we review the advantages and disadvantages of each of them in the context of solving MILPs serially. As discussed in Chapter 2, when searching in parallel, we can only implement a given search strategy locally for the subtree that is being explored. The global search strategy involves a number of other factors, primarily the load balancing strategy.

Depth-first Search

Depth-first search selects the node at maximum depth in the current tree as the next to be processed. Compared with other types of search, depth-first search has the following advantages:

- feasible solutions can be found quickly because such solution are usually found deep in the tree;

- the setup cost (loading basis, refactorization, updating variable bounds in the LP solver, etc.) involved in solving a new subproblem is small when solving the new LP relaxation because there is not much change in the LP relaxation to be solved from parent to child (generally just a change in bound for one of the variables); and

- the storage required to implement a depth-first strategy is low. The set of candidate subproblems is generally smaller than with other strategies.

A disadvantage of the depth-first strategy is that it usually results in the processing of nodes that would otherwise have been pruned. Depth-first has the potential to become "stuck" in one portion of the search tree, and if that portion of the search tree does not contain good solutions, then many low quality nodes may be processed and computation time may be wasted. Therefore, in practice, depth first is not the preferred search strategy for solving MILPs unless users just want a feasible solution or lack of memory is a big issue.

Best-bound Search

The *best-bound* search strategy chooses the next node to be processed to be the one with highest quality, where quality is usually measured as the negation of the objective value of the LP relaxation. This strategy has the advantage that it tends to minimize the number of nodes processed overall and improves the lower bound quickly. By employing a best-bound strategy, tree search will never process a node whose lower bound is greater than the optimal value. However, the best-bound search strategy does not focus on improvement on the upper bound and its memory usage is high because the set of subproblems waiting to be processed is usually larger than that of depth-first search. In

addition, the setup cost to process a new node is high since the best-bound search tends to choose nodes in different parts of the tree. The best-bound search strategy is still widely used but with some modification to overcome these deficiencies.

Best-estimate Search

Best-estimate search strategies attempt to combine the advantages of depth-first and best-bound strategies, while at the same time avoiding the disadvantages. The idea is to try to select a node with high probability of producing a good solution by computing an estimate of the optimal value of each candidate subproblem. B*é*nichou *et al.* [14] and Forrest *et al.* [40] describe how to compute the estimated values for a given node N^i. The node with lowest estimate value is then selected to be processed next. The estimated value is defined as

$$E_i = z_i + \sum_{j \in I} \min(|P_i^- f_i|,\ |P_i^+(1 - f_i)|)). \tag{3.7}$$

This estimate takes into account the fractionality and pseudocosts of the integer variables, and attempts to balance the goals of improving both the upper and lower bounds. However, this method has some of the disadvantages of best-bound search, such as high setup cost and memory usage.

Hybrid Search

Another way of balancing the advantages of depth-first or best-bound search strategies is to use a *hybrid* strategy. For instance, Forrest *et al.* [40] propose a *two-phase* method that first selects nodes according to the best-estimate criterion and switches to a method

that chooses nodes maximizing a different criterion called *percentage error* once a feasible solution is found. For a node i, the percentage error (PE_i) is defined as

$$PE_i = 100 \times \frac{z_u - E_i}{z_i - z_u}. \tag{3.8}$$

Percentage error can be thought of as the amount by which the estimate of the solution obtainable from a node must be in error for the current solution to not be optimal.

Next, we discuss an useful hybrid strategy. This method first chooses a best node based on lower bound or estimate. The search continues in a depth-first fashion until the node to be processed next is worst than the best node in active node set $\mathcal{L}$ by a pre-specified percentage. The backtracking method then chooses the best node in $\mathcal{L}$ in a fashion similar to backtracking in depth-first search. Depending on the pre-specified percentage, this method can be made to behave either like depth-first or best-bound.

As discussed in Chapter 2, the default search strategy of ALPS is a hybrid one. Also, many MILP solvers (like CPLEX, SYMPHONY, and XPRESS) use some kinds of hybrid search strategies as their defaults.

3.1.3 Valid Inequalities

A linear inequality defined by $(a, b) \in \mathbb{R}^{\kappa} \times \mathbb{R}$ called a *valid inequality* if $ax \leq b$, $\forall x \in \mathcal{F}$, where $\mathcal{F}$ is the feasible region of the solution x. Valid inequalities are mainly used to improve the lower bounds yielded by solving an LP relaxation. A valid inequality is said to be *violated* by $\hat{x} \in \mathbb{R}^{\kappa}$ if $a\hat{x} > b$. Valid inequalities violated by the solution of the current LP relaxation are also called *cutting planes* or *cuts*, because adding the violated valid inequalities to the formulation "cuts off" the solution to the LP relaxation and

improves lower bound produced by the LP relaxation in most cases. Given a solution to the current LP relaxation that is not feasible to the MILP, the *separation* problem is to find a valid inequality violated by that solution. Sometimes, the separation problem can be expensive to solve. We also have to be careful not to add too many valid inequalities to the formulation or the LP relaxations can become difficult to solve. We need balance the trade-off between the benefits of applying valid inequalities and the cost of finding and using them.

Valid inequalities can be divided into two broad classes. The first group are the valid inequalities that are based only on the integrality of variables, and do not use any problem structure. The second group exploits partial or full problem structure. Next, we briefly introduce the typical inequalities used in branch-and-cut.

Gomory's Mixed Integer Inequalities

Consider an integer program $min_{x \in \mathcal{P} \cap \mathbb{Z}_+} c^T x$, where and $\mathcal{P} = \{x \in \mathbb{R} \mid Ax = b,\ x \geq 0\}$, $A \in \mathbb{Q}^{m \times n},\ b \in \mathbb{Q}^m, c \in \mathbb{Q}^n$. Suppose x^* is an optimal solution of the LP relaxation of this integer program, $B \subseteq \{1, \ldots, n\}$ is the set of basis variables and $N = \{1, \ldots, n\} \backslash B$ is the set of nonbasic variables. Let A_B be the submatrix formed only by selecting the columns corresponding to the basic variables from matrix A and let A_N be the submatrix formed only by selecting the columns corresponding to the non-basic variables from matrix A. Also, let $x_B^* = A_B^{-1} - A_B^{-1} A_N X_N^*$. If x^* is not integral, then at least one of the components of x_B^* is fractional. Let $i \in B$ be such that $x_i^* \notin \mathbb{Z}$. Since every feasible integral solution $x \in X$ satisfies $x_B = A_B^{-1} - A_B^{-1} A_N x_N$, we have

$$A_{i.}^{-1} b - \sum_{j \in N}^{-1} A_{i.}^{-1} A_{.j} x_j \in \mathbb{Z} \ \ \forall x \in \mathcal{P}. \tag{3.9}$$

Note the term on the left of (3.9) remains integral when adding integer multiples of x_j, $j \in N$, or an integer to $A_{i\cdot}^{-1}$b. Now if we let $f(\alpha) = \alpha - \lfloor \alpha \rfloor$, for $\alpha \in \mathbb{R}$, then we get

$$f(A_{i\cdot}^{-1}) - \sum_{j \in N}^{-1} f(A_{i\cdot}^{-1} A_{\cdot j}) x_j \in \mathbb{Z} \ \ \forall x \in \mathcal{P}. \tag{3.10}$$

Since $0 \leq f(\cdot) \leq 1$ and $\mathcal{P} \in \mathbb{R}_+$, we obtain

$$f(A_{i\cdot}^{-1}) - \sum_{j \in N}^{-1} f(A_{i\cdot}^{-1} A_{\cdot j}) x_j \leq 0 \ \ \forall x \in \mathcal{P}. \tag{3.11}$$

Let $\bar{a}_j = A_{i\cdot}^{-1} A_{\cdot j}$, $\bar{b} = A_{i\cdot}^{-1} b$, $f_j = f(\bar{a}_j)$, $f_0 = f(\bar{b})$, and $N^+ = \{j \in N : \bar{a}_j \geq 0\}$ and $N^- = N \backslash N^+$. Expression (3.9) is equivalent to $\sum_{j \in N} \bar{a}_j x_j = f_0 + k$ for some $k \in \mathbb{Z}$. We now consider two cases: if $\hat{x} \in \mathcal{P}$ is such that $\sum_{j \in N} \bar{a}_j \hat{x}_j \geq 0$, then

$$\sum_{j \in N^+} \bar{a}_j \hat{x}_j \geq f_0 \tag{3.12}$$

must hold. Otherwise, we must have $\sum_{j \in N^-} \bar{a}_j \hat{x}_j \leq f_0 - 1$, which is equivalent to

$$-\frac{f_0}{1 - f_0} \sum_{j \in N^-} \bar{a}_j \hat{x}_j \geq f_0. \tag{3.13}$$

Let $\mathcal{P}^1 = \mathcal{P} \cap \{x : \sum_{j \in N} \bar{a}_j x_j \geq 0\}$ and $\mathcal{P}^2 = \mathcal{P} \cap \{x : \sum_{j \in N} \bar{a}_j x_j \leq 0\}$. Suppose (a^1, b^1) and (a^2, b^2) are valid inequalities for polyhedron P^k for $k = 1, 2$, then we have

$$\sum_{i=1}^{n} \min(a_i^1, a_i^1) x_i \leq \max(\alpha^1, \alpha^2) \tag{3.14}$$

for all $x \in \mathcal{P}^1 \cup \mathcal{P}^2$ and $x \in \text{conv}(\mathcal{P}^1, \mathcal{P}^2)$. Applying this observation to the inequalities

(3.12) and (3.13) valid for $\mathcal{P}^1$ and $\mathcal{P}^2$ respectively, we obtain the inequality

$$\sum_{j \in N^+} \bar{a}_j x_j - \frac{f_0}{1 - f_0} \sum_{j \in N^-} \bar{a}_j x_j \geq f_0, \tag{3.15}$$

which satisfied for all $x \in \mathcal{P}$. This inequality can be strengthened in the following way. Note the derivation of 3.15 remains valid if adding integer multiples to the integer variables, By adding integer multiplies, we may put each integer variable either in the set N^+ or N^-. If a variable is in N^+, the final coefficient in (3.15) is $\bar{a}_j$ and the best possible coefficient after adding integer multiplies is $f_j = f(\bar{a_j})$. In the variable is in N^-, the final coefficient in (3.15) is $-\frac{f_0}{(1-f_0)\bar{a}_j}$ and $\frac{f_0(1-f_0)}{1-f_0}$ is the best choice. Overall, we obtain the best possible coefficient of the variable by using $\min(f_j, \frac{f_0(1-f_j)}{1-f_0})$. This gives us a Gomory's mixed integer inequality

$$\begin{array}{l} \sum_{\substack{j: f_j \leq f_0 \\ x_j \in \mathcal{Z}}} f_j x_j + \sum_{\substack{j: f_j \geq f_0 \\ x_j \in \mathcal{Z}}} \frac{f_0(1-f_j)}{1-f_0} x_j + \\ \sum_{\substack{j \in N^+ \\ x_j \in \mathcal{Z}}} \bar{a}_j x_j - \sum_{\substack{j \in N^- \\ x_j \in \mathcal{Z}}} \frac{f_0}{1-f_0} \bar{a}^j \geq f_0 \ \ \forall x \in \mathcal{P}. \end{array} \tag{3.16}$$

Gomory [45] showed that an algorithm based on iteratively adding these inequalities solves $min\{c^T x : x \in \mathcal{P} \cap \mathbb{Z}\}$ in a finite number of steps.

Knapsack Cover Inequalities

The concept of a cover has been used extensively in the literature to derive valid inequalities for (mixed) integer sets. In this section, we first show how to use this concept to derive *cover inequalities* for a 0-1 knapsack set. We then discuss how to extend these inequalities to more complex mixed integer sets.

Consider the 0-1 knapsack set

$$P = \{x \in \mathbb{B}^N : \sum_{k \in N} a_k x_k \leq b\}$$

with non-negative coefficients, i.e., $a_k \geq 0$ for $k \in N$ and $b \geq 0$. The set $C \subseteq N$ is a cover if

$$\lambda = \sum_{k \in C} a_k - b > 0. \tag{3.17}$$

In addition, the cover C is said to be *minimal* if $a_k \geq \lambda$ for all $k \in C$. To each cover C, we can associate a simple valid inequality that states "not all decision variables x_k for $k \in C$ can be set to one simultaneously".

Let $C \subseteq N$ be a cover. $\sum_{k \in C} x_k \leq |C| - 1$ is a *cover inequality* and satisfied by every $x \in P$. If C is minimal, then the inequality defines a facet of conv(P_C) where $P_C = P \cap \{x : x_k = 0,\ k \in N \backslash C\}$. Cover inequalities are generally not facet-defining, but they can be strengthened through *sequential or simultaneous lifting*. More information about cover inequality and lifting procedures can be found in [52, 78, 110, 113].

Flow Cover Inequalities

It is quite common that MILPs have special structures called *single node flow model*, such as that shown in Figure 3.1. The single node flow model has following representation

$$X = \{(x, y) \in \mathbb{R}^n_+ \times B^n : \sum_{k \in N^+} x_k - \sum_{k \in N^-} x_k \leq d,\ x_k \leq m_k y_k,\ k \in N\}, \tag{3.18}$$

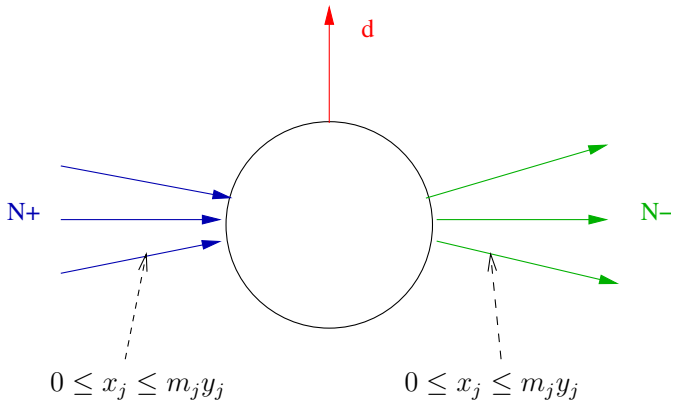

Figure 3.1: Single Node Flow Model

where $N = N^+ \cup N^-$ and $n = |N|$. The x variables are arc flows that need satisfy external demand of d and the variable upper bound constraints. m_k is the arc k and y_k is a binary variable that indicates whether arc j is open.

Let $C^+ \subseteq N^+$ and $C^- \subseteq N^-$. Then, the set $C = C^+ \cup C^-$ is called a *flow cover* if $\sum_{k \in C^+} m_k - \sum_{k \in C^-} m_k = d + \lambda$ with $\lambda > 0$. The inequality

$$0 \leq d + \sum_{k \in C^-} m_k - \sum_{k \in C^+} x_k - \sum_{k \in C^{++}} (m_k - \lambda)(1 - y_k) + \sum_{k \in L^-} \lambda y_k + \sum_{k \in L^{--}} x_k, \quad (3.19)$$

where $C^{++} = \{k \in C^+ : m_k > \lambda\}$ $L^- = \{k \in N^- \backslash C^- : m_k > \lambda\}$ and $L^{--} = N^- \backslash (L^- \cup C^-)$, is called a *simple generalized flow cover inequality* (SGFCI) and is satisfied by every $(x, y) \in X$.

A SGFCI can be strengthened through a lifting procedure known as *sequence independent lifting*. When applying the lifting procedure to a SGFCI, we obtain a *lifted*

simple generalized flow cover inequality (LSGFCI)

$$\begin{aligned}\sum_{k\in C^+} x_k - \sum_{k\in C^{++}}(m_k-\lambda)(1-y_k) + \sum_{k\in N^+\setminus C^+}\alpha x_k - \sum_{k\in N^+\setminus C^+}\beta y_k \\ \leq d' - \sum_{k\in C^-} g(m_k)(1-y_k) + \sum_{L^-}\lambda y_k + \sum_{k\in L^{--}} x_k \;\; \forall (x,y)\in X,\end{aligned} \tag{3.20}$$

where g is a superadditive lifting function. Generally, LSGFCI have better performance that SGFCI. Gu, *et al.* [51] study flow cover inequalities and superadditive lifting functions in more detail.

Clique Inequalities

The constraints of many MILPs imply logical relationships between binary variable, such as $x_j = 1 \Rightarrow x_k = 0$, $x_j = 1 \Rightarrow \bar{x}_k = 0$, $\bar{x}_j = 1 \Rightarrow x_k = 0$ and $\bar{x}_j = 1 \Rightarrow \bar{x}_k = 0$, where x_j and x_k are variables, and $\bar{x}_j = 1 - x_j$ is the complement of x_j and $\bar{x}_k = 1 - x_k$ is the complement of x_k [98]. Such logical implications can be used to generate *clique inequalities*, which are composed of a set of binary variables.

Consider an integer program $min_{x\in\mathcal{P}\cap\mathbb{Z}_+} c^T x$, where and $\mathcal{P} = \{x \in \mathbb{R} \mid Ax = b,\ x \geq 0\},\ A \in \mathbb{Q}^{m\times n},\ b \in \mathbb{Q}^m, c \in \mathbb{Q}^n$. A clique inequality is an inequality of the form

$$\sum_{k\in C} x_k \leq 1, x_k \in \{0,1\} \;\; \forall x \in \mathcal{P}, \tag{3.21}$$

which states no more than one of the variables x_k and $k \in C$ can take value 1. The basic idea of finding set C is to first use the logical implications of the constraints defining $\mathcal{P}$ to construct a *conflict graph* $G = (B^o \cup B^c, E)$, where B^o is the set of original binary variables, B^c is the set of complemented binary variables, and E is the set of edges. In the conflict graph, there is an edge between two variables if they cannot both

have the value 1 at the same time. The logic implications used to construct the conflict graph may be present in the original formulation or can be derived from preprocessing [13, 98]. Every clique C of G defines a valid clique inequality. One way of using the conflict graph is to find all cliques during preprocessing and store them in a clique table, or find cliques dynamically during search. Atamtürk, *et al.* [13] performed a number of tests on how to generate and use clique inequalities.

