

Topics in Parallel Integer Optimization

A THESIS
Presented to
The Academic Faculty

by

Jeffrey T. Linderoth

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in Industrial and Systems Engineering

Georgia Institute of Technology
August 11, 1998

*To my parents and to my grandmothers.
The people whose love made this possible.*

Topics in Parallel Integer Optimization

Approved:

Martin Savelsbergh, Chairman

Lloyd Clarke

Ellis Johnson

George Nemhauser

Karsten Schwan

Date Approved

ACKNOWLEDGEMENTS

I first would like to express my sincere appreciation to my advisor Martin Savelsbergh for his wonderful guidance that made this work a reality. Simply, one could not ask for a better thesis advisor. I would also like to thank Ellis Johnson and George Nemhauser for sharing their brilliance with me on many subjects during my tenure at Georgia Tech. Thanks are due to Lloyd Clarke and Karsten Schwan for their friendship, encouragement, and assistance in preparing this work. The financial assistance of the many professors of the Logistics Institute who supported my work is gratefully acknowledged.

Like most large software systems, PARINO and PSP were not undertaken alone. Kalyan Perumalla deserves special mention for writing the NAYLAK library, writing the initial PARINO implementation, and for his patience and guidance while I familiarized myself with this software. Martin Savelsbergh also provided me with portions of the MINTO code which found their way into PARINO. I would like to thank Alper Atamtürk, Eva Lee, and Martin Savelsbergh for writing modules of the PSP system.

The list of students and colleagues who made life bearable over the years in Atlanta and in room 402 is quite long. I would like to thank Alper Atamtürk,

Diego Klabjan, Joe Hartman, Ladislav Lettovský, Dirk Güether, Vijay Nori, Petra Bauer, Hamish Waterer, Dieter Vandenbussche, Ann Campbell, Jill Hardin, Andrew Miller, Andrew Schaefer, Jarrod Goentzel, Ismael Farias, Zonghao Gu, Kevin Gue, Dasong Cao, Andrea VanHull, Noha Tohamy, and other “402”ers. My non-optimizing friends have also helped keep my life and work in perspective, and for that I thank the entire Moe’s and Joe’s gang.

I have the most wonderful family in the world, who with their love and support have made everything in my life possible. Thanks Mom, Dad, Unk, Sean, Grandma, Jenny, and Jill. The love and encouragement of Helen Chou was of immeasurable comfort during the final phases of this work. I love you all very much.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	ii
LIST OF TABLES	viii
LIST OF FIGURES	xiii
SUMMARY	xvii
1 Introduction	1
1.1 Literature Review	4
1.2 Thesis Outline	7
2 Integer Programming Preliminaries	10
2.1 Mixed Integer Programming	10
2.1.1 Branch and Bound	10
2.1.2 Other Algorithmic Enhancements	14
2.1.3 Cutting Planes	15
2.2 Set Partitioning Problem	21
2.2.1 Preprocessing and Probing	22

2.2.2	Solving the Linear Program	24
2.2.3	Cutting Planes	28
2.2.4	Primal Heuristics	31
2.2.5	Reduced Cost Fixing	35
3	Parallel Computing Preliminaries	36
3.1	Shared Memory versus Message Passing	36
3.2	Advantages of Parallel Computing	39
3.2.1	Concurrent Computing	39
3.2.2	Information Sharing	48
3.3	NAYLAK	57
4	Parallel Branch and Bound	61
4.1	Introduction	61
4.2	Sharing Lower Bound Information	62
4.3	Search Strategies for Mixed Integer Programming	63
4.3.1	Using Pseudocosts	63
4.3.2	Node Selection	75
4.3.3	Branch Selection	79
4.3.4	Computational Results	80
4.4	Pseudocosts in Parallel Branch and Bound	82
4.4.1	Observations	83
4.4.2	An Implementation Scheme	87

4.4.3	Previous Work	90
4.5	Node Selection in Parallel Branch and Bound	90
4.5.1	Observations	91
4.5.2	An Implementation Scheme	94
4.5.3	Previous Work	98
4.6	The PARINO System	102
4.6.1	PARINO Entities	103
4.6.2	PARINO Specifics	109
4.7	Computational Results	110
4.7.1	Preliminaries	110
4.7.2	Pseudocosts	119
4.7.3	Active Set Management	122
5	Parallel Branch and Cut	126
5.1	Effectively Using Cuts	126
5.1.1	How Often to Generate Cuts	128
5.1.2	Tailing Off	130
5.1.3	The Number of Cuts Per Round	131
5.1.4	Deleting Cuts from the Active Formulation	132
5.1.5	The Importance of a Cutting Plane	133
5.2	Issues in Cut Sharing	140
5.2.1	Cut Pools	141
5.2.2	Algorithmic Issues	143

5.2.3	Cut Sharing using Pools	148
5.3	Implementing Distributed Cut Management in PARINO	154
5.4	Computational Results	155
5.4.1	Default Algorithm	157
5.4.2	Test Suite	157
5.4.3	Basic Cut Sharing Paradigms	158
5.4.4	Improving on the Basic Cut Sharing Paradigms	163
6	Solving Set Partitioning Problems in Parallel	169
6.1	Routing	170
6.2	Parallelizing the Sequential Algorithm	172
6.2.1	Duplicate Column Removal	173
6.2.2	Dominant Row Identification	174
6.2.3	Solving the Linear Program and Finding Feasible Solutions .	176
6.2.4	Reduced Cost Fixing	177
6.2.5	Probing	178
6.2.6	Cut Generation	179
6.3	Computational and Control Issues	179
6.3.1	Algorithm Flow	179
6.3.2	Heuristics	184
6.3.3	Choosing Columns	185
6.4	PSP – A Parallel Code for Set Partitioning	186
6.5	Computational Experiments	187

6.5.1	A Test Suite of Problems	187
6.5.2	Speedup and Performance	190
7	Conclusions	195
7.1	Contributions	195
7.2	Future Research	197
A	Tables of Sequential Branch and Bound Experimental Results	199
B	Tables of Parallel Branch and Bound Experimental Results	218
C	Tables of Sequential Branch and Cut Experimental Results	227
D	Tables of Parallel Branch and Cut Experimental Results	236
	BIBLIOGRAPHY	250

LIST OF TABLES

4.1	Percentage of Fractional Integer Variables	66
4.2	Summary of Pseudocost Initialization Experiment	70
4.3	Summary of Pseudocost Update Experiment	73
4.4	Summary of Degradation Use Experiment	75
4.5	Summary of Node Selection Method Experiment	82
4.6	Relative Importance of Pseudocost Initialization	86
4.7	Relative Importance of Pseudocost Sharing	87
4.8	Statistics of All MIP Test Instances	112
4.9	Run Variance for Solved Instances	115
4.10	Run Variance for Unsolved Instances	116
4.11	Speedup of the Default Algorithm	117
4.12	Summary of Pseudocost Buffersize Experiment	120
4.13	Average Performance Ranking of Node Selection Schemes	123
4.14	Summary Statistics for Node Selection Methods Experiment – Solved Instances	124
4.15	Summary Statistics for Node Selection Methods Experiment – In- stances Sometimes Solved	125

5.1	Instances Used for Sequential Cut Strategy Study	129
5.2	Summary of Skip Factor Experiment	130
5.3	Summary of Weak Cut Deletion Experiment	132
5.4	Relationship Between γ and d – Estimate of β_1 and its Variance . .	136
5.5	Relationship Between γ and D – Estimate of β_1 and its Variance . .	138
5.6	Additional Instances Used for Cut Sharing Study	158
5.7	Summary of Basic Cut Sharing Experiment – Solved Instances . . .	159
5.8	Performance of Basic Cut Sharing Experiment – Unsolved Instances	160
5.9	Cut Generation and Waiting Times for Various Instances	160
5.10	Average Performance Ranking of Prefetch Sharing Schemes	164
5.11	Summary of Using Skip Factor in Master-Worker Sharing Scheme Experiment – Solved Instances	166
5.12	Summary of Using Skip Factor in Master-Worker Sharing Scheme Experiment – Unsolved Instances	166
6.1	Characteristics of Set Partitioning Problem Test Instances	189
6.2	PSP Performance on Solved Instances	191
6.3	Percentage of Time in Solution Phases	192
6.4	PSP Performance on Unsolved Instances	194
A.1	Comparison of Pseudocost Initialization Methods	199
A.2	Comparison of Pseudocost Update Methods	202
A.3	Comparison of Combination of Up and Down Pseudocost Information	204
A.4	Comparison of Node Selection Methods	207

B.1	Effect of Pseudocost Buffer Size. 16 Processors. Solved Instances. .	218
B.2	Effect of Pseudocost Buffer Size. 16 Processors. Instances Sometimes Solved.	220
B.3	Effect of Pseudocost Buffer Size. 16 Processors. Unsolved Instances.	220
B.4	Comparison of Parallel Pseudocost Initialization with Other Schemes. 16 Processors. Solved Instances.	221
B.5	Comparison of Parallel Pseudocost Initialization with Other Schemes. 16 Processors. Instances Sometimes Solved.	222
B.6	Comparison of Parallel Pseudocost Initialization with Other Schemes. 16 Processors. Unsolved Instances.	222
B.7	Comparison of Parallel Node Selection Methods. 8 Processors. . . .	223
B.8	Comparison of Parallel Node Selection Methods. 16 Processors. . . .	225
C.1	Effect of Varying the Skip Factor	227
C.2	Effect of Adding Multiple Cuts Per Round	230
C.3	Effect of Varying the Tailing Off Percentage	232
C.4	Effect of Deleting Weak Cuts	234
D.1	Comparison of Basic Cut Sharing Strategies. 4 Processors. Solved Instances.	236
D.2	Comparison of Basic Cut Sharing Strategies. 4 Processors. Unsolved Instances.	237
D.3	Comparison of Basic Cut Sharing Strategies. 8 Processors. Solved Instances.	238

D.4	Comparison of Basic Cut Sharing Strategies. 8 Processors. Unsolved Instances.	239
D.5	Comparison of Basic Cut Sharing Strategies. 16 Processors. Solved Instances.	240
D.6	Comparison of Basic Cut Sharing Strategies. 16 Processors. Unsolved Instances.	241
D.7	Comparison of Prefetch Sharing Schemes. 16 Processors. Solved Instances.	242
D.8	Comparison of Prefetch Sharing Schemes. 16 Processors. Unsolved Instances.	243
D.9	Effect of Varying Skip Factor in Master-Worker Sharing Scheme. 16 Processors. Solved Instances	244
D.10	Effect of Varying Skip Factor in Master-Worker Sharing Scheme. 16 Processors. Instances Sometimes Solved.	245
D.11	Effect of Varying Skip Factor in Master-Worker Sharing Scheme. 16 Processors. Unsolved Instances.	245
D.12	Effect of Varying Skip Factor in Broadcast-Distributed Sharing Scheme. 16 Processors. Solved Instances.	246
D.13	Effect of Varying Skip Factor in Broadcast-Distributed Sharing Scheme. 16 Processors. Instances Sometimes Solved.	247
D.14	Effect of Varying Skip Factor in Broadcast-Distributed Sharing Scheme. 16 Processors. Unsolved Instances.	247

D.15 Effect of Varying Skip Factor in No Sharing Scheme. 16 Processors.	
Solved Instances.	248
D.16 Effect of Varying Skip Factor in No Sharing Scheme. 16 Processors.	
Instances Sometimes Solved.	249
D.17 Effect of Varying Skip Factor in No Sharing Scheme. 16 Processors.	
Unsolved Instances.	249

LIST OF FIGURES

2.1	Single Node Variable Upper Bound Flow Model.	19
3.1	A Ring-Connected Parallel Computer Topology	39
3.2	An Assignment of Grid Points to Processors for Jacobi Iteration . .	41
3.3	A Task Graph of One Jacobi Iteration	43
3.4	A Gantt Chart of One Jacobi Iteration	43
3.5	Superlinear Speedup in Branch and Bound	49
3.6	NAYLAK System Architecture	59
4.1	Characteristics of All Sequential Computational Experiments	65
4.2	Observed Pseudocosts as a Function of Number of Branches	68
4.3	Node Selection Rules Investigated	80
4.4	A Branch and Bound Tree	84
4.5	PARINO System Architecture	105
4.6	Default PARINO Settings	111
5.1	Default Cutting Plane Strategy for Sequential Experiments	128
5.2	Relationship Between γ and d for Test Instances	135

5.3	Relationship Between γ and D for Test Instances	137
5.4	The Percentage of Time in Which Cuts Are More Useful at Descendant Nodes and Nondescendant Nodes as a Function of D	139
5.5	Cut Pools	143
5.6	PARINO System Architecture	156
5.7	Default PARINO Cut Management Settings	157
6.1	A Vehicle Routing Problem and its Set Partitioning Formulation	171
6.2	Set Partitioning Heuristic	181
6.3	Main Control Decision	183
6.4	Settings for PSP	190

SUMMARY

A mixed integer program (MIP) is the problem of maximizing a linear function over linear constraints subject to integrality restrictions on some or all of the variables. A rich class of real world problems can be modeled as MIPs, so many advanced techniques have been developed for their solution. From a computational standpoint, some of the most effective algorithms are linear programming based branch and bound algorithms. It is these algorithms on which we will focus.

We study how to combine the power of linear programming based techniques for solving MIP with the power of parallel computing. In a wide range of scientific fields, the introduction of parallel computers consisting of many microprocessors has made possible the solutions of problems impossible to consider solving on a single processor. The field of Optimization should be no exception. This is a computationally oriented thesis, in which we build two separate parallel optimization codes. Each code solves a specific type of MIP and investigates issues related to the parallelization of sophisticated solution techniques for this particular problem type. The most interesting parallelization issues occur when the processors must communicate by message passing, so we focus our algorithm design for these architectures.

The core issue that must be addressed is how to deal with the *globally useful information* that the algorithm generates. There is a tradeoff between complete sharing of information between processors, requiring potentially significant overhead, and little sharing of information, where potentially valuable information is lost. We give a categorization of ways in which information might be shared, and discuss the advantages and disadvantages of each. The manner in which information is best shared depends in large part on the underlying importance of the information. Therefore, we undertake a number of studies aimed at determining the importance of the types of information used in advanced MIP solvers such as pseudocosts and cutting planes. Our research showed that the proper initialization of pseudocosts is extremely important for an effective algorithm, that averaging pseudocost observations is only mildly effective, and that cutting planes generated near the top of the branch and bound tree are the most important. Each of these observations impacts the design of our parallel algorithm.

Armed with this knowledge, we developed PARINO, a powerful, flexible parallel branch and cut system for solving general MIPs. Using PARINO, we are able to verify our conjectures about how to best use pseudocost and cutting plane information in a parallel branch and cut algorithm. Also, we suggest different schemes for distributed active set management and node selection in a parallel algorithm. Depth-first oriented search methods are very powerful in a distributed setting.

The second code (PSP) is specialized to solve the Set Partitioning Problem

(SPP). The SPP is a MIP with wide applicability in areas such as scheduling and routing. A variety of techniques have been developed in order to exploit the problem structure of SPP resulting in the ability to solve much larger instances of SPP than of MIP. Our implementation is carefully designed to exploit parallelism to greatest advantage in advanced techniques like preprocessing and probing, primal heuristics, and cut generation. In addition, a primal-dual subproblem algorithm is used for solving the linear programming relaxation. This method outperforms standard simplex algorithms, reduces memory requirements, and breaks the linear programming solution process into natural phases from which we can exploit information to find good solutions on the various processors. Implications from the probing operation are another type of globally useful information that are shared among the processors. Combining these techniques allows us to obtain solutions to extremely large and difficult problems in a reasonable amount of computing time.

CHAPTER 1

Introduction

A mixed integer program (MIP) can be stated mathematically as follows:

$$\begin{aligned} \text{Maximize} \quad & z_{MIP} = \sum_{j \in I} c_j x_j + \sum_{j \in C} c_j x_j \\ \text{subject to} \quad & \sum_{j \in I} a_{ij} x_j + \sum_{j \in C} a_{ij} x_j \sim b_i \quad i = 1, \dots, m \\ & l_j \leq x_j \leq u_j \quad j \in N \\ & x_j \in \mathcal{Z} \quad j \in I \\ & x_j \in \mathcal{R} \quad j \in C, \end{aligned}$$

where \sim denotes one of \leq , \geq , or $=$, I is the set of integer variables, C is the set of continuous variables, and $N = I \cup C$. The lower and upper bounds l_j and u_j may take on the values of plus or minus infinity.

We refer to $\sum_{j \in N} c_j x_j$ as the *objective function*. A *feasible solution* is any vector x that satisfies all of the constraints specifying MIP, and the *feasible region* is the set of all feasible solutions. A feasible solution with the largest objective function value is called the *optimal solution*.

Consider the optimization problem obtained by relaxing the integrality requirements on the variables of MIP. The resulting linear program, called the LP relaxation of MIP (RMIP), provides an upper bound on the value of the optimal solution:

$$\begin{aligned}
 \text{Maximize} \quad & z_{LP} = \sum_{j \in I} c_j x_j + \sum_{j \in C} c_j x_j \\
 \text{subject to} \quad & \sum_{j \in I} a_{ij} x_j + \sum_{j \in C} a_{ij} x_j \leq b_i \quad i = 1, \dots, m \\
 & l_j \leq x_j \leq u_j \quad j \in N \\
 & x_j \in \mathfrak{R} \quad j \in C \cup I.
 \end{aligned}$$

In a “branch and bound” solution approach to MIP, the variables are systematically set to integral values, resulting in a number of linear programs. From the solution of these linear programs an upper bound on the value of the optimal solution can be derived. Any feasible integral solution to the linear program provides a lower bound on what the true optimum may be. In this way, a series of valid upper and lower bounds on the true optimum value are obtained. The process is complete when these two bounds are the same or their difference is less than some prescribed tolerance.

The introduction of parallel computers consisting of many microprocessors has made possible the solutions of problems impossible to consider solving on a single processor. A few examples of such problems are computational fluid dynamics problems (such as those arising in aerodynamics and weather prediction), image

processing, pattern recognition, artificial intelligence, finite element modeling, and search problems [72] [105]. The fastest computers existing in the world today are parallel machines [33].

Branch and bound is a natural paradigm to map to a multi-processor computer, since exploration of the search space consists of evaluating many independent portions of the space corresponding to different sets of variables that are fixed at their bounds.

Many advanced mathematical techniques have been developed that are able to improve the upper and lower bounding procedures for each subproblem. Among these techniques are logical preprocessing [110], pseudocosts [14], and cutting planes [48] [57] [99]. When cutting planes are incorporated into the branch and bound procedure, the resulting algorithm is called *branch and cut* [100]. These techniques, in conjunction with faster and faster sequential machines, have made possible the solution of integer programs a few years ago thought unsolvable.

A very important special case of MIP is the Set Partitioning Problem (SPP). The SPP is the case of MIP where we have $a_{ij} \in \{0,1\} \forall i = 1, \dots, m, j \in N$, $b_i = 1 \forall i = 1, \dots, m$, $C = \emptyset$, $l_j = 0$ and $u_j = 1 \forall j \in N$, and the sense of each constraint is $=$. A large number of real-life problems, including vehicle routing, airline crew scheduling, and airline crew recovery can be formulated as SPPs, so the problem has received a good deal of study [62] [5] [81]. Instances of practical problems modeled in this way tend to be very large. Specialized logical processing, cutting planes, and heuristics for this problem allow the solution of much larger

instances of SPPs than for general MIPs.

Even though the power and importance of using advanced mathematical techniques to solve difficult instances of MIP is documented, the issues involved in marrying these techniques to the parallel computer is not well studied. The aim of this research is to show the viability of combining the advanced mathematical techniques used in solving difficult integer programs and parallel computation. Many issues will be discussed and tested, such as how work should be allocated among the processors and how much information should be shared among the processors depending on the problem type to be solved.

Because the parallel solution techniques we employ are quite different, MIP and SPP are good prototypical problems on which to study the effects of combining parallel processing and optimization. Our solution approach to MIP can naturally be broken into relatively large chunks of computation, which is called *coarse-grained* parallelism. The algorithm for solving SPP is decomposed into smaller units of computation, which is called *fine-grained* parallelism. From studying the effects of combining optimization with these two fundamentally different types of parallelism, we can gain insight as to how to parallelize other advanced optimization techniques.

1.1 Literature Review

Since branch and bound is a natural idea to use on parallel computers, there is a wealth of documentation on approaches and experiences with parallel branch and

bound, dating back to the 1970's. In an historical note [103], Pruul, Nemhauser, and Rushmeier report results of a simulated parallel branch and bound algorithm for solving the traveling salesman problem. Given the vast amount of research done on parallel branch and bound, there is surprisingly little that focuses specifically on the general mixed integer program. For a survey of the body of work on parallel branch and bound, the reader is referred to the article of Gendron and Crainic [46].

Boehning *et al.* [18] present an implementation of a simple linear programming based branch and bound algorithm for solving MIP, but incorporate no “advanced” techniques into the procedure. Ashford *et al.* [3] present a more sophisticated branch and bound algorithm for a transputer network of up to eight nodes. Eckstein [37] [38] was among the first to write an industrial strength parallel branch and bound code for general mixed integer programming. The main enhancements he added over a naive implementation were reduced cost fixing, a primal heuristic, and pseudocost based branching. Eckstein's early work made use of many of the advanced features of the particular architecture on which he was working. His later work [40] has shown that these features are generally not necessary to obtain effective parallel algorithms. Laundry [80] has described the parallel implementation of the XPRESS-MP commercial IP solver. A main emphasis of this work is to build a *fault-tolerant* system – a system that works if some of the processors become unavailable during the algorithm's execution. In order to build fault-tolerance into the system, the parallelization strategy used is somewhat simple. Experimental results on a network of up to four workstations are reported. Hajian *et*

al. [56] describe a novel, multi-stage parallelization scheme for solving MIP. In the first stage, all processors begin searching in (possibly overlapping) portions of the search space. The second stage resembles the more “classical” branch and bound approach. Limited results on a configuration of up to thirty workstations are reported.

Cannon and Hoffman [22] were the first to report results of a parallel branch and cut implementation. Their code performed nearly all of the advanced integer programming techniques described in Section 2.1. Because parallel computing tools were not as developed as today, Cannon and Hoffman resorted to operating system constructs in order to perform many of the required parallelization tasks. They report results on a small suite of problems on up to eight processor configurations. Bixby *et al.* [15] use a parallel software platform called TreadMarks [68] in order to parallelize the branch and cut algorithm. The use of packages such as TreadMarks to build parallel software systems will be discussed in Chapter 3. A large number of computational results are reported on a parallel system of eight workstations. Homeister [63] gives a relatively simple implementation of the branch and cut algorithm that does not rely on advanced operating system features or on outside packages such as TreadMarks. Jünger and Störmer [66] describe an implementation of a parallel branch and cut for the traveling salesman problem. Their work is distinctive, since a number of processors are dedicated to tasks such as performing heuristics and managing the cutting planes in addition to processors exploring the branch and bound search space. The computational results presented, on up

to 64 processors, indicate that parallelism is mildly effective in helping solving the problems by branch and cut. Ralphs [104] describes a parallel branch and cut implementation and uses this implementation to solve the capacitated vehicle routing problem. Ladányi [76] extends the ideas described by Ralphs. The work of Ralphs and Ladányi is geared for specific combinatorial problems, but many of the ideas they develop are similar to ours for solving MIP in a general form.