3.1.4 Primal Heuristics

Besides improving the lower bound by adding valid inequalities, another important aspect of branch-and-cut algorithms is improving the upper bound by finding feasible solutions. A feasible solution will eventually be found during the search, as long as one exists, but finding a feasible solution in the early stages of the search sometimes has tremendous impact on search efficiencies because the search may be terminated early if the user does not require a provably optimal solution. More importantly, feasible solutions help to prune low quality nodes so that fewer nodes are required to be processed. Finally, some search techniques may require feasible solutions in order to be successfully employed. A typical example is *reduced cost tightening*, which uses the reduced costs of the LP to fix or tighten variables.

There are several ways to accelerate the process of finding good feasible solutions. For instance, we have already mentioned that employing depth-first search usually helps find feasible solutions quickly. Patel and Chinneck [79] propose a branching method that helps find the first feasible solution as quickly as possible. However, the most commonly used techniques are *primal heuristics*, which are procedures dedicated specifically to finding solutions quickly. There are a variety of existing primal heuristics, many of

which use LP solutions as input and try to find solutions by correcting the integer infeasibility. The well-known *rounding* [75], *diving* [60], *pivot and shift* [15], and *feasibility pump* [38] heuristics belong to this category. Others take feasible solutions as input and try to come up with better feasible solution. This category includes *local branching* [39] and *relaxation induced neighborhood search* (RINS) [28].

Some heuristics, like rounding, are computationally inexpensive, and can be called frequently during search, while others like diving and RINS, are relatively expensive. For the expensive heuristics, dedicated control schemes are required to balance the cost of spending time executing heuristics and the benefit of finding additional feasible solutions.

3.2 Knowledge Management

There are a number of types of knowledge that must be shared in branch and cut. Note that some knowledge types, such as bounds and pseudo-costs are not treated as abstract knowledge types because of their simple structure and sharing mechanisms.

3.2.1 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 lower bounds is important mainly for load balancing purposes and is shared according to the scheme described in Chapter 2. Distribution of information regarding upper bounds is mainly important for avoiding performance of redundant

work, i.e., processing of nodes whose lower bound is above the optimal value. Distribution of this information is relatively easy to handle because relatively few changes in upper bound typically occur throughout the algorithm. In our implementation, a workers periodically sends its lower bound information to its hub, which periodically sends the lower bound information of the cluster as a whole to the master. The upper bound is broadcast to other processes immediately once a process finds a feasible solution.

3.2.2 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. Linderoth [69] studies the methods and effect of sharing pseudocosts among processes. In his study, pseudocosts were shared in a buffered-distributed manner with buffer sizes of 1, 25, 100, and ∞. A buffer size of 1 means that new pseudocosts will be broadcast immediately; a buffer size of 25 means that new pseudocosts will be broadcast once the number of new pseudocosts reaches 25; and a buffer size of ∞ means that pseudocosts are not shared among the processes. His study showed there is no need to share pseudocost if the node processing time is short, and pseudocost should be shared for problems with long node processing time. Eckstein [32] also found that sharing pseudocost can be very important. We use a scheme similar to that in PARINO [69]. Pseudocosts are shared in a buffered-distributed manner. Users can adjust the frequency with which pseudocosts are shared by changing the buffer sizes. Also, users can choose to sharing pseudocost only during rampup.

3.2.3 Objects

An *Object* is an entity that has a domain. Typical objects include *variables* and *constraints*. Objects are treated as knowledge during the search. The main usage of objects is to describe tree nodes. A node description consists mainly of

- a list of variables that are in the formulation of subproblem; and
- a list of constraints that are in the formulation of the subproblem.

For data-intensive applications, the number of objects describing a tree node can be huge. However, the set of objects may not change much from a parent node to its child nodes. We can therefore store the description of an entire subtree very compactly using a *differencing* scheme. We discuss the details of the object handling and differencing schemes in section 3.4.1.

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.

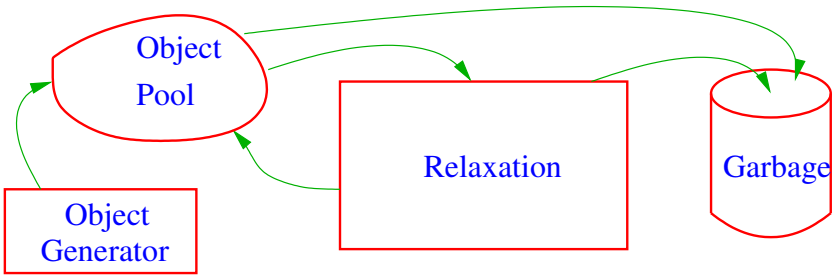

Figure 3.2: Objects Handling

3.3 Task Management

3.3.1 BiCePS Task Management

BiCePS assumes that applications employ an iterative bounding scheme. During each iteration, new objects might be generated to enhance the problem formulation. The newly generated objects are either stored in object pools or used to augment the current model. Since the number of objects can be huge, so duplicate and weak objects can be removed based on their hash keys and their effectiveness. It also possible that some objects are *dominated* by the others, which means we can safely remove these objects to save storage space. Furthermore, invalid and ineffective objects can be purged periodically. Figure 3.2 shows the object handling scheme of BiCePS.

3.3.2 BLIS Task Management

In BLIS, there are a number of distinct tasks to be performed and these tasks can be assigned to processors in a number of ways. Figure 3.3 is a simplified flowchart that shows how the main tasks are scheduled. In the rest of the section, we briefly discuss the tasks to be performed in BLIS.

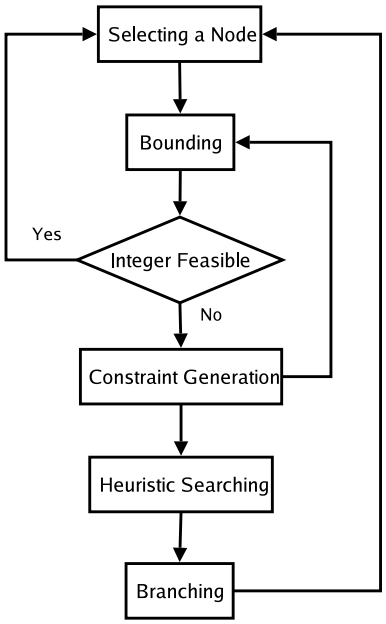

Figure 3.3: BLIS Tasks

Bounding. From the description of a candidate node, the bounding procedure produces either an improved lower bound or a feasible solution to the original MILP. The lower bound is obtained by solving the linear relaxation of the subproblem formulated at the candidate node.

Branching. From the description of a processed node, the branching procedure is used to select a method of branching and subsequently producing a set of children to be added to the candidate list.

Constraint 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).

Heuristic Searching. From a the given state, the heuristics are used to search for feasible solutions, which provide upper bounds to the original MILP.

3.4 The Class Hierarchies of BiCePS and BLIS

The Branch, Constrain, and Price Software (BiCePS) and the BiCePs Linear Integer Solver (BLIS) are the two libraries, in addition to ALPS, that comprise the CHiPPS hierarchy. Figure 3.4 shows the complete library hierarchy of CHiPPS. BiCePS implements the so-called *branch, constrain, and price* algorithm and is the data-handling layer of CHiPPS for relaxation-based descrete optimization. BiCePS is the data-handling layer for relaxation-based discrete optimization. It introduce the concept of objects and assumes an iterative bounding procedure. BLIS is a concretization of BiCePS and specific to models with *linear* constraints and *linear* objective function.

3.4.1 BiCePS

Figure 3.5 shows the class hierarchy of BiCePS. The class names with prefix `Alps` belongs to ALPS library, while the class names with prefix `Bcps` are the classes in BiCePS. BiCePS introduces a new type of knowledge called `BcpsObject` from which `BcpsConstraint` and `BcpsVariable` are derived. Also, BiCePS adds a new type of knowledge pool `BcpsObjectPool` from which `BcpsConstraintPool` and `BcpsVariablePool` are derived. BiCePS also derives subclasses `BcpsSolution`, `BcpsTreeNode`, `BcpsNodeDesc`, and `BcpsModel` from the base classes in ALPS.

Data-intensive applications are those in which the amount of information required to describe each node in the search tree is very large. An efficient storage scheme is

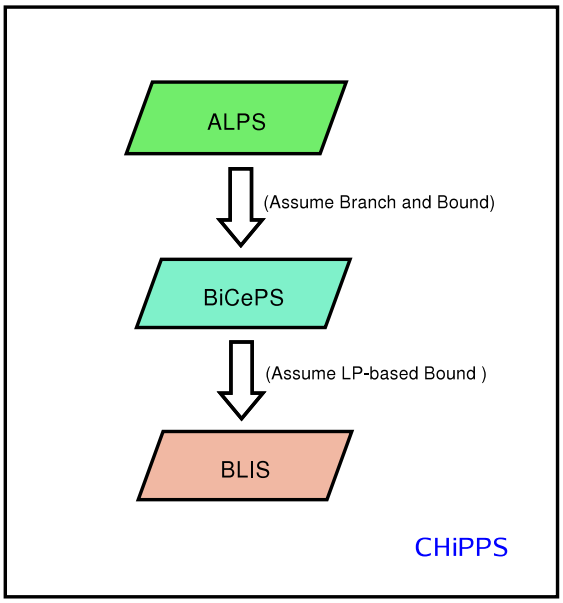

Figure 3.4: Library Hierarchy of CHiPPS

required to avoid memory issues and increased communication overhead. In the BiCePS library, we have developed compact data structures based on the ideas of *object*s and *differencing*.

BiCePS Objects

In BiCePS, we introduce the concept of objects, which are divided into two groups: *core* objects and *extra* objects. Core objects are active in all subproblems, which means that they are in the formulation of all subproblems. Extra objects can be added and removed. Since core variables are always in the problem formulation, no bookkeeping is required and communication overhead is small. However, problem size can become very large if a lot of objects are designated as core objects.

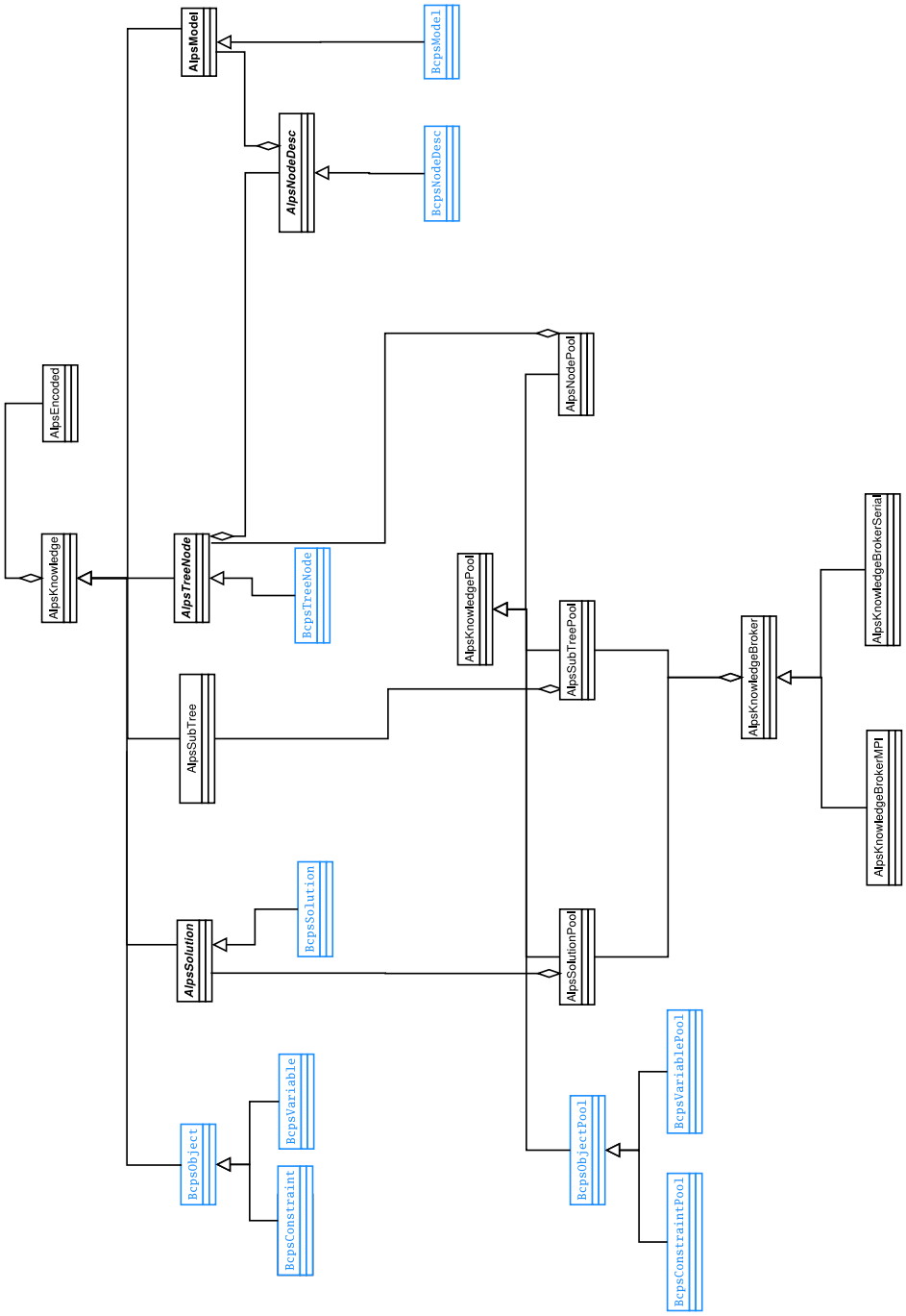

Figure 3.5: Class Hierarchy of BiCePS

The extra objects are also subdivided into two categories: *indexed* objects and *algorithmic* objects. A indexed object has a unique global user index assigned by the user. This index represents the object's position in a "virtual" global list known only to the user. For instance, in problem like the TSP, where the variable correspond to the edges of an underlying graph, the index could be obtained from a lexicographic ordering of the edges. The indexing scheme provides a compact object representation, as well as a simple way of moving in and out of the problem formulation. However, it requires the user have a priori knowledge of all objects and a method for indexing them. For some problems, like airline scheduling, the number of objects is not known in advance. In such cases, the user can use algorithmic objects, which require an algorithm to generate them on demand. Subtour elimination constraints for the TSP are typical examples of algorithmic objects since there are exponential number of them ($2^n - 2$ for an instance with n nodes) and we may not have enough indices for them.

The Differencing Scheme

The *differencing* scheme we use only stores the difference between the descriptions of a child node and its parent. In other words, given a node, we store a *relative* description that has the newly added objects, removed objects and changed objects compared with its parent. For the root of a subtree, the differencing scheme always stores an *explicit* description that has all the objects active at the root. Our differencing scheme for storing the search tree means that whenever we want to retrieve a node, we have to spend time constructing its explicit description. This is done by working back up the tree undoing the differencing until an explicit description of the node is obtained and exemplifies the trade-off of saving space and increasing search time. To prevent too much time being

```
template <class T> struct BcpsFieldListMod
{
    bool relative;
    int  numModify;
    int* posModify;
    T*   entries;
};
```

Figure 3.6: BiCePS field modification data structure.

spent in recovering node descriptions, the differencing scheme has the option to store an explicit description for any node that is a specified number of levels deeper than its first ancestor with an explicit description.

To store a relative description for a node, the differencing scheme identifies the relative difference of this node with its parent. The data structures used to record these differences are `BcpsFieldListMod` and `BcpsObjectListMod`.

`BcpsFieldListMod` stores a single field modification, such as the lower bounds of variables. Figure 3.6 lists the data members of `BcpsFieldListMod`. Member `relative` indicates how the modification is stored. If `relative` is `true`, then it means complete replacement; otherwise, it means it is relative to those of its parent node. Member `numModify` records the number of entries to be modified. Member `posModify` stores the positions to be modified. Member `entries` has the new values.

`BcpsObjectListMod` stores modifications of a particular type of objects. Member `mumRemove` records the number of the objects to be deleted. Member `posRemove` stores the positions of the objects to be deleted. Member `numAdd` records the number

```
struct BcpsObjectListMod
{
    int  numRemove;
    int* posRemove;
    int  numAdd;
    BcpsObject **objects;
    BcpsFieldListMod<double> lbHard;
    BcpsFieldListMod<double> ubHard;
    BcpsFieldListMod<double> lbSoft;
    BcpsFieldListMod<double> ubSoft;
};
```

Figure 3.7: BiCePS object modification data structure.

of the objects to be added. Member `objects` stores the objects to be added. Member `lbHard`, `ubHard`, `ubSoft` and `ubSoft` store the modification in the individual fields like lower bounds and upper bounds.

3.4.2 BLIS

The BLIS library is built on top of BiCePS. It provides the functionality needed to solve mixed integer linear programs. Figure 3.8 shows the class hierarchy of BLIS. The class names with prefix `Blis` belongs to BLIS library. BLIS derived following subclasses from their corresponding base classes in BiCePS:

- `BlisConstraint`,
- `BlisVariable`,
- `BlisSolution`,
- `BlisTreeNode`,

- `BlisNodeDesc`, and
- `BlisModel`.

Note that we only put the classes that show the inheritance relation with ALPS and BiCePS in the Figure 3.8. BLIS has a number of other classes, such as `BlisHeuristic` and `BlisConGenerator`.

Branching Scheme

The branching scheme of BLIS is similar to that of COIN/CBC [70]. It comprises three components:

- *branching Objects*: BiCePS objects that can be branched on, such as integer variables and SOS sets;
- *candidate branching objects*: objects that do not lie in the feasible region or objects that will be beneficial to the search if they are branched on; and
- *branching methods*: methods to compare objects and choose the best one.

Figure 3.9 shows the major steps of BLIS' branching scheme. The branching method is the core component of BLIS' branching scheme. When expanding a node, the branching method first chooses a set of objects as candidates, and then selects the *best* object based on the rules it specifies. `BcpsBranchStrategy` in BiCePS provides the base class for implementing various branching methods. `BcpsBranchStrategy` has the following member data:

- `BcpsBranchObject ** branchObjects_`: the set of candidate branching objects,

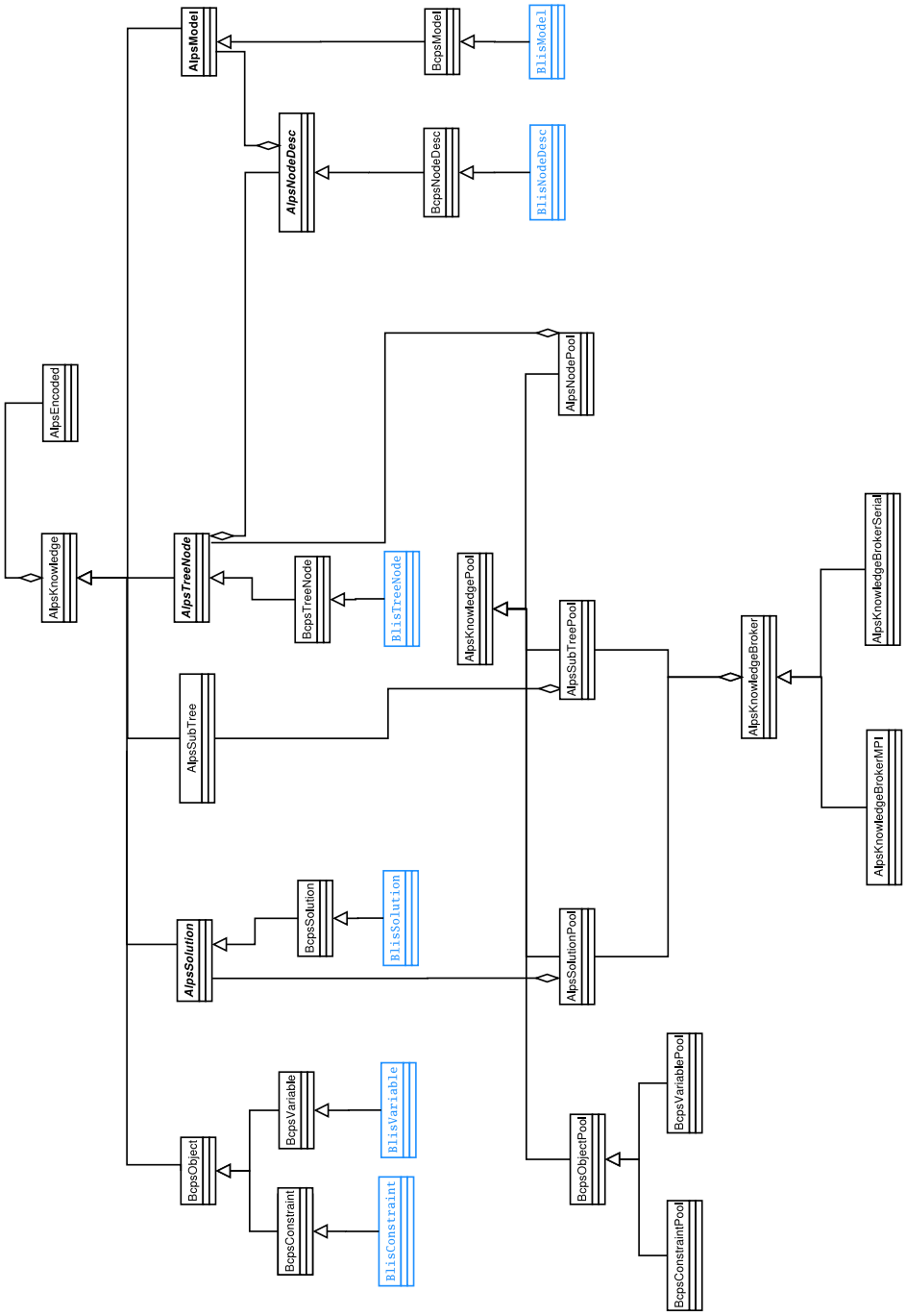

Figure 3.8: Class Hierarchy of BLIS

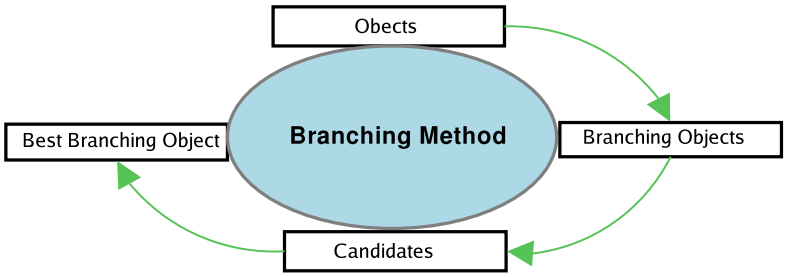

Figure 3.9: Branching Scheme

- `int numBranchObjects_`: the number of candidate branching objects, and
- `BcpsBranchObject * bestBranchObject_`: the best branching object found.

`BcpsBranchStrategy` also declares a number of virtual member functions:

- `virtual int createCandBranchObjects()`: creates a set of candidate branching objects,
- `virtual int betterBranchObject()`: compares two branching object and identify the better one, and
- `bestBranchObject()`: compares a set of branching objects and identify the best one.

By deriving subclasses from `BcpsBranchStrategy`, BLIS implements several branching methods include *strong branching*, *pseudocost branching*, *reliability branching* and *maximum infeasibility branching*, as described in Section 3.1.1. Users can develop customized branching method by deriving subclass from `BcpsBranchStrategy`.

Constraint Generator

A BLIS constraint generator provides an interface between BLIS and the algorithms in COIN/Cgl, which have all the generators to find the inequalities discussed in Section 3.1.3. A BLIS constraint generator has the ability to specify rules to control the generator:

- where to call the generator?
- how many constraints can the generator generate at most?
- when to activate or disable the generator?

It also contains the statistics to guide the cut generation strategy.

Class `BlisConGenerator` provide necessary member data and functions for implementing various constraint generators. The major member data includes the following.

- `CglCutGenerator * generator_`: The COIN/Cgl cut generator object if using a generator in COIN/Cgl.
- `BlisCutStrategy strategy_`: The strategy used to generate constraints. The strategy can be "only generating constraints at the root", "according to certain frequency", "automatically decided by the search", or "disabled".
- `int cutGenerationFrequency_`: The frequency of calls to the constraint generator.
- `std::string name_`: The name of the generator.

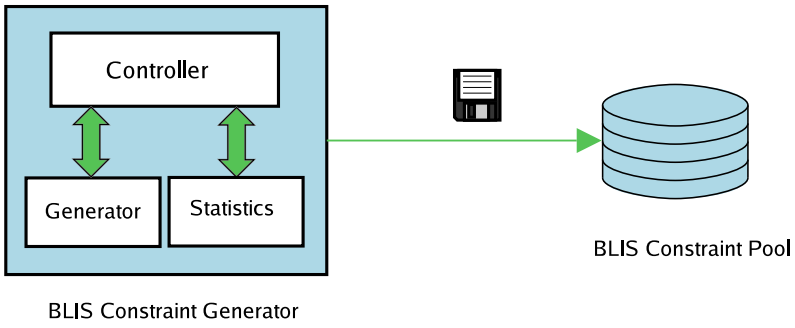

Figure 3.10: Constraint Generator

- `int numConsGenerated_`: The number of constraints generated by the generator.
- `double time_`: The time in seconds used by the generator.
- `int calls_`: The number of times the generator is called.

The key member function is `generateConstraints`, which evaluates the state of the search and generates constraints if necessary.

As Figure 3.10 shows, BLIS stores the generated constraints in constraint pools. Later, BLIS augments the current subproblem with some or all of the newly generated constraints. BLIS can use all the constraint generators in COIN/Cgl, a library of cut generation methods. Users can develop a COIN/Cgl constraint generator and then add into BLIS by using the interface provided by the BLIS constraint generator.

Primal Heuristics

The BLIS primal heuristic class declares the methods needed to search heuristically for feasible solutions. The class also provides the ability to specify rules to control the heuristic, such as

- where to call the heuristic?
- how often to call the heuristic?
- when to activate or disable the heuristic?

The methods in the class collect statistics to guide searching and provides a base class for deriving various heuristics. Currently, BLIS has only a simple round heuristics.

Class `BlisHeuristics` provide necessary member data and functions for implementing various heuristics. The major member data includes the following.

- `BlisHeurStrategy strategy_`: The strategy of using the heuristic. The strategy can be "only call the heuristic at the root", "according to certain frequency", "automatically decided by the search", or "disabled".
- `int heurCallFrequency_`: The frequency of calls to the heuristic.
- `std::string name_`: The name of the heuristic.
- `int numSolutions_`: The number of solutions found by the heuristic.
- `double time_`: The time in seconds used by the heuristic.
- `int calls_`: The number of times the heuristic is called.

The key member function is `searchSolution()`, which evaluates the state of the search and searches for solutions necessary.

3.5 Developing Applications

3.5.1 Procedure

In BLIS, users can derive problem-specific heuristics and constraint generators, and also define their own search strategy and branching rules. Although BLIS has the flexibility for users to control the search process, there are situations in which users might need to develop their own applications to solve problems that cannot be solved efficiently by a generic MILP solver. For instance, the formulation of some problems may have an exponential number of constraints. To develop applications based on BLIS, the user needs to follow two steps:

- derive problem-specific subclasses from BLIS' or ALPS' base classes, and
- write a `main()` function.

The base classes that the user might need to derive from include

- `BlisModel`,
- `BlisSolution`,
- `BlisConstraint`,
- `BlisConGenerator`,
- `BlisVariable`, and
- `AlpsParameterSet`.

The user-derived subclasses contain problem-specific data and functions that are needed to formulate and solve the problem. In the following sections, we first introduce the

background of the vehicle routing problem (VRP), then we discuss how to develop applications to solve VRPs.

3.5.2 Example: The Vehicle Routing Problem

The class of problems that we discuss here was introduced by Dantzig and Ramser [29]. In the Vehicle Routing Problem (VRP) formulation, a quantity $d_i \in \mathcal{Q}$ of a single commodity must be delivered to each customer $i \in N = \{1, \ldots, n\}$ from a central depot $\{0\}$ using k identical delivery vehicles of capacity C. The objective is to minimize total cost, with $c_{ij} \in \mathcal{Q}$ denoting the transit cost from i to j, for $0 \leq i, j \leq n$. We assume that c_{ij} does not depend on the quantity transported and that the cost structure is *symmetric*, i.e., $c_{ij} = c_{ji}$ and $c_{ii} = 0$. A solution for this problem consists of a partition of N into k routes $R_1, \ldots, R_k$, each satisfying $\sum_{j \in R_i} d_j \leq C$, and a corresponding permutation σ_i of each route specifying the service ordering.

We can associate a complete undirected graph with this problem. The graph consists of nodes $N \cup \{0\}$, edges E, and edge-traversal cost $c_{ij} \in E$. A solution is the union of k cycles whose only intersection is the depot node. Each cycle corresponds to the route serviced by one of the k vehicles. By associating a binary variable with each edge in the graph, we can formulate the VRP as the following integer program:

$$min \sum_{e \in E} c_e x_e$$

$$\sum_{e=\{0,j\} \in E} x_e = 2k, \tag{3.22}$$

$$\sum_{e=\{i,j\}\in E} x_e = 2 \ \ \forall i \in N, \tag{3.23}$$

$$\sum_{\substack{e=\{i,j\}\in E \\ i\in S, j\notin S}} x_e \geq 2b(S) \ \ \forall S \subset N, \ |S| > 1, \tag{3.24}$$

$$0 \leq x_e \leq 1 \ \ \forall e = \{i,j\} \in E, \ i,j \neq 0, \tag{3.25}$$

$$0 \leq x_e \leq 2 \ \ \forall e = \{i,j\} \in E, \tag{3.26}$$

$$x_e \in \mathbb{Z} \ \ \forall e \in E. \tag{3.27}$$

For ease of computation, we define $b(S) = |(\sum_{i\in S} d_i)/C|$, which is a lower bound on the number of trucks needed to service the customers in set S. Constraints (3.22) ensure that there are exactly k vehicles. Constraints (3.23) ensure that each customer is serviced by exactly on vehicle, as well as ensuring that the solution is the union of edge sets of routes. Constraints (3.24) ensure that every route includes the depot and that no route has total demand greater than the capacity C.

To solve the VRP problems described above, we designed and implemented a VRP solver. The core of the VRP solver is a branch-and-cut algorithm that has been studied in [85, 89, 86]. Following the steps to develop a BLIS application, we first derived four subclasses (`VrpModel`, `VrpSolution`, `VrpVariable`, and `VrpCutGenerator` from BLIS' base classes. Because we need special parameters to control the VRP solver, we also derived `VrpParams` from ALPS' base classes.

The Model Subclass

Subclass `VrpModel` is derived from `BlisModel`. Its member data includes:

- `int vertnum_`: the number of vertices in the graph,
- `int edgenum_`: the number of edges in the graph,
- `int capacity_`: specifies the capacity of a vehicle, and
- `int *demand_`: stores the demand of the customers.

`VrpModel` also has the member data that specifies the location of each customer. The important member functions of `KnapModel` are as follows.

- `readInstance()`: Reads an instance from a date file and creates variables and constraints.
- `userFeasibleSolution()`: Checks whether a VRP solution is feasible according to the user's criteria. A solution can only be accepted as a VRP solution if it is integral and associated subgraph is connected.

In our formulation, constraints (3.22) and (3.23) are designated as core constraints, while constraints (3.24) are designated as algorithm constraints because there are exponentially many of them. Constraints (3.25) and (3.26) set the bounds of the variables. Constraints (3.27) restrict all variable to be integral. We assume that there is an edge between every pair of nodes in the graph, so the number of edges is $n(n-1)/2$ edges. We associate a variable with each edge, and treat all variables as core variables. Therefore, the initial formulation of the VRP includes constraints (3.22), (3.23), (3.25), (3.26)

and (3.27), and all variables. In order to broadcast the model to all processes for parallel execution, `VrpModel` defines `encode()` and `decode()` functions to convert the model data into a `string` and re-construct a model from a string.

The Solution Subclass

As mentioned previously, a solution for the VRP consists of a partition of customers into several routes with each routes having a specific sequence of customers. The `VrpSolution` subclass uses a linked list to store the partition of customers and the visit sequences. It provides several functions to access and display the solution and defines `encode()` and `decode()` functions for parallel execution.

The Variable Subclass

Since a variable in the VRP corresponds to an edge in the graph, `VrpVariable` stores the endpoints of an edge by using data member `int ends_[2]` and assigns an index for each variable. `VrpVariable` defines functions to access and construct the associated edge for a variable. `VrpVariable` also overrides `encode()` and `decode()` functions.

The Constraint Generator Subclass

The cut generator in the VRP solver generates only constraint (3.24). A description of the methods used to generate these constraints is in [89]. The generated constraints are global valid, which means they can been added to any subproblems, and can be shared among processes. `VrpCutGenerator` defines `generateConstraints()`, which uses the methods described in [89] to generate constraints.

The Parameter Subclass

The VRP parameter subclass `VrpParams` is derived. from ALPS' base parameter class, The parameters first needs to be classified into the following types:

- `boolParams`: the bool parameters,
- `intParams`: the integer parameters,
- `dblParams`: the double parameters,
- `strParams`: the string parameters, and
- `strArrayParams`: the string array parameters.

`VrpParams` have 4 `boolParams` parameters and 7 intParams parameters. It does not have other types of parameters. `VrpParams` define several member functions to access and modify those parameters and overrides functions `encode()` and `decode()` for parallel execution.

The `main()` Function

A `main()` function for the VRP solver is shown in figure 3.11 and is based on the template provided by ALPS. The `main()` function is quite simple. The user first declares a LP solver that is going to be used. As Figure 3.11 shows, here COIN/Clp [70] is use as the LP solver. To run parallel code, the user needs to declare a parallel broker, such as `AlpsKnowledgeBrokerMPI`; to run serial code, the user needs to declare a `AlpsKnowledgeBrokerSerial` broker. Then, the broker searches for solutions and prints search results.

```
int main(int argc, char* argv[])
{
    // Set up lp solver
#ifdef  COIN_HAS_CLP
    OsiClpSolverInterface lpSolver;
    lpSolver.getModelPtr()->setDualBound(1.0e10);
    lpSolver.messageHandler()->setLogLevel(0);
#endif

    // Create VRP model
    VrpModel model;
    model.setSolver(&lpSolver);

#ifdef  COIN_HAS_MPI
    AlpsKnowledgeBrokerMPI broker(argc, argv, model);
#else
    AlpsKnowledgeBrokerSerial broker(argc, argv, model);
#endif

    // Search for best solution
    broker.search(&model);

    // Report the best solution found and its objective value
    broker.printBestSolution();

    return 0;
}
```

Figure 3.11: `Main()` Function of VRP

Chapter 4

Computational Results

We conducted a number of experiments to test the scalability of CHiPPS and the effectiveness of the methods that have been implemented to improve performance. In these experiments, we solved several sets of Knapsack, generic MILP, and VRP instances that were either generated by us or selected from public sources. Since KNAP is built on top of ALPS, the experiments with KNAP allowed us to look independently at aspects of our framework having to do with the general search strategy, load balancing mechanism, and the task management paradigm. Generic MILPs are what most practitioners are interested in solving, so how well the framework scales when solving generic MILPs is a key measure of its usefulness. The VRP is an important class of problem since it exemplifies typical properties of combinatorial optimization problems. Testing VRP instances provided us extra insights into the advantages and disadvantages of the framework. We used both the KNAP application and BLIS to solve the knapsack instances, BLIS to solve the MILPs, and the VRP application to solve the VRP instances.