Very little research has been done on parallelizing sophisticated algorithms for the set partitioning problem. Smith and Thompson [112] present a parallel version of the column subtraction method of Harche and Thompson [58], and Levine [82] describes a parallel genetic algorithm approach in his thesis. However, the column subtraction method and genetic algorithm for solving the set partitioning problem are generally quite inferior to a branch and cut approach, like the one presented by Hoffman and Padberg [62]. Being a branch-and-cut approach, the method of Hoffman and Padberg could be parallelized by any of the branch and cut schemes mentioned in this section. Our parallelization approach is different.

1.2 Thesis Outline

In Chapter 2, we explain the advanced solution techniques we use in solving integer programs. Techniques for solving both the general MIP and the SPP are described.

Chapter 3 gives an introduction to parallel computing. The two main categories of parallel computers are explained, and we provide a justification for gearing our algorithms towards one of these two architecture categories. The issues and ad-

vantages of designing an algorithm for this type of parallel computer are discussed. The main question that must be addressed is how to handle information generated by the processors. A categorization of ways in which information may be shared among processors is given. Finally, we describe a software system on top of which our algorithms are built.

In Chapter 4, we investigate issues relating specifically to parallel branch and bound for solving MIP. First, basic search strategy concepts are presented and analyzed, including a computational study in a sequential environment from which we draw conclusions on which to base our parallel implementation. The next sections make some basic observations about performing the sequential search techniques in parallel. From these observations, we describe general implementation schemes for effective pseudocost usage and active set management in parallel branch and bound. Next, we describe PARINO, our software package that implements our parallel branch and bound scheme for solving MIP. Finally, a test suite of problems is introduced and computational results testing and comparing the concepts presented are reported.

Chapter 5 deals with the issues in parallelizing a branch and cut algorithm. We are mostly interested in how to effectively share cutting planes in a parallel branch and cut algorithm. We begin with a study of the importance of cutting planes and cut management issues in a sequential algorithm. Next, we describe a framework for investigating many cut sharing schemes. We next show how the cut sharing schemes are implemented in PARINO. Extensive computational results comparing

cut sharing schemes are presented.

In Chapter 6, we present a parallel linear programming based heuristic for solving large scale set partitioning problems. The parallelization strategy for this algorithm is quite different than for the branch and bound and branch and cut algorithms. We discuss the computational and control issues that arise, and we give computational results for our algorithm.

In the last chapter, we conclude with the main contributions of this thesis and provide suggestions for future research.

CHAPTER 2

Integer Programming Preliminaries

2.1 Mixed Integer Programming

Since one of the goals of this research is to build an industrial strength mixed integer optimizer using a parallel processing machine, we will briefly highlight the main features of such a system. We assume that the reader is familiar with most of these concepts and include a discussion here merely for self-containment. A full treatment of these topics can be found in the text by Nemhauser and Wolsey [94].

2.1.1 Branch and Bound

The term “branch and bound” was originally coined by Little *et al.* [86] in their study of such an algorithm to solve the traveling salesman problem. However, the idea of using a branch and bound algorithm for integer programming using linear programming relaxations was proposed somewhat earlier by Land and Doig [79]. The process involves keeping a list of linear programming problems obtained by

relaxing some or all of the integer requirements on the variables x_j , $j \in I$. To precisely define the algorithm, let us make a few definitions. We use the term *node* or *subproblem* to denote the problem associated with a certain portion of the feasible region of MIP. Define z_L to be a lower bound on the value of z_{MIP} . For a node N^i , let z_U^i be an upper bound on the value that z_{MIP} can have in N^i . The list \mathcal{L} of problems that must still be solved is called the *active set*. Denote the optimal solution by x^* . Algorithm 2.1 is an LP-based branch and bound algorithm for solving MIP.

Algorithm 2.1 The Branch and Bound Algorithm

0. **Initialize.**

$\mathcal{L} = \text{RMIP}$. $z_L = -\infty$. $x^* = \emptyset$.

1. **Terminate?**

Is $\mathcal{L} = \emptyset$? If so, the solution x^* is optimal.

2. **Select.**

Choose and delete a problem N^i from \mathcal{L} .

3. **Evaluate.**

Solve the LP relaxation of N^i . If the problem is infeasible, go to step 1, else let z_{LP}^i be its objective function value and x^i be its solution.

4. **Prune.**

If $z_{LP}^i \leq z_L$, go to step 1. If x^i is fractional, go to step 5, else let $z_L = z_{LP}^i$, $x^* = x^i$, and delete from \mathcal{L} all problems with $z_U^j \leq z_L$. Go to step 1.

5. **Divide.**

Divide the feasible region of N^i into a number of smaller feasible regions $N^{i1}, N^{i2}, \dots, N^{ik}$ such that $\cup_{j=1}^k N^{ij} = N^i$. For each $j = 1, 2, \dots, k$, let $z_U^{ij} = z_{LP}^i$ and add the problem N^{ij} to \mathcal{L} . Go to 1.

This description makes it clear that there are various choices to be made during

the course of the algorithm. Namely, how do we select which node to evaluate, and how do we divide the feasible region. A partial answer to these two questions is now provided.

Problem Division

For MIP, the obvious way to divide the feasible region of N^i is to choose a variable j that has fractional value x_j^i in the solution to the LP relaxation of N^i and impose the constraint $x_j \leq \lfloor x_j^i \rfloor$ to define one subregion and $x_j \geq \lceil x_j^i \rceil$ to define another subregion. We call this branching on a *variable dichotomy*, or simply branching on a variable.

If there are many fractional variables in the solution to the LP relaxation of N^i , we must choose one variable to define the division. Because the effectiveness of the branch and bound method strongly depends on how quickly the upper and lower bounds converge, we would like to branch on a variable that will improve these bounds. It seems difficult to select a branching variable that will affect the lower bound. Doing this amounts to heuristically finding feasible solutions to MIP and is very problem specific. However, there are ways to attempt to predict which fractional variables will most improve the upper bound when required to be integral. One of the most effective ways involves the use of *pseudocosts* [14] [44].

Pseudocosts work as follows: with each integer variable x_j , we associate two quantities P_j^- and P_j^+ that attempt to measure the per unit decrease in objective function value if we fix x_j to its rounded down value and rounded up value, re-

spectively. Suppose that $x_j^i = \lfloor x_j^i \rfloor + f_j^i$, with $f_j^i > 0$. Then by branching on x_j , we will estimate a decrease of $D_j^{i-} = P_j^- f_j^i$ on the down branch of node i and a decrease of $D_j^{i+} = P_j^+(1 - f_j^i)$ on the up branch of node i . Bénichou *et al.* call the values P_j^- and P_j^+ *down* and *up pseudocosts* [14].

One way to obtain values for P_j^- and P_j^+ is to simply observe the true degradation in objective function value. Let N^{i-} and N^{i+} denote the nodes for the down and up branches of node N^i , then the pseudocosts are computed as

$$P_j^- = \frac{z_{LP}^{i-} - z_{LP}^i}{f_j^i} \quad P_j^+ = \frac{z_{LP}^{i+} - z_{LP}^i}{1 - f_j^i}. \quad (2.1)$$

Driebeek [34] gave a method for determining lower bounds on the degradations by implicitly performing one dual simplex pivot.

Node Selection

We now deal with the “select” portion of the branch and bound algorithm. When we make a decision to branch, we are solely concerned with maximizing the change in z_{LP}^i between a node N^i and its children. In selecting a node, our purpose is twofold: to find good integer feasible solutions or to prove that no solution better than our current one with value z_L exists. Therefore, the quality of the current solution value z_L and an estimate of the optimal solution z_E are important criteria in determining which node to select for evaluation. We will motivate and describe various node selection rules in Section 4.3.2. Some node selection rules use pseudocosts to calculate z_E , so pseudocosts are important for both the branching and node selection decisions of the branch and bound algorithm for MIP. For an

in depth discussion of many search strategies for mixed integer programming, see the paper by Linderoth and Savelsbergh [85].

2.1.2 Other Algorithmic Enhancements

Logical Preprocessing

Preprocessing and probing techniques for mixed integer programming are ways of tightening the linear programming relaxation of MIP. Using these techniques, infeasible instances can be recognized, redundant constraints can be removed, variables' bounds can be improved (variables may even be fixed), and matrix coefficients can be improved. An extended discussion, including more advanced preprocessing and probing techniques, can be found in the paper of Savelsbergh [110]. Preprocessing and probing techniques play an especially crucial role in solving SPP, and we will discuss the techniques more fully in Section 2.2.1.

Primal Heuristic

A primal heuristic is a procedure that attempts to generate a feasible integral solution, and hence a valid lower bound. Many heuristics have been developed for specific optimization problems [25] [47]. Indeed, we will discuss three “problem-specific” heuristics in to solving SPP in Section 2.2.4.

Far fewer heuristics have been developed for solving MIP in a general form. Notable exceptions include the pivot and complement heuristic of Balas and Martin [10] and the OCTANE heuristic of Balas *et al.* [9]. One simple, yet effective

heuristic is known as the *diving* heuristic. In the diving heuristic, some integer variables are fixed and the linear program resolved. The fixing and resolving is iterated until either an integral solution is found or the linear program becomes infeasible.

Reduced Cost Fixing

After the LP-relaxation of MIP is solved, the reduced costs of nonbasic binary variables can be used to attempt to fix binary variables for the subtree rooted at that node. Let \bar{c}_j be the reduced cost of a nonbasic binary variable x_j and node N^i . If $x_j = l_j$ and $z_{LP}^i + \bar{c}_j \leq z_L$, then x_j can be fixed to l_j for all nodes in the subtree rooted at N^i . Likewise, if $x_j = u_j$ and $z_{LP}^i - \bar{c}_j \leq z_L$, then x_j can be fixed to u_j . If node N^i is the root node of the branch and bound tree, we call this procedure *global reduced cost fixing*.

2.1.3 Cutting Planes

An inequality $\sum_{j \in N} \pi_j x_j \leq \pi_0$ is called a *valid inequality* if the inequality is satisfied by all feasible solutions to MIP. For a given LP relaxation of MIP, there may exist many valid inequalities that cut away portions of the feasible region of LP without cutting any portion of the feasible region of MIP. Adding these inequalities to the LP relaxation provides a tighter upper bound on the value of z_{MIP} . Branch and cut is the name for a systematic procedure for adding valid inequalities to tighten the linear programming relaxation of a problem. Branch and cut has allowed the

solution of many difficult MIP problems unsolvable by other techniques [29] [108] [7] [62] [27].

Branch and cut is a simple extension to the basic branch and bound algorithm. The sole difference in the two occurs at the **evaluate** stage. To evaluate a node in branch and cut, we solve the linear programming relaxation of a given problem and then look for valid inequalities that exclude the current linear programming solution. The problem of finding such valid inequalities is called the *separation problem*. The valid inequalities are added to the formulation and the linear program resolved. Algorithm 2.2 is a generic branch and cut algorithm.

The strongest types of valid inequalities are called *facet defining inequalities*. Many researchers have studied structures that are common to many instances of MIP and discovered facet defining inequalities for these structures. In the following paragraphs, we describe a few such special structures and their associated cutting planes.

Often MIP has rows that are of the form

$$\sum_{j \in B} a_j x_j \leq b, \tag{2.2}$$

where $B \subseteq I$ is the set of binary variables of MIP. Considering only (2.2), we have a relaxation to MIP. Inequalities valid to a relaxation of the problem must also be valid for the original problem. Thus, an important class of valid inequalities for MIP can be derived from the set of feasible solutions to a 0-1 knapsack problem:

$$S = \{x \in \{0, 1\}^{|B|} : \sum_{j \in B} a_j x_j \leq b\}.$$

Algorithm 2.2 The Branch and Cut Algorithm

0. Initialize.

$\mathcal{L} = \text{MIP}$. $z_L = -\infty$. $x^* = \emptyset$.

1. Terminate?

Is $\mathcal{L} = \emptyset$? If so, the solution x^* is optimal.

2. Select.

Choose and delete a problem N^i from \mathcal{L} .

3. Evaluate.

Solve the LP relaxation of N^i . If the problem is infeasible, go to step 1, else let z_{LP}^i be its objective function value and x^i be its solution. Solve the separation problem to find valid inequalities violated by x^i . If no valid inequalities are found, go to 4, else add the inequalities to the description of N^i and go to 3.

4. Prune.

If $z_{LP}^i \leq z_L$, go to step 1. If x^i is fractional, go to step 5, else let $z_L = z_{LP}^i$, $x^* = x^i$, and delete from \mathcal{L} all problems with $z_U^j \leq z_L$. Go to step 1.

5. Divide.

Divide the feasible region of N^i into a number of smaller feasible regions $N^{i1}, N^{i2}, \dots, N^{ik}$ such that $\cup_{j=1}^k N^{ij} = N^i$. For each $j = 1, 2, \dots, k$, let $z_U^{ij} = z_{LP}^i$ and add the problem N^{ij} to \mathcal{L} . Go to 1.

Many researchers have studied the underlying polyhedral structure and found facets of S [6] [57] [119].

A set $C \subseteq B$ is called a *cover* if $\sum_{j \in C} a_j > b$. A cover C is *minimal* if there does not exist a $k \in C$ such that $C \setminus \{k\}$ is also a cover. Consider a partition of a minimal cover C into disjoint sets C_1 and C_2 with $C_1 \neq \emptyset$. A facet-inducing *lifted cover inequality* for S is given by

$$\sum_{j \in C_1} x_j + \sum_{j \in B \setminus C} \alpha_j x_j + \sum_{j \in C_2} \gamma_j x_j \leq |C_1| - 1 + \sum_{j \in C_2} \gamma_j.$$

The coefficients α_j and γ_j are called *lifting coefficients*. The details of their computation is beyond the scope of the thesis. The interested reader is referred to the paper by Gu *et al.* [53] and to the book of Nemhauser and Wolsey [94].

Another important structure found or derived in many MIP problems is that of a variable upper bounded single node flow model. We have a network on which the flow along an arc is bounded if the arc is open, or the flow is zero if the arc is closed. In addition, we have the restriction that the net flow out of a node in the network is also bounded. This situation is described pictorially in Figure 2.1.

The flow model was first investigated by Padberg, Van Roy, and Wolsey [99], and they found many classes of useful inequalities. These inequalities were used by Van Roy and Wolsey [108] to solve very difficult instances of mixed integer programs. Algebraically, we can model such a situation using the mixed integer region

$$S = \{(x, y) \in R_+^n \times \{0, 1\}^n : \sum_{j \in N^+} x_j - \sum_{j \in N^-} x_j \leq d, x_j \leq m_j y_j, j \in N\},$$

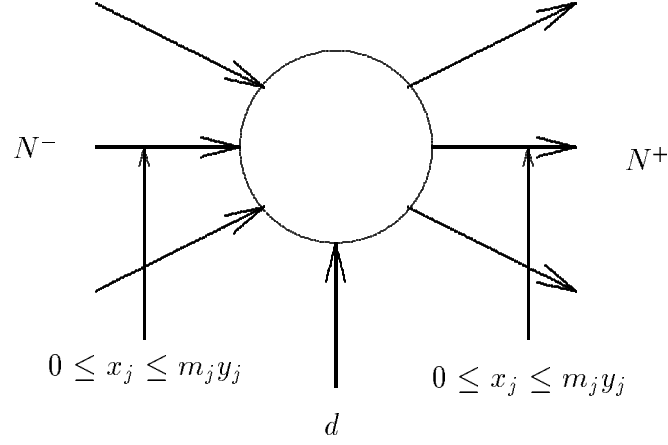


Figure 2.1: Single Node Variable Upper Bound Flow Model.

where $N = N^+ \cup N^-$ and $n = |N|$.

A set $C = C^+ \cup C^-$ is called a *flow cover* if $C^+ \subseteq N^+$, $C^- \subseteq N^-$ and $\sum_{j \in C^+} m_j - \sum_{j \in C^-} m_j > d$.

For any flow cover C , the inequality

$$\sum_{j \in C^+} x_j + \sum_{j \in C^{++}} (m_j - \lambda)(1 - y_j) \leq d + \sum_{j \in C^-} m_j + \sum_{j \in L^-} \lambda y_j + \sum_{j \in L^{--}} x_j,$$

where $\lambda = \sum_{j \in C^+} m_j - \sum_{j \in C^-} m_j - d$, $C^{++} = \{j \in C^+ : m_j > \lambda\}$, $L^- \subseteq (N^- \setminus C^-)$ and $m_j > \lambda$ for $j \in L^-$, and $L^{--} = N^- \setminus (L^- \cup C^-)$, is called a *simple generalized flow cover inequality* and is valid for S .

Just as the case with knapsack cover inequalities, this inequality can be strengthened through a procedure known as lifting. When we do so, we obtain an inequality called the *lifted simple generalized flow cover inequality*. The full details of obtaining such an inequality are given by Gu *et al.* [52].

Separation

Given a fractional solution x^* to the solution of the LP relaxation at a node, the *separation problem* for a class of inequalities is the problem of finding an inequality of this class that is violated by x^* . We will now briefly discuss the separation problem for the classes of inequalities we have introduced as well as discuss lifting violated inequalities.

Knapsack Covers Given a fractional point x^* , the separation problem for cover inequalities seeks a subset of the variables C with $\sum_{j \in C} a_j > b$ and $\sum_{j \in C} x_j^* > |C| - 1$. Introduce a characteristic vector $z \in \{0, 1\}^{|B|}$ to denote the variables in the cover C . We seek a z such that $\sum_{j \in B} a_j z_j > b$ and $\sum_{j \in B} x_j^* z_j > \sum_{j \in B} z_j - 1$. Through simple algebraic manipulation, the separation problem (KSP) becomes

$$\xi = \min\left\{\sum_{j \in B} (1 - x_j^*) z_j \mid \sum_{j \in B} a_j z_j > b, z \in \{0, 1\}^{|B|}\right\}.$$

If $\xi < 1$, then the inequality $\sum_{j \in C} x_j \leq |C| - 1$ cuts off the point x^* . KSP is a knapsack problem which is known to be NP-Complete [70]. Thus, we solve the separation problem heuristically. The exact details of the heuristic procedure to solve the separation problem and the computational of the lifting coefficients can be found in Gu *et al.* [51]. The main point to be made is that since the separation problem is solved heuristically, there may be cover inequalities that will cut off x^* but that are not found by our separation procedure.

Flow Covers

Nemhauser and Wolsey [94] show that the separation problem for generalized flow cover inequalities is a nonlinear integer program – an extremely difficult prob-

lem to solve from a computational point of view. They also suggest relaxing the (difficult) true separation problem to

$$\begin{aligned} \xi &= \max\{\sum_{j \in N^+} (x_j^* - 1)\alpha_j + \sum_{j \in N^-} x_j^* \beta_j\} \\ &\text{subject to } \sum_{j \in N^+} a_j \alpha_j - \sum_{j \in N^-} a_j \beta_j > b \\ &\alpha \in \{0, 1\}^{|N^+|} \quad \beta \in \{0, 1\}^{|N^-|}, \end{aligned}$$

where α is the characteristic vector of the set C^+ and β is the characteristic vector of the set C^- . If $(\sum_{j \in C^+} a_j - \sum_{j \in C^-} a_j - b)(\xi - (\sum_{j \in N^-} x_j^* - 1)) > 0$, then there is a violated generalized flow cover inequality. The relaxed separation problem is a knapsack problem, so like in the case of knapsack covers, a heuristic procedure is used for its solution. Since the separation procedure for flow cover inequalities is to heuristically solve a relaxed version of the true separation problem, it is quite likely that there are flow cover inequalities that cut off x^* but that are not found by the separation procedure. The full details of solving the lifting problem and computing the lifting coefficients for flow cover inequalities can be found in the work of Gu [50] and Gu *et al.* [52].

2.2 Set Partitioning Problem

We now turn our attention to the second problem under study, the Set Partitioning Problem (SPP). SPP is the special case of MIP, where $A \in \{0, 1\}^{m \times n}$ and $x \in \{0, 1\}^n$. It is written formally as

$$\min\{c^T x : Ax = 1, x \in \{0, 1\}^n\}.$$

Balas and Padberg [11] give a survey of some applications and early solution methods. Hoffman and Padberg [62] give a branch and cut approach that is very successful in solving airline crew scheduling SPPs. Borndorfer [19] gives a branch and cut approach to solving SPPs arising from handicapped bus scheduling. A heuristic combining ideas of Hoffman and Padberg with those of Wedelin is given by Atamtürk *et al.* [5].

Due to the wide practical applicability of the SPP, solving large, difficult SPPs in a short amount of time is important. Our goal is to show how to exploit parallelism in a linear programming based solution procedure for solving SPP. The procedure is very similar to that presented by Atamtürk *et al.* [5]. In particular, both of the procedures are heuristics. Thus, finding optimal solutions to the SPP is not our goal, but rather to find provably good feasible solutions in a reasonable amount of time.

In the remainder of this section, we will describe the components of the heuristic procedure.

2.2.1 Preprocessing and Probing

A number of authors (in particular Borndorfer [19]) suggest simple methods for identifying variables that may be fixed and rows that may be removed from the problem. We perform three such preprocessing methods.

Duplicate Column Removal

In many applications, the way in which the columns of A are generated does not ensure that they are unique. If duplicate columns exist, obviously we need keep only the one with minimum cost. Because the number of columns is typically quite large, performing a pairwise comparison of columns in order to find duplicates is inefficient. Instead, a random hash function is used [71]. For each $i = 1, \dots, m$, let u_i be an integer chosen uniformly from the interval $[0, 2^{32})$. The hash value for each column A_j $j = 1, \dots, n$ is computed as

$$h(A_j) = \left(\sum_{i=1}^m u_i a_{ij} \right) \bmod 2^{32}.$$

If $h(A_j) = h(A_k)$ for two columns $A_j, A_k, j \neq k$, then a pairwise comparison is done to determine if columns A_j and A_k are really duplicates.

Dominant Row Removal

For each row $i = 1, \dots, m$, let $T(i) = \{j : a_{ij} = 1\}$. If $T(i) \subseteq T(j)$ for two rows $i \neq j$, we say that row i *dominates* row j . If row i dominates row j , then row j is redundant to the formulation and can be removed. In addition, we may set $x_k = 0 \forall k \in T(j) \setminus T(i)$. In order to determine dominant rows, a pairwise comparison is performed. Variables appearing in a row are stored in increasing order of their indices so that non-dominance can be detected as early as possible when comparing two rows.

Probing

As mentioned in Section 2.1, probing is a general technique used in solving MIP. Probing plays a crucial role in our heuristic for SPP so we will briefly describe it here in the context of SPP. For SPP, probing is performed by tentatively setting a variable to one and observing the logical implications. The logical implications deduced from probing on variable x_j take the form

$$x_j = 1 \Rightarrow x_k = 0 \text{ or } x_j = 1 \Rightarrow x_k = 1.$$

Many implications not immediately evident from the constraint matrix may be deduced in this way – in fact, probing may lead to a logical contradiction resulting in the fixing of variables. In Example 2.1, by probing on variable x_1 , we can deduce that $x_1 = 0$ in all feasible solutions.

Probing is one of the main techniques used in constraint programming [116], which is an effective method for solving tightly constrained scheduling problems. Because obtaining a feasible solution to an instance to SPP can be a very difficult problem, we would expect that probing is effective for SPP as well.