4.1 Hardware

We used a cluster at Clemson University for initial and small-scale testing. This cluster has 52 nodes, each node with a single IBM Power5 PPC64 chip. Each chip has two cores that share 4 GB RAM, as well as L2 and L3 cache. Each core has its own L1 cache and can be hyper-threaded with two threads. The core speed is 1654 MHz. For large-scale testing, we primarily used the Blue Gene system at the San Diego Supercomputer Center (SDSC) [91]. The Blue Gene system has three racks with $3,072$ compute nodes ($6,144$ processors) and 384 I/O nodes. Each node consists of two PowerPC processors that run at 700 MHz and share 512 MB of memory. We also used a PC running Fedora 4 Linux to test the differencing scheme. The PC had a 2.8 GHz Pentium processor and 2 GB of RAM. Unless pointed out specifically, we limited each processor to one parallel process.

4.2 Test Suite

The knapsack instances were generated based on the method proposed by Martello and Toth [72]. Three sets of instances were used. Table 4.1 shows basic statistics for the ten *easy* instances when solving them sequentially. Column *Item* is the number of items available to be placed in the knapsack. Column *BestSol* is the path cost of the best solution found. Column *Node* is the number of nodes processed and column *NodeLeft* is the number of node left when solving them serially using KNAP with a time limit of 7200 seconds. We chose 7200 as the time cutoff, since KNAP can solve most knapsack instances in this amount of time. The last column *Time* shows the wallclock time in seconds used for each instance.

Table 4.1: Statistics of the Easy Knapsack Instances

Instance	Item	BestSol	Nodes	NodeLeft	Time
input55_1	55	1127	859210	0	16.58
input55_2	55	1151	466579	0	7.64
input55_3	55	1163	342777	0	4.27
input55_4	55	2145	704644	0	9.31
input55_5	55	2068	629662	0	8.19
input60_1	60	1192	479314	0	9.17
input60_2	60	1234	303637	0	4.29
input60_3	60	1222	738416	0	12.49
input60_4	60	1306	342432	0	5.58
input60_5	60	1419	607930	0	15.59

Table 4.2: Statistics of the Moderately Difficult Knapsack Instances

Instance	Item	BestSol	Nodes	NodeLeft	Time
input100_1	100	19387	19489172	0	3696.48
input100_2	100	15668	20630783	0	5144.68
input100_3	100	18024	7334000	346716	7200.00
input100_4	100	18073	6078869	0	428.24
input100_5	100	9367	4696000	3290611	7200.00
input75_1	75	6490	13858959	0	7004.81
input75_2	75	8271	13433001	0	6968.56
input75_3	75	8558	18260001	724141	7200.00
input75_4	75	8210	9852551	0	3006.69
input75_5	75	6315	10647047	0	1398.48

Table 4.3: Statistics of the Difficult Knapsack Instances

Instance	Item	BestSol	Nodes	NodeLeft	Time
input100_30	100	4334	450193400	0	158.94
input100_31	100	4479	295018523	0	117.01
input100_32	100	4378	374169758	0	136.46
input100_34	100	193071	355850972	0	130.14
input100_35	100	187491	381809948	0	137.68
input100_36	100	151934	684364180	0	255.75
input100_37	100	211231	440881122	0	169.57
input100_38	100	221009	399288703	0	141.76
input100_39	100	196402	366764346	0	134.63
input150_1	150	266157	1107247315	0	471.11
input150_2	150	258789	962872398	0	411.63
input150_4	150	335900	1414324008	0	766.87
input150_5	150	283645	334366142	0	144.43
input150_7	150	149820	243837878	0	106.38
input150_8	150	75449	946658511	0	456.24
input150_9	150	71315	688020337	0	298.13
input150_10	150	101254	520055323	0	213.49
input150_11	150	102443	652738452	0	275.91
input150_12	150	111715	477186428	0	191.24
input150_13	150	106253	766822612	0	366.28
input150_14	150	141827	394270918	0	173.48
input150_17	150	30261	348443929	0	135.35
input150_18	150	53463	453055347	0	188.22
input150_19	150	55784	314137323	0	131.19
input150_20	150	48727	269752458	0	111.28
input175_0	175	126364	1091614792	0	473.33

Table 4.2 shows basic statistics for the ten *moderately difficult* instances when solving them sequentially. The columns have the same meanings as those in Table 4.1. As Table 4.2 shows that serial KNAP could not solve instances *input100_3*, *input100_5*, or *input75_3* within the 7200 second time limit.

Table 4.3 shows the basic statistics for the 26 *difficult* instances. The columns *Instance*, *Item*, and *BestSol* have the same meaning as in Table 4.1. Column *Node* is the number of processed nodes and column *NodeLeft* is the number of nodes left when solving them in *parallel* using 64 processors on the SDSC Blue Gene system. Column *Time* lists the wallclock time used to solve these instance when using 64 processors. We used 64 processors to solve the difficult instance because the running time to solve them sequentially would have been too long.

To test the scalability when solving generic MILPs, we selected 18 MILP instances from Lehigh/CORAL, MIPLIB 3.0, MIPLIB 2003 [3], BCOL [12], and [81]. These instances took at least two minutes but no more than 2 hours to be solved by BLIS sequentially. Table 4.4 shows the problem statistics and the results of solving them sequentially. Column *Rows* and *Cols* list the number of rows and columns of each instance. Other columns have the same means as those in Table 4.1.

4.3 Overall Scalability

We first report results of solving knapsack instances with the simple knapsack solver KNAP that we discussed in Chapter 2. This solver uses a straightforward branch-and-bound algorithm with no cut generation. We tested KNAP using two different test sets, one composed of the moderately difficult instances shown in Table 4.2 and the other composed of the much more difficult instances shown in Table 4.3. Next, we report results of solving a set of generic MILPs with BLIS and discuss the cause of the increase in communication overhead. Finally, we report results of solving VRP instances with the VRP solver that we discussed in Chapter 3. Figure 4.1 shows several important

Table 4.4: Statistics of the Generic MILPs

Instance	Rows	Cols	Nodes	NodeLeft	Time
aflow30a	479	842	1003443	0	4346.61
bell5	91	104	644677	0	882.96
bienst1	576	505	14425	0	672.99
bienst2	576	505	179059	0	6233.11
blend2	274	353	259043	0	358.61
cls	180	181	98099	0	303.72
fiber	363	1298	10335	0	101.00
gesa3	1368	1152	106071	0	2904.33
markshare_4_1	4	30	3615047	0	762.42
markshare_4_3	4	30	2446893	0	790.62
mas76	12	151	986843	0	887.84
misc07	212	260	17899	0	111.36
pk1	45	86	885313	0	729.69
rout	291	556	1001147	0	7109.07
stein45	331	45	116677	0	208.63
swath1	884	6805	50125	0	1728.45
swath2	884	6805	196163	0	5252.10
vpm2	234	378	178697	0	437.02

defaults parameter setting of the framework. Unless explicitly specified otherwise, we used the default setting.

4.3.1 Solving Moderately Difficult Knapsack Instances

We tested the ten instances in the moderately difficult set by using 4, 8, 16, 32, and 64 processors on the Clemson cluster. The default algorithm was used to solve those instances, except that

- the static load balancing scheme was two-level root initialization,
- the number of nodes generated by the master during ramp up was 3000, and

1. The search strategy was the best-first strategy during the ramp up phase, and switches to the hybrid strategy after the ramp up phase.
2. The static load balancing scheme was spiral initialization.
3. Both the inter-cluster and intra-cluster dynamic load balancing are was used.

Figure 4.1: Default Setting

- the number of nodes designated as a unit work was 300.

Although these instances were difficult to solve when using sequential KNAP, they were all solved to optimality quite easily using just 4 processors. Table A.1 presents the detailed results of this experiment. As shown in Table 4.2, instances *input100_3*, *input100_5* and *input75_3* are unsolvable with sequential KNAP in 7200 seconds. However, it took KNAP 158.60 seconds, 87.50 seconds, and 56.29 seconds to solve them respectively when using 4 processors. When we increased the number processors used to 128, KNAP was able to solve them in 4.53 seconds, 2.81 seconds and 1.77 seconds, respectively.

Because the results of the ten instances show a similar pattern, we aggregated the results, which are shown in Table 4.5. The column headers have the following interpretations.

- *Nodes* 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 being done. Ideally, the total number of nodes explored does not grow as the number of processors is increased and may

actually decrease in some cases, due to earlier discovery of feasible solutions that enable more effective pruning.

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

- *Idle* is the average percentage of total wallclock running time each processor spent idle due to work depletion, i.e., waiting for more work to be provided.

- *Ramp-down* is the average percentage of total wallclock running time time each processor spent idle during the ramp-down phase.

- *Wallclock* is the total wallclock running time (in seconds) for solving the 10 knapsack instances.

- *Eff* is the parallel efficiency and is equal to the total wallclock running time for solving the 10 instances with p processors divided by the product of 4 and the total running time with four processors. Note that the efficiency is being measured here with respect the solution time on four processors, rather than one, because of the memory issues encountered in the single-processor runs.

The parallel efficiency is very high if computed in the standard way (see Equation (1.35)). For instance, the efficiency is 16.13 when solving *input100_1* on 64 processors. This abnormality is mainly caused by memory issues (size limitation and fragmentation) when solving knapsack instances serially. To properly measure the scalability of ALPS, we use the wallclock on 4 processors as the base for comparison. We therefore define

Table 4.5: Scalability for solving Moderately Difficult Knapsack Instances

P	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
4	193057493	0.28%	0.02%	0.01%	586.90	1.00
8	192831731	0.58%	0.08%	0.09%	245.42	1.20
16	192255612	1.20%	0.26%	0.37%	113.43	1.29
32	191967386	2.34%	0.71%	1.47%	56.39	1.30
64	190343944	4.37%	2.27%	5.49%	30.44	1.21

the efficiency as

$$Efficiency = \frac{Wallclock\ time\ using\ 4\ processors \times 4}{Wallclock\ time\ using\ P\ processors \times P}. \tag{4.1}$$

The last column in Table 4.5 lists the efficiency of KNAP. We found that the efficiencies were greater than 1.0 when using 8, 16, 32, and 64 processors. Efficiency should normally represent approximately the average fraction of time spent by processors doing "useful work," where the amount of useful work to be done, in this case, is measured by the running time with four processors. Efficiencies greater than one are not typically observed and when they are observed, they are usually the result of random fluctuations in the total number of nodes processed. Here we observe that the total number of nodes remained relatively constant, so the so-called "superlinear speedup" observed may again be due to minor memory issues encountered in the four processor runs. In any case, it can be observed that the overhead due to ramp-up, ramp-down, and idle time remains low as the number of processors is scaled up. Other sources of parallel overhead were not measured directly, but we can infer that these sources of overhead are most likely negligible.

4.3.2 Solving Difficult Knapsack Instances

We further tested the scalability of ALPS by using KNAP to solve the set of very difficult knapsack instances shown in Table 4.3. The tests were conducted on the SDSC Blue Gene system. The default algorithm was used to solve those instances except that

- the static load balancing scheme was the two-level root initialization,
- during ramp up, the number of nodes generated by the master varied from 10000 to 30000 depends on individual instance,
- during ramp up, the number of nodes generated by a hub varied from 10000 to 20000 depends on individual instance,
- the number of nodes specified as a unit work was 300, and
- multiple hubs were used (the actual number depended on the number of processors).

Table A.2 shows the detailed testing results. Table 4.6 summarizes the aggregated results. For this test, we defined the efficiency as

$$Efficiency = \frac{Wallclock\ time\ using\ 64\ processors \times 64}{Wallclock\ time\ using\ P\ processors \times P}. \tag{4.2}$$

The experiment shows that KNAP scales well, even when using several thousand processors. This is primarily because the node evaluation time is generally short and it is easy to generate a large number of nodes quickly during ramp-up. For these instance, the workload is already quite well balanced at the beginning of the search. As the number of processors is increased to 2048, we observe that the wallclock running time of the algorithm becomes shorter and shorter, while ramp-up and ramp-down overhead inevitably

Table 4.6: Scalability for solving Difficult Knapsack Instances

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

increase as percentages of the overall running time. This effect is known as Amdahl's Law [6], which predicts a limit to the efficiency that can be achieved by increasing the number of processors used to solve a given fixed problem instance due to the inherently sequential parts of the algorithm. Ideally, as the number of processors increases, we should also be scaling the size of the instances to estimate the iso-efficiency function of Kumar and Rao [65], which might be a more effective measure of scalability than the standard measures presented here.

Both scalability experiments show that ALPS scales well when solving knapsack problem instances. This provides support for the ideas that we are using to improve scalability. During the experiments, we found that node evaluation time were generally short and a large number of nodes could be generated in a short time during ramp-up, so that the workload was already quite balanced at the beginning of the search. These special characteristics of the knapsack problem help KNAP to obtain good parallel efficiency. In Chapter 3, we discuss ideas for achieving speedup when node evaluation times are long and the number of nodes required to solve a problem is small. We discuss computational results in this setting next.

Table 4.7: Scalability for generic MILP

Instance	Nodes	Ramp -up	Idle	Ramp -down	Comm Overhead	Wallclock	Eff
1 P	11809956	–	–	–	–	33820.53	1.00
Per Node		–	–	–	–	0.00286	
4P	11069710	0.03%	4.62%	0.02%	16.33%	10698.69	0.79
Per Node		0.00%	4.66%	0.00%	16.34%	0.00386	
8P	11547210	0.11%	4.53%	0.41%	16.95%	5428.47	0.78
Per Node		0.00%	4.52%	0.53%	16.95%	0.00376	
16P	12082266	0.33%	5.61%	1.60%	17.46%	2803.84	0.75
Per Node		0.27%	5.66%	1.62%	17.45%	0.00371	
32P	12411902	1.15%	8.69%	2.95%	21.21%	1591.22	0.66
Per Node		1.22%	8.78%	2.93%	21.07%	0.00410	
64P	14616292	1.33%	11.40%	6.70%	34.57%	1155.31	0.46
Per Node		1.38%	11.46%	6.72%	34.44%	0.00506	

4.3.3 Solving Generic MILP Instances

In the experiment, we test the scalability of BLIS by solving the MILPs shown in Table 4.4. We used four cut generators from the COIN-OR Cut Generation Library [70]:

- `CglGomory`,
- `CglKnapsackCover`,
- `CglFlowCover`, and
- `CglMixedIntegerRounding`.

Pseudocost branching was used to select branching objects. We used two hubs for the runs with 64 processors. For all other runs, one hub was used.

Each instance was solved twice: once with two-level root initialization and once with spiral initialization. Table 4.7 shows the results of these computations. Note that in this

table, we have added an extra column to capture sources of overhead other than idle time, i.e., communication overhead, since these become significant here. However, the effect of these sources of overhead is estimated as the difference between the efficiency and the fraction of time attributed to the other three sources. The accuracy of this estimate is difficult to determine. Note also that in these experiments, the master process does not do any exploration of subtrees, so an efficiency of $(p-1)/p$ is the best that can be expected, barring anomalous behavior. To solve these instance, BLIS needs to process a relatively large number of nodes and the node processing time is not too long. These two properties tend to lead to good scalability and the results reflect this to a large extent. However, as expected, percentage overhead increased across the board as the number of processors increased. For this test set, it looks as though BLIS would not scale well beyond 64 processors.

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

Table 4.8 summarizes the results of load balancing and subtree sharing when solving the 18 instances that were used in the scalability experiment. The column labeled *P* is the number of processors. The column labeled *Inter* is the number of inter-cluster balancing

Table 4.8: Load Balancing and Subtrees Sharing

P	Inter	Intra	Starved	Subtree	Split	Whole
4	0	87083	22126	42098	28017	14081
8	0	37478	25636	41456	31017	10439
16	0	15233	38115	55167	44941	10226
32	0	7318	44782	59573	50495	9078
64	494	3679	54719	69451	60239	9212

operations performed (this is zero when only one hub is used). The column labeled *Intra* is the number of intra-cluster balancing operations performed. The column labeled *Starved* is the number of times that workers reported being out of work and proactively requested more. The column labeled *Subtree* is the number of subtrees shared. The column labeled *Split* is the number of subtrees that were too large to be packed into a single message buffer and had to be split when sharing. The column labeled *Whole* is the number of subtrees that did not need to be split. As Table 4.8 shows, the total number of inter- and intra-cluster load balancing goes down when the number of processors increases. However, worker starvation increases. Additionally, the number of subtrees needing to be split into multiple message units increases. These combined effects seem to be the cause of the increase in communication overhead.

In order to put these results in context, we have attempted to analyze how they compare to results reported in paper in the literature, of which there are few. Eckstein *et al.* tested the scalability of PICO [33] by solving 6 MILP instances from MIPLIB 3.0. For these tests, PICO was used as a pure branch and bound code and did not have any constraint generation, so it is difficult to do direct comparison. One would expect that the lack of cut generation would increase the size of the search tree while decreasing the node processing time, thereby improving scalability. Their experiment showed an

efficiency of 0.73 for 4 processors, 0.83 for 8 processors, 0.69 for 16 processors, 0.65 for 32 processors, 0.46 for 64 processors, and 0.25 for 128 processors. Ralphs [87] reported that the efficiency of SYMPHONY 5.1 is 0.81 for 5 processors, 0.89 for 9 processors, 0.88 for 17 processors, and 0.73 for 33 processors when solving 22 MILP instances from Lehigh/CORAL, MIPLIB 3.0, and MIPLIB 2003. All in all, our results are similar to those of both PICO and SYMPHONY. However, since we are using a more sophisticated algorithm that is more difficult to implement scalably, this can be seen as an improvement. Also, SYMPHONY was not tested beyond 32 processors and it is unlikely that it would have scaled well beyond that. In section 4.4, we show this by comparing the performance of BLIS and SYMPHONY directly when solving knapsack problem.

4.3.4 Solving VRP Instances

In this experiment, we wanted to assess how the framework scales when solving VRP instances. We performed this experiment on the Clemson Cluster. The default setting was used except that the search strategy was best-first and the branching method was strong branching. Table 4.9 shows the aggregated results of solving the 16 VRP instances by using 1, 4, 8, 16, 32, and 64 processors. When using 4 processors, the VRP solver achieved an efficiency of 0.90. However, as the number of processors increased, the number of nodes increased significantly and the efficiency dropped dramatically. The overhead also increased as the number of processors increased. With increased number of processors, we found that feasible solutions were found very late in the search process for many instances. Hence, a lot redundant work was performed. Overall, the VRP solver does not seem to scale well for this set of VRP instances.

Table 4.9: Scalability for VRP

P	Nodes	Ramp-up	Idle	Ramp-down	Wallclock	Eff
1	40250	–	–	–	19543.46	1.00
4	36200	7.06%	7.96%	0.39%	5402.95	0.90
8	52709	9.88%	6.15%	1.29%	4389.62	0.56
16	70865	14.16%	8.81%	3.76%	3332.52	0.37
32	96160	15.85%	10.75%	16.91%	3092.20	0.20
64	163545	18.19%	10.65%	19.02%	2767.83	0.11

4.4 Comparison with SYMPHONY

It is not straightforward to compare the performance of two parallel solvers, in part because of the fact that it is difficult to find a set of test instances appropriate for testing both solvers simultaneously. SYMPHONY implements a highly customizable branch-and-bound algorithm and can be used to solve the knapsack problem. Ralphs [87] reported that SYMPHONY is very effective for small number of processors, but is not scalable beyond about 32 processors. The test set that we used is the *easy* instance set described in Table 4.1. We performed the experiment on the Clemson cluster.

First, we compared KNAP with SYMPHONY. To make SYMPHONY execute more like a simple branch and bound code, we turned off *reduced cost fixing* and *cut generation*, set the number of candidates for *strong branching* to one and set the maximum number of simplex iteration when solving LPs to one during strong branching. Table 4.10 shows the aggregated results using KNAP and SYMPHONY on using 4, 8, 16, and 32 processors. Column *P* is the number of processors. Column *Solver* is the solver used. The other columns have the same meaning as those in Table 4.7. Even for these easy instances, KNAP showed quite good efficiency. It achieved super-linear speedup when using 4, 8 and 16 processors. The efficiency of SYMPHONY was not good for these

Table 4.10: Scalability of KNAP and SYMPHONY for the Easy Instances

P	Solver	Node	Ramp-up	Idle	Ramp-down	Wallclock	Eff
1	KNAP	5474601	–	–	–	93.11	1.00
	SYM	5467094	–	–	–	2835.72	1.00
4	KNAP	5471538	6.67%	2.40%	1.20%	16.67	1.40
	SYM	5060032	0.01%	32.73%	0.24%	1706.58	0.42
8	KNAP	5482627	12.43%	0.52%	0.65%	7.64	1.52
	SYM	5060364	0.01%	48.41%	2.05%	1118.48	0.32
16	KNAP	5496753	21.50%	0.74%	2.42%	4.14	1.40
	SYM	5070008	0.02%	68.75%	4.09%	913.07	0.19

instance. When using 4 processors, the efficiency of SYMPHONY was 0.42. When using 16 processors, the efficiency of SYMPHONY went down to 0.19. The numbers of nodes for these instances are around half a million and node evaluation time was quite small. The number of nodes processed by KNAP was roughly the same as that of SYMPHONY for every instance. For SYMPHONY, the master was overwhelmed by the requests of nodes and workers spends long time waiting for instructions.

Next, we compared BLIS and SYMPHONY for solving the same knapsack instances. The default settings of BLIS and SYMPHONY were used. The aggregated results are summarized in Table 4.11. It can be seen from these results that BLIS scales reasonably well and the overhead is relatively small. However, SYMPHONY has trouble scaling as the number processors increase, since it suffers from the synchronization bottleneck that happens when many workers are being controlled by a single master. In SYMPHONY, the workers are not as autonomous as in BLIS and must check in with the master much more frequently. The effect of this is seen in the results here, as the efficiency is already extremely low with just 16 processors. BLIS also begins to show efficiency degradation when using 16 processors, but this is mainly due to the fact that

Table 4.11: Scalability of BLIS and SYMPHONY for Knapsack Instances

P	Solver	Nodes	Ramp-up	Idle	Ramp-down	Wallclock	Eff
1	BLIS	450644	–	–	–	124.67	1.00
	SYM	5465932	–	–	–	3205.77	1.00
4	BLIS	435990	0.08%	3.16%	2.46%	38.63	0.80
	SYM	4935156	0.01%	28.40%	0.12%	1659.55	0.48
8	BLIS	347126	0.30%	4.74%	6.37%	16.65	0.94
	SYM	4944232	0.49%	42.92%	1.73%	1070.03	0.37
16	BLIS	369586	0.77%	8.00%	18.32%	10.37	0.75
	SYM	4957982	0.16%	62.77%	3.78%	847.43	0.23

these problems are too easy and Amdahl's law is coming into play, as discussed earlier. As a side note, BLIS appears to be much more effective on these instances than SYMPHONY, as the number of nodes generated is an order of magnitude different. We suspect this is due to SYMPHONY's less effective cut generation strategy.

4.5 The Effect of the Master-Hub-Worker Paradigm

An important aspect of the scheme for scalability improvement in ALPS is the implementation of the master-hub-worker paradigm. By inserting the hubs in between the master and the workers, we hoped that parallel overhead would be reduced. In this test, we examine whether the use of hubs does actually help to improve performance. The test was conducted on the SDSC Blue Gene system. We used the moderately difficult knapsack instance set and ran KNAP on 1024 processors. Table 4.12 shows aggregated results using 1, 2, 4, 8, 16, 32, 64 and 128 hubs. The column labeled *Hubs* is the number of hubs used. Column *ClusterSize* is the number of processors in each cluster. From the table, we can see that the use of multiple hubs does improve efficiency as long as

Table 4.12: Effect of Master-hub-worker Paradigm

Hubs	Cluster Size	Node	Ramp-up	Idle	Ramp-down	Wallclock
1	1024	2189900404	14.78%	2.25%	38.17%	103.79
2	512	2181608684	16.43%	4.04%	29.64%	91.74
4	256	2671387391	18.76%	3.17%	18.13%	91.10
8	128	2138187406	20.45%	2.57%	17.27%	74.57
16	64	2132342280	22.38%	1.71%	11.99%	70.14
32	32	2132560921	22.61%	1.04%	11.17%	71.44
64	16	2136262257	20.38%	0.75%	13.78%	82.13
128	8	2150789246	14.69%	0.85%	14.12%	118.31

the cluster size is not too small. In particular, we see that the hubs have a difficult time balancing the workload of workers at the end of the search if the number of hubs used is small and the master has a difficult time balancing the workload among the clusters if the number of hubs is large. For this test, a cluster size of 32 or 64 seems to be optimal.

4.6 Choice of Static Load Balancing Scheme

In this experiment, we compared the two static load balancing schemes implemented in ALPS. The experiment was performed on the Clemson cluster and used 64 processors. One hub was used. Table B.2 shows the detailed results of the 19 instances. Column *Scheme* is the static load balancing scheme used. Other columns have the same meaning as those in Table 4.7. The spiral scheme was better for 8 instances, and the two-level root initialization scheme was better for 4 instances. The two schemes were approximately tied for 7 instances (the running time difference is less than 10%). Table 4.13 shows the total number of nodes, etc. of these instances. The two schemes are not very

Table 4.13: Static Load Balancing Comparison

Scheme	Node	Ramp-up	Idle	Ramp-down	Wallclock
Root	15876829	55.24%	4.31%	11.42%	3690.30
Spiral	15849764	1.74%	10.75%	18.21%	1419.37

different with regarding to the number of nodes processed. However, the root initialization scheme has almost 100 times more ramp-up overhead than the spiral scheme does. For instances `markshare_4_1` and `markshare_4_3`, root initialization had difficulty in generating enough nodes for workers and almost all the search time spent in ramp-up phase.

From this experiment, we found that root initialization was better for instances with short node processing time and large search trees, while spiral initialization is better for instances with long node processing time and small search trees. For problems like the knapsack problem, root initialization distributes work more evenly than spiral initialization. Overall, the performance of spiral initialization is more reliable than root initialization. For instances with long node processing time, the ramp-up overhead of spiral initialization is generally much smaller than that of root initialization.

4.7 The Effect of Dynamic Load Balancing Schemes

Because ALPS was a decentralized implementation, dynamically balancing the workload would seem to be absolutely necessary. However, we still performed an experiment to verify this. We performed the test on the Clemson cluster and used 64 processors of which two processors were designated as hubs. Table B.3 shows the results of searching with and without dynamic load balancing enabled. Column *Balance* indicates whether

Table 4.14: The Effect of Load Balancing

Balance	Node	Ramp-up	Idle	Ramp-down	Wallclock
No	17900506	8.69%	0.00%	79.67%	9718.67
Yes	16057620	32.50%	6.81%	22.32%	2589.43

Table 4.15: Effect of inter-cluster Load Balancing

Hub	Inter	Node	Ramp-up	Idle	Ramp-down	Wallclock
16	Yes	2132342280	22.38%	1.71%	11.99%	70.14
	No	2131879616	22.52%	1.73%	11.57%	69.85
32	Yes	2132560921	22.61%	1.04%	11.17%	71.44
	No	2131980156	22.43%	0.90%	12.07%	72.01
64	Yes	2136262257	20.38%	0.75%	13.78%	82.13
	No	2136236456	20.35%	0.40%	14.33%	82.34
128	Yes	2150789246	14.69%	0.85%	14.12%	118.31
	No	2149541063	14.62%	0.17%	15.33%	119.01

dynamic load balancing schemes were used or not. Other columns have the same meaning as those in Table 4.7. Not surprising, using the dynamic load balancing scheme helped reduce overhead and reduce solution time for all instances. Table 4.14 shows the aggregated results of the 18 instances. When dynamic load balancing was disabled, the total number of nodes increased about 12.5%, and the running time increased almost three times. These results show that dynamic load balancing is important to the performance of parallel branch and cut. Note that there is no idle time without dynamic load balancing since the overhead is alll counted as ramp-down time.

Table 4.16: Effect of Intra-cluster Load Balancing

P	Intra	Node	Ramp-up	Idle	Ramp-down	Wallclock
64	Yes	190343944	4.37%	2.27%	5.49%	30.44
	No	190344811	3.38%	0.00%	28.06%	39.35
128	Yes	190885657	7.84%	3.97%	10.14%	19.14
	No	190932329	6.38%	0.00%	30.59%	23.18

4.7.1 Inter-cluster Dynamic Load Balancing

In this experiment, we wanted to assess the effectiveness of the inter-cluster load balancing mechanism. The test was conducted on the SDSC Blue Gene system. We solved the ten moderately difficult knapsack instances with KNAP using 1024 processors. We compared the results using inter-cluster dynamic load balancing with those when not using it. Table 4.15 presents the aggregated results of the ten instances. The column labeled *Hubs* indicates how many hubs were used. A *Yes* in the second column means the inter-cluster dynamic load balancing was turned on, and a *No* means it was turned off. The results show that for this test set, parallel efficiency does not change much when inter-cluster dynamic load balancing is turned on. As we mentioned above, we suspect that this is because the static load balancing scheme works well for these relatively well behaved instances. However, we do not believe this result will carry over into other domains.

4.7.2 Intra-cluster Dynamic Load Balance

We tested the effect of the intra-cluster load balancing by comparing the results obtained with and without the dynamic load balancing schemes. For this experiment, we used *one* hub and solved the ten moderately difficult knapsack instances with KNAP on 64

and 128 processors. The experiment was conducted on the Clemson cluster. Table 4.16 presents the aggregated results of the ten instances. A *Yes* in the second column indicates that the intra-cluster dynamic load balancing was turned on, and a *No* means it was turned off. The results demonstrate that the load balancing scheme is very important in achieving scalability. Without dynamic load balancing, the wallclock running time increases substantially. Furthermore, knapsack instances tend to be very well behaved with respect to the effect of the initial static load balancing. This indicates that the effect might be even more pronounced for other types of problems.

4.8 The Impact of Dedicated Hubs

In our design, hubs can also process nodes in addition to managing a cluster of workers. In this section, we answer the question "Should a hub also explore subtrees?". For this test, we used the 10 instances in the moderately difficult set. The experiment was conducted on the Clemson cluster and used 64 processors. Table 4.17 shows the aggregated results. Column *ClusterSize* is the number of processors in each cluster. Column *Work* indicates whether hubs also functioned as workers or not. Other columns have the same meaning as those in Table 4.12. We found that the idle time and ramp-down time increased significantly when hubs also functioned as workers. The reason is that hubs are only able to process messages after completing a unit of work when they also explore subtrees. Thus, other processors that need to communicate with the hubs have to wait. We can change the size of a unit of work for the hubs so that they can handle messages in a more timely fashion. However, it would require extensive performance tuning to find a suitable size. Therefore, we suggest that hubs should be dedicated to handle messages

Table 4.17: Effect of Hubs Work

ClusterSize	Work	Node	Ramp-up	Idle	Ramp-down	Wallclock
16	Yes	2296738608	0.45%	17.05%	47.49%	1015.67
	No	2294060630	1.23%	0.87%	1.89%	373.94
8	Yes	2322525423	0.29%	32.67%	45.12%	1679.84
	No	2370103928	1.12%	2.11%	2.27%	427.51
4	Yes	2163174835	0.22%	52.51%	32.51%	2371.59
	No	2165083195	1.11%	3.09%	3.38%	481.24

and not to explore subtrees when there are a large number of processors available.

4.9 The Impact of Problem Properties

As we have mentioned several times, the properties of individual instances can significantly effect scalability, independent of the implementation of the solver itself. Table 4.18 shows the detailed results of solving three specific MILP instances with BLIS. The test was performed on SDSC Blue Gene. We used the time for the 64-processor run as the baseline for measuring efficiency.

Instance `input150_1` is a knapsack instance. When using 128 processors, BLIS achieved superlinear speedup mainly due to the decrease in tree size. BLIS showed good parallel efficiency as the number processors increased to 256 and 512. Instance `fc_30_50_2` is a fixed-charge network flow instance. It exhibited very significant increases in the size of its search tree (indicating the performance of redundant work) as the number of processors increased, resulting in decreased efficiency. It was found that the optimality gap of instance`fc_30_50_2` improved very slowly during search, which

caused a large number of node to be processed. Instance `pk1` is a small integer program with 86 variables and 45 constraints. It is relatively easy to solve. Although the efficiency is reasonable good when using 128 processors, it is eventually wiped out by significant increased in ramp-up and ramp-down overhead as the number of processors increased.

The results in Table 4.18 showed that properties of each particular instance can have tremendous impact on scalability. For instances with large tree size and short node processing time, it is not difficult to achieve good speedup. For instances that are easy to solve or for which the upper bounds are hard to improve, it is not easy to achieve good speedup. When there are more processors available, we should use them solve more difficult instances if we want to achieve good speedup. Parameter turning may help to improve scalability. However, good scalability is difficult, if not impossible, to achieve without suitable instances, since limiting one type of overhead may increase another type of overhead.

4.10 The Effect of Sharing Pseudocosts

As discussed in Section 3.1.1, the pseudocost branching method uses pseudocosts to choose branching objects. Linderoth [69] studied methods for and effect of sharing pseudocosts. His study showed there is no need to share pseudocost if the node processing time is short and that pseudocosts should be shared for problems with long node processing time. Eckstein [32] also found sharing pseudocosts can be important. During the search, ALPS periodically checks to see if there are any of pseduocosts to be shared. If process k has pseudocosts to share, ALPS forms all processors into a binary tree with

Table 4.18: Problem Properties and Scalability

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

process k as the root. Process k then sends the pseudocosts to its two children, and its children send the pseudocosts to its children's children, and so on.

We performed a test to find out if there is any benefit of sharing pseudocost during the ramp-up phase. The test was conducted on the Clemson cluster and 64 processors were used. Table 4.19 also has the aggregated results. In column *Share*, *No* means no pseudocost were shared, while *Yes* means pseudocost were shared. Here we only shared pseudocost during ramp-up phase, since branching decisions are more important at the top of the tree. In term of total tree sizes and running time, sharing pseudocost is slightly better than not sharing. However, the difference is not significant. Table B.4 has the detailed testing results. For some instances, such as `aflow30a` and `swath2`, sharing pseudocosts helped improve performance, but for instances like `rout`, not sharing is better.

Table 4.19: The Effect of Sharing Pseudocost

Share	Node	Ramp-up	Idle	Ramp-down	Wallclock
No	16839373	0.94%	11.16%	18.47%	1431.04
Yes	15865523	0.96%	10.87%	17.69%	1394.76

4.11 The Impact of Hardware on Performance

In this experiment, we studied the impact of hardware on parallel overhead and load balancing. We selected 8 relatively easy instances from the generic MILP test set described in section 4.3.3, and solved them with BLIS by using 64 processors on the Clemson Cluster and the SDSC Blue Gene system. One hub was used for this experiment.