2.2.2 Solving the Linear Program

In order to solve the linear programming relaxation of the SPP, Hu and Johnson describe a technique called the *primal-dual subproblem simplex method* [64]. For problems with few rows and many columns (as is the case for our instances of SPP), the primal-dual subproblem simplex method has been shown to be more

$$x_1 + x_2 + x_4 = 1 \quad (\text{R1})$$

$$x_2 + x_3 = 1 \quad (\text{R2})$$

$$x_1 + x_3 + x_4 = 1 \quad (\text{R3})$$

$$x_1 = 1 \Rightarrow x_2 = 0 \quad \text{by (R1)}$$

$$\Rightarrow x_3 = 1 \quad \text{by (R2)}$$

$$\Rightarrow x_4 = 0 \quad \text{by (R3)}$$

- x_1 may be fixed at 0

Example 2.1: Fixing Variables by Probing

effective than the standard simplex method and the primal subproblem simplex method (or SPRINT approach) developed by John Forrest and described by Anbil *et al.* [2]. We now briefly discuss the primal-dual subproblem simplex method and its application to the set partitioning problem.

Consider the linear programming relaxation (P) to SPP and its dual (D).

$$\begin{array}{ll}
 \text{(P): } \min & c^T x \\
 \text{(s.t.) } & Ax = 1 \\
 & x \geq 0
 \end{array}
 \qquad
 \begin{array}{ll}
 \text{(D): } \max & 1^T \pi \\
 \text{(s.t.) } & \pi^T A \leq c
 \end{array}$$

We will solve a sequence of subproblems where only a subset of the columns are considered. Let $K \subseteq \{1, 2, \dots, n\}$ be the index set of columns considered in a subproblem, and let A_K , c_K , and x_K be the restrictions of A , c , and x to K . We also use the notation $\bar{c}^\pi \equiv c - \pi^T A$ to denote the reduced costs with respect to a dual solution π .

A primal optimal solution to a subproblem, extended by adding 0 to the columns not in the subproblem, is a primal feasible solution for (P). However, a dual optimal solution for the subproblem is usually not feasible for (D). If it is feasible for (D), then it is also optimal.

Let $\mathcal{F}(\mathcal{P})$ be the feasible solution set of (P) and $\mathcal{F}(\mathcal{D})$ be the set of feasible solutions for (D). Given $x \in \mathcal{F}(\mathcal{P})$, $\pi \in \mathcal{F}(\mathcal{D})$, Algorithm 2.3 is a basic description of the primal-dual subproblem simplex method. If $x \in \mathcal{F}(\mathcal{P})$ is not known, then a two-phase approach using artificial variables is used [26]. In our applications, we

have $c \geq 0$, so $\pi = \mathbf{0} \in \mathcal{F}(\mathcal{D})$.

Algorithm 2.3 The Primal-Dual Subproblem Simplex Method

1. Let $x \in \mathcal{F}(\mathcal{P})$, $\pi \in \mathcal{F}(\mathcal{D})$. The initial set of columns K consists both of $F = \{j : x_j > 0\}$ and the $|K| - |F|$ columns (not in F) with smallest reduced costs \bar{c}^π .
2. Solve the subproblem $\min\{c_K^T x_K : A_K x_K = 1, x \geq 0\}$. Call the corresponding dual solution ρ .
3. If $\rho \in \mathcal{F}(\mathcal{D})$, or $\pi^T 1 = c_K^T x_K$ then stop. The optimal solution to (P) is x_K .
4. Let $\pi' = \theta\pi + (1 - \theta)\rho$, for $0 \leq \theta \leq 1$ such that $\pi'^T A \leq c$, and $\pi'^T 1$ is maximized. In closed form,

$$\theta = \max_{j:\bar{c}_j^\rho < 0} \left\{ 0, \frac{-\bar{c}_j^\rho}{(\bar{c}_j^\pi - \bar{c}_j^\rho)} \right\}.$$

5. Construct a new set of columns K' consisting of the basic columns of x_K and the $|K| - m$ (nonbasic) columns with the smallest reduced costs $\bar{c}^{\pi'}$. Let $\pi = \pi'$, $K = K'$, and go to 2.
-

Hu and Johnson [64] discuss a number of techniques to improve the performance of the algorithm. One algorithmic issue not covered in their discussion is the choice of an appropriate size of $|K|$. We have found that a subproblem size of

$$|K| = \min(\lfloor 2m + 0.025n \rfloor, 2000)$$

works well on a large majority of the problems, and we use this subproblem size in our implementation.

2.2.3 Cutting Planes

The set packing problem (PACK) is closely related to SPP and can be written as

$$\min\{c^T x : Ax \leq 1, x \in \{0, 1\}^n\}.$$

Since PACK is a relaxation of SPP, valid inequalities for PACK can be used as valid inequalities for SPP. A number of authors have studied the polyhedral structure of PACK and derived classes of facet inducing inequalities for it [97] [96] [23].

An important concept for generating these inequalities is that of the conflict graph CG , where $V(CG) = \{1, 2, \dots, n\}$ and

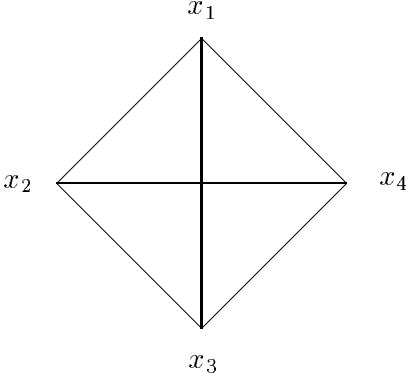
$$E(CG) = \{(i, j) \in V : x_i \text{ and } x_j \text{ cannot both be one in a feasible solution to SPP}\}.$$

A discussion of conflict graphs in a general context is given by Atamtürk *et al.* [4]. For SPP, many edges $e \in E(CG)$ can be found by direct inspection of the matrix A . In particular, if columns A_j and A_k are not orthogonal, then $(j, k) \in E(CG)$. Edges $e \in E(CG)$ are also found as implications during probing.

In our algorithm, we use two classes of valid inequalities that have been shown to be effective in improving the linear programming relaxation of SPP. The first class is the class of *clique inequalities*. Recall that a clique C in a graph is a set of nodes such that each pair of nodes is connected by an edge. If $C \subseteq V(CG)$ is a clique in the conflict graph, then the clique inequality

$$\sum_{j \in C} x_j \leq 1$$

is a valid inequality for PACK (and hence for SPP). Example 2.2 shows a fractional solution to the linear programming relaxation of PACK and a clique inequality that cuts off this fractional solution.

$x_1 + x_2 \leq 1$ $x_1 + \quad + x_3 + x_4 \leq 1$ $\quad x_2 + x_3 + x_4 \leq 1$ <p>LP Solution: $x_1 = x_2 = x_3 = x_4 = \frac{1}{3}$</p>	
<p>Clique inequality: $x_1 + x_2 + x_3 + x_4 \leq 1$</p>	

Example 2.2: A Clique Inequality

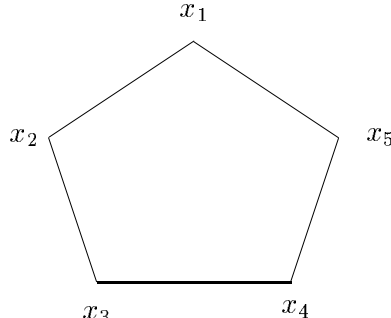
In its general form, the separation problem for clique inequalities is known to be NP-Complete. Therefore, we use a simple greedy heuristic in order to identify violated clique inequalities.

The second class of inequalities used in our algorithm is the *odd-cycle* inequalities. If $H \subseteq V(CG)$ is a cycle in the conflict graph and $|H|$ is odd, then the odd-cycle inequality

$$\sum_{j \in H} x_j \leq \frac{|H| - 1}{2}$$

is valid for PACK and SPP.

Example 2.3 gives a fractional linear programming solution and an odd-cycle inequality that cuts off this solution.

$x_1 + x_2$	≤ 1	
$+ x_2 + x_3$	≤ 1	
$x_3 + x_4$	≤ 1	
$x_4 + x_5$	≤ 1	
$x_1 + x_5$	≤ 1	
<p style="text-align: center;">LP Solution:</p> $x_1 = x_2 = x_3 = x_4 = x_5 = \frac{1}{2}$ <p style="text-align: center;">Clique inequality: $x_1 + x_2 + x_3 + x_4 + x_5 \leq 2$</p>		

Example 2.3: An Odd-Cycle Inequality

The separation problem for odd cycle inequalities is solvable in polynomial time [23]. However, as pointed out by Hoffman and Padberg [62], the polynomial time separation procedure is not computationally suitable for our problem. Therefore, we separate for odd cycle inequalities using a slight variation of the limited enumeration heuristic of Bixby and Lee [17].

For SPP, the size of the conflict graph can be very large, which may cause the procedures for finding violated inequalities to be very time consuming. We improve the efficiency of the separation procedures by considering only the subgraph CG_F of CG induced by the fractional variables in a solution to the linear programming relaxation. Specifically, given a fractional linear programming solution \hat{x} , we

construct CG_F with vertex set $V(CG_F) = \{j : 0 < \hat{x}_j < 1\}$ and edge set

$$E(CG_F) = \{(j, k) \in E(CG) : j \in V(CG_F) \cap k \in V(CG_F)\}$$

and look for violated inequalities in CG_F .

The clique inequalities are lifted by a simple exact procedure. For a clique inequality with corresponding clique C , the lifting coefficient a_j for $j \notin F$ is given by

$$a_j = \begin{cases} 1, & \text{if } (j, k) \in E(CG) \forall k : a_k = 1, \\ 0, & \text{otherwise.} \end{cases}$$

Note that a_k could equal one because $k \in C$ or because the variable with index $k \notin F$ had already been lifted. In practice, even this simple procedure can be very time consuming, so we decided to improve coefficients (or lift) only a fixed fraction of the variables. The variables to lift are chosen in order of increasing reduced cost.

2.2.4 Primal Heuristics

Obtaining provably good feasible integer solutions quickly is the goal of our heuristic. For this purpose, three different heuristics for the set partitioning problem are included as components of our heuristic.

Heuristic I

The first heuristic is a slight modification of a dual based heuristic due to Fisher and Kedia [41]. Given a feasible dual solution, a 3-opt procedure looks for an improved

dual feasible solution by adjustments involving exactly three dual variables. Our 3-opt procedure has been randomized so that 3-exchanges which do not improve the dual objective value are also performed with some probability.

Given a dual feasible solution $\hat{\pi}$ obtained from the randomized 3-opt procedure, a primal feasible solution is computed by a simple greedy procedure. The choice of variables to be one is made in a greedy fashion as suggested by Chvátal for the set covering problem [25], with the modification that the variables are ordered by their reduced costs instead of their cost coefficients. Note that (as we would expect), there is no guarantee of obtaining a feasible solution in this manner. Indeed, since the problem of simply finding a feasible solution to SPP is NP-Complete, none of the heuristics we present is guaranteed to find a feasible solution.

Heuristic II

The second heuristic is the Lagrangian dual cost perturbation heuristic of Wedelin [118]. A Lagrangian relaxation of SPP is obtained by moving the equality constraints to the objective:

$$L(\pi) = \mathbf{1}^T \pi + \min_{x \in \{0,1\}} (c - \pi^T A)x.$$

The Lagrangian dual is

$$\max_{\pi \in \mathfrak{R}^m} L(\pi).$$

For a reduced cost vector $\bar{c}^\pi = c - \pi^T A$, if the Lagrangian dual has an optimal solution $(\hat{x}, \hat{\pi})$ such that \hat{x} is feasible to SPP and $\bar{c}_j^\pi < 0$ or $\bar{c}_j^\pi > 0$ for all j , then \hat{x} is an optimal solution to SPP.

The Lagrangian dual is solved by an iterative coordinate search method. Let A_i denote the i th row of A , and let e_i be the i th coordinate vector. Finding an optimal step size in the i th coordinate direction from a dual solution π amounts to solving the problem

$$\max_{\theta \in \mathfrak{R}} L(\pi + \theta e_i).$$

Some simple manipulation shows that

$$\begin{aligned} L(\pi + \theta e_i) &= \mathbf{1}^T \pi + \theta + \min_{x \in \{0,1\}} (\bar{c}^\pi - \theta A_i)x \\ &= \mathbf{1}^T \pi + \theta + \sum_{j=1}^n \min(0, \bar{c}_j^\pi - \theta a_{ij}). \end{aligned}$$

To find the optimal step size length, we look for θ satisfying

$$\frac{\partial}{\partial \theta} L(\pi + \theta e_i) = 1 + \sum_{j: a_{ij}=1} \frac{\partial}{\partial \theta} \min(0, \bar{\pi}_j - \theta a_{ij}) = 0.$$

For any j ,

$$\frac{\partial}{\partial \theta} \min(0, \bar{\pi}_j - \theta a_{ij}) = \begin{cases} 0, & \text{if } \theta < \bar{c}_j^\pi, \\ -1 & \text{if } \theta > \bar{c}_j^\pi. \end{cases}$$

Therefore, if r^- and r^+ are the smallest and second smallest reduced costs of variables with coefficient 1 in A_i , then $L(\pi + \theta e_i)$ is stationary for $r^- \leq \theta \leq r^+$. Since L is a piecewise linear concave function, we can conclude that the stationary point is indeed a maximum and $\theta \in [r^-, r^+]$ is an optimal step size.

The Lagrangian dual is solved by starting from an arbitrary π and moving in each coordinate direction by an amount $\theta^* = (r^+ + r^-)/2$. Hence, if $T(i)$ is the set of variables appearing in row i , we update \bar{c}_j^π , $j \in T(i)$ by $\bar{c}_j^\pi := \bar{c}_j^\pi - \theta^*$.

The solution of the Lagrangian dual by this method is unlikely to yield solutions that have all $\bar{c}_j^\pi \neq 0$. Wedelin proposes a scheme for perturbing the vector c in such a way such that when the Lagrangian dual is solved, the resulting reduced costs satisfy either $\bar{c}_j^\pi < 0$ or $\bar{c}_j^\pi > 0$ for all j . Specifically,

$$\bar{c}_j^\pi := \begin{cases} \bar{c}_j^\pi - \theta^* - \frac{\kappa(r^+ - r^-)}{1 - \kappa} - \delta, & \text{if } \bar{c}_j^\pi \leq r^-, \\ \bar{c}_j^\pi - \theta^* - \frac{\kappa(r^+ - r^-)}{1 - \kappa} + \delta, & \text{if } \bar{c}_j^\pi \geq r^+, \end{cases} \quad (2.3)$$

where $\kappa \in [0, 1]$ and δ is a small constant.

The solution quality depends highly on the parameter κ , and it is difficult to know beforehand what a suitable value of κ might be. Wedelin [118] discusses the algorithm in full detail and gives an adaptive “sweep” strategy for choosing appropriate values for the parameters κ and δ . We will discuss our implementation of such a strategy in Section 6.3.2.

Heuristic III.

The final heuristic is a primal based heuristic and is based on the observation that the linear programming relaxations of small sized set partitioning problems often have integer solutions or yield integer solutions with relatively little branching in a branch and bound procedure. Ryan and Falkner [109] cite a result of Padberg [98] and give some theoretical evidence to support this empirical observation. Our heuristic is to choose a “suitable” subset of the columns of A and to solve the integer program (with a suitable time limit) over these columns. Of course, choosing the

columns is the most difficult part of this heuristic. Our strategy for performing the column choice will be discussed in Section 6.3.1.

2.2.5 Reduced Cost Fixing

Variables may be fixed based on their reduced costs as discussed in Section 2.1. When solving the SPP using a branch and cut approach, reduced cost fixing is often an extremely powerful tool – allowing as many as 90% of the variables to be fixed. However, obtaining a good feasible solution is critical in allowing a large percentage of variables to be fixed by reduced cost.

A flowchart of the heuristic and a discussion of the specific control issues is delayed until Section 6.3.1.

CHAPTER 3

Parallel Computing Preliminaries

In this chapter, we describe the two main types of parallel computer architectures and give our justifications for focusing our algorithms on one of the two architectures. The next section in the chapter raises issues that must be addressed when designing algorithms for this parallel architecture. We discuss how and why benefits from parallelism might be attained, and we give a categorization of ways in which information may be shared among processors and discuss the advantages and disadvantages of each category. Finally, we describe a software system that facilitates the development of our parallel algorithms.

3.1 Shared Memory versus Message Passing

Parallel computing architectures can vary widely, and there are a variety of parameters that can be used to help classify the architectures. We will be most concerned about processor interconnections, or the way in which processors of the parallel

computer exchange information. Loosely speaking, the two extreme alternatives for processor interconnection are *shared memory* and *message passing*.

Just as the name suggests, one processor of a shared memory machine can communicate with another by writing the information into a global shared memory location and having the second processor read directly from that location. This makes inter-processor communication very easy, but introduces problems having to do with simultaneous access of a unique memory location by multiple processors. The more processors the machine has, the bigger the problem, so shared memory machines tend to have fewer processors than their message passing counterparts.

In a message passing architecture, each processor has its own local memory. For this reason, this architecture is often given the name *distributed memory*. Processors are connected on a communication network and share information by passing it through this network. The communication network connecting the processors can be quite complicated, consisting of a complex mesh of interconnections all connected by a special bus designed for the purpose, or it can be as simple as an ethernet network to which all the processors are attached.

We choose to tailor our parallel algorithms to a machine with a distributed memory architecture. We have chosen to not focus on the shared memory architecture for a variety of reasons.

The first of the reasons arises from the fact that a network of simple workstations connected by an ethernet can be conceptualized as a parallel, message passing computer. This is a very economical way of having parallel processing capabilities,

so building algorithms to take advantage of this parallelism is important. Second, there are physical limits on the number of processors that can exist on shared memory machines; hence, in the future, massively parallel machines will have a distributed memory. Currently, there is a focus on combining shared memory and message passing machines. Indeed, the world's fastest computer (currently under development) is a such a hybrid [88]. For these hybrid machines as well, message passing is an issue.

The interconnection topology of the parallel computer also can be a large factor in designing efficient algorithms. In some topologies, (for example the simple ring network pictured in Figure 3.1) a message must make a number of point to point hops in order to travel from the source processor to the destination processor. Our main focus will be on machines that have no strong notion of such a processor "neighborhood". We will assume that the cost of communication between any two processors is roughly the same. There are two reasons that we concentrate on such architectures. First, even for machines where processors do have distinct neighborhoods, advanced routing techniques have made the cost of communication between any two processors roughly the same. Second, the network of workstations on a common ethernet subnetwork has no notion of a processor neighborhood and is likely where parallel optimization algorithms will find the most widespread use.

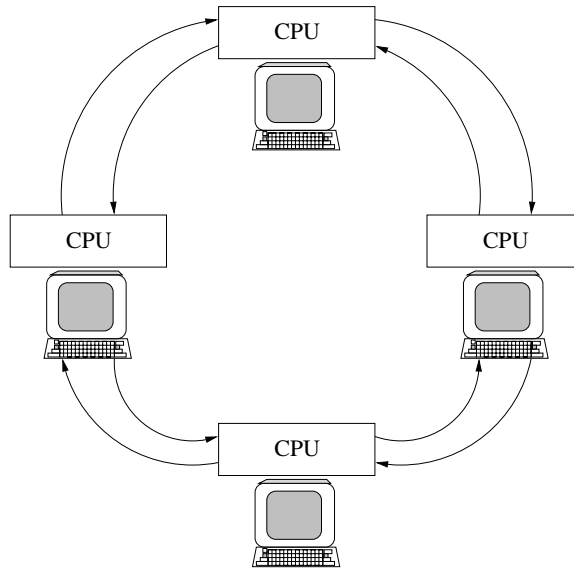


Figure 3.1: A Ring-Connected Parallel Computer Topology

3.2 Advantages of Parallel Computing

In this section, we ask the question of what we hope to gain by parallel computing, and introduce some challenges that must be overcome in order to achieve these gains.

3.2.1 Concurrent Computing

The obvious way in which parallelism can be exploited is by performing portions of the computing task concurrently. Breaking a problem into slices that can be performed simultaneously is often not a trivial matter. If a problem can be decomposed, it is usually done in one of two ways. Either the data is divided among the

processors, which is called *domain decomposition* or the program tasks themselves are divided, which is called *control decomposition* or *functional decomposition*. See Lewis and El-Rewini [83] or Foster [43] for more explanation of these parallel programming methods.

In the domain decomposition approach to problem partitioning, we partition the computation that is to be performed by associating each operation with the data on which it operates. This partitioning yields a number of tasks, each comprising some data and a set of operations on that data. An operation may require data from several tasks. In this case, communication is required to move data between tasks and to synchronize the operations. The classic example of a domain decomposition method occurs in solving finite difference equations. In this application, we have a mesh of points $x_{i,j}$, $i = 1, \dots, n$, $j = 1, \dots, n$, and at each iteration t of the algorithm, for each point we must compute

$$x_{ij}^t = \frac{x_{i-1,j}^{t-1} + x_{i,j-1}^{t-1} + x_{i+1,j}^{t-1} + x_{i,j+1}^{t-1}}{4}. \quad (3.1)$$

To parallelize this algorithm, different points in the mesh are assigned to different processors, as depicted in Figure 3.2. Mesh points on the boundaries of the regions assigned to different processors must share data in order to perform an iteration. Thus, data transfer and synchronization between tasks is required.

In functional decomposition, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The goal is to divide the computation into disjoint tasks that can be executed in parallel. If the data requirements of the divided tasks are also disjoint, then we have suc-

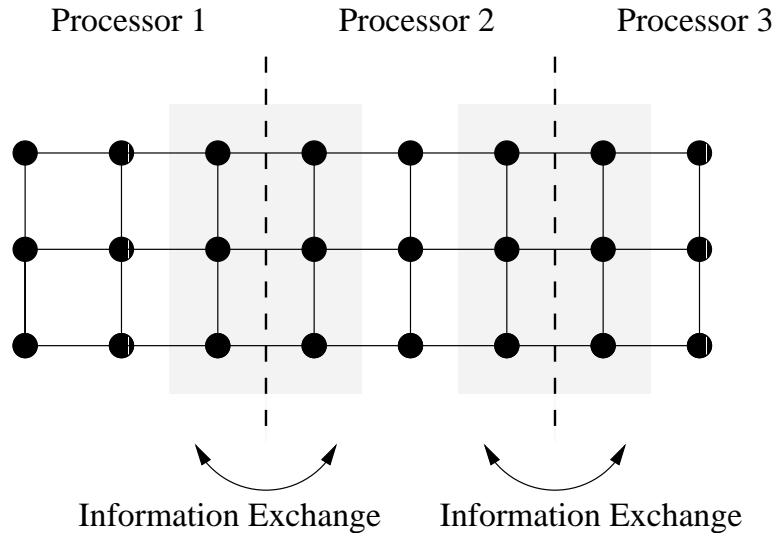


Figure 3.2: An Assignment of Grid Points to Processors for Jacobi Iteration

cessfully parallelized the program. Usually there is at least some overlap in the data requirements of the tasks, in which case communication will be needed. A good example of where functional decomposition can be effective is in tree search techniques, such as the branch and bound procedure.

Define $T(p)$, $p \geq 1$, to be the time required to solve a problem on p processors and the *speedup* of the algorithm on this problem as

$$S(p) = \frac{T(1)}{T(p)}.$$

Many algorithms do not readily break into pieces that may be parallelized. In general, perhaps a fraction β of the algorithm is inherently serial. Ignoring communication overhead, we would expect to require a time

$$T(p) = T(1)\beta + \frac{1}{p}T(1)(1 - \beta) \tag{3.2}$$

to perform the algorithm on p processors. Substituting this expression into the definition of speedup, gives us a limit on the speedup we can expect from an algorithm:

$$S(p) \leq \frac{p}{\beta p - \beta + 1}.$$

This expression is known as Amdahl's Law [1]. Define the *efficiency* of the algorithm as

$$E(p) = S(p)/p.$$

We say an algorithm has a *linear speedup* if $E(p) = 1$ and a *superlinear speedup* if $E(p) > 1$. An algorithm exhibits a *speedup anomaly* if for $p_1 < p_2$, $T(p_1) < T(p_2)$. Superlinear speedup and speedup anomalies are not predicted by Amdahl's law, and we delay discussion of effects causing these occurrences until the next section. Instead, we now focus on stumbling blocks to achieving linear or near linear speedup.

Amdahl's law states that if β is large then the speedup will be small. The parameter β is difficult to quantify for many algorithms, but it might be thought of as both the portion of the algorithm that cannot be divided into disjoint tasks and a measure of the synchronization and communication required between tasks. An example will make our point clear. Consider our domain decomposition example in Figure 3.2. Suppose that computing x_{ij}^t by equation (3.1) takes $1/3$ of a time unit, and suppose that exchanging the necessary values of x_{ij}^t between two processors takes one time unit. Since processor 2 must share data with both processors 1 and 3, then the total time to share data takes 2 time units. Figure 3.3 shows a "task

graph”, or order in which the various computing tasks must be performed, and Figure 3.4 gives a Gantt chart of the processors’ utilization during one iteration of the method.

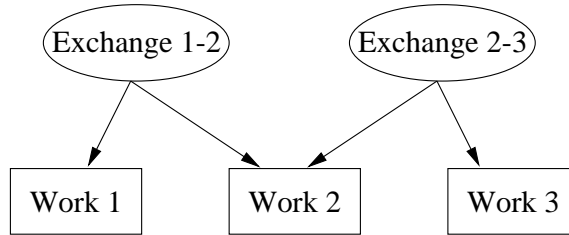


Figure 3.3: A Task Graph of One Jacobi Iteration

	Processor 1	Processor 2	Processor 3
1	Exchange 1-2	Exchange 1-2	
2		Exchange 2-3	Exchange 2-3
3	Work 1	Work 2	Work 3
4			
5			

Figure 3.4: A Gantt Chart of One Jacobi Iteration

For one iteration, we have $T(1) = 8$ and $T(3) = 5$. Substituting into equation (3.2) and solving for β yields $\beta = 7/16$. The contributions to β come from two sources. First, a contribution of $2/5$ comes from the fact that two of the five time

units required to perform an iteration are spent in communicating necessary data between processors. The remaining $3/80$ of β stems from the fact that we were unable to “perfectly” divide the work among the processors.

Note that we could have divided the work among the processors equally, but at an increase in the number of mesh points for which data must have been shared. This type of communication versus computation tradeoff will arise frequently in our parallel algorithms.

In the early days of parallel computing, it was thought that Amdahl’s law would limit the use and effectiveness of parallel computing. However, researchers today rarely find that Amdahl’s law severely limits the ability to effectively parallelize a computing task. If β is large, then we may wish to design a different algorithm. A good example of how to overcome the limitation of Amdahl’s law is given in the context of a non-computing task by Foster [43].

Assume that 999 of 1000 workers on an expressway construction project are idle while a single worker completes a “sequential component” of the project. We would not view this as an inherent attribute of the problem to be solved, but as a failure in management. For example, if the time required for a truck to pour concrete at a single point is a bottleneck, we could argue that the road should be under construction at several points simultaneously. Doing this would undoubtedly introduce some inefficiency – for example, some trucks would have to travel further to get to their point of work – but would allow the entire task to be

finished more quickly.

The same types of issues arises when designing parallel optimization algorithms. Suppose that in doing parallel branch and bound, we would like to to evaluate node N^i that has the highest value of z_U^i . Consider a strategy where at each iteration of the branch and bound algorithm, the p nodes with the highest values of z_U^i are evaluated, one by each of the p available processors. Then, the generated children nodes are passed back to a master process, and the process is repeated. Performing branch and bound in this way increases the amount of synchronization of the algorithm, and increases the value of β . This synchronization is not necessary for the correctness of the algorithm. In fact, in order for the branch and cut algorithm to be correct, we can use any valid bound information, cut information, and pseudocost information. Because of this, we say that the parallel branch and bound algorithm has *weak synchronization requirements*, and we hope to exploit this fact to design effective parallel branch and bound algorithms. Mohan [92] and Kumar *at al.* [74] showed that asynchronous implementations of branch and bound algorithms can be significantly more efficient than synchronous implementations.

Another way to lessen the value of β is to increase the “grain” of computation. For example, a processor could evaluate more than one node before passing the generated children back to a master process. In different contexts, this has been referred to as “scaling-up” the parallel algorithm [54] [83] [73] [113]. Care must be taken when scaling-up a parallel algorithm for any computing task where the work is determined in a dynamic fashion, such as is the case in the branch and bound

algorithm. For example, we may evaluate more nodes when processors evaluate multiple nodes in one iteration, which reduces the speedup. In general, we would like to always exploit parallelism at the coarsest grain possible, while minimizing the negative effects that increasing the unit of computation might have.

Gustafson *et al.* [55] were the first to demonstrate how to “beat” Amdahl’s law by scaling-up the problem size to achieve near linear speedups. This basic concept was formalized by Kumar and Rao [75], who introduced the concept of the *isoefficiency function*. The isoefficiency function $f(n, p)$ of a parallel algorithm is the rate of growth of problem size (n) required to keep the efficiency of a parallel algorithm constant as the number of processors (p) increases. For example, if $f(n, p)$ is an exponential function, the algorithm is poorly scalable since it will be difficult to obtain good speedups on large numbers of processors unless the problem size is enormous.

The optimization problems we study are NP-Complete, meaning that all known algorithms require an amount of work exponential in the problem size in order to find a solution in the worst case. Therefore, for a reasonable implementation, a linear increase in the number of processors requires only a logarithmic increase in the problem size in order to maintain the same efficiency. At first glance, the sublinearity of the isoefficiency function seems to be a good thing, since it implies that the procedures we consider, such as branch and bound, should scale well. However, it also means that increasing the number of processors by a small (or linear) amount will not lead to a similar growth in the size of the problem that we

are able to solve.

In all the preceding discussion we have neglected the effects of communication overhead in determining the speedup of parallel programs. Certainly, the overhead of communication between two processors cannot *help* to improve the speedup. (Note that the communication of “useful” information between processors *can* produce speedup. We delay the discussion of these effects until Section 3.2.2). The overhead required to pass a message can be broken into three parts. First the message must be packed. Second, the message must be sent over the communication channel to the receiving process. Third, the message must be unpacked, interpreted, and dispatched. The efficiency of the packing, sending, and unpacking procedures depends on the message passing interface used and the underlying computer hardware. The important point to make in our context is that the work and time required to pass messages takes the place of other computing tasks which might be more useful to the algorithm.

Regardless of the message passing library being used, to each message there is appended a small amount of information pertaining to the source and destination of the message, as well as other technical details. Because of this header, it is generally more efficient to try to send a few large messages, rather than many small messages.

The message passing libraries we consider have message queues to handle incoming and outgoing messages. The use of these queues adds an extra memory copy instruction each time a message is to be sent. The message queues are pro-

cessed in a first-in-first-out order.

3.2.2 Information Sharing

Motivation

The fact that information needs to be shared among processors is one of the complications of parallel computing, but it can also be one of the major sources of improved performance. In the parallel algorithms we study, information generated at the various processors may be beneficial to the work occurring at other processors. Figure 3.5 of the search tree of a branch and bound procedure performed both by one and two processors makes it clear how superlinear speedup can occur. Suppose that z_{MIP} , which is found at node 9, is such that $z_{UB}^j \leq z_{MIP} \forall j = 3, 4, \dots, 7$. Therefore, when z_{MIP} is found, these nodes can be pruned. The left portion of the figure shows a sequential branch and bound tree, where the numbers by the nodes indicate in which order the nodes were searched. Assuming it takes one time unit to evaluate each node, we require ten time units to complete the search. The right portion of the figure shows a possible allocation of work to two processors when the same problem is to be solved. In this case, since z_{MIP} is found earlier, nodes N^3, N^4, \dots, N^7 are not searched, and we require only three time units to solve the problem – a superlinear speedup. Nearly every author who has published on parallelizing the branch and bound algorithm has reported on the existence of superlinear speedup in their computational experiments [103] [37] [90].

A more disturbing consequence of the fact that information must be shared is

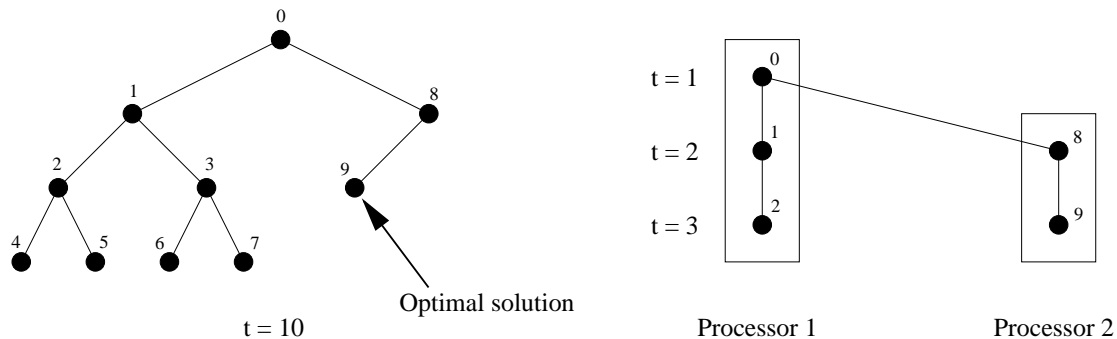


Figure 3.5: Superlinear Speedup in Branch and Bound

when information is not shared intelligently, it is possible that a computing task takes longer on a parallel machine than on a sequential machine. Some authors have explained conditions under which these *speedup anomalies* occur [77] [32].

Finding feasible solutions earlier in the branch and bound process is not the only way in which superlinear speedup can occur. Conceivably, any information which could help improve the bounds earlier in a parallel algorithm than in a sequential one could lead to superlinear speedup. In fact, the *only* way in which superlinear speedup can occur is through information sharing.

TreadMarks [68] is a package that supports parallel computing by providing a shared memory emulation. The user can program as if there is a common memory space, and TreadMarks handles the message passing required in order to retrieve the information. Bixby *at al.* [15] build a parallel MIP solver using TreadMarks. TreadMarks, or any other general shared memory emulation system, cannot take advantage of the particular information access patterns for the problem it is solving. For optimization algorithms, we are likely to know which types of information are

likely to be accessed by a processor, and through an intelligent implementation of a parallel system, we can hope to achieve a much higher efficiency than with a shared memory emulation system.

Goals

In order to share information on a distributed memory computer architecture, a message must be passed, for which some overhead is incurred. In this section, we state the goals of an information sharing system, and describe sharing paradigms whereby one may achieve these goals. Trienekens and de Bruin also discuss issues in information sharing for general parallel branch and bound algorithms [115].

The goals of an information sharing system are the following.

Goal I. Maximize “useful” sharing of information.

Goal II. Minimize latency of access to information.

Goal III. Minimize the number of messages passed.

Goal IV. Minimize memory use and memory use imbalance.

The goals are stated (roughly) in their level of importance. In addition, these goals should be accomplished in the least “disruptive” way possible. That is, processors should be kept busy doing work necessary to solve the problem, not simply sharing information. Throughout the thesis, we will continually refer back to these goals when thinking about appropriate designs for our parallel algorithms. Here we briefly describe the idea behind each goal.

Certainly it is difficult to quantify what the “useful” sharing of information mentioned in Goal I is; indeed, it is one of the purposes of this research to provide a partial answer to this question in the case of the parallel optimization algorithms we study.

The *latency* of an access to information, mentioned in Goal II, as the amount of time spent waiting for the information. In certain cases, we will refer to the time required to *generate* information as being latency as well. This is somewhat different than the standard definition of latency [43] [65]. The definition that is convenient to us is to define latency is the amount of time between when a request for information is made and the information is available – either by generating the information “from scratch” or by retrieving the information from a source.

We would like to achieve Goal III for two reasons: First, a large number of messages will use a large amount of bandwidth – perhaps even flooding the communication channel. Second, we would like to minimize the amount of processing time spent packing and unpacking messages, which also will help us perform information sharing in the least “disruptive” way possible.

If a large amount of information is generated by an algorithm, we may wish to distribute the information evenly over the available memory at many processors, thus achieving Goal IV of our information sharing scheme. Distributing the memory over all processors will enable us to make fullest use of the resources at our disposal and to potentially solve larger problems on a parallel machine than we could on a single processor machine.

It is easy to see that the goals are interrelated, and tradeoffs may be exploited by varying the degrees to which each is satisfied. The weak synchronization requirements of our algorithms previously mentioned gives a great degree of flexibility in meeting each of these goals.

We now introduce some information sharing strategies and state the degree to which these strategies satisfy the various goals. Our discussion uses the following abstract setting. Suppose there are a number of units of work that make up our computing task. For these units of work to be accomplished, there exists information that either may be distributed or must be distributed to the processors. The processors perform units of work, and potentially generate new information and new units of work. The distinction between information that *may* be shared and information that *must* be shared is an important one, and will be discussed later in the section.

The crucial question to be dealt with is how the information will be accessed. We break the information access pattern into two broad categories. In the *fetch* information sharing strategy, one processor specifically requests information from another processor. In the *prefetch* information sharing strategy, information simply appears at a processor, and the processor may access the information from its local memory.

Another important point to make is that there need not be only one way in which all types of information should be shared. The type of information sharing should depend on

- the “importance” of the information being shared,
- the speed and ease of generating the information locally, and
- the underlying computer architecture and network architecture.

Our research will find suitable information sharing schemes for the types of information generated by our algorithms.

Fetch

The most basic fetch sharing mechanism is to designate one processor responsible for keeping track of all the necessary information. The remaining processors are responsible solely for performing units of work. We refer to this as the *master-worker* information sharing paradigm. Since all information is available in one memory address space, the master-worker strategy can be thought of as an attempt to emulate the shared memory computer architecture. Thus, the master-worker approach has many of the same advantages and disadvantages that shared memory computer architectures have with their message passing counterparts.

The main advantage of the master-worker approach lies in the fact that there exists a central repository for information. Thus, maximizing useful information sharing (accomplishing Goal I) is a relatively simple implementation detail.

The obvious drawback of the master-worker paradigm is that the master process may become swamped with requests for information. If this happens, *contention* is said to occur. Contention shows its negative effects in the ability of the information

sharing system to meet Goal II. If the master has a large number of requests from workers, then a worker may wait idle until all the requests are served. Another drawback with the master-slave information sharing scheme occurs when there is a large amount of information to be shared among the processors. In this case, a significant imbalance in the memory usage occurs – Goal IV is not accomplished. The memory imbalance may limit the ability of the parallel computer to perform its required task.

In order to alleviate some of the drawbacks of the master-worker scheme, we might consider having a designated number of “servers” of information. When a processor requires information, it queries the appropriate server to retrieve it. This can be thought of as a *client-server* approach to information sharing.

The advantage of the client-server approach is that by designating an appropriate number of information servers, we can alleviate some of the contention and memory imbalance effects of the master-slave approach – helping us achieve Goals II and IV to higher degrees.

However, the implementation of such approach is significantly more complicated. Specifically, what is an “appropriate” number of servers for a given type of information? Also, if there is more than one server for a specific type of information, in order to have an effective parallel algorithm, we may need schemes to share information among the servers themselves. Each of these problems may lessen the degree to which we are able to accomplish Goal I. A final drawback of the client-server approach, but a very important one, is that using processors only

to serve information to other processors may not be making use of the computing resources to their fullest potential. The information sharing scheme is *disrupting* the normal work of the algorithm. We may therefore wish to consider allowing information servers to also perform units of work. In this case, requests for information may not be served until after a unit of work is completed on the server processor, leading to high latency – conflicting with Goal II.

Prefetch

In contrast to the fetch information sharing system, where a processor requests information, the prefetch strategy distributes the information to the processors where it can be retrieved locally.

The simplest prefetch sharing scheme is to broadcast the information to all processors once it becomes available. We will refer to this device as a *broadcast-prefetch* information sharing scheme. Advantages of the scheme are that the information is always available to a processor and that there are no contention effects. Thus, we can achieve Goals I and II. However, these goals are accomplished at the expense of passing a large number of messages, so Goal III is not met. Also, storing (by replicating) information on each processor may use a significant amount of memory – contrary to Goal IV.

To lessen some of the negative effects with respect to communication of the broadcast-prefetch scheme, we might consider having each processor collect and hold information locally for a time before passing it to other processors. We call

such a scheme a *buffered-prefetch* information sharing scheme. Analogous to the relationship between the client-server and master-worker sharing schemes in fetch, the degree to which some of the goals are met can be increased but at the price of reducing the degree to which other goals are met. In particular, for the buffered-prefetch scheme, we reduce the number of messages sent (helping achieve Goal III). However, useful information may be sitting in buffers, and Goal I is satisfied to a lesser degree.

Also, the complexity of the implementation of a buffered-prefetch scheme is somewhat high. In particular, appropriate buffer sizes for the information must be determined that balance the number of messages sent with the availability of information at all the processors. An aim of the research is to examine this question for our parallel optimization algorithms.

If we could somehow know on which processors information was likely to be useful, we could simply pass the useful information to these processors. We call such a scheme a *selective-prefetch* information sharing scheme. The advantage of a selective-prefetch sharing scheme is obvious – information is shared with no contention effects and with minimum number of messages. The disadvantage of a selective-prefetch sharing scheme is being able to know precisely whether or not sharing information will be useful and exactly which processors the information will be useful. For the optimization algorithms we study, we will attempt to determine whether or not one can develop selective-prefetch schemes.

These ideas of information sharing are not unlike ideas of prefetching or caching

found in the fields of filesystems [93], databases [30], or multi-processor shared memory computer architectures [111].

No information sharing

For some types of information, it may be possible for a processor to attempt to generate the information itself, removing the need for the information to be shared. The advantages of this scheme are obvious: no bandwidth or information processing time is needed, and no contention effects or memory imbalances are present when information is not shared. Thus, Goals III and IV are fulfilled. However, obviously Goal I cannot be achieved, since no information is shared.

Minimizing the latency of access to information *may or may not* be accomplished by this scheme, depending on how quickly the information is locally generated as opposed to retrieved from its location under a sharing scheme. Another point about not sharing information is that for certain types of information, a processor may fail in generating useful information, even if such information exists elsewhere in the system.

3.3 NAYLAK

We build our parallel optimization algorithms on top of NAYLAK. NAYLAK is a software system for message passing that is based on an “entity-FSM” paradigm written by K. Perumalla [102]. NAYLAK provides a higher software interface layer above traditional process-based message passing systems. Traditionally, bulky pro-

cesses (usually UNIX processes) are the units which communicate by exchanging messages. In NAYLAK, an additional layer is added over this basic process-level message-passing by introducing the concept of lighter units called *entities*. Each process consists of a set of entities, and message-passing is performed at the entity level rather than process level. Thus, entities send messages to other entities, as opposed to processes sending messages to other processes. Entities can be created and terminated dynamically. The mapping of entities to UNIX-level processes, and the mapping of the processes to the parallel processors can be controlled at runtime. Entities can create or terminate other entities dynamically.

Another feature lacking in traditional message-passing interfaces, such as PVM [45], is that of *context* maintenance. If a given computation consists of several stages, with some message-passing/synchronization occurring between the stages, then either barriers or artificial message tags are used to realize the multi-stage computation. In NAYLAK, the concept of a finite state machine (FSM) is supported to directly facilitate multi-stage computation. An FSM is an arbitrary graph of *states*, where each state is a set of statements that are executed indivisibly (atomically). Every FSM is associated with a single entity, called its owner. An entity can have zero or more FSMs running for it. Each FSM is, in effect, a thread of computation, and each FSM state is a unit of computation. Each FSM state dynamically designates its next state. Since FSM states are executed indivisibly (atomically), no synchronization is required for access to common data across FSM's and FSM states. This is one of the main differentiating features

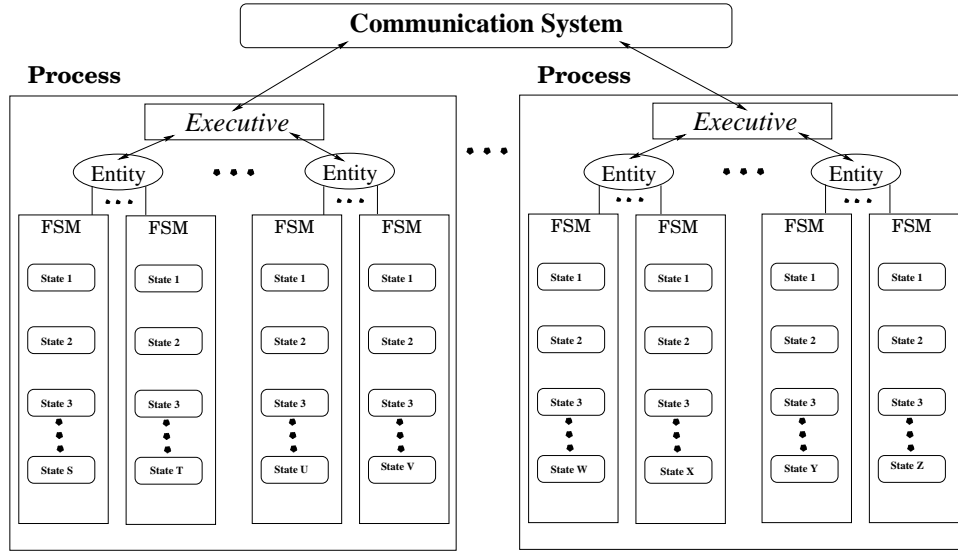


Figure 3.6: NAYLAK System Architecture

from other systems that combine conventional message passing with threads; in such threads-based systems, explicit synchronization primitives, such as mutexes or semaphores, should be used to ensure the atomic nature of the most common sequence of operations, whereas such operation is provided by default in this entity-FSM architecture. This greatly eases the burden on the programmer, and helps in a natural flow of control and data. FSM's can be started, paused, resumed and terminated dynamically. Figure 3.3 illustrates the NAYLAK system architecture.

Although abstractions usually entail overheads, the implementation of the NAYLAK system has been carefully designed in order to minimize the overhead imposed by the layering. As compared to the execution time of the majority of the procedures in our optimization algorithms, the overhead incurred (mostly

from extra memory-copying instructions) due to the NAYLAK runtime system is small. The advantages in using NAYLAK – portability, ease of development, and extendability – outweigh this overhead. NAYLAK is extremely useful for developing algorithms wherein the communication is inherently asynchronous and the processes are conceptually multi-threaded.

NAYLAK provides another layer above the message passing library. Its treatment of arriving messages is handled in a way very similar to that of PVM [45]. Since it is built on top of the message passing library, NAYLAK also handles the interface with the message passing library itself. Currently, NAYLAK has been ported to use either the PVM (3.3) message passing library [45], or the MPI (1.0) message passing standard [59].

CHAPTER 4

Parallel Branch and Bound

4.1 Introduction

In our version of the branch and bound algorithm for solving MIP, there are three main types of information to share among the processors:

- lower bounds on the optimal solution value,
- nodes of the branch and bound tree, and
- pseudocosts.