Table 4.20 shows the results of solving these instances on the two machines. Because the Clemon cluster has faster processors and more memory than the Blue Gene system, it was not surprising to find that solving the instances on the Blue Gene system took longer than on the Clemson cluster. However, it was surprising to find that overall overhead when solving on the Blue Gene system was much larger than that when using the Clemson cluster. Although ramp-up overhead is small, idle and ramp-down overhead become a severe issue when solving these instances on the Blue Gene system.

We suspect that the reason for the loss of performance for the Blue Gene system run is due to a lack of effectiveness of the dynamic load balancing. We collected statistics on dynamic load balancing (see Table 4.21). Column *Intra* is the number of intra-cluster balancing completed. Column *Asked* is the number of requests that workers made. Column *Subtree* is the number of subtrees shared. Column *Split* is the number of subtrees split, and Column *Whole* is the number of subtrees shared without splitting. The number of subtrees shared is equal to the sum of the number of subtrees split and the number of

Table 4.20: The Hardware Impact on Overhead

Instance	Machine	Nodes	Ramp-up	Idle	Ramp-down	Wallclock
bell5	Clemson	188717	6.66%	11.81%	12.48%	6.01
	BlueGene	227133	0.75%	25.85%	50.34%	60.97
bienst1	Clemson	44527	1.66%	5.87%	11.09%	55.37
	BlueGene	52687	3.79%	23.18%	23.71%	443.95
cls	Clemson	382121	7.10%	12.33%	11.66%	38.59
	BlueGene	638267	1.03%	50.31%	8.25%	439.9
gesa3	Clemson	56553	11.91%	9.95%	29.89%	20.91
	BlueGene	118513	0.48%	24.33%	63.98%	856.22
mas76	Clemson	1758445	0.94%	5.61%	7.73%	25.48
	BlueGene	2074333	0.34%	38.36%	5.11%	123.27
misc07	Clemson	14337	10.07%	12.03%	27.09%	11.22
	BlueGene	11743	1.99%	10.81%	56.94%	64.4
pk1	Clemson	1559947	0.70%	8.79%	9.30%	21.28
	BlueGene	3101103	0.32%	42.76%	19.47%	282.53
vpm2	Clemson	443899	6.74%	9.12%	7.30%	24.79
	BlueGene	895465	1.24%	38.06%	6.07%	195.65
Total	Clemson	4448546	4.78%	8.70%	12.98%	203.64
	BlueGene	7119244	1.26%	32.34%	33.64%	2466.89

whole subtree. Note that ALPS will do intra-cluster balancing only after the previous intra-cluster balancing has been completed. Workers will ask for a new subtree only if the previous request has been completed.

As Table 4.21 shows, ALPS was able to complete significantly more intra-cluster load balancing for 7 out of 8 instances when running on the Clemson cluster. The number of subtrees shared when running on the Clemson cluster is 3 times larger than that when running on the Blue Gene system. Considering the fact that total wallclock time when running on the Clemson cluster was less than 10% of that on the Blue Gene system, we can conclude that our suspicion that the performance loss was caused by ineffective load balancing makes sense. This raises another question as to why dynamic

Table 4.21: The Hardware Impact on Dynamic Load Balancing

Instance	Machine	Intra	Asked	Subtree	Split	Whole
bell5	Clemson	58	1436	1729	1523	206
	BlueGene	8	596	570	544	26
bienst1	Clemson	96	902	1324	966	358
	BlueGene	15	484	470	453	17
cls	Clemson	367	4152	4366	4252	114
	BlueGene	1839	520	576	552	24
gesa3	Clemson	278	2472	2484	2449	35
	BlueGene	9	719	763	748	15
mas76	Clemson	340	5211	6289	6030	259
	BlueGene	198	1218	1323	1264	59
misc07	Clemson	28	355	352	304	48
	BlueGene	9	289	236	227	9
pk1	Clemson	154	2613	3966	3123	843
	BlueGene	12	1940	2008	1945	63
vpm2	Clemson	186	4590	5289	5013	276
	BlueGene	13	1238	1264	1235	29
Total	Clemson	1507	21731	25799	23660	2139
	BlueGene	2103	7004	7210	6968	242

load balancing on the Blue Gene system is not as effective as on the Clemson cluster. We found that the Blue Gene system uses a three-dimensional (3D) torus network in which the nodes are connected to their six nearest-neighbor nodes in a 3D mesh, while the Clemson cluster employs a high-speed Myrinet networking system. Myrinet is a lightweight protocol with little overhead that allows it to operate with throughput close to the base signaling speed of the physical layer [102]. Myrinet is well-known for its low latency and high throughput performance. We believe the difference in networking systems of the two machines caused the performance difference.

1. The search strategy is the default one of ALPS (hybrid method) in which one of the children of a given node was retained as long as its bound did not exceed the best available by more than a given threshold.
2. The branching method is pseudocost branching.
3. Four cut generators from the COIN-OR Cut Generation Library [70], `CglMixedIntegerRounding`, `CglKnapsackCover`, `CglFlowCover`, and `CglGomory`, are used.
4. A rounding heuristic is used.

Figure 4.2: Setting for Testing Differencing

4.12 The Effectiveness of Differencing

As described above, the differencing scheme is an important aspect of the implementation of BiCePS for handling data-intensive applications. Here, we test the effect of it on a set of MILP instances using serial BLIS. Figure 4.2 lists the important settings.

Table B.1 lists the number of nodes, execution time and peak memory required to solve each instance with and without the differencing scheme. For every instance, the number of nodes required with the differencing scheme is the same as that without differencing. This is the expected behavior, since the differencing scheme does not affect the search order. There is little difference between the solution times (CPU seconds) with and without the differencing. However, the memory required to solve each instance is quite different. For almost all instances, using the differencing scheme require less memory.

Table 4.22: The effect of Differencing

	Without Differencing	With Differencing	Geometric mean
Total Time	2016 seconds	1907 seconds	1.0
Total Peak Memory	1412 MB	286 MB	4.3

We added up the solution times and memory usage of all instances. Table 4.22 shows the aggregated results. The total solution time is 2016 seconds without differencing, while the total solution time is 1907 seconds with differencing. The geometric mean of the solution time ratios is 1.0, which means there is no discernable difference in solution time. The total peak memory usage is 1412 MB without differencing, while the total peak memory usage is 286 MB when with differencing. The geometric mean of the peak memory usage ratios is 4.3. The results show that using differencing can save a significant amount of memory.

4.13 Search Strategies and Branching Methods

In this experiment, we tested different search strategies and branching methods for solving the VRP sequentially by using the VRP solver that we developed. We wanted to determine the best search strategy and branching method for solving VRP instances. For the test, we selected 16 instances from [84]. The time limit was 4 hours. Table 4.23 shows the results of solving the instances by using the default, best-first, and best-estimate search strategies (see Section 3.1.2). Note strong branching was used to choose branch variables and the running time is in seconds. The values in *Total* only include those of the instances that can be solved by all strategies, so instances `B-n45-k6` and `B-n57-k7` are excluded in computing values in the *Total* row. As the table shows,

Table 4.23: Search Strategies Comparison

	Default		Best-First		Best-Estimate	
Instance	Nodes	Time	Nodes	Time	Nodes	Time
A-n37-k6	5781	743.65	2981	510.81	2687	341.63
A-n39-k6	845	64.12	283	29.22	537	46.02
A-n44-k6	4859	1133.13	2835	811.36	2829	616.19
A-n45-k6	659	158.88	241	76.27	587	145.41
A-n48-k7	7903	1995.49	5293	1668.53	19377	4275.19
A-n53-k7	1995	973.92	601	360.04	761	388.59
A-n55-k9	7041	1554.09	1873	519.48	1829	423.17
A-n65-k9	16245	9853.71	6467	4427.57	17569	8706.63
B-n43-k6	501	64.75	251	47.87	1297	144.88
B-n45-k6	1237	242.10	1577	399.75	99614	TimeLimit
B-n57-k7	2215	532.27	1233	247.63	33708	TimeLimit
B-n57-k9	28441	8723.16	14083	9408.07	8645	2309.73
dantzig42	3603	688.80	309	91.39	563	116.86
P-n101-k4	157	183.35	107	194.91	297	397.75
P-n50-k7	4753	1105.54	1971	532.13	3719	850.67
P-n76-k4	683	873.50	145	218.43	223	274.65
Total	83466	28116.09	37440	18896.08	60920	19037.37

best-first search required the fewest number of node, and shortest running time. Best-estimate search was faster for the difficult instance `B-n57-k9`, but it could not solve two instances in the time limit. The default search solved all instances, but took longer than the best-first search.

Table 4.24 shows the results of using 3 different branching methods: strong branching, pseudocost branching, and reliability branching (see Section 3.1.1). Note that the best-first search strategy was used in all cases. Strong branching solved all instances and used less computing time than either pseudocost branching or reliability branching. Reliability branching was better than pseudocost in terms of the overall running time, but it still failed to solve two instances. It looks as though the best strategy is to always use

Table 4.24: Branch Methods Comparison

	Strong		Pseudocost		Reliability	
Instance	Nodes	Time	Nodes	Time	Nodes	Time
A-n37-k6	2981	510.81	24293	1693.37	9113	875.70
A-n39-k6	283	29.22	8925	465.23	1263	57.87
A-n44-k6	2835	811.36	27675	4683.02	13991	2344.72
A-n45-k6	241	76.27	14289	3002.47	4741	845.44
A-n48-k7	5293	1668.53	23619	7927.34	7877	2483.22
A-n53-k7	601	360.04	6939	3129.83	3315	1585.75
A-n55-k9	1873	519.48	32369	TimeLimit	20718	TimeLimit
A-n65-k9	6467	4427.57	11164	TimeLimit	9843	TimeLimit
B-n43-k6	251	47.87	3957	381.27	717	93.39
B-n45-k6	1577	399.75	8053	1858.38	1753	270.86
B-n57-k7	1233	247.63	4595	490.58	2247	466.65
B-n57-k9	14083	9408.07	47180	TimeLimit	28917	4953.88
dantzig42	309	91.39	23685	2963.86	3795	490.23
P-n101-k4	107	194.91	2361	1603.27	515	466.80
P-n50-k7	1971	532.13	20575	2839.64	10191	1916.23
P-n76-k4	145	218.43	1127	891.19	2143	2297.62
Total	17827	5199.34	170093	31929.45	61661	14194.48

the latest pseudocosts (as strong branching does) when choosing branching objects for VRP instances. As discussed in Section 3.1.1, both pseudocost and reliability branching will stop computing pseudocosts when certain conditions are satisfied , while strong branching always computes and uses the latest pseudocosts.

Chapter 5

Conclusions

5.1 Contributions

This is a computationally oriented thesis in which we have studied many algorithmic and implementational issues arising in the design of parallel tree search algorithms. In Chapter 2, we discussed a number of techniques to improve the scalability of parallel tree search. These techniques include the master-hub-worker paradigm, the knowledge management system, the three-tier load balancing schemes, and the mechanism for adjusting granularity. We illustrated the steps involved in implementing applications based on ALPS by describing the development of a knapsack solver.

In Chapter 3, we extended our search framework to handle discrete optimization problems. We proposed an object handling system and a differencing scheme for storage to handle large-scale optimization problem. We developed a MILP solver (BLIS) that employs a branch-and-cut algorithm and can be used to solve general MILPs. BLIS uses

the COIN/Cgl cut generators and provides base classes for users to develop problem-specific cut generators. BLIS also has it own primal heuristics routines and provides interface for users to add their new ones. We demonstrated the procedure for implementing applications based on BLIS by developing an application that can be used to solve both vehicle routing problem and the symmetric traveling salesman problem.

In Chapter 4, we described a number of experiments performed to test the scalability of our framework and the effectiveness of the methods that have been implemented to improve performance. In these experiments, we solved several sets of knapsack, generic MILP, and VRP instances. Our results show that overall scalability is relatively easy to achieve when solving the knapsack instances. We were able to obtain good scalability even when using several hundreds or thousands of processors. However, we failed to achieve good scalability for the VRP instances due to the fact that the number of nodes increases significantly as the number of processors increases. For generic MILPs, overall scalability was quite instance dependent. We have achieved almost linear speedup for some instances, but we got poor results for others.

In addition to testing overall scalability, we also tested the effectiveness of some of the specific scalability features of the framework. The results of our experiments show that effective knowledge sharing is the key to improving parallel scalability. An asynchronous implementation, along with effective load balancing is the most essential component of a scalable algorithm. The Master-hub-worker programming paradigm provides extra improvement for large-scale parallel computing. The differencing scheme that we use to handle data-intensive applications can significantly reduce memory usage without slowing down the search. We performed an experiment to test the effect of sharing pseudocosts during ramp-up. Finally, we tested the impact of hardware on scalability

and our results shows that hardware has significant impact on scalability, especially fast communication helps

5.2 Future Research

5.2.1 Improving Current Implementation

There a number of further investigations we hope to take up in future research. First of all, the current implementation of ALPS still needs some improvement. For example, in dynamic load balancing, the current implementation only allow the donor cluster to send one subtree to the receiver cluster. We need to study whether donating more that one subtree is helpful. Also, we need study rules for picking which subtree to share. Our results show the differencing scheme is useful for sequential search; however, we need verify and make sure it is still effective for parallel search. Furthermore, we plan to generalize the current implementation of BLIS in order to better support column generation and related algorithms, such as branch and price.

5.2.2 Overhead Reduction

Ramp-up overhead is one of the most important scalability issues for data-intensive applications because it may take long time to process a node. We have implemented the spiral initialization scheme to reduce ramp-up time. However, there are still several more ideas that we can try. We can use different branching rules (produce more children) or branch more quickly to expedite ramp-up phase. It is also possible that, during the ramp-up phase, workers can do other useful work instead of waiting idle. For example, workers may try to find good initial solutions via heuristics or test different search

settings.

Ramp-down time is also a big scalability issue. Currently, we do not adjust or use different load balancing schemes during ramp down. Probably, we can perform load balancing more frequently or use different load balancing schemes as the number of nodes available decreases.

Our experiment shows that the "communication overhead" became significant for MILPs as the number of processor increases. We guess it is caused by load balancing overhead such as splitting subtrees. We still need investigate more deeply to see if there are other reasons and study the methods that can be used to deal with this issue.

5.2.3 Fault Tolerance

Some applications are expected to run for days or even weeks on thousands of processors. Because of the large number of processors, the complexity of the computing platforms, and the long execution times, it is reasonable to expect at least one failure (unresponsive node or network failure) to occur during any single run. Right now, there is no routine to efficiently deal with failures, so when one occurs, the application crashes. The possible solutions are

- to periodically save the search state (checkpoint), or
- to replicate the search state.

However, supporting fault tolerance will increase overhead. That is a trade-off we need balance.

5.2.4 Grid Environment

We plan to make the framework useful in implementing applications that run on computational grids. One approach to achieve this is to derive a knowledge broker subclass from `AlpsKnowledgeBroker`. This subclass will use the MW tool [46] that support grid computing applications.

Appendix A

Tables of KNAP Experimental Results

Table A.1: Scalability of KNAP for the Moderately Difficult Instances

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock
input100_1	4	19490283	0.28%	0.02%	0.00%	60.14
	8	19490687	0.60%	0.16%	0.08%	25.12
	16	19491621	1.18%	0.76%	1.43%	11.9
	32	19498171	2.28%	1.47%	5.70%	6.14
	64	19531215	3.91%	5.03%	11.45%	3.58
	128	19626565	7.73%	6.76%	11.11%	2.07
input100_2	4	20631005	0.25%	0.02%	0.00%	63.72
	8	20631687	0.52%	0.07%	0.04%	26.71
	16	20630851	1.13%	0.16%	0.24%	12.35
	32	20631425	2.12%	0.65%	1.79%	6.13
	64	20632389	3.94%	1.82%	5.76%	3.3
	128	20636123	7.80%	3.90%	12.68%	2.05
input100_3	4	50683752	0.11%	0.04%	0.01%	158.6
	8	50517810	0.23%	0.08%	0.14%	66.06
	16	50212121	0.46%	0.17%	0.07%	30.17
	32	49844536	0.95%	0.54%	1.09%	14.69
	64	48052067	1.83%	1.97%	6.16%	7.63
	128	48069313	3.53%	4.64%	7.73%	4.53
input100_4	4	6081321	0.97%	0.05%	0.05%	18.62
	8	6082149	2.03%	0.00%	0.00%	7.88
	16	6078083	4.04%	0.27%	1.08%	3.71
	32	6087963	6.86%	1.96%	1.96%	2.04
	64	6104790	12.28%	2.63%	8.77%	1.14
	128	6203162	18.29%	2.44%	13.41%	0.82
input100_5	4	28988167	0.18%	0.02%	0.00%	87.5
	8	28988539	0.38%	0.03%	0.03%	36.56
	16	28992879	0.77%	0.12%	0.18%	16.9
	32	29004061	1.58%	0.36%	0.36%	8.23
	64	29026767	2.93%	1.81%	3.16%	4.43
	128	29095310	5.34%	2.49%	12.81%	2.81
input75_1	4	13859837	0.39%	0.00%	0.00%	41.08
	8	13862388	0.82%	0.06%	0.00%	17.15
	16	13869359	1.64%	0.13%	0.25%	7.92
	32	13883137	3.29%	0.51%	0.51%	3.95
	64	13896876	6.16%	1.42%	4.74%	2.11