In this chapter we examine approaches for sharing information in a branch and bound method for solving mixed integer programming. We focus solely on search issues, postponing the discussion of sharing cuts until Chapter 5. In the first section, we describe how lower bound information is shared among the processors. The next section is a comprehensive study of search strategies for mixed integer

programming in a sequential environment. The study is not only interesting in its own right, but allows us to make observations about how to perform and implement search strategies in a parallel algorithm as well. The next section discusses issues relating to the use of pseudocosts in a parallel algorithm. We make observations about the relative importance of pseudocosts in a sequential versus a parallel algorithm, survey previous work, and give a pseudocost implementation scheme for the parallel algorithm. Next, we discuss effective node selection schemes in a parallel branch and bound algorithm for solving MIP. We describe new node selection schemes for parallel branch and bound algorithms that are based on emulating sequential schemes. The next section is a diversion in order to explain PARINO (PARallel INTeger Optimizer), the code allowing us to test our various parallel branch and bound ideas. Finally, we discuss and perform appropriate computational experiments to verify our intuition and draw conclusions.

4.2 Sharing Lower Bound Information

In branch and bound, the search tree can only be pruned if there exist valid lower and upper bounds for each node of the tree. Hence, the bound information is *extremely* important. When a valid lower bound is found by one processor, either by a heuristic procedure or because the linear programming relaxation at a node was feasible to MIP, we choose to broadcast this information to the other processors as soon as possible. Since the global lower bound is just a scalar, the communication cost for this broadcast is relatively small. This immediate broadcast of updated

bound information is the same approach taken by other researchers [37] [22].

4.3 Search Strategies for Mixed Integer Programming

4.3.1 Using Pseudocosts

As mention in Section 2.1, the use of pseudocosts is a well-known and effective device that enables the branch and bound tree to be searched in an efficient manner. However, the discussion in Section 2.1 left unanswered two important questions about how to implement a pseudocost-based branching scheme:

- To what value should the pseudocosts be initialized?
- How should the pseudocosts be updated from one branch to the next?

The question of initialization is an important one. By the very nature of the branch and bound process, the branching decisions made at the top of the tree are the most crucial. As pointed out by Forrest *et al.* [42], if at the root node we branch on a variable that has little or no effect on the LP solution at subsequent nodes, we have essentially doubled the total amount of work required.

An obvious method for initialization is to simply let the pseudocosts for a variable be the value of its objective function coefficient, since if a variable were unrelated to the other variables in the problem, its pseudocost would be precisely its objective function coefficient.

Bénichou *et al.* [14] and Gauthier and Ribière [44] experimented with explicitly computing the pseudocosts of each variable and initializing the pseudocosts to the computed values. Explicitly computing the pseudocost for a variable involves imposing the bounds that would be implied by the branching rule, solving the resulting linear programs, and applying the formulas (2.1). The authors report that initializing pseudocosts with an explicit computation can be effective in reducing the number of nodes evaluated, but that the time spent computing these values explicitly is quite significant. In fact, Gauthier and Ribière conclude that the computational effort is too significant compared to the benefit obtained. Even though today’s situation may be slightly different, due to faster computers and better LP solvers, it clearly indicates that care has to be taken when pseudocosts are explicitly computed for each variable.

Throughout the course of this section, we will be introducing experiments aimed at establishing the effectiveness of different pseudocost utilization techniques for mixed integer programming. The techniques under investigation were incorporated into the mixed integer optimizer MINTO (v2.0) [95]. MINTO is an integer programming solver that uses many state-of-the-art solution techniques. These features include preprocessing and probing, cut generation, and reduced cost fixing. We have chosen to perform a majority of the testing on a suite of 14 problems from the newest version of MIPLIB [16]: *air04*, *arki001*, *bell3a*, *bell5*, *gesa2*, *harp2*, *l152lav*, *mod011*, *pp08a*, *qiu*, *qnet1*, *rgn*, *stein45*, *vpm2*. The instances were chosen more or less at random, but exhibit a wide range of problem characteristics. The

characteristics of the instances on which we perform our pseudocost utilization experiments are listed in Table 4.8 in Section 4.7.1. Unless otherwise noted, all experiments for our study of sequential branch and bound search strategies were run with the settings shown in Figure 4.1. Other characteristics about the experiments will be mentioned as needed.

Figure 4.1: Characteristics of All Sequential Computational Experiments

- Code compiled with the IBM XLC compiler, optimization level -O2.
- Code run on an RS/6000 Model 590.
- CPU time limited to one hour.
- Memory limited to 100MB.

The first experiment is aimed at determining when to initialize pseudocosts. There are two main alternatives. The pseudocosts for all integer variables could be initialized at the outset of the computations, or each pseudocost could be initialized the first time an integer variable is fractional in a solution at a node. For a branch and bound computation, let f_I denote the percentage of integer variables that ever take on a fractional value in a linear programming relaxation at a node. Let f_{NB} denote the percentage of integer variables that are nonbasic at the root node, but take on a fractional value in a linear programming relaxation to a different node. (Recall that if an integer variable is nonbasic, then it is *not* fractional). We define $f_{NB} \equiv 0$ if no integer variables are nonbasic at the root node. For our test instances, Table 4.1 shows the value of f_I and f_{NB} .

Since for many instances, f_I is relatively small, it makes little sense to initialize the pseudocosts for all integer variables. We conjecture that if pseudocosts are going to be explicitly initialized, then they should be computed only for the fractional variables as needed. The fact that f_{NB} is small implies that if the pseudocosts for the basic integer variables are initialized at the root node, then very little other pseudocost initialization needs to be done. This observation will affect the design of our parallel algorithm.

Table 4.1: Percentage of Fractional Integer Variables

Problem	f_I	f_{NB}
air04	13.9	10.3
arki001	44.2	0.0
bell3a	56.3	0.0
bell5	96.6	0.0
gesa2	30.3	0.0
harp2	20.7	17.5
l152lav	20.1	16.4
mod011	83.3	0.0
qiu	100	0.0
qnet1	13.27	0.0
rgn	44.0	0.0
stein45	100.0	100.0
vpm2	49.4	0.0

Yet another pseudocost initialization alternative, suggested by Eckstein [37], keeps track of the average value of all pseudocosts for both the up and down branches. For each variable that has yet to be arbitrated, the pseudocosts are set to these average values. This method has the disadvantage that variables not yet branched on are ranked in importance only by how fractional they are.

In the course of the solution process, the variable x_j may be branched on many times. How should the pseudocosts be updated from one branch to the next? Bénichou *et al.* [14] state that the pseudocosts vary little throughout the branch and bound tree, and suggest fixing P_j^- and P_j^+ to the values observed the first time when variable x_j is branched on. Alternatively, as suggested in Forrest *et al.* [42], one could also fix the pseudocosts to the values obtained from the last time when x_j was branched on. Forrest *et al.* [42] and Eckstein [37] suggest averaging the values from all x_j branches.

We performed an experiment to verify the observations of Bénichou *et al.* [14], i.e., that the pseudocosts are relatively constant throughout the branch and bound tree. Suppose the (either up or down) pseudocost P_j is some linear function of the number of times N_j variable x_j is branched on. We can express this relationship as

$$P_j = \beta_0 + \beta_1 N_j.$$

For the suite of 14 problems from MIPLIB presented in Table 4.1, we explicitly computed the regression coefficients β_0 and β_1 for each variable and direction on which we chose to branch more than seven times. This gave us 693 variable-

direction pairs. For these 693, zero was in the 95% confidence interval of the regression coefficient β_1 673 times, which would imply there was statistical reason to believe that the pseudocosts are constant throughout the branch and bound tree for these variable-direction pairs. However, we also observed that from node to node, pseudocosts can vary significantly. In Figure 4.2, we plot the observed pseudocosts as a function of the number of times we branch on variable 219 in the problem *pp08a*. Therefore, we believe that updating the pseudocosts by averaging the observations should be the most effective.

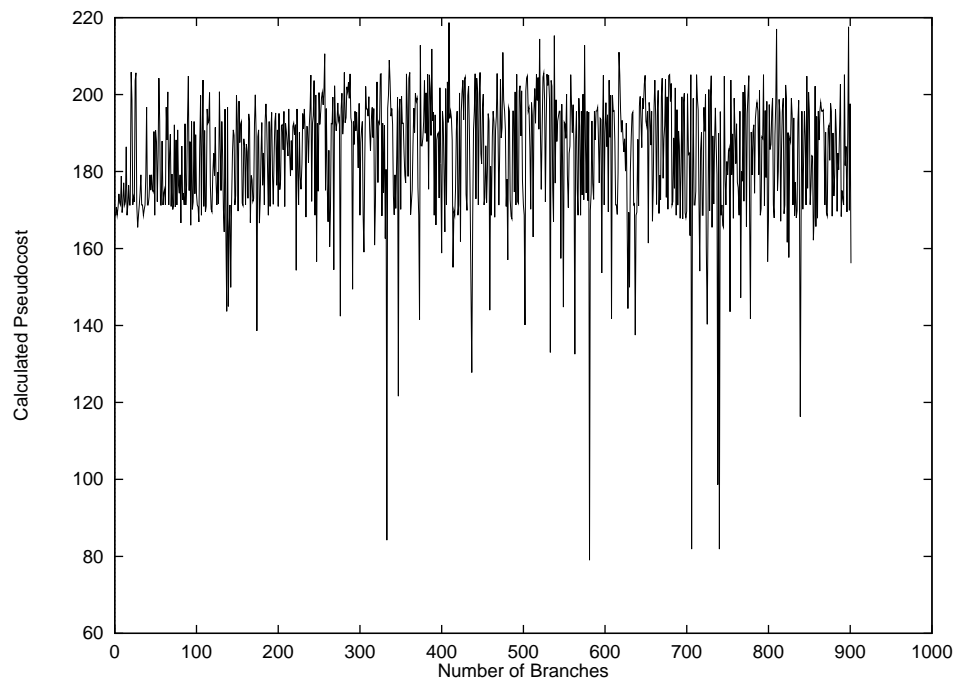


Figure 4.2: Observed Pseudocosts as a Function of Number of Branches

The issues of initialization of pseudocosts and updating of pseudocosts are unrelated. Generally once a variable is branched on, the initial pseudocost value is

discarded and replaced by the true (observed) pseudocost. Therefore, we will deal with the initialization and update issues separately.

We now describe an experiment that aims at establishing the best pseudocost initialization method. Since determining the best initialization method is our goal here, we have fixed the updating method in these runs to be the averaging suggestion. We branch on the variable x_j for which the sum of the estimated degradations (as defined in Section 2.1.1) $D_j^{i-} + D_j^{i+}$ is the largest. This choice will be discussed in more detail in Section 4.3.1 As stated in Section 2.1, the main focus of choosing a branching variable is to choose one that will most improve the upper bound of the child nodes from the parent. By setting z_L to the value of the optimal solution to the problem in our computational experiments, we minimize factors other than branching that determine the size of the branch and bound tree. For example, the node selection rule has no effect on the size of the branch and bound tree. Just for completeness, we mention that we use the “best bound” node selection rule, where at the **Select** portion of Algorithm 2.1, we choose to evaluate the active node with the largest value of z_U^i . In our tables of computational results, the “Final Gap” is computed as

$$\text{Final Gap} \equiv \frac{\max_{i \in \mathcal{L}} z_U^i - z_L}{z_L}, \quad (4.1)$$

and an “XXX” in the solution time column signifies that the memory limit was reached.

For many experiments, we will be including summary tables that rank the performance of the techniques under investigation. We rank techniques related to

branching methods using the following rules:

1. A method that proves the optimality of the solution is ranked higher than one that does not.
2. If two methods prove the optimality of the solution, the one with shorter computation time is ranked higher.
3. If two methods do not prove the optimality of the solution, the one with smaller final gap is ranked higher.
4. Ties are allowed.

Table A.1 in Appendix A shows the results of solving various problems using four pseudocost initialization methods. A summary of the experiment is given in Table 4.2. The pseudocosts were initialized with objective function coefficients, by averaging observations, by explicitly computing them for all variables at the root node, and explicitly computing them for the fractional variables only as needed.

Table 4.2: Summary of Pseudocost Initialization Experiment

Initialization Method	Avg. Ranking
Obj. Coef.	2.93
Averaged	3.07
Computed All	2.50
Computed Fractional	1.50

Examination of the results shows that explicitly computing initial pseudocosts for fractional variables as needed is clearly the best method. This result is different than the conclusion reached by Gauthier and Ribière [44]. The faster simplex algorithm and computers of today now make it possible to invest more effort into the (often very important) initial branching decisions.

We conclude that a good pseudocost initialization strategy should allow for initially computing pseudocosts explicitly, take care not to expend too much computation time accomplishing this task, and allow for spending more time computing explicit pseudocosts at the top of the branch and bound tree where branching decisions are more crucial. After further experimentation, we adopted the following pseudocost initialization strategy. Let T be the maximum amount of time per node we would like to spend to initialize pseudocosts for variables on which we have yet to branch. In this time, we wish to gain useful branching information on all fractional variables. We therefore impose a limit L on the number of simplex iterations used in solving the linear program necessary to compute one particular pseudocost. Let γ be an estimate of the number of simplex iterations performed per unit time, obtained by

$$\gamma \equiv \frac{\text{Number of iterations needed to solve the initial LP}}{\text{Time to solve the initial LP}}.$$

Let η be the number of fractional variables in initial LP solution. Then we compute L as

$$L = \frac{T\gamma}{2\eta}. \tag{4.2}$$

As we develop the branch and bound tree, if there is a fractional variable x_j upon which we have never branched, we perform L simplex pivots after fixing the bounds of this variable to $\lfloor x_j \rfloor$ and $\lceil x_j \rceil$ in order to explicitly determine the pseudocosts.

Gauthier and Ribière [44] also proposed a pseudocost initialization strategy that uses a limited number of simplex iterations, but our approach is fundamentally different. They purposely limit the number of simplex iterations to a small number, while we set T to a large number hoping to be able to compute a “true” pseudocost. The proposed pseudocost initialization method is also very similar to a technique known as *strong branching* [28]. In strong branching, however, the pseudocosts for a variable are reinitialized each time a branching decision for a fractional variable is to be made.

We now turn our attention to the question of how to update the pseudocosts from one branch to the next. As mentioned above, our initial experiments lead us to believe that updating the pseudocosts by averaging the observations would be the most computationally effective.

We empirically verified this conjecture by solving instances of MIPLIB where the pseudocosts were updated in each of the three ways suggested. For these runs we have initialized the pseudocosts by our strategy that explicitly computes them, with a limit on the number of iterations used. As in our previous experiment, we branch on the variable x_j for which $D_j^{i-} + D_j^{i+}$ is the largest, set z_L to be the known optimal solution to the problem, and we use the best bound node selection rule.

Table A.2 shows the full results of this experiment. For the pseudocost update experiment, Table 4.3 shows the average ranking over all the instances of each method.

Table 4.3: Summary of Pseudocost Update Experiment

Update Method	Avg. Ranking
First	2.43
Last	1.64
Average	1.43

From the results of the experiment we see that our intuition is correct. For the most part, it seems to be best to average the degradations when branching on a variable to determine its pseudocosts, and the overhead necessary to perform the averaging does not outweigh its benefits.

For the remainder of this section, when we refer to *pseudocost branching*, this will imply that we have used our strategy of explicitly computing the initial pseudocosts with a simplex iteration limit, and we update the pseudocosts by averaging the observations.

Using Degradation Estimates

Once we have computed estimates or bounds on the degradation of the objective function given that we branch on a specific variable, we still must decide how to use this information to make our branching choice. Our goal is to maximize the

difference in LP value of the relaxation from a parent to its children, but since there are two children of each parent, there are different measures of change. Gauthier *et al.* [44] suggest trying to maximize the sum of the degradation on both branches, i.e. branch on the variable x_{j^*} with

$$j^* = \arg \max_j \{D_j^+ + D_j^-\}.$$

Bénichou *et al.* [14] and Beale [13] suggest instead to branch on the variable for which the smaller of the two estimated degradations is as large as possible. That is,

$$j^* = \arg \max_j \{\min\{D_j^+, D_j^-\}\}.$$

Eckstein [37] suggests combining these ideas by branching on the variable

$$j^* = \arg \max \{\alpha_1 \min\{D_j^+, D_j^-\} + \alpha_2 \max\{D_j^+, D_j^-\}\} \quad (4.3)$$

Note that we can maximize the sum of the degradation on both branches by letting $\alpha_1 = \alpha_2 = 1$ in equation (4.3), and we can maximize the minimum degradation on both branches by letting $\alpha_1 = 1, \alpha_2 = 0$.

Table 4.4 shows a summary of the effect of varying the parameters α_1 and α_2 for our MIPLIB test instances. The full results can be found in Table A.3. For these runs, we have set z_L to be the known optimal solution to the problem and used the best bound node selection rule. From this experiment, we draw the following conclusions about using the degradation estimates to choose a branching variable:

- Both the up and down degradations should be considered.

- The importance of a variable is more related to the smaller of the two degradations.
- Using $(\alpha_1, \alpha_2) = (2, 1)$ in equation (4.3) appears to be a good choice.

Table 4.4: Summary of Degradation Use Experiment

(α_1, α_2)	Avg. Ranking
(1,0)	4.71
(10,1)	2.86
(2,1)	1.86
(1,1)	2.86
(1,2)	3.64
(1,10)	4.29

4.3.2 Node Selection

Here we provide a brief survey of node selection methods. We categorize the node selection methods as *static* methods, *estimate-based* methods, and *backtracking* methods. In addition, we introduce a new calculation on which to base estimation methods, and we perform experiments to test the effectiveness of the various methods.

Static Methods

A popular way to choose which subproblem to explore is to choose the one with the largest value of z_U^i . There are theoretical reasons for making this choice, since

for a fixed branching rule, selecting problems in this way minimizes the number of evaluated nodes before completing the search. This node selection rule is usually called *best first* or *best bound* search.

At the other extreme is a selection rule called *depth-first* search. As the name suggests, the solution space is searched in a depth-first manner.

Both these methods have inherent strengths and weaknesses. Best first search will tend to minimize the number of nodes evaluated and at any point during the search is attempting to improve the global upper bound on the problem. Therefore best first search concentrates on proving that no solution better than the current one exists. Memory requirements for searching the tree in a best first manner may become prohibitive if good lower bounds are not found early, leading to relatively little pruning of the tree. Also, the search tree tends to be explored in a breadth-first fashion, so one linear program to be solved has little relation to the next – leading to higher computation times.

Depth-first search overcomes both of these shortcomings of best first search. In fact, searching the tree in a depth-first manner will tend to minimize the memory requirements, and the changes in the linear program from one node to the next are minimal – usually just changing one variable’s bound. Depth-first search has another advantage over best first search in finding feasible solutions since feasible solutions tend to be found deep in the search tree. Depth-first search was the strategy proposed by Dakin [31] and Little *et al.* [86], primarily due to the small memory capabilities of computers at that time. Despite its advantages, depth-first

search can lead to extremely large search trees. This stems from the fact that we may evaluate a good many nodes that would have been fathomed had a better value of z_L been known. For larger problems, depth-first search has been shown to be impractical [42]. However, this conclusion was made in the days before primal heuristics were incorporated into most MIP codes, so depth-first search deserves to be re-examined.

Estimate-based Methods

Neither best first search nor depth-first search make any intelligent attempt to select nodes that may lead to improved integer feasible solutions. What would be useful is some estimate of the value of the best feasible integer solution obtainable from a given node of the branch and bound tree. The *best projection* criterion, introduced by Hirst [61] and Mitra [91] and the *best estimate criterion* found in Bénichou *et al.* [14] and Forrest *et al.* [42], are ways to incorporate this idea into a node selection scheme. The best projection method and the best estimate method differ in how they determine an estimate of the best solution obtainable from a node N^i . Given an estimate E^i , they both select the node in the active set for which this value is largest. Here we discuss only the best estimate criterion. For the best estimate criterion,

$$E_i = z_U^i + \sum_{j \in I} \min(P_j^- f_j, P_j^+(1 - f_j)). \quad (4.4)$$

This estimate is logical since the pseudocost is an estimate of the rate of change of the objective function value.

The estimate of the best solution obtainable from a node given by (4.4) assumes that we will always be able to round a fractional variable to the closest integer and obtain a feasible integer solution, which is somewhat optimistic. A more realistic estimate would be the following:

$$\begin{aligned}
E_i = z_U^i &+ \sum_{j \in I: f_j \leq 0.5} (f_j P_j^- q_j + (1 - f_j) P_j^+ (1 - q_j)) \\
&+ \sum_{j \in I: f_j > 0.5} (f_j P_j^- (1 - q_j) + (1 - f_j) P_j^+ q_j), \quad (4.5)
\end{aligned}$$

where q_j is the “probability” that we will be able to round a fractional solution to the closest integer and obtain a feasible integer solution. Note that if $q_j = 1 \forall j \in I$, then (4.5) reduces to (4.4). Linderoth and Savelsbergh [85] give a suggestion for computing q_j .

Backtracking Methods

Define a *superfluous node* as a node N^i that has $z_{LP}^i < z^*$. Searching the tree in a best first manner will ensure that no superfluous nodes are evaluated. If, however, one can be assured that all (or most) of the superfluous nodes will be fathomed, (which is the case if $z_L = z^*$), the memory and speed advantages of depth-first search make this method the most preferable. Various authors have proposed strategies that attempt to go depth-first as much as possible while minimizing the number of superfluous nodes evaluated [14] [20] [44] [28]. We call these types of methods *backtracking methods*. Given some estimate E_0 of the optimal objective function value z^* , the tree is searched in a depth-first fashion as long as $z_{LP}^i > E_0$.

If $z_{LP}^i \leq E_0$, then a node is selected by a different criterion such as best first or best estimate. The methods differ in the manner in which they obtain E_0 and in which criterion they use when deciding to backtrack.

There is a tradeoff in how “aggressive” one wants to be when calculating an estimate. If the estimate is too large, then the tree is searched in a best first or best estimate fashion and none of the advantages of going depth-first is obtained. If the estimate is too small, then the tree is searched in a more depth-first fashion and many superfluous nodes may be evaluated. This point must be kept in mind when selecting an estimation method on which to base a backtracking node selection rule.

4.3.3 Branch Selection

Typical branching is based on a dichotomy that creates two new nodes for evaluation. The node selection scheme must also answer the question of how to rank the order of evaluation for these nodes. Schemes that prioritize the nodes based on an estimate of the optimal solution obtainable from that node have a built-in answer to this question, since distinct estimates are assigned to the newly created nodes. For schemes that do not distinguish between the importance of the two newly created nodes, such as depth-first, researchers have made the following suggestion. Suppose that we have based the branching dichotomy on the variable x_{j^*} , then we select the down node first if $f_{j^*}^i < 1 - f_{j^*}^i$ and the up node first otherwise [78]. If estimates are not available, we will use this rule for selecting whether to evaluate

the down or up child of a node.

4.3.4 Computational Results

We performed an experiment to compare many of the node selection rules we have discussed. Each of the problems in Table 4.8 was solved using the node selection methods detailed in Table 4.3. For a branching method, we used a hybrid pseudo-cost method. The method is described in detail in Linderoth and Savelsbergh [85], where it is denoted as branching method B6. All advanced features of MINTO were used, which includes a diving heuristic that is invoked every ten nodes of the branch and bound tree.