Continued on next page

Table A.1 – continued from previous page

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock
	128	13922154	10.56%	4.23%	9.86%	1.42
input75_2	4	13435041	0.41%	0.03%	0.00%	39.14
	8	13436166	0.85%	0.12%	0.24%	16.55
	16	13439373	1.70%	0.13%	0.00%	7.65
	32	13443556	3.41%	0.79%	0.52%	3.81
	64	13461398	6.10%	1.88%	3.29%	2.13
	128	13536843	8.97%	2.76%	12.41%	1.45
input75_3	4	19146699	0.28%	0.00%	0.00%	56.29
	8	19077120	0.60%	0.04%	0.00%	23.49
	16	19017985	1.20%	0.28%	0.18%	10.83
	32	19020815	2.43%	0.56%	0.56%	5.35
	64	19053853	4.71%	0.72%	1.45%	2.76
	128	19150187	8.47%	4.52%	6.21%	1.77
input75_4	4	9853803	0.55%	0.03%	0.00%	28.94
	8	9853319	1.15%	0.08%	0.00%	12.21
	16	9853237	2.30%	0.18%	0.18%	5.64
	32	9858999	4.45%	0.34%	0.68%	2.92
	64	9883815	8.23%	1.90%	2.53%	1.58
	128	9955088	14.29%	2.86%	7.62%	1.05
input75_5	4	10887585	0.49%	0.03%	0.03%	32.87
	8	10891866	1.02%	0.22%	0.22%	13.7
	16	10670103	2.20%	0.47%	1.10%	6.36
	32	10694723	4.14%	0.64%	1.27%	3.14
	64	10700774	7.30%	3.37%	5.62%	1.78
	128	10690912	12.71%	3.39%	10.17%	1.18

Table A.2: Scalability of KNAP for the Difficult Instances

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock
input100_30	64	450193400	0.99%	1.54%	1.13%	158.94
	128	447704010	1.90%	3.17%	4.44%	82.67
	256	440847253	2.33%	4.33%	10.92%	44.59
	512	441705395	6.13%	5.66%	16.50%	25.45
	1024	443356768	11.37%	3.23%	23.81%	14.87
	2048	454050799	7.17%	2.76%	23.72%	7.25
input100_31	64	295018523	0.21%	6.17%	6.79%	117.01
	128	294993812	2.97%	1.96%	3.45%	53.6
	256	295029677	3.06%	6.52%	18.42%	34.04
	512	295346607	8.77%	3.35%	19.83%	17.9
	1024	297443751	16.01%	3.48%	27.42%	11.49
	2048	296750161	9.96%	3.52%	25.59%	5.12
input100_32	64	374169758	1.17%	2.89%	2.77%	136.46
	128	374191065	2.36%	2.61%	2.97%	67.79
	256	374190786	2.75%	5.77%	9.99%	38.15
	512	374232687	7.37%	5.40%	14.65%	21.3
	1024	374350599	13.92%	3.97%	24.85%	13.36
	2048	374870108	5.10%	6.23%	32.44%	7.06
input100_34	64	355850972	1.24%	3.27%	2.32%	130.14
	128	345678249	2.34%	10.26%	3.79%	68.91
	256	341502395	5.57%	4.45%	16.79%	38.23
	512	293980068	7.43%	7.10%	28.82%	21.41
	1024	312696260	14.42%	5.22%	30.63%	12.83
	2048	316668953	4.79%	6.29%	43.09%	7.31
input100_35	64	381809948	1.17%	2.66%	1.58%	137.68
	128	381542889	2.26%	4.12%	3.97%	71.35
	256	382584306	5.68%	5.10%	5.29%	37.84
	512	383189419	6.83%	7.00%	18.22%	23.44
	1024	388970274	14.22%	4.26%	20.23%	13.15
	2048	399546872	6.95%	3.21%	33.16%	7.48
input100_36	64	684364180	0.63%	5.14%	2.23%	255.75
	128	679456710	1.17%	6.45%	8.80%	137.7
	256	676044591	3.13%	8.15%	5.54%	68.09
	512	677306425	6.27%	8.26%	16.99%	42.61
	1024	681981001	12.43%	5.71%	16.65%	21.56

Continued on next page

Table A.2 – continued from previous page

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock
	2048	691970626	4.09%	6.40%	32.92%	12.97
input100_37	64	440881122	0.97%	5.73%	4.55%	169.57
	128	441118207	1.82%	6.62%	9.19%	90.02
	256	438120922	4.40%	7.35%	14.06%	49.09
	512	432528103	8.72%	5.35%	26.66%	31.21
	1024	441967447	17.70%	4.60%	19.94%	15.65
	2048	461446346	5.91%	5.70%	34.72%	9.13
input100_38	64	399288703	1.14%	2.04%	1.25%	141.76
	128	396777320	2.22%	4.08%	3.04%	73
	256	416006687	4.89%	4.22%	11.70%	43.32
	512	370375938	6.89%	7.93%	18.40%	22.94
	1024	392728769	13.78%	4.59%	21.85%	13.5
	2048	397019339	7.30%	4.92%	29.21%	7.12
input100_39	64	366764346	1.21%	4.37%	1.75%	134.63
	128	352189300	2.32%	7.63%	6.58%	70.4
	256	377854493	5.36%	5.79%	11.17%	39.93
	512	354283715	11.59%	5.34%	18.53%	23.2
	1024	365731001	12.54%	4.29%	33.13%	14.91
	2048	324128459	6.82%	4.98%	43.77%	7.63
input175_0	64	1091614792	0.39%	0.58%	0.44%	473.33
	128	1099282426	0.61%	17.42%	4.34%	299.42
	256	898334310	1.23%	1.10%	3.63%	99.46
	512	310337557	16.68%	1.82%	6.38%	20.38
	1024	329269870	18.43%	1.30%	11.57%	11.5
	2048	823706045	14.15%	0.32%	2.99%	12.37
input150_1	64	1107247315	0.39%	3.08%	0.56%	471.11
	128	1106901522	0.75%	4.29%	3.74%	244.14
	256	1107133612	0.78%	12.04%	13.63%	153.82
	512	1111310554	4.86%	4.23%	10.46%	69.57
	1024	1110490269	5.89%	5.40%	15.20%	36.65
	2048	1123112986	6.30%	3.51%	19.31%	19.37
input150_10	64	520055323	0.82%	1.72%	1.21%	213.49
	128	520342395	1.55%	4.58%	3.72%	112.36
	256	521419528	1.77%	9.34%	10.36%	64.87
	512	523450894	9.72%	3.17%	9.93%	32.82
	1024	525028667	11.32%	3.79%	16.49%	18.19

Continued on next page

Table A.2 – continued from previous page

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock
	2048	533194171	5.38%	4.69%	27.08%	10.23
input150_11	64	652738452	0.63%	3.83%	0.83%	275.91
	128	652991048	1.23%	4.10%	3.72%	141.45
	256	653578790	2.73%	4.60%	17.16%	83.85
	512	655240782	8.38%	4.02%	11.05%	41.54
	1024	661342072	8.78%	4.35%	17.65%	22.78
	2048	664773821	4.22%	4.91%	29.32%	13.03
input150_12	64	477186428	0.89%	2.53%	1.93%	191.24
	128	487239170	1.73%	3.52%	2.54%	98.64
	256	462070673	4.68%	2.02%	4.85%	48.06
	512	422860938	10.86%	2.55%	18.00%	29
	1024	443618162	14.50%	2.87%	13.89%	14.62
	2048	428534193	6.64%	2.01%	28.45%	7.98
input150_13	64	766822612	0.46%	5.97%	8.12%	366.28
	128	769348911	0.79%	5.60%	21.91%	216.41
	256	734038137	2.32%	7.96%	15.89%	96.09
	512	723551622	6.13%	4.61%	20.46%	51.02
	1024	772499481	16.45%	1.91%	10.48%	24.62
	2048	674117750	9.68%	2.94%	19.01%	11.57
input150_14	64	394270918	1.12%	0.09%	0.12%	173.48
	128	355312724	2.51%	0.18%	0.42%	77.74
	256	251750819	4.55%	0.42%	1.48%	28.38
	512	348058879	9.05%	1.08%	3.82%	21.21
	1024	102825860	43.71%	0.19%	5.03%	5.17
	2048	271476227	13.90%	0.23%	3.87%	4.39
input150_17	64	348443929	1.28%	0.72%	0.50%	135.35
	128	368502776	2.33%	1.32%	3.97%	74.3
	256	309026062	3.53%	2.08%	5.73%	32.28
	512	315576256	9.40%	1.59%	9.56%	18.2
	1024	275223835	21.07%	1.39%	10.37%	9.35
	2048	269492098	23.96%	0.83%	9.38%	4.8
input150_18	64	453055347	0.95%	3.15%	0.94%	188.22
	128	453367417	1.79%	5.20%	4.21%	99.43
	256	454095303	4.23%	5.48%	12.66%	56.25
	512	456442211	10.09%	4.11%	17.09%	32.6
	1024	459135871	11.96%	4.23%	22.39%	17.73

Continued on next page

Table A.2 – continued from previous page

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock
	2048	457358640	14.02%	2.90%	19.35%	8.63
input150_19	64	314137323	1.38%	2.79%	2.04%	131.19
	128	314345732	2.57%	4.27%	6.82%	70.33
	256	315316104	3.09%	9.59%	8.51%	38.78
	512	317732319	7.89%	8.59%	16.48%	22.7
	1024	320513829	16.48%	3.96%	19.41%	12.62
	2048	325350109	19.12%	2.53%	15.01%	6.33
input150_2	64	962872398	0.43%	3.23%	0.75%	411.63
	128	963209404	0.83%	6.86%	2.61%	215.67
	256	963948282	0.97%	8.55%	9.45%	121.03
	512	967535513	5.32%	4.07%	13.67%	62.2
	1024	965626180	13.18%	3.12%	11.60%	31.72
	2048	972658946	7.08%	3.63%	18.86%	16.81
input150_20	64	269752458	1.56%	2.59%	1.14%	111.28
	128	269885124	3.01%	3.53%	3.38%	57.75
	256	270695598	3.16%	6.35%	18.18%	36.35
	512	272287026	8.60%	4.90%	21.31%	19.99
	1024	270788793	17.75%	4.19%	20.96%	11.21
	2048	279991651	20.49%	3.00%	15.19%	5.66
input150_4	64	1414324008	0.25%	17.04%	2.46%	766.87
	128	1447906947	0.53%	11.50%	2.79%	363.96
	256	898884789	1.31%	5.14%	1.04%	97.47
	512	1264535885	4.62%	0.48%	8.01%	77.29
	1024	1231439139	11.17%	1.00%	7.65%	42.07
	2048	1325447920	12.79%	2.45%	12.79%	24.08
input150_5	64	334366142	1.34%	1.07%	0.50%	144.43
	128	380811420	2.36%	1.27%	1.49%	82.58
	256	546098309	2.09%	1.58%	3.80%	61.36
	512	346443319	8.82%	3.02%	6.63%	21.87
	1024	430128080	15.31%	1.24%	8.58%	14.57
	2048	277659553	2.81%	3.75%	36.25%	6.4
input150_7	64	243837878	1.82%	1.46%	1.47%	106.38
	128	238604790	3.62%	2.22%	3.47%	53.62
	256	274353503	3.79%	3.76%	9.54%	33.76
	512	237310725	11.20%	3.27%	14.47%	17.14
	1024	356962624	16.97%	1.61%	12.54%	13.08

Continued on next page

Table A.2 – continued from previous page

Instance	P	Node	Ramp-up	Idle	Ramp-down	Wallclock
	2048	249173559	27.09%	1.11%	11.32%	5.39
input150_8	64	946658511	0.37%	3.41%	10.89%	456.24
	128	946982344	0.78%	6.39%	4.42%	216.58
	256	948089177	0.77%	21.42%	9.37%	145.36
	512	948725768	4.95%	4.66%	16.17%	63.58
	1024	949136593	14.46%	2.00%	6.90%	28.57
	2048	955634637	10.29%	2.99%	14.10%	15.74
input150_9	64	688020337	0.57%	4.40%	2.30%	298.13
	128	688060032	1.13%	2.83%	5.78%	150.75
	256	688714214	2.72%	5.41%	10.86%	82.42
	512	689599891	6.71%	4.53%	16.27%	46.97
	1024	693724499	8.13%	5.13%	17.84%	23.99
	2048	697294621	13.10%	4.12%	15.20%	12.37

Appendix B

Tables of Integer Programming Experimental Results

Table B.1: The Results of Differencing

Instance	Without Differencing			With Differencing		
	Nodes	Time	Memory(KB)	Node	Time	Memory(KB)
10teams	989	232.27	39579	989	234.26	2606
align1	11	7.47	1872	11	7.40	1764
align2	1357	20.74	15076	1357	20.76	3216
air03	1	0.43	10748	1	0.43	10748
bell3a	35363	97.09	88423	35363	72.00	12147
cap6000	4475	82.14	429385	4475	81.59	13401
dano3imp	47	337.85	25313	47	336.63	10684
Camelot	1133	4.20	17450	1133	4.19	1439
submit	737	33.28	34150	737	33.16	2428
egout	21	0.18	347	21	0.18	228
enigma	7525	6.41	13146	7525	6.16	2377
fiber	583	6.60	17667	583	6.56	2078
flugpl	111	0.39	207	111	0.40	157
gen	1	0.08	789	1	0.09	789
gesa2	357	7.35	14161	357	7.29	1766
gesa3_o	1821	138.33	59303	1821	139.22	4580
gesa3	1117	143.42	35906	1117	144.18	2817
khb05250	281	1.37	11648	281	1.33	1217
l152lav	783	14.15	37336	783	14.07	3466
lseu	187	1.53	980	187	1.52	437
misc03	895	3.61	3678	895	3.61	830
misc06	41	1.56	3147	41	1.54	1400
mitre	75	67.19	35735	75	68.19	11237
mod008	2697	4.71	19442	2697	4.38	1449
mod010	113	2.95	8418	113	2.89	2221
gamsmod	19	226.57	228935	19	227.47	141483
p0033	1	0.02	109	1	0.03	109
p0201	167	4.89	2132	167	4.91	1122
p0282	23	3.31	900	23	3.29	706
p0548	43	9.31	2082	43	9.23	1125
p2756	181	12.09	19946	181	12.13	3245
qnet1_o	349	8.90	14628	349	8.88	1809
qnet1	111	7.48	5202	111	7.43	1537
rentacar	439	128.11	105040	439	128.08	9342
rgn	2273	4.96	13266	2273	4.75	1477

Continued on next page

Table B.1 – continued from previous page

	Without Differencing			With Differencing		
Instance	Nodes	Time	Memory(KB)	Node	Time	Memory(KB)
test3	1823	2.99	7838	1823	2.79	1154
stein27	4625	7.35	5084	4625	6.94	1954
stein45	56433	384.31	82962	56433	301.78	25532

Table B.2: Static Load Balancing Comparison

Instance	Scheme	Node	Ramp-up	Idle	Ramp-down	Wallclock
aflow30a	Root	1450327	1.61%	12.09%	16.78%	251.18
	Spiral	914829	0.34%	12.79%	11.68%	144.35
bell5	Root	566539	4.29%	6.95%	4.58%	12.26
	Spiral	397265	1.49%	12.30%	3.39%	8.22
bienst1	Root	43077	13.72%	5.44%	6.85%	57.45
	Spiral	44527	1.66%	5.87%	11.09%	55.37
bienst2	Root	330563	3.28%	4.71%	1.39%	275.95
	Spiral	318789	0.39%	4.32%	2.61%	259.86
blend2	Root	452563	2.58%	10.32%	5.28%	29.35
	Spiral	507943	0.19%	9.33%	6.19%	30.62
cls	Root	382121	7.10%	12.33%	11.67%	38.59
	Spiral	494207	1.90%	12.48%	10.96%	44.79
fiber	Root	27115	16.65%	8.78%	16.36%	13.06
	Spiral	9627	7.94%	12.42%	23.32%	5.85
gesa3	Root	56553	11.91%	9.97%	29.90%	20.91
	Spiral	139283	1.61%	18.52%	18.17%	47.57
markshare_4_1	Root	3473213	99.98%	0.00%	0.01%	779.05
	Spiral	2335049	99.99%	0.00%	0.01%	1136.54
markshare_4_1	Root	3436579	0.16%	12.58%	10.13%	16.30
	Spiral	2789645	0.19%	13.33%	7.62%	12.90
mas76	Root	2112409	1.74%	6.26%	9.42%	27.16
	Spiral	2298361	0.08%	16.09%	16.76%	37.18
misc07	Root	14337	10.10%	11.99%	27.05%	11.22
	Spiral	10343	0.30%	22.23%	53.44%	23.34
mitre	Root	391	90.07%	0.00%	9.49%	25.28
	Spiral	2221	42.48%	5.19%	47.42%	26.32
nw04	Root	3063	12.39%	2.56%	74.62%	415.57
	Spiral	629	4.22%	1.80%	88.87%	166.27
pk1	Root	1778037	2.20%	6.10%	8.13%	22.87
	Spiral	1559947	0.72%	8.79%	9.30%	21.28
rout	Root	2087679	0.77%	13.04%	3.99%	247.47
	Spiral	2075145	0.08%	11.55%	4.18%	238.21
stein45	Root	112715	9.86%	20.52%	6.12%	6.23
	Spiral	101161	1.90%	25.22%	9.18%	5.37
swath1	Root	57343	6.07%	17.82%	13.63%	104.96

Continued on next page

Table B.2 – continued from previous page

Instance	Scheme	Node	Ramp-up	Idle	Ramp-down	Wallclock
	Spiral	59849	0.44%	21.17%	8.66%	94.22
swath2	Root	154446	3.45%	16.39%	5.17%	190.30
	Spiral	131145	0.28%	18.07%	4.79%	151.76
vpm2	Root	439289	6.65%	9.41%	7.73%	24.92
	Spiral	558269	1.18%	10.99%	7.32%	29.58
Total	Root	15876829	55.24%	4.31%	11.42%	3690.30
	Spiral	15849764	1.74%	10.75%	18.21%	1419.37

Table B.3: The Effect of Dynamic Load Balancing

Instance	Balance	Node	Ramp-up	Idle	Ramp-down	Wallclock
aflow30a	No	1783235	0.28%	0.00%	76.41%	899.22
	Yes	1447135	1.15%	10.30%	8.20%	221.04
bell5	No	371095	0.46%	0.00%	93.02%	94.74
	Yes	245991	7.35%	8.01%	11.76%	5.95
bienst1	No	51589	2.44%	0.00%	66.48%	168.58
	Yes	46293	6.28%	8.48%	14.25%	65.67
bienst2	No	330713	0.43%	0.00%	69.09%	888.72
	Yes	297963	1.46%	5.72%	6.15%	263.60
blend2	No	345909	0.30%	0.00%	79.93%	84.42
	Yes	452563	2.58%	10.32%	5.28%	29.35
cls	No	538665	1.42%	0.00%	67.85%	130.39
	Yes	388511	4.23%	8.81%	24.14%	44.41
fiber	No	16817	23.24%	0.00%	42.90%	11.70
	Yes	9627	7.94%	12.42%	23.32%	5.85
gesa3	No	190489	0.36%	0.00%	92.62%	471.12
	Yes	56849	7.29%	9.89%	19.86%	23.36
markshare_4_1	No	3463729	80.61%	0.00%	19.36%	433.14
	Yes	3463729	79.53%	0.00%	20.45%	433.31
markshare_4_3	No	2613747	74.87%	0.00%	25.12%	552.98
	Yes	2613747	75.79%	0.00%	24.19%	551.38
mas76	No	2017019	0.35%	0.00%	72.09%	78.12
	Yes	2050675	1.09%	3.44%	8.76%	25.47
misc07	No	13145	3.16%	0.00%	74.70%	17.95
	Yes	12535	4.43%	12.90%	48.18%	12.84
mitre	No	391	48.22%	0.00%	45.45%	33.36
	Yes	391	48.34%	0.00%	45.45%	33.20
nw04	No	1821	10.91%	0.00%	81.38%	341.18
	Yes	1721	19.13%	3.26%	65.73%	194.28
pk1	No	2469351	0.25%	0.00%	74.04%	109.03
	Yes	2671467	0.68%	10.54%	15.10%	39.84
rout	No	2430157	0.13%	0.00%	82.43%	1090.06
	Yes	1501183	0.52%	21.58%	20.83%	261.28
stein45	No	103995	4.14%	0.00%	61.08%	9.57
	Yes	103953	6.90%	9.92%	26.24%	5.75
swath1	No	83719	0.22%	0.00%	95.42%	1361.25

Continued on next page

Table B.3 – continued from previous page

Instance	Balance	Node	Ramp-up	Idle	Ramp-down	Wallclock
	Yes	58473	2.27%	16.75%	34.26%	134.70
swath2	No	182801	0.12%	0.00%	95.08%	2654.43
	Yes	148243	1.48%	13.17%	14.73%	211.41
vpm2	No	892119	0.38%	0.00%	86.87%	288.70
	Yes	486571	4.06%	7.57%	10.26%	26.72
Total	No	17900506	8.69%	0.00%	79.67%	9718.67
	Yes	16057620	32.50%	6.81%	22.32%	2589.43

Table B.4: The Effect of Sharing Pseudocost

Instance	Share	Node	Ramp-up	Idle	Ramp-down	Wallclock
aflow30a	No	1196395	0.25%	13.71%	15.27%	193.69
	Yes	914829	0.34%	12.79%	11.68%	144.35
bell5	No	339245	1.44%	12.85%	8.06%	8.32
	Yes	397265	1.49%	12.30%	3.39%	8.22
bienst1	No	47109	1.67%	5.56%	5.59%	55.29
	Yes	44527	1.66%	5.87%	11.09%	55.37
bienst2	No	309189	0.40%	4.52%	2.48%	249.36
	Yes	318789	0.39%	4.32%	2.61%	259.86
blend2	No	525193	0.18%	10.62%	4.32%	31.46
	Yes	525855	0.18%	9.95%	5.28%	31.41
cls	No	489855	1.66%	11.71%	19.17%	50.95
	Yes	494207	1.90%	12.48%	10.96%	44.79
fiber	No	10535	5.89%	9.38%	40.05%	7.90
	Yes	9695	6.85%	12.01%	33.96%	6.77
gesa3	No	116577	1.98%	18.81%	25.63%	38.54
	Yes	139283	1.61%	18.52%	18.17%	47.57
markshare_4_1	No	3435775	0.71%	13.75%	8.95%	16.49
	Yes	3436579	0.16%	12.58%	10.13%	16.30
markshare_4_3	No	2790777	0.19%	12.14%	8.48%	13.02
	Yes	2789645	0.19%	13.33%	7.62%	12.90
mas76	No	2666601	0.07%	18.30%	13.85%	42.69
	Yes	2298361	0.08%	16.09%	16.76%	37.18
misc07	No	10629	0.29%	25.09%	54.95%	24.98
	Yes	10343	0.30%	22.23%	53.44%	23.34
nw04	No	645	4.15%	2.05%	88.51%	168.67
	Yes	629	4.22%	1.80%	88.87%	166.27
pk1	No	2111119	0.28%	10.26%	6.99%	28.06
	Yes	1559947	0.72%	8.79%	9.30%	21.28
rout	No	1874903	0.09%	8.32%	1.81%	200.56
	Yes	2075145	0.08%	11.55%	4.18%	238.21
stein45	No	97333	1.93%	27.62%	6.76%	5.20
	Yes	101161	1.90%	25.22%	9.18%	5.37
swath1	No	52161	0.44%	20.38%	13.16%	94.32
	Yes	59849	0.44%	21.17%	8.66%	94.22
swath2	No	139117	0.24%	20.94%	5.00%	168.17

Continued on next page

Table B.4 – continued from previous page

Instance	Share	Node	Ramp-up	Idle	Ramp-down	Wallclock
	Yes	131145	0.28%	18.07%	4.79%	151.76
vpm2	No	626215	1.05%	10.73%	6.90%	33.36
	Yes	558269	1.18%	10.99%	7.32%	29.58
Total	No	16839373	0.94%	11.16%	18.47%	1431.04
	Yes	15865523	0.96%	10.87%	17.69%	1394.76

Bibliography

[1] T. Achterberg. SCIP—a framework to integrate constraint and mixed integer programming. Technical Report 04–19, Zuse Institute Berlin, 2004. `http://www.zib.de/Publications/abstracts/ZR-04-19/`.

[2] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters*, 33:42–54, 2005.

[3] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):1–12, 2006. See `http://miplib.zib.de`.

[4] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc, New Jersey, USA, 1st edition, 1993.

[5] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc, CA, USA, 1st edition, 1989.

[6] 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.

[7] T. Anderson, D. Culler, and D. Patterson. A case for now (networks of workstations, 1995.

[8] Answers.com. `http://www.answers.com/topic/optimization`.

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

[10] K. R. Apt. *Principles of constraint programming*. Cambridge University Press, USA, 2003.

[11] A. C. Arpaci-Dusseau, D. E. Culler, and A. Mainwaring. Scheduling with implicit information in distributed systems. In *Proceedings of 1998 SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 233–243, 1988.

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

[13] A. Atamtürk, G. Nemhauser, and M.W.P. Savelsbergh. Conflict graphs in solving integer programming problems. *European J. Operational Research*, 121:40–55, 2000.

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

[15] E. Balas, S. Schmieta, and C. Wallace. Pivot and shift—a mixed integer programming heuristic. *Discrete Optimization*, 1:3–12, 2004.

[16] P. Banerjee, J. A. Chandy, M. Gupta, E. W. Hodges IV, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The paradigm compiler for distributed-memory multicomputers. *IEEE Computer*, 28(10), October 1995.

[17] E. M. L. Beale and J. A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. In J. Lawrence, editor, *Proceedings of the 5th International Conference on Operation Research*, pages 447–454. Tavistock Publications, 1969.

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

[19] D. P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, USA, 2nd edition, 2000.

[20] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation Numerical Methods*. Prentice Hall, New Jersey, USA, 1st edition, 1989.

[21] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with polaris. *IEEE Computer*, 29(12):78–82, 1996.

[22] D. R. Butenhof. *Programming with $POSIX^{@}$ Threads*. Prentice-Hall, Inc, London, 1997.

[23] R. Buyya, editor. *High Performance Cluster Computing: Architecture and Systems*, volume 1. Prentice-Hall, Inc, New Jersey, USA, 1999.

[24] R. Buyya, editor. *High Performance Cluster Computing: Programming and Applications*, volume 2. Prentice-Hall, Inc, New Jersey, USA, 1999.

[25] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, USA, 1st edition, 2007.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, volume 2. The MIT Press, Massachusetts, USA, 2001.

[27] G. Cornuéjols and M. Dawande. A class of hard small 0-1 programs. In R. E. Bixby, E. A. Boyd, and R. Z. Rios-Mercado, editors, *Integer Programming and Combinatorial Optimization*, pages 284–293, Berlin, German, 1998. Springer-Verlag.

[28] E. Danna, E. Rothberg, and C. L. Pape. Exploring relaxation induced neighborhoods to improve mip solutions. *Mathematical Programming*, 102(1):71–90, 2005.

[29] G. B. Dantzig and R. H. Ramster. The truck dispatching problem. *Management Science*, 6:80–91, 1959.

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

[31] ILOG CPLEX Division. `http://www.cplex.com`.

[32] J. Eckstein. Distributed versus centralized storage and control for parallel branch and bound: Mixed integer programming on the cm-5. *Comput. Optim. Appl.*, 7(2):199–220, 1997.

[33] 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.

[34] J. Eckstein, C. A. Phillips, and W. E. Hart. PEBBL 1.0 user guide, 2007.

[35] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. Harness and fault tolerant mpi. *Parallel Computing*, 27(11):1479–1495, 2001.

[36] M. Feeley, M. Turcotte, and G. Lapalme. Using Multilisp for solving constraint satisfaction problems: an application to nucleic acid 3D structure determination. *Lisp and Symbolic Computation*, 7(2/3):231–247, 1994.

[37] R. Feldmann, P. Mysliwietz, and B. Monien. Game tree search on a massively parallel system. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess7*, pages 203–218, Masstricht, The Netherlands, 1994. University of Limburg.

[38] M. Fischetti, F. Glover, and A. Lodi. The feasibility pump. *Mathematical Programming A*, 104:91–104, 2005.

[39] M. Fischetti and A. Lodi. Local branching. *Mathematical Programming B*, 98:23–47, 2003.

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

[41] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[42] G. C. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works*. Morgan Kaufmann Publishers, San Francisco, California, 1994.

[43] T. Fruhwirth and S. Abdennadher. *Essentials of constraint programming*. Springer, 2003.

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

[45] R. E. Gomory. A algorithm for the mixed integer problem. Technical report, The RAND Cooperation, 1960.

[46] J. P. Goux, S. Kulkarni, J. T. Linderoth, and M. E. Yoder. Master-worker: An enabling framework for applications on the the computational grd. *Cluster Computing*, 4:63–70, 2001.

[47] A. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel Distrib. Technol.*, 1(3):12–21, 1993.

[48] A. S. Grimshaw and W. A. Wulf. A view from 50,000 feet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, Los Alamitos, California, 1996. IEEE Computer Society Press.

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

[50] M. Grötschel, M. Jünger, and G. Reinelt. A cutting plane algorithm for the linear ordering problem. *Operations Research*, 32(6):1195–1220, 1984.

[51] Z. Gu, G. L. Nemhauser, and M. W. P. Savelsberg. Lifted flow cover inequalities for mixed 0-1 integer programs. *Mathematical Programming*, 85:439–468, 1999.

[52] Z. Gu, G. L. Nemhauser, and M. W. P. Savelsberg. Sequence independent lifting in mixed integer programming. *Journal on Combinatorial Optimization*, 4(6):109–129, 2000.

[53] J. L. Gustafson. Reevaluate amdahl's law. *Comm. ACM*, 31(1):532–533, 1988.

[54] P. B. Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, 1993.

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

[56] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT press, USA, 1989.

[57] K. Hoffman and M. Padberg. Lp-based combinatorial problem solving. *Annals of Operations Research*, 4(6):145–194, 198586.

[58] Holger Hopp and Peter Sanders. Parallel game tree search on SIMD machines. In *Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 349–361, 1995.

[59] IBM. Grid computing: Past, present and future.

[60] E. L. Johnson, G. L. Nemhauser, and M. W. P. Savelsbergh. Progress in linear programming based branch-and-bound algorithms: An exposition. *INFORMS Journal on Computing*, 12:2–23, 2000.

[61] M. Karamanov and G. Cornuejols. Branching on general disjunctions. *Mathematical Programming (to appear)*.

[62] R. E. Korf. Artificial intelligence search algorithms. In *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[63] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings, Redwoood City, CA, USA, 1st edition, 1993.

[64] 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.

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

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

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

[68] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessor*. PhD thesis, Department of Computer Science, Yale University, 1986.

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

[70] R. Lougee-Heimer. The common optimization interface for operations research. *IBM Journal of Research and Development*, 47(1):57–66, 2003.

[71] D. Loveman. High-performance fortran. *IEEE Parallel and Distributed Technology*, 24, 1993.

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

[73] F. Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2:161–175, 1987.

[74] S. Mehrotra and Z. Li. On generalized branching methods for mixed integer programming. Technical report, Department of Industrial Engineering, Northwestern University, Evanston, Illinois 60208, 2004.

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

[76] A. Osman and H. Ammar. Dynamic load balancing strategies for parallel computers.

[77] M. Padberg and G. Rinaldi. A branch-and-cut algorithm for the resolution of large-scale traveling salesman problems. *SIAM Review*, 33:60–100, 1991.

[78] M. W. Padberg. A node on zero-one programming. *Operations Research*, 23:833–837, 1975.

[79] J. Patel and J. W. Chinneck. Active-constraint variable ordering for faster feasibility of mixed integer linear programs. *Mathematical Programming, to appear*, 2006.

[80] J. F. pekny89. *Exact Parallel Algorithms for Some Members of the Traveling Salesman Problem Family*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, USA, 1989.

[81] Marc Pfetsch. Markshare instances, 2005.

[82] M. Platzner and B. Rinner. Design and implementation of a parallel constraint satisfaction algorithm. *International Journal of Computers and Their Applications*, 5(2):106–116, 1999.

[83] D. Pritchard. Mathematical models of distributed computation. In *OUG-7, Parallel Programming on Transputer Based Machines*. IOS Press, 1992.

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

[85] T. K. Ralphs. *Parallel Branch and Cut for Vehicle Routing*. PhD thesis, Field of Operations Research, Cornell University, Ithaca, NY, USA, 1995.

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

[87] T. K. Ralphs. Parallel branch and cut. In E. Talbi, editor, *Parallel Combinatorial Optimization*, pages 53–102. Wiley, USA, 2006.

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

[89] T. K. Ralphs, L. Kopman, W. R. Pulleyblank, and L. E. Trotter Jr. On the capacitated vehicle routing problem. *Mathematical Programming Series B*, 94:343–359, 2003.

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

[91] D. A. Reed. Grids, the teragrid, and beyond. *Computer*, 36(1):62–68, 2003.

[92] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., USA, 2nd edition, 2003.

[93] P. Sanders. Parallelizing np-complete problems using tree shaped computations.

[94] P. Sanders. Analysis of random polling dynamic load balancing. Technical Report iratr-1994-12, 1994.

[95] P. Sanders. Randomized static load balancing for tree shaped computations, 1994.

[96] P. Sanders. A scalable parallel tree search library, 1996.

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

[98] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[99] 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.

[100] 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.

[101] Top 500 Supercomputer sites. `http://www.top500.org`.

[102] A. J. Steen and J. J. Dongarra. Overview of recent supercomputers. `http://www.netlib.org/utk/papers/advanced-computers`.

[103] T. Sterling. *Beowulf Cluster Computing with Linux*. The MIT Press, USA, 1st edition, 2001.

[104] T. Sterling, J. S. Donald, J. B. Becker, and D. F. Savarese. *How to Build a Beowulf: A Guide to the Implementation and Application of PC Clusters*. MIT Press, USA, 1st edition, 1999.

[105] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the condor experience. *Concurrency—Practice and Experience*, 17(2–4):323–356, 2005.

[106] 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.

[107] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[108] S. Tschöke and T. Polzer. Portable parallel branch and bound library, 1996.

[109] D. Wang, W. Zheng, and J. Xiong. Research on cluster of workstations. In *ISPAN '97: Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97)*, page 275, Washington, DC, USA, 1997. IEEE Computer Society.

[110] R. Weismantel. On the $0/1$ knapsack polytope. *Mathematical Programming*, 77, 1997.

[111] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Worksations and Parallel Computers*. Prentice-Hall, Inc, New Jersey, USA, 1st edition, 1999.

[112] G. Wilson. *Parallel Programming for Scientists and Engineers*. MIT Press, MA, USA, 1996.

[113] L. A. Wolsey. Valid inequalities for 0-1 knapsacks and mips with generalized upper bound constraints. *Discrete Applied Mathematics*, 29, 1990.

Vita

Education

- Fudan University, China, M.S. in Management Science, 1996
- Fudan University, China, B.S. in Mechanics (Mathematical), 1993

Current Positions

- Optimization Development Lead, Advanced Analytics, SAS Institute Inc., Cary, NC, 2004 – present

Previous Positions

- Intern, Service Parts Solutions, IBM, Mechanicsburg, PA, 2001 – 2004
- Research Assistant, Department of Engineering Management, City University of Hong Kong, Hong Kong, 1998 – 2000
- Senior Analyst, Asset Management, Pacific Mechatronics (Group) Co., Ltd., Shanghai, China, 1997 – 1998
- Research Staff, Research and Development, Shanghai Erfangji Co., Ltd., Shanghai, China, 1996 – 1997