Node Selection Method	Description
N1	Best Bound.
N2	Depth-First.
N3	Best Estimate (normal pseudocost).
N4	Best Estimate (adjusted pseudocost).
N5	Backtrack. Best Estimate (normal pseudocost). When backtracking, select node by best bound criterion.
N6	Backtrack. Best Estimate (adjusted pseudocost). When backtracking, select node by best bound criterion.

Figure 4.3: Node Selection Rules Investigated

Each of the node selection methods were tested for all the MIP instances in Table 4.8. One exception was the instance *arki001*, which was excluded because during the heuristic phase, we encounter a linear program which takes more than one hour of CPU time to solve. This left us with 56 problems in the test suite. Table A.4 shows the full results of this experiment, and Table 4.5 shows a summary of the results. When ranking the performance of a node selection method on a given instance, we use the following rules:

1. Methods are ranked first by the value of the best solution obtained.
2. If two methods find the same solution, the method with the lower provable optimality gap is ranked higher.
3. If both methods find the same solution and optimality gap, they are ranked according to computation time.
4. Ties are allowed.

In computing the rankings, instances where each node selection method was able to prove the optimality of the solution and the difference between best and worst methods' computation times was less than 30 seconds were excluded. This left us with 33 instances.

From the tables it is difficult to determine a clear winner among the node selection methods, but we can make the following observations:

- Pseudocost-based node estimate methods or combining a pseudocost based estimate method in backtracking seems to be the best idea for node selection.

Table 4.5: Summary of Node Selection Method Experiment

Method	Ranking (Min, Avg, Max)	Times No Sol'n Found	Times Optimal Sol'n Found	Computation Time (sec.)
N1	(1, 3.24, 6)	2	8	68189
N2	(1, 4.18, 6)	1	3	80005
N3	(1, 2.97, 6)	2	11	66147
N4	(1, 3.00, 6)	1	8	67823
N5	(1, 3.36, 6)	1	8	66475
N6	(1, 3.42, 6)	1	8	68106

- There is no node selection method which clearly dominates the others (note that almost all methods came in last at least once), so a sophisticated MIP solver should allow many different options for selecting the next node to evaluate.
- Even in the presence of a primal heuristic, depth-first search performs poorly in practice.

4.4 Pseudocosts in Parallel Branch and Bound

In this section, we begin by making some observations about the relative importance of sharing and initializing pseudocosts in a parallel algorithm for MIP. We then describe our implementation scheme and survey pseudocost sharing approaches that others have taken.

4.4.1 Observations

One of the main conclusions of our experiments about the effective use of pseudocosts in a sequential setting was that the initialization of pseudocosts is *very* important. Recall the observation of Forrest *et al.* [42] that for every “mistake” that is made in choosing a branching variable, the amount of work required to process the subtree rooted at that node doubles. This same rationale implies that the initialization of pseudocosts in a parallel algorithm is more important than in a sequential one. An example will make this point clear. Consider the partial branch and bound tree shown in Figure 4.4.1. The order in which the nodes were evaluated is indicated by the number next to the node. At node 1, variable x_{bad} is branched on. Suppose that x_{bad} is a poor choice of variable on which to branch and that this poor branching choice was made because the pseudocosts are not initialized explicitly. Subsequently, on processor 2, suppose that the variable x_{bad} is fractional at node 5. If the pseudocost information indicating that x_{bad} is a poor variable to branch on is not shared among the processors, then it is likely that x_{bad} will be branched on again at node 5. In general, we might make the same “mistake” on all p processors. Thus, we conclude that if pseudocosts are not shared, then good pseudocost initialization is more important for a parallel algorithm than for a sequential one. Using the same reasoning, one can also conclude that if pseudocosts are not initialized intelligently, then pseudocost sharing is more important for a parallel algorithm than a sequential one.

If pseudocosts are shared among the processors, then (assuming the same

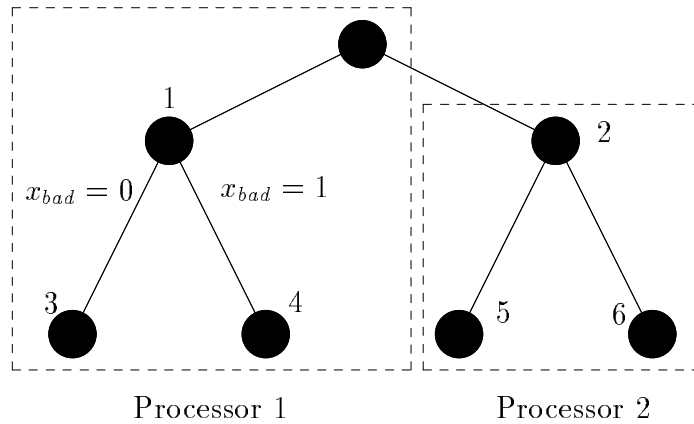


Figure 4.4: A Branch and Bound Tree

branching order) a parallel branch and bound algorithm would need to make the same number of branching decisions with uninitialized pseudocosts as would a sequential algorithm. Therefore, we conclude that if pseudocosts are shared among the processors, then pseudocost initialization has equal importance for parallel and sequential branch and bound algorithms.

Thus, regardless if pseudocosts are to be shared among the processors, we see that pseudocost initialization can only be *more* important in a parallel branch and bound algorithm than in a sequential algorithm. Since our experiments in Section 4.3.1 showed that pseudocosts initialization was important for a sequential algorithm, we will only consider schemes where the pseudocosts are explicitly initialized by the strategy presented in Section 4.3.1.

If we consider only schemes where the pseudocosts are explicitly computed as needed, what can we hope to gain by sharing pseudocosts?

- A reduction in the total time involved in initializing pseudocosts.
- A reduction in the number of nodes in the branch and bound tree (by making more intelligent branching decisions).

Reducing the total time involved in initializing the pseudocosts can be seen as an attempt to satisfy Goal II (reducing information latency) of an information sharing scheme. If pseudocosts are not shared, then many of the processors may need to perform the computations necessary to perform the pseudocost initialization. If the time required to initialize a pseudocost is sufficiently large, then we may hope see some benefit in sharing the pseudocosts by reducing the amount of time required to make a branching decision for a node.

On the other hand, it may be more beneficial for each processor to initialize its own pseudocosts. This is because pseudocosts initialized at a node are in some sense the “best” local information that the branch and bound algorithm can use when deciding on a branching variable. In this case, not sharing pseudocosts implies that the algorithm will explicitly initialize more pseudocosts, which could result in a more effective algorithm. Of course, the sequential algorithm could explicitly compute “local” pseudocosts whenever a branching decision is to be made. This is akin to the idea of *strong branching* mentioned in Section 4.3.1.

Reducing the number of nodes in the branch and bound tree is striving to achieve Goal I (maximizing useful sharing) of an information sharing scheme. In our sequential study, aggregating the pseudocost measurements taken throughout the branch and bound tree was shown to be mildly effective. When pseudocosts

are initialized, then the sole benefit of sharing pseudocost information comes from the averaging effect. One of the main conclusions of our sequential pseudocost study was that the averaging of pseudocosts was only mildly effective. In a parallel algorithm, the pseudocosts for a given variable can be collected and averaged locally at a processor. Assuming that the marginal benefit of averaging pseudocost information decreases as the number of observations increases, then the further pseudocost averaging benefit obtained by sharing the observations among the processors decreases. In a distributed setting, sharing pseudocosts information requires that messages be passed. We conjecture that the marginal benefit obtained due to sharing of pseudocosts does not outweigh the overhead incurred due to the message passing.

In Table 4.6 we summarize the conjectured importance of pseudocost initialization in a parallel algorithm depending on whether or not pseudocosts are shared. In Table 4.7 we summarize the conjectured importance of pseudocost sharing in a parallel algorithm depending on the pseudocost initialization scheme. The *Relative Importance* mentioned in the heading is the importance of performing the action in the parallel algorithm relative to a sequential algorithm.

Table 4.6: Relative Importance of Pseudocost Initialization

Pseudocosts Shared	Relative Importance of Initialization
No	More important
Yes	Equally important

Table 4.7: Relative Importance of Pseudocost Sharing

Pseudocosts Initialized	Speed of Initialization	Relative Importance of Sharing
No	(N/A)	More important
Yes	Fast	Less important
Yes	Slow	More important

4.4.2 An Implementation Scheme

Initialization

Our implementation will explicitly initialize all pseudocosts. At the root node of the branch and bound tree, it is likely that many integer variables have fractional values in the LP relaxation. In fact, as shown in Table 4.1, in many cases *all* of the variables that ever take on fractional values are fractional at the root node. On a parallel computer, we have the opportunity to divide the work necessary to explicitly compute the pseudocosts for these variables among the processors. The natural way to divide the work is a domain decomposition approach, where each processor is assigned a number of fractional variables. After computing the pseudocosts assigned to it, each processor shares the information with the other processors. We call this *parallel pseudocost initialization*. Parallel pseudocost initialization shares the pseudocosts information at the root node, so Table 4.7 tells us about the importance of performing this technique. If pseudocosts take a long time to initialize, then it is likely that parallel pseudocost initialization will be effective.

If pseudocost initialization requires little effort, parallel pseudocost initialization will likely be ineffective. A computational experiment presented in Section 4.7.2 is aimed at determining the effectiveness of parallel pseudocost initialization.

Sharing

In Section 4.4.1 we conjecture that pseudocost sharing might be beneficial in reducing the total time involved in initializing pseudocosts, and that pseudocost sharing is unlikely to significantly reduce the number of nodes in the branch and bound tree.

We wish to verify these conjectures experimentally, so we will discuss a scheme for sharing pseudocosts. First, since the target architecture for which we are designing the branch and bound algorithm does not have a large number of processors, and the marginal benefit of pseudocost information is likely small, we assume that dedicating a processor to act as a server of pseudocost information will be a waste of resources – the processor would be more useful evaluating nodes than in serving requests for pseudocost information. Therefore, we will consider a prefetch pseudocost sharing scheme. Further, a pseudocost is a small amount of information that is generated very often during the algorithm, which implies that a buffered-prefetch scheme would be most effective. The question as to an appropriate buffer size that balances the number of messages passed against the benefits of sharing will be examined.

The observations about the relative importance of pseudocost sharing and ini-

tialization made in Tables 4.7 and 4.6 all implicitly assumed pseudocosts were shared in an undelayed fashion. That is, once a pseudocost was computed, the information was immediately made available to all other processors. If there is a delay in the pseudocost information being made available to other processors, as is definitely the case in a buffered-prefetch sharing scheme, then the observations change somewhat.

For the observations of Table 4.6, the easiest way to see the effect of buffering the pseudocosts is to consider no sharing as a buffered sharing scheme with infinite buffer size and an undelayed sharing scheme as a buffered sharing scheme with buffer size zero. The larger the buffer size, the more important explicit pseudocost initialization becomes. Buffering the pseudocosts does *not* change the conclusion that pseudocost initialization is extremely important for a parallel branch and bound algorithm.

From the observations of Table 4.7, in cases where sharing pseudocosts is more important – when pseudocosts are not initialized or when pseudocost initialization is slow – it is also more important to have small pseudocost buffer sizes. If pseudocost initialization is fast, large pseudocost buffer sizes should be adequate.

A number of observations and conjectures about how to best share pseudocost information have been made in this section. Computational experiments verifying these observations are delayed until after we have cited previous work in this area and more accurately detailed our implementation.

4.4.3 Previous Work

The only published work on using pseudocosts in a parallel branch and bound algorithm for solving MIP is that of Eckstein [37] [39]. In his work, Eckstein performs a systematic study of the effects of sharing pseudocost information. He compares a master-worker pseudocost sharing scheme, a client-server sharing scheme, and a buffered-distributed pseudocost sharing scheme. Although unable to declare a “winner” among the sharing methods, Eckstein concludes that the efficiency of the branch and bound algorithm decreases as pseudocost storage become less centralized. Eckstein uses the averaging method in order to initialize the pseudocosts, which was the *least* effective initialization scheme presented in Section 4.3. The importance of pseudocost sharing may be quite different if a more effective initialization scheme is used. The observations summarized in Tables 4.7 are able to explain the observed importance of pseudocost sharing in Eckstein’s experiments [39], where pseudocosts are not initialized in an effective manner.

4.5 Node Selection in Parallel Branch and Bound

In this section, we will discuss issues related to the sharing of node information in a parallel branch and bound algorithm for MIP. In so doing, we will also survey approaches others have taken for performing the sharing. Sharing node information is very closely related to the idea of active set management for a parallel branch and bound algorithm. We must perform active set management in order to develop

a node selection scheme for the parallel algorithm.

4.5.1 Observations

When solving some instances of MIP, the size of the active set can grow very large. In order to allow as large an active set as possible (by making use of all the physical memory at our disposal), we would like to distribute the unevaluated nodes among all the processors. We consider achieving Goal IV (to minimize the memory imbalance) to be the most important feature of a node sharing scheme for parallel branch and bound. A distributed active set has the further advantage that it will reduce contention effects at the processor acting as the node server, so it also helps us achieve Goal II of an information sharing scheme.

Since there is no centralized location of the active set of nodes, information about the various active sets must be shared among the processors. The first goal of an information sharing scheme is to maximize the useful sharing of information. What information about the active sets at the processors is likely to be useful? Clearly, the most important piece of information about an active set at a processor is whether or not it is empty. If a processor has no nodes to evaluate, we would like to ensure that it receives nodes in a timely fashion; or, if no processors contain nodes to evaluate, that the program stops and declares the incumbent solution optimal. We refer to ideas concerning keeping a processor's active set nonempty as *quantity balancing*. *Termination detection* is the problem of determining when all of the processors' active sets are empty. Just as in a sequential algorithm, we

would like for all processors to be working on “non-superfluous” nodes or on nodes that are likely to lead to an improved integer feasible solution. We refer to ideas concerning keeping all processors working on “useful” nodes as *quality balancing*.

Quantity Balancing and Termination Detection

We would like some way to ensure that processors do not run out of work while others have a large number of nodes to evaluate. The most obvious way to accomplish this task is to have an idle processor broadcast a message saying it requires work. A suitable processor can then send a portion of its work to the idle processor. An obvious extension of the “wait until empty” approach of quantity balancing would be to not wait to fetch more nodes until completely running out of nodes in a active set; instead, a threshold, k , could be used so that whenever the number of nodes in the active set drops below k , a request for more work could be sent. The main drawback of this *anticipative quantity balancing scheme* is that the complexity of the implementation needs to be increased for two reasons. First, termination detection is more difficult. Secondly, the implementation must guard against the phenomenon of *thrashing*. When the number of nodes in all of the active sets are low, it may be the case that when a processor passes nodes in order to satisfy another processors request, the donating processor also falls below the node threshold k . In the extreme case, the nodes are simply passed back and forth between two processors.

Quality Balancing

We would like to ensure that each processor is doing useful work. One definition of useful work in this context is the evaluation of a non-superfluous node. By definition, if we evaluate nodes in order of the best upper bound on that node, we are always doing useful work. There are two points to make. First, since we have distributed our active set among all the processors, it is not an easy task to always ensure that a processor is working on the node with the best upper bound. Second, as pointed out in Section 4.3, even in a sequential setting, there are a variety of reasons why the best bound node selection rule is not always the most computationally effective.

There is a reason to believe that node selection rules that have a depth-first flavor are even *more* likely to outperform best first strategies in a parallel setting. Some of the information generated by the branch and bound algorithm (pseudo-costs and cuts) may be the most effective if used on nodes that are “close” to the node for which the information was first generated. In Section 5.1.5 we will provide some empirical evidence that this is indeed the case for cuts. By keeping a processor working on nodes that are local in the tree (say by searching in a depth-first manner), then the information generated locally at a processor does not have to be shared with other processors in order for it to be effective. Also, depth-first search techniques fail miserably when the search gets “stuck” in an unrewarding section of the search tree. By having many processors perform depth-first search in parallel, it is unlikely that all processors will be searching in unrewarding sections

of the tree.

4.5.2 An Implementation Scheme

Quantity Balancing and Termination Detection

Preliminary computational experience demonstrated that for nearly all MIP instances, processors run out of work very infrequently compared to the total number of nodes generated. Thus, the benefit obtained in developing an anticipative quantity balancing scheme would not justify the increased implementation complexity. Our implementation uses a naive quantity balancing and termination detection scheme, which is implemented in the following manner. When the active set at a processor becomes empty, the processor with the largest active set sends a portion of its active set to the empty processor. Once the nodes are passed, we would like for both processors to have a roughly equal amount of work. Note that this is *not* equivalent to the processors having an equal number of nodes in their active sets. Rather, we would like to balance the number of nodes stemming from the subtrees rooted at the nodes in the active sets. Since the exact size of the remaining subtree from a node is not known at the time nodes are to be passed, the amount of work cannot be balanced exactly. If the nodes $\{N^1, N^2, \dots, N^k\}$ at the donating processor are ordered so that $z_U^1 \geq z_U^2 \geq \dots \geq z_U^k$, then nodes $\{N^2, N^4, \dots, N^{2\kappa}\}$ are passed to the requesting processor. We take $\kappa = \lfloor k/4 \rfloor$ in our implementation. We limit the number of nodes passed to a maximum of κ for three reasons. First, nodes N^i with small values of z_U^i are more likely to be superfluous, so we certainly

do not want to pass them from processor to processor. Second, limiting κ keeps the size of the message passed small, and still we are able to roughly balance the amount of work between the two processors. Third, small values of κ reduce the following thrashing effect. If the same processor serves many requests for nodes, and if κ is too large, then by serving all the requests, its active set will become small, in which case, it will soon be requesting nodes of its own.

From the preceding discussion, it is not clear how a processor knows that it has the largest active set among all processors, so that it is to satisfy requests for nodes from processors whose active sets are empty. To accomplish this, there is a monitor of the branch and bound process. The monitor uses a simple “check-in” mechanism, where at fixed time intervals, the size of the active set of each processor is passed to the monitor. Requests for nodes are made to the monitor, who forwards this message to the processor with the largest active set. The discussion of the exact implementation of this monitor entity is delayed until Section 4.6.

The use of the check-in mechanism makes it a bit difficult to categorize our quantity balancing scheme as one of our basic information sharing schemes described in Chapter 3. Certainly it falls under the general category of a fetch sharing scheme – when a processor runs out of work, it makes a request for more nodes to evaluate. Since there are multiple servers of node information (each processor has its own active set), then the quantity balance scheme is most like the client-server information sharing scheme. The only difference from the basic client-server scheme is that the server designated to serve a request for information is done in

a selective fashion based on information that is known to a master, or monitor, process.

Quality Balancing

We saw in Section 4.3.2 that there was no clear best rule for selecting nodes in a sequential branch and bound algorithm for solving MIP. It seems unlikely that there is a best strategy for choosing nodes in a parallel branch and bound algorithm either. We would therefore like the flexibility to explore different node selection strategies in a parallel environment. Nearly all of the prior work in quality balancing of a distributed active set can be seen as an attempt to emulate the best bound node selection rule. We have chosen to emulate four different sequential search strategies in parallel: best bound, best estimate, a backtracking method, and a modified depth-first method. Here we will briefly describe each and explain how they are emulated.

The best bound node selection rule is emulated as follows. A weight function, or *quality*, is calculated for the active set of each processor. We define the *quality* of an active set to be the average of the largest m upper bound values of the nodes in the active set. Let w_{\max} be the maximum quality of an active set and let w_{\min} be the minimum quality of an active set. There is a quality imbalance between two active sets when

$$\frac{w_{\max} - w_{\min}}{w_{\max}} > \Delta, \quad (4.6)$$

where Δ is the *quality tolerance*. We correct the quality imbalance by swapping

unevaluated nodes between the two. A simple “shuffling scheme” where every other node from the first m in the two pools is swapped is used to balance the quality of the pools. In order to collect the active set qualities and send messages signaling that nodes should be exchanged, we use the same monitor process and polling mechanism used for quantity balancing.

To emulate the best estimate node selection rule, an estimate of the best solution obtainable z_E^i from a node N^i is computed by equation (4.5), where $q_j = 0.8 \forall j$. Note that pseudocosts are required to make this calculation, so sharing of pseudocost information in this case has some interaction with the sharing of node information (or the parallel node selection scheme). Given that we have computed the estimates z_E^i , the emulation of the best estimate node selection rule works in a similar fashion to the best bound. We simply redefine the quality of an active set to be in this case the average of the m largest estimate values z_E^i for the nodes in the set.

The goal of a backtracking method is to evaluate nodes in a depth-first fashion until the upper bound at a node falls below the estimate of the best optimal solution. To emulate the backtracking node selection rule, the values z_U^i and z_E^i are computed for each node N^i . Let $z_U^{\max}(j)$ and $z_E^{\max}(j)$ be the largest value of z_U^i and z_E^i in the active set at processor j . Define $z_E \equiv \max(z_L, \max_j z_E^{\max}(j))$ to be the estimate of the optimal solution. By a check-in and broadcast mechanism, each processor is made aware of z_E and locally performs node selection by a backtracking rule using the value z_E . If $z_U^{\max}(j) < z_E$ for some processor j , then it is deemed that

processor j has no “useful” nodes to evaluate and an exchange of nodes between processors is made in a manner similar to that used in the best bound emulation node selection rule.

The final node selection emulation scheme implemented is a modified depth-first (or *plunging*) scheme, which works in the following manner. Given a node to evaluate, depth-first moves are made until the node becomes infeasible or an integer feasible solution is obtained. Then, the node with the highest value of z_U^i is chosen to evaluate, and the process is repeated. Quality balancing is performed like in the best bound emulation scheme.

Each of the quality balancing schemes we describe can be categorized as a prefetch information sharing scheme – suitable nodes to evaluate simply appear at a processor. By using a polling mechanism and a monitor process, we are able to “intelligently” share the nodes among the processors, so we categorize each of the node selection emulation schemes as a selective-prefetch information sharing scheme.

4.5.3 Previous Work

In order to achieve a memory balance among the processors, our implementation will distribute the active set of nodes over all the processors. Eckstein [37] also cites the significant memory imbalance that occurs as a major drawback to the centralized active set approach. In this section, we survey the approaches others have taken to perform quantity and quality balancing with a distributed active set

in a parallel branch and bound algorithm. The problems of quantity and quality balancing are not unique to solving MIP in parallel, so we also mention ideas springing from “generic” parallel branch and bound implementations.

Lüling and Monien [87] suggest assigning a weight function to the active set of each processor and to attempt to balance the weight functions among processors. Given a set of nodes $N = \{N^1, N^2, \dots, N^k\}$, some weight functions they suggest are

$$w(N) = k,$$

$$w(N) = \max_j \{z_U^j\}, \text{ and}$$

$$w(N) = \sum_{j=1}^k z_U^j.$$

If $N(j)$ is the active set of node at processor j , Lüling and Monien suggest keeping

$$\frac{w(N(j)) - w(N(k))}{w(N(j))} \leq \Delta \quad \forall j, k.$$

In their work, they suggest a somewhat complex scheme for achieving this balance. Note that the concept of quantity and quality balancing are both encompassed by balancing a suitable weight function. In later work, Tschöke, Lüling, and Monien suggest that quality and quantity balancing should be performed separately [117]. The main difference between this scheme and our implementation scheme described in Section 4.5.2 is the definition of the weight function for an active set. In the case of MIP, it makes little sense to include nodes with low values of z_U^i in the calculation of quality since these nodes may not be evaluated anyway.

Dutt and Mahapatra [36] use an anticipative quantity balancing scheme in solving the traveling salesman problem, but they do not report on the effectiveness of this scheme in comparison to a nonanticipative quantity balancing scheme. In order to describe their quality balancing scheme, we assume that the set of nodes $\{N^1(j), N^2(j), \dots, N^{n_j}(j)\}$ at processor j are ordered such that $z_U^1(j) \leq z_U^2(j) \leq \dots \leq z_U^{n_j}(j)$. Processor i sends “a few good” nodes to processor j if $z_U^1(i) > z_U^5(j)$. The exact details of how many nodes are sent are omitted. Mahapatra and Dutt [89] extend this quality balancing scheme and call it the *quality equalizing* strategy. The quality equalizing strategy has been combined with a naive branch and bound algorithm to solve some instances of MIPLIB on a parallel machine with 128 processors [35].

Eckstein [37] has an interesting solution to the memory imbalance problem and to some of the contention effects that occur from a master-worker node sharing scheme. He uses *tokens* to effectively distribute the bulk of the active set. After a worker creates a node, a small “token” of information is passed to the master. The token contains the information z_U^i , the processor number that generated the node, and the local memory address of the node at the generating processor. Using this information, the master is able to allocate nodes to the workers. If a processor becomes idle, it passes a message to the master. The master then determines the processor location of a suitable node (say the node with the greatest value of z_U^i) to transfer to the idle processor. A message is sent to the processor where the node is located in order to initiate the transfer. A similar token-based scheme was

suggested by Rayward-Smith *et al.* [106].

One main drawback of the token approach is that it requires passing a large number of messages, conflicting with Goal III of our information sharing scheme. Eckstein's code was designed for the special computer architecture of the CM-5 [114]. In the CM-5 architecture, two processors could communicate rapidly and independently of all other processors. Further, Eckstein's implementation of the approach made use of a feature called *active messages* available on the CM-5. In order to make sure nodes are passed in a timely fashion from one worker to another, we would like to be able to interrupt the current work on the donating processor and initiate transfer of the node immediately. Active messages allow us to accomplish this.

A second version of Eckstein's code [37] uses a more decentralized approach, where quantity and quality balancing is performed by a simple randomized scheme due to Karp and Zhang [67]. When a new node is generated, it is sent to the active set of a randomly chosen processor. Karp and Zhang show that the expected quality balance between processors should be good for such a scheme. In his computational work, Eckstein found that the scheme is not particularly well suited for quantity balancing in solving instances of MIP [37]. A further disadvantage of such a scheme in our context is the large number of messages that are generated. Again, such a scheme was suitable for the CM-5 computer architecture.

To improve the quantity balancing performance of the randomized strategy, Eckstein [39] combined the scheme with a "rendezvous" algorithm, which was

originally conceived of as a parallel storage allocation method [24] [60]. In the rendezvous algorithm, a weight function similar to that suggested by Lüling and Monien is computed for the active set at each processor. At periodic time intervals, the average weight function $\bar{w} \equiv \sum_1^p w(N(j))/p$ of the active sets is computed. Processors whose active set weight function falls a fraction $\alpha < 1$ below \bar{w} are assigned work from processors whose weight functions are larger than $\beta\bar{w}$, $\beta > 1$. The assignment of nodes from donating processors to receiving processors is done so that the new weight functions of all processors nearly satisfy $\alpha\bar{w} \leq w(N_j) \leq \beta\bar{w}$. In order to compute the average weight function of the active sets, a broadcast-reduction scheme that takes advantage of the communication abilities of the CM-5 was used.

Eckstein [40] also designed a code with greatly reduced communication requirements, making it more amenable to parallel computers not having the sophisticated features of the CM-5. The quantity and quality balancing scheme is identical to his previous work [39], but in order to compute \bar{w} a polling mechanism is used in place of the broadcast-reduction scheme. To collect the weight functions of the active sets, a “monitor” process, which periodically queries the processors, is used.

4.6 The PARINO System

In this section, we describe PARINO (PARallel INteger Optimizer). PARINO uses the entity-FSM structure of NAYLAK since the functionality required to solve MIP can be mapped naturally to computational entities. The use of the entity-FSM

architecture allows for easy extension of PARINO’s functionality, and allows us to easily test and compare many of the ideas presented. Most features of the PARINO system can be configured or fine tuned appropriately for optimal performance for different computation and communication platforms.

We can conceptualize the parallel branch and bound algorithm as consisting of a number of interacting entities. In this section, we define the entities we use in PARINO, show how these entities interact with each other, show how to map the entities to the processors, and describe how to implement our parallel branch and bound algorithm with these entities.

4.6.1 PARINO Entities

The basic entities of PARINO are the *primer*, *distributor*, *loader*, *worker* and *logger*. Each entity has a specific set of tasks and FSMs to perform these tasks.

The primer entity is responsible for initially creating all the entities, reading the initial problem, preprocessing the problem, solving the initial LP relaxation, and passing necessary startup information to the other entities. There is only one FSM in the primer entity.

The distributor is responsible for the coordination of quantity balancing and quality balancing across the distributed active sets. It is the “monitor” of the branch and bound process. It has two FSM’s, one for quantity balancing and one for quality balancing.

Each loader entity is responsible for the maintenance of the active set at each

processor. Loaders act on commands from the distributor to achieve quantity and quality balancing. The loader has an FSM to initialize the active set, an FSM to request work, FSMs to send and receive work based on messages from the distributor, and an FSM to calculate the current load and active set quality and send this information to the distributor.

The duties of a worker are to request a node for computation from its loader, evaluate the node, generate new nodes if necessary, and report results back to the loader. The worker is also responsible for sending and receiving pseudocost information. It has one FSM for performing these tasks.

Exactly one instance of the primer entity, logger entity, and distributor entity type are created for a given run. All of these entities exist on a common processor. On all other processors save for this “controlling” processor, one instance of the loader entity and one instance of the worker entity are created.

Figure 4.5 shows a graphical description of the system. The dashed lines in the figure depict the mapping of entities to processors.

PARINO’s Quantity Balancing

Initially, the first loader’s active set contains the root node, and the active sets of the other loaders are empty. The distributor acts as the monitor process described in Section 4.5.2 and keeps track of the current levels of loads and qualities of the active sets at each loader. The loaders periodically send updates of their current loads and qualities to the distributor. In order to avoid contention effects at the

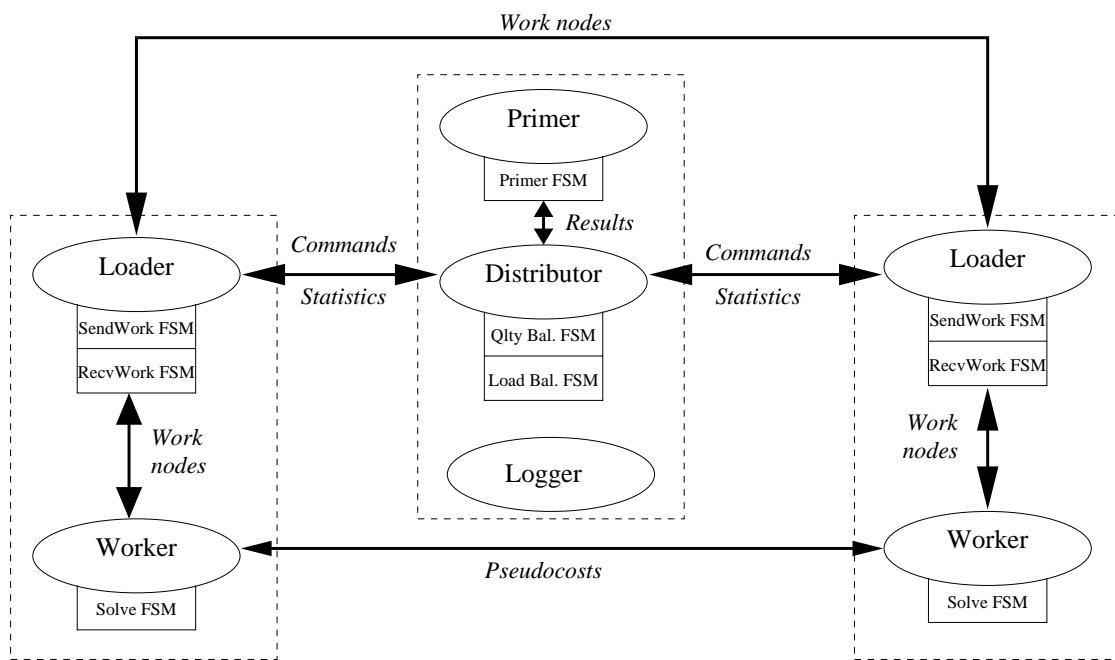


Figure 4.5: PARINO System Architecture

processor containing the distributor, and to make the implementation scalable, the rate γ of “check-ins” to the distributor is kept relatively constant regardless of the number of processors. Namely, the loader sends updated quantity and quality information to the distributor every p/γ seconds, where p is the number of loaders.

Whenever a worker asks its loader for a node from the active set and the loader finds its active set empty, the loader sends an out-of-work message to the distributor entity and waits for a response. The distributor then arranges for some unevaluated nodes to be sent from an appropriately selected loader to this loader. The distributor never actually sends or receives any nodes, it only directs the loaders to exchange nodes appropriately. The quantity-balancing FSM of the distributor acts on out-of-work messages from the loaders. All out-of-work messages are collected in a FIFO queue. Suppose the first request in the queue is from L_i . Let L_j be the loader with the maximum load (based on the most recent information available to the distributor). The distributor sends a message to L_j asking it to send part of its active set to L_i , and waits for a response from L_j . Loader L_j responds with a success status if and when it has successfully completed the transfer of part of its active set to loader L_i . If L_j could not transfer any nodes at all (which is possible if L_j had just run out of work itself), it responds with a failure status. If the response is success, then the distributor dequeues the out-of-work request from L_i from its queue, and sends the loader L_i a success response. Loader L_i is at that point guaranteed to have a non-empty active set. The distributor’s quantity-balancing FSM then moves on to process the next out-of-work message,

if any, from other loaders. If the response from L_j is that of failure, then the distributor does not dequeue the request of L_i , but instead continues the preceding steps all over again – this time using the updated information of the quantity levels which were implied by the failure message. The distributor uses the same request queue to detect termination of the parallel branch and bound search – termination is detected if all the loaders are waiting on the queue.

Quality Balancing Mechanism

The quality-balancing FSM of the distributor continually checks if the qualities of the active sets need to be balanced based on the particular node selection emulation scheme. Recall that updates about the quality of each active set are arriving to the distributor at the rate of γ per second. Every p/γ seconds, the distributor checks to see if nodes must be swapped in order to balance the qualities of the active sets. As discussed in Section 4.5.2, the criterion under which the decision to swap nodes is to be made depends on the node selection emulation scheme being used. For the best bound and best estimate node selection emulation methods, when the inequality (4.6) is violated, the distributor sends messages to the loaders with qualities w_{\max} and w_{\min} directing them to exchange nodes.

The implementation of the backtracking emulation method requires a bit more explanation. Along with the quality and quantity information passed to the distributor, each loader j passes an estimate of the best solution among the nodes in its active set ($z_E^{\max}(j)$) and the best upper bound ($z_U^{\max}(j)$). With this information,

the distributor can compute $z_E \equiv \max(z_L, \max_j z_E^{\max}(j))$. If z_E has changed since the last quality balancing step, the new value of z_E is broadcast to the loaders, so that depth-first moves can be made based on this new value. As described in Section 4.5.2, if $z_U^{\max}(j) < z_E$, the distributor directs loader $k \equiv \arg \max_j z_U^{\max}(j)$, and loader j to swap nodes as in the best bound emulation scheme.

Pseudocost Update Mechanism

The task of handling the pseudocosts is done by the workers. Each worker has a local pseudocost table that is used to make branching decisions. Pseudocosts generated by the processor are continually averaged into the local table. If pseudocosts are to be shared among the processors, then each worker also has a table (or buffer) containing the most recent pseudocost observations. When “enough” pseudocosts are collected (the buffer is full), the pseudocosts in the buffer are passed to the other workers, and the buffer is flushed. When worker i receives pseudocosts from worker j , the pseudocosts are immediately averaged into worker i ’s local pseudocost table.

Parallel pseudocost initialization is handled as follows. When each worker is initialized, it is passed (among other things) the problem formulation, the basis of the solution to the linear programming relaxation of the root node of branch and bound tree, and an iteration limit. The worker first loads the formulation and initial basis. Suppose that the fractional integer variables in the initial LP solution are x_1, x_2, \dots, x_l . Worker j computes the initial pseudocosts for variables

$x_{[(j-1)l/p]+1}, x_{[(j-1)l/p]+2}, \dots, x_{[j]l/p}$ (i.e. the j th fraction of the fractional integer variables). The initialized pseudocosts are immediately passed to the other workers.

The parallel pseudocost initialization can be thought of as performing the “domain decomposition” approach to parallel programming, but there is no synchronization that occurs at this step. If for some reason a worker does not receive pseudocosts by the time it needs to make a branching decision, then the worker will go ahead and compute the unknown pseudocosts itself.

4.6.2 PARINO Specifics

PARINO interacts with a linear programming solver through an objected oriented class library LPSOLVER [84]. Currently, LPSOLVER is available for CPLEX versions 4 and 5 [28], but its object oriented design will allow for easy porting to other linear programming solvers. PARINO is written in C++ and contains over 15,000 lines of code, not including NAYLAK and LPSOLVER. For the architectures on which we test, PARINO has been compiled with the GNU g++ compiler, version 2.8.1, with optimization level -O3. For the large majority of our experiments, the NAYLAK library (and hence PARINO) has been configured to use the the MPICH implementation of the MPI 1.0.8 message passing interface standard [49].

4.7 Computational Results

In this section, we perform a variety of experiments aimed at testing the implementation ideas for parallel branch and bound that we have described.

4.7.1 Preliminaries

Hardware

Our computational experiments on parallel architectures were performed on clusters of Pentium II servers running Solaris x86 2.5.1. There were two cluster of these machines at our disposal. The first cluster, called **beetle**, consists of 48 dual Pentium II 300 MHz servers, each with 512MB RAM. The beetle machines are linked via Fast Ethernet to a Cisco 5500 network switch. The second cluster, referred to as **danish**, consists of 16 Quad Pentium Pro 200 MHz servers, each with 256MB RAM. The danish machines are linked via Fast Ethernet to a Cisco 5505 network switch. When reporting the results of experiments, we will note on which of these clusters the experiments were run.

Due to the large amount of time necessary to perform all the experiments we require to fully test our ideas, obtaining dedicated access to a cluster was impossible. However, the load on the clusters of machines was light enough so that for each experiment, processors could be dedicated to running PARINO.

Default Algorithm

Since there are many implementation choices to be made in the branch and bound algorithm, comparing the performance of each of the possible combinations would require too much effort. Instead, we will give choices for a “default” parallel branch and bound algorithm that performs well and requires limited communication among the processors. Figure 4.6 describes the implementation choices of our default algorithm.

- A diving primal heuristic is performed every 10 nodes of the branch and bound tree.
- The best bound emulation node selection rule is used with parameters $\Delta = 5\%$ in the inequality (4.6) and a check-in rate (described in Section 4.6.1) of $\gamma = 6$.
- Lifted knapsack covers and lifted simple generalized flow cover inequalities are added, but not shared among the processors.
- Pseudocost based branching is used, with explicit initialization of all pseudocosts as described in Section 4.3.1. T in equation (4.2) is two minutes.
- Pseudocosts are not shared among the processors, and parallel pseudocost initialization is not performed.

Figure 4.6: Default PARINO Settings

Test Suite

The problems on which we experiment are taken from MIPLIB 3.0 [16], which is a public repository of mixed integer programming instances arising mostly from real-world applications. Table 4.8 shows the size and number of integer variables for the instances of MIPLIB. The heading “0/1” is the number of binary variables. The final two columns of Table 4.8 show the results of running the default implementation of PARINO on the **beetle** cluster with one worker entity for a maximum of one hour. The Final Gap is computed by equation (4.1).

Table 4.8: Statistics of All MIP Test Instances

Name	Rows	Cols	Integers	0/1	Continuous	Sol Time	Final Gap
10teams	230	2025	1800	ALL	225	3600	1.18%
air03	124	10757	10757	ALL	0	46	0%
air04	823	8904	8904	ALL	0	3600	1.10%
air05	426	7195	7195	ALL	0	3600	1.56%
arki001	1048	1388	538	415	850	3600	(N/A)
bell3a	123	133	71	39	62	411	0%
bell5	91	104	58	30	46	53	0%
blend2	274	353	264	231	89	406	0%
cap6000	2176	6000	6000	ALL	0	3600	(N/A)
danoint	664	521	56	ALL	465	3600	4.2%
dcmulti	290	548	75	ALL	473	32	0%
egout	98	141	55	ALL	86	1	0%
enigma	21	100	100	ALL	0	190	0%
fiber	363	1298	1254	ALL	44	20	0%
fixnet6	478	878	378	ALL	500	13	0%
flugpl	18	18	11	0	7	35	0%
gen	780	870	150	144	726	1	0%

Name	Rows	Cols	Integers	0/1	Continuous	Sol Time	Final Gap
gesa2	1392	1224	408	240	672	3600	0.42%
gesa2_o	1248	1224	720	336	504	3600	0.044%
gesa3	1368	1152	384	216	768	126	0%
gesa3_o	1224	1152	672	336	480	131	0%
gt2	29	188	188	24	0	792	0%
harp2	112	2993	2993	ALL	0	3600	0.16%
khb05250	101	1350	24	ALL	1326	3	0%
l152lav	97	1989	1989	ALL	0	349	0%
lseu	28	89	89	ALL	0	10	0%
misc03	96	160	159	ALL	1	12	0%
misc06	820	1808	112	ALL	1696	6	0%
misc07	212	260	259	ALL	1	3520	0%
mitre	2054	10724	10724	ALL	0	3600	(N/A)
mod008	6	319	319	ALL	0	55	0%
mod010	146	2655	2655	ALL	0	10	0%
mod011	4480	10958	96	ALL	0	3600	13.3%
modglob	291	422	98	ALL	0	3600	0.26%
p0033	16	33	33	ALL	0	1	0%
p0201	133	201	201	ALL	0	6	0%
p0282	241	282	282	ALL	0	9	0%
p0548	176	548	548	ALL	0	11	0%
p2756	755	2756	2756	ALL	0	108	0%
pk1	45	86	55	ALL	31	3600	54.9%
pp08a	136	240	64	ALL	176	3600	7.01%
pp08aCUTS	246	240	64	ALL	176	3600	7.27%
qiu	1192	840	48	ALL	792	3600	180.8%
qnet1	503	1541	1417	1288	124	41	0%
qnet1_o	456	1541	1417	1288	124	35	0%
rgn	24	180	100	ALL	80	54	0%
rout	291	556	315	300	241	3600	4.63%

Name	Rows	Cols	Integers	0/1	Continuous	Sol Time	Final Gap
set1ch	492	712	240	ALL	472	3600	(N/A)
seymour	4944	1372	1372	ALL	0	3600	(N/A)
stein27	118	27	27	ALL	0	65	0%
stein45	331	45	45	ALL	0	2800	0%
vpm1	234	378	168	ALL	210	3	0%
vpm2	234	378	168	ALL	210	3600	0.86%

Many of the problem instances are “easy” in the sense that they take a very short amount of time to solve by a sophisticated branch and cut algorithm. We will focus our attention on the more difficult instances, where parallelism is likely to be helpful.

Variance in Parallel Algorithms

Due to the asynchronous nature of the algorithm, the nondeterminism of running times of various components of the algorithm, and the nondeterminism of the communication times between processors, the order in which the nodes are searched and the number of nodes searched can vary significantly when solving the same problem instance. Other researchers have noticed the stochastic behavior of asynchronous parallel branch and bound implementations [37] [32]. Because we were unable to have dedicated access to the cluster for all the experiments, the network traffic during the time of the experiment could potentially vary significantly, which further increases the variance of the algorithm’s performance.

Tables 4.9 and 4.10 give an indication of the magnitude of the variance. Each of the instances in this table was run seven times using the default version of PARINO

on sixteen processors in the **beetle** cluster. The symbol σ is used to denote the standard deviation of either the number of nodes or the computing time.

Table 4.9: Run Variance for Solved Instances

Name	Number of Nodes				Time (sec.)			
	Min	Avg	Max	σ	Min	Avg	Max	σ
air04	5104	6927.3	8824	1385.80	2280.0	2941.7	3550.0	466.62
bell5	14186	16039.3	18552	1765.02	8.7	11.3	15.5	2.15
blend2	3305	4399.7	5378	672.587	19.5	22.5	28.3	2.92
gesa2_o	89659	105707.4	128211	14121.1	505.0	637.1	792.0	96.47
l152lav	1717	3015.6	3993	852.418	28.1	36.6	44.2	5.69
misc07	51307	53670.1	57779	2319.60	152.0	162.3	173.0	7.61
p2756	1167	1286.7	1596	149.853	50.0	59.0	72.3	7.90
stein45	104423	109071.7	112926	2898.93	131.0	135.6	141.0	4.28
vpm2	98854	106135.6	126030	9929.89	151.0	165.4	198.0	16.52

For the remaining experiments, a concerted effort was made to run each instance multiple times to reduce the effect of this variance. We would like to ensure that the variations we observe between different strategies come from differences in the schemes themselves, rather than from randomness in the algorithm.

Before presenting the computational results, it is appropriate to give a brief warning. When comparing computational strategies in the branch and bound algorithm, it is in general difficult to draw conclusions about the relative performance of various strategies – what works well on one instance may not work well

Table 4.10: Run Variance for Unsolved Instances

Name	Number of Nodes				Final Gap (%)			
	Min	Avg	Max	σ	Min	Avg	Max	σ
pp08a	957733	1143938.8	1236064	103536.	2.18	3.06	3.98	0.61
rout	69438	90476.4	106012	11704.1	2.72	4.06	4.61	0.66

on another. Due to the randomness introduced by an asynchronous implementation, this problem is only compounded when trying to compare parallel branch and bound algorithms, which is why we often rely on intuition generated from sequential studies or on some underlying theory in order to search for trends and rationalize the choices we make in our implementation.

Speedup of the Default Algorithm

To show that the default algorithm performs reasonably well, ten instances from Table 4.8 were solved to optimality on 2, 4, 8, 16, and 32 processors in the **beetle** cluster. Each of the instances was solved three times. Table 4.11 shows the average number of nodes and average time required to solve each instance. The column heading p stands for the number of processors, \bar{N} denotes the average number of nodes required, and \bar{T} is the average solution time in seconds. If $\bar{N}(p)$ is the average number of nodes required on p processors, then $\Delta \% \equiv 100(\bar{N}(p) - \bar{N}(2))/\bar{N}(2)$. The minimum size of a configuration is $p = 2$, which means there is one processor evaluating nodes. The definitions of speedup and efficiency from Chapter 3 do

not directly apply, since we have no value for $T(1)$, the time required to solve the instance on one processor. Instead, we compute a relative efficiency

$$\hat{E}(p) = \frac{T(2)}{(p-1)T(p)}. \quad (4.7)$$

Table 4.11: Speedup of the Default Algorithm

Name	p	\bar{N}	σ	Δ %	\bar{T}	σ	\hat{E}
vpm2	2	117977.0	0.00	0.00	4696.7	5.77	1.0000
vpm2	4	137499.0	4904.02	16.55	1520.0	43.59	1.0300
vpm2	8	113963.0	8477.78	-3.40	421.0	61.51	1.5937
vpm2	16	117569.7	8691.88	-0.35	195.3	25.15	1.6030
vpm2	32	125106.0	8632.14	6.04	106.2	7.52	1.4271
misc07	2	57849.0	0.00	0.00	2876.7	20.82	1.0000
misc07	4	59773.7	4530.15	3.33	949.3	53.72	1.0101
misc07	8	61594.7	1761.12	6.47	423.0	18.00	0.9715
misc07	16	55644.0	3272.25	-3.81	175.3	12.86	1.0938
misc07	32	50104.3	2053.73	-13.39	81.2	3.56	1.1428
stein45	2	110587.0	0.00	0.00	2373.3	5.77	1.0000
stein45	4	107981.0	2809.62	-2.36	632.7	20.84	1.2504
stein45	8	109022.3	1159.91	-1.41	280.3	6.11	1.2094
stein45	16	110098.7	3949.72	-0.44	144.3	2.08	1.0962
stein45	32	105751.3	396.42	-4.37	80.8	5.98	0.9475
air04	2	4551.0	0.00	0.00	38766.7	208.17	1.0000
air04	4	8666.3	1094.51	90.43	17466.7	3450.12	0.7398
air04	8	7714.7	1112.75	69.52	6020.0	849.23	0.9199
air04	16	7089.0	1073.70	55.77	2733.3	298.38	0.9455
air04	32	12813.0	3129.32	181.54	2946.7	654.24	0.4244
gesa2_o	2	71563.0	0.00	0.00	6260.0	10.00	1.0000
gesa2_o	4	94465.3	4932.39	32.00	2750.0	113.58	0.7588
gesa2_o	8	130806.0	4623.46	82.78	1713.3	41.63	0.5220

Name	p	\bar{N}	σ	$\Delta \%$	\bar{T}	σ	\hat{E}
gesa2_o	16	110006.7	15558.37	53.72	699.0	81.73	0.5970
gesa2_o	32	141181.3	5213.85	97.28	499.3	31.21	0.4044
blend2	2	4477.0	0.00	0.00	339.3	1.15	1.0000
blend2	4	3083.0	272.94	-54.09	51.8	4.60	3.2723
blend2	8	2249.0	173.95	-66.51	19.2	1.34	3.7971
blend2	16	4046.5	60.10	-39.74	20.8	0.78	1.6353
blend2	32	5721.0	876.36	27.79	30.6	4.32	0.3573
l152lav	2	1711.0	0.00	0.00	291.0	2.00	1.0000
l152lav	4	2022.0	59.40	-21.22	80.2	0.92	1.8131
l152lav	8	1886.0	1537.25	-26.51	39.8	20.93	1.5668
l152lav	16	1986.5	256.68	-22.60	35.8	4.17	0.8140
l152lav	32	1725.7	560.82	0.86	31.5	2.78	0.2983
p2756	2	141.0	0.00	0.00	91.0	0.21	1.0000
p2756	4	265.7	16.77	88.42	46.2	0.66	0.6568
p2756	8	814.0	29.14	477.30	58.1	3.63	0.2237
p2756	16	1512.7	83.72	972.81	60.0	5.95	0.1011
p2756	32	2644.3	213.97	1775.41	84.2	1.19	0.0349
bell5	2	9821.0	0.00	0.00	41.8	0.26	1.0000
bell5	4	15206.0	1404.31	3.22	22.7	0.85	0.9207
bell5	8	100056.5	122262.30	579.20	70.3	84.43	0.1274
bell5	16	73746.5	85755.79	400.60	33.9	32.39	0.1233
bell5	32	1815893.3	3095418.10	18389.90	618.7	1023.04	0.0022

In most cases the default algorithm performs well, and we even see many cases where “superlinear speedup” of $\hat{E} > 1$ is achieved. A notable exception is the instance *bell5*. Note that the standard deviation of the measures of performance for *bell5* are also very high; the number of nodes required to prove optimality were 23,966, 33,544, and 5,390,170. Solving *bell5* requires an enormous number of nodes if a bad search strategy is used. Table A.4 shows that only three of the sequential node selection techniques were able to solve the problem. In the case where over 5

million nodes were required to solve the problem, the quality balancing procedure did not ensure that the best bound node selection rule was emulated “well-enough,” and the search order did not happen to find the optimal solution until after a large number of nodes were evaluated. This is a (somewhat-extreme) example of how much the behavior of a parallel branch and bound algorithm can vary, and it gives motivation to study how to perform parallel search effectively.

4.7.2 Pseudocosts

For our pseudocost experiments, we have chosen to perform testing on the instances *bell3a*, *gesa2_o*, *misc07*, *p2756*, *stein45*, *vpm2*, *air04*, *air05*, *mod011*, *pp08a*, *rout*. These instances constitute a good blend of small and large problems. All experiments in Section 4.7.2 were run on 16 processors of the **danish** cluster. Each instance/setting combination was run three times in order to reduce the chances of randomness skewing the results.

The Effect of Pseudocost Buffer Size

We first performed an experiment aimed at determining the effect of varying the pseudocost buffer size. For each of the instances in our test suite, pseudocosts were shared in a buffered-distributed manner with buffer sizes of 1, 25, 100, 1000, and ∞ . A buffer size of 1 is the broadcast-prefetch sharing scheme, and a buffer size of ∞ means that pseudocosts are not shared among the processors. Tables B.1, B.2, and B.3 list the full numerical results for the experiment.

In order to best interpret the results, we divide the instances in our test suite into two classes depending on the length of time required to solve the initial linear

programming relaxation. The time to solve the initial linear programming relaxation is a good indicator of both how quickly a node of the branch and bound tree can be evaluated and how quickly pseudocosts can be explicitly initialized. The first class consists of the instances *bell3a*, *misc07*, *stein45*, *vpm2*, *pp08a*, and *rout* for which the initial linear programming relaxation takes less than 200 milliseconds to solve. The second class consists of the remaining instances, for which more than 200 milliseconds are required to solve the initial linear programming relaxation. Table 4.12 shows the average ranking of the performance of PARINO using different pseudocost buffer sizes on the two classes of instances. The ranking method presented in Section 4.3.2 is used to rate the performance.

Table 4.12: Summary of Pseudocost Buffersize Experiment

	Average Ranking	
Buffer Size	Class I	Class II
1	4.17	2.60
25	3.33	2.00
100	2.83	2.20
1000	2.50	4.00
∞	1.67	4.20

Clearly for instances in class I, there is no need to share pseudocost information between processors – the overhead for performing the sharing is too great. For the instances in class II, some positive effects of sharing can be seen. The relative unimportance of sharing pseudocost information is different than the observation of Eckstein [39]. However, Eckstein does not explicitly initialize pseudocosts, so

our observations in Table 4.7 tells us that indeed sharing pseudocosts *should* be very important in this case.

Parallel Pseudocost Initialization

We saw in the last section that there clearly is some benefit of sharing pseudocosts when solving large problems. Can parallel pseudocost initialization lessen this effect, so that pseudocosts do not need to be shared? Recall the conclusion drawn from Table 4.1 that if an integer variable is nonbasic at the root node, then it rarely takes on a fractional value in a solution at any point of the branch and bound tree. This implies that explicitly initializing the pseudocosts may need to be done only at the root node and that a parallel initialization scheme may be very effective. Tables B.4, B.5, and B.6 show the results of parallel pseudocost initialization when compared to a scheme where the pseudocosts are not shared, and a scheme where the pseudocosts are not initialized in parallel, but are shared in a buffered-distributed manner with buffer size 1.

Again, the difference in performance between the methods is not large. For the instances requiring a large amount of time to explicitly initialize pseudocosts (*air04* and *air05*), there is some positive benefit of initializing the pseudocosts in parallel. However, it appears that for these instances, it also makes sense to share pseudocost information among the processors. The positive benefit of the sharing comes from avoiding the duplicate work of initialization at non-root nodes. We conclude that the best pseudocost sharing scheme for large instances combines parallel pseudocost initialization with a broadcast of pseudocost initialization information. From the results summarized in Table 4.1, we expect the number of broadcasts of pseudocost initialization information to be small.

This concludes our experiments with pseudocosts. From the results of this chapter, we make the following recommendation about pseudocost use in a parallel branch and bound algorithm.

- Pseudocosts should be explicitly initialized.
- If the solution time of the initial linear programming relaxation is small, then there is no need to initialize the pseudocosts in parallel or share the pseudocosts.
- If the solution time of the initial linear programming relaxation is large, then initialize the pseudocosts in parallel. Whenever a pseudocost is explicitly initialized in the branch and bound tree, broadcast the initialization information.

4.7.3 Active Set Management

In this section, we test the effectiveness of the various node selection emulation schemes introduced in Section 4.5. The methods under investigation are a best bound emulation method, a best estimate emulation method, a backtracking method, and a plunging method. Preliminary computational testing showed that $\Delta = 5\%$ in Equation (4.6) was an appropriate choice for the quality tolerance, so we use this value in both the best bound and best estimate emulation schemes.

On both eight and sixteen processors of the **danish** cluster, each of the node selection emulation schemes was used to solve the instances *air04*, *air05*, *bell5*, *gesa2-o*, *misc07*, *p2756*, *qiu*, *rout*, *set1ch*, and *vpm2* three times. The full results of the experiment are listed in Tables B.7 and B.8. In these tables, we report the

average number of nodes and the average percentage gap $(z_{OPT} - z_L)/z_{OPT}$ between the best solution found by the algorithm z_L and the optimal solution z_{OPT} . We also report the average provable optimality gap (from Equation (4.1)) and the average solution time. Table 4.13 shows the average ranking of the different schemes using the ranking scheme presented in Section 4.3.2.

Table 4.13: Average Performance Ranking of Node Selection Schemes

Selection Scheme	Avg. Rank 8 Proc.	Avg. Rank 16 Proc.
Best Bound	2.5	2.7
Best Estimate	2.6	1.9
Backtracking	2.6	2.4
Plunging	2.5	3.1

We make two observations about Table 4.13. First, the plunging method begins to perform worse as the number of processors increases, but the backtracking method does not suffer from a similar performance degradation. From this, we conclude that depth-first search methods can provide effective and scalable search techniques in a parallel algorithm if used intelligently. A second observation is the improved performance of the best estimate method as the number of processors increases. Finding good feasible solutions early in the branch and bound procedure means that the burden of trying to ensure that processors are working on “non-superfluous” nodes is lessened, and parallelism can have a large benefit. From the results of this experiment, we conjecture that a backtracking method that selects the node with the best estimate value when backtracking should be a very effective node selection technique to emulate in a parallel environment.

To present other summary statistics, we break the instances up into two classes.

The first class consists of instances solved to proven optimality in all cases in less than one hour: *bell5*, *gesa2_o*, *misc07*, *p2756*, and *vpm2*. Table 4.14 shows the total number of nodes (NN) and the total time (T) required to solve all instances in the first class.

Table 4.14: Summary Statistics for Node Selection Methods Experiment – Solved Instances

Selection Scheme	8 Proc.		16 Proc.	
	NN	T	NN	T
Best Bound	937161	9515.9	826197	4669.2
Best Estimate	798019	6878.2	884777	3817.8
Backtracking	1049192	8811.5	1753242	5123.9
Plunging	982346	9151.2	3983495	6579.3

The best estimate estimate node selection rule seems to be the “winner” in terms of the total time on these problems. A significant portion of the difference in the best estimate method with the other methods in terms of time is due to the performance on the instance *gesa2_o*. As mentioned in Section 4.7.1, depth-first search type node selection rules perform very badly on the instance *bell5*. To some extent, this accounts for the large number of extra nodes evaluated by the backtracking and plunging methods – especially on 16 processors.

The second class consists of problems not always solved to proven optimality in less than one hour: *air04*, *air05*, *qiu*, *rout*, and *set1ch*. Table 4.15 shows the total number of nodes evaluated and the number of cases in which the instance was solved to proven optimality for the problems in the second class.

Table 4.15: Summary Statistics for Node Selection Methods Experiment – Instances Sometimes Solved

Selection Scheme	8 Proc.		16 Proc.	
	NN	# Solved	NN	# Solved
Best Bound	900895	2	1811278	5
Best Estimate	998168	3	1933281	7
Backtracking	1344186	3	2552630	7
Plunging	1336534	5	2641998	6

The best estimate and backtracking methods seem to be performing the best. One surprising result is that the best estimate node selection method is able to find the optimal solution to *rout* and prove it optimal in less than one hour. This is a very difficult MIP instance, either unsolvable or requiring many hours to find the solution on sophisticated MIP solvers such as MIPO [101] or *bc-opt* [27].

This concludes our experiments on node selection techniques for a parallel branch and bound algorithm. Based on the discussion and results of this chapter, we make the following suggestions.

- A combination of backtracking with best estimate techniques appears to be a good parallel node selection method.
- No one node selection emulation technique dominates the performance, so a sophisticated solver should allow a wide range of different node selection strategies.

CHAPTER 5

Parallel Branch and Cut

Branch and bound alone is not enough to solve many large MIP instances. The LP relaxation to MIP must be systematically strengthened by introducing cuts. Parallel branch and cut can offer advantages over its sequential counterpart. In this chapter, we deal with the question of how to effectively share cuts in a parallel branch and cut system. The first section will be a study of cutting planes in a sequential environment. The second section will discuss issues related to sharing cuts among the processors in a parallel algorithm and will provide a framework for experimenting with various cut sharing schemes. The next section describes the implementation of our cut sharing schemes in PARINO. Finally, extensive computational experiments comparing cut sharing schemes are performed and computational results are reported.

5.1 Effectively Using Cuts

The effective management of cutting planes in the branch and cut algorithm has been the subject of very little research. Balas *et al.* [8] do a thorough examination of the computational issues arising when incorporating disjunctive cuts into

a branch and cut algorithm, and Bauer *et al.* [12] perform a limited study of cut management in the context of a branch and cut approach to solving a specific combinatorial problem – the cardinality constrained circuit problem. For a branch and cut algorithm based on lifted cover inequalities, the author is aware of no published research on effective cut handling strategies. The thesis of Gu [50] mentions issues related to cut management but does not offer any guidelines about implementing a cut management strategy. For some problem instances, a significant number of lifted cover inequalities are found, and a number of computational issues arise that should be addressed. A few of these issues are

- Should cuts be generated at every node of the search tree?
- When generating cuts for a node, should the separation procedure be repeated as long as it is successful?
- Should all the generated cuts for a given fractional LP point be used?
- Should weak cuts be deleted from the active linear programming formulation, and if yes, then how often?

An even more important issue, that is at the heart of any cut management scheme, is how to measure the importance of a cut with respect to the overall effectiveness of the algorithm. In this section, we hope to provide some insight in what makes a cut important as well as answers to the more practical computational issues raised above. We will do so on the basis of several computational experiments with lifted knapsack and flow cover inequalities.

Given the large number of choices that have to be made when designing a cut management scheme, we have again chosen the approach of defining a suit-

able “default” strategy, and investigating the effects of varying from this strategy. Figure 5.1 lists the features of our default cut management strategy.

- Cuts are generated at every 10 nodes of the search tree.
- We stop generating cuts at a node if the percentage change in the objective function value over the last three solutions to the linear program is less than 0.2% at the root node and 0.5% at every other node.
- At most 20 inequalities that cut off the current fractional solution are added to the linear program at one time. The cuts added are the ones for which $(b - a^T x^*)/b$ is the largest.
- Cuts are deleted from the linear programming formulation if the dual variable associated with the cut has been 0 in 50 consecutive solutions to the linear programming relaxation.

Figure 5.1: Default Cutting Plane Strategy for Sequential Experiments

Ten instances from Table 4.8 for which knapsack and flow cover inequalities are found were used as our test set. Table 5.1 shows the instances in our test set, as well as the number and type of inequalities added during the solution by the default branch and cut algorithm. All experiments in this section were performed with the sequential mixed integer programming package MINTO (v3.0) [95].

5.1.1 How Often to Generate Cuts

The first experiment is aimed at determining an appropriate balance between cutting and branching. In this experiment, cuts are generated at the root node of the branch and bound tree, and every κ nodes thereafter. If $\kappa = \infty$, then cuts

Table 5.1: Instances Used for Sequential Cut Strategy Study

Name	Knapsack Covers	Flow Covers
blend2	382	183
cap6000	1436	0
fiber	88	148
harp2	2978	0
l152lav	23	0
mitre	4423	0
mod011	0	130
modglob	0	359
p2756	772	0
vpm2	19	90

are generated at the root node only, and the procedure is sometimes called *cut and branch*. Balas *et al.* [8] call κ the *skip factor*. Each of the instances in Table 5.1 was solved using various values of κ . Table 5.2 shows a summary of this experiment. All cut strategy settings except for at which nodes cuts are generated are as described in Figure 5.1. The method used to rank the performance on an instance is the ranking method presented in Section 4.3.4. The full results of the experiment are presented in Table C.1 in Appendix C.

From Table 5.2, we see the tradeoff between cutting and branching – more cuts lead to fewer nodes. However, adding cuts at every node of the branch and bound tree may not be the most computationally efficient strategy. The ramifications of this observation for a parallel branch and cut algorithm will be discussed in more detail in Section 5.2.

Table 5.2: Summary of Skip Factor Experiment

κ	Average Ranking	Total Cuts	Total Nodes
1	3.3	34825	62824
5	3.6	18391	83783
10	3.4	12500	85420
100	2.7	6307	107572
1000	3.8	5167	98604
∞	3.2	5019	119040

5.1.2 Tailing Off

The change in the value of the objective function of the LP relaxation has a tendency to decrease as the number of rounds of cut generation increases at a node in the search tree. This is known as the *tailing-off* effect. If the change in the value of the objective function from one round of cut generation to the next becomes “too small”, we may decide to branch rather than to generate additional cuts, since it appears that spending more time on cut generation is not worthwhile at the moment. If the change in the value of the objective function is less than $\alpha\%$ over the last two rounds of cut generation, we say tailing-off has occurred, and no more cutting planes will be added at the current node.

Table C.3 in Appendix C shows the results of an experiment designed to show the effect of varying α . For the instances in our test suite, the tailing off percentage seems to have little effect on the overall performance of our branch and cut algorithm.

5.1.3 The Number of Cuts Per Round

For each relaxation of the problem that gives rise to a structure from which we can derive valid inequalities, there is a separation problem to be solved. For example, we can find violated knapsack cover inequalities from each row of the matrix having the form $\sum_{j \in B} a_j x_j \leq b$. Solving all these separation problems is called a *round* of cut generation. In a round of cut generation, many different inequalities cutting off the same fractional point x^* may be found. To conserve memory and to speed up the solution of the active linear program, it may be more efficient to add only a subset of the generated violated inequalities, for example the λ “best” cuts. We have used the relative violation $(b - a^T x^*)/b$ from the LP solution x^* to the hyperplane $a^T x = b$ defining a cut as the measure of quality of a cut. If $b = 0$, then we use the absolute violation $b - a^T x^*$ as the measure of quality.

Table C.2 in Appendix C shows the results of an experiment where for each instance in Table 5.1 we added at most $\lambda = 1, 10, 20, 50$, and all violated inequalities found by the separation problems to the linear programming relaxation. Cut strategy decisions except for the number of cuts to add per round were as shown in Figure 5.1. The choice of $\lambda = 1$ was generally inferior to the other choices of λ , however it is very difficult to determine any other trends. The results of this experiment show that the issue of the number of cuts to add per round is not an extremely important one for building a branch and cut algorithm based on lifted cover inequalities.

5.1.4 Deleting Cuts from the Active Formulation

If the dual variable associated with a cut has value zero in the solution to a linear programming relaxation, then the removal of this cut would not worsen the objective function, and the cut is deemed *useless* for this particular linear program. We would like to have very few useless cuts in our linear programming relaxation, since their inclusion will slow down the linear programming procedure. For each cut, we keep a counter of the number of consecutive rounds for which it has been useless. Cuts that have been useless for the last η rounds are removed. Removed cuts are not deleted entirely, but instead placed in a cut pool. If inequalities in the cut pool are violated by fractional solutions at subsequent nodes, they are returned to the linear programming formulation.

Table C.4 gives the results of an experiment aimed at determining the effect of varying η for the instances in our test suite. Except for the removal of useless inequalities, all other cut strategy decisions are made as listed in Figure 5.1. Table 5.3 gives a summary of the experiment.

Table 5.3: Summary of Weak Cut Deletion Experiment

η	Average Ranking
1	1.4
25	2.3
50	2.5
100	3.1
∞	3.4

The first conclusion that can be drawn from the results summarized in Table 5.3

experiment is that deleting cuts regularly from the linear programming relaxation is quite important. A closer look at the instance by instance performance in Table C.4 reveals behavior that seems to defy intuition. In general, we would expect that instances solved with small values of η would require more nodes than instances solved with large values of η . Also, the smaller the value of η the faster the linear programs should presumably solve. The aim of this experiment was to attempt to quantify the tradeoff and determine an acceptable level of η . For many instances, such as *mod011*, *harp2*, and *vpm2*, with $\eta = 1$ either fewer nodes were needed to prove optimality or the final gap was smaller in the same amount of computing time. Close inspection of the output showed that good feasible solutions were found more quickly by MINTO's primal heuristic when $\eta = 1$. Thus, we conclude that it is important to delete cuts from the linear programming relaxation regularly for two reasons:

- The linear programs solve faster and
- diving heuristics are more likely to find good feasible solutions.

5.1.5 The Importance of a Cutting Plane

When a cutting plane is generated, is it possible to know whether or not this inequality will be violated at other nodes of the branch and bound tree? Two statistics which seem like reasonable indicators of the likelihood of a cutting plane being “important” are the depth D in the branch and bound tree at which the cutting plane was generated and the distance d by which the cutting plane $a^T x \leq b$ cuts off the current fractional solution x^* :

$$d = \frac{a^T x^* - b}{a^T a}.$$

In order to test if these statistics truly are meaningful, we conducted an experiment on the instances *dcmulti*, *dsbmip*, *fiber*, *gesa3*, *l152lav*, *lseu*, *mod011*, *modglob*, *p0282*, *p0548*, *p2756*, *pp08a*, *rout*, *set1ch*, *vpm2*. At each node of the branch and bound tree, all the cutting planes generated up until that time were removed. The linear program was resolved, resulting in a solution \hat{x} , and the cutting planes were checked to see if they cut off \hat{x} . For each cut we computed γ , the percentage of nodes in the branch and bound tree after the node in which it was generated for which the cut was violated.

Figure 5.2 shows two scatter plots of γ against d for all cuts generated for the instances in Table 5.1. The second scatter plot in the figure focuses on a subrange of d .

By inspection of Figure 5.2, it appears that there is little relationship between γ and d . In order to statistically verify this observation, the regression coefficient β_1 in the expression $\gamma = \beta_1 d + \beta_0$ was computed for each instance. Table 5.4 shows the value of β_1 and the estimate of the standard deviation of β_1 for each instance. For seven of the fifteen instances β_1 is negative. Our intuition is that γ and d are positively correlated, but the results of this experiment do not seem to support this notion. We thus conclude that the original distance by which an inequality cuts off a fractional solution is not a reliable indicator of the likelihood that the inequality will be violated at other nodes.

Figure 5.3 is a scatter plot of γ against D for all cuts generated for the instances in our test suite. There appears to be a downward trend in Figure 5.3, which coincides with our intuition that cuts generated near the top of the branch and bound tree are more useful than others. We statistically verified this conjecture by computing the regression coefficient β_1 in the expression $\gamma = \beta_1 D + \beta_0$ as was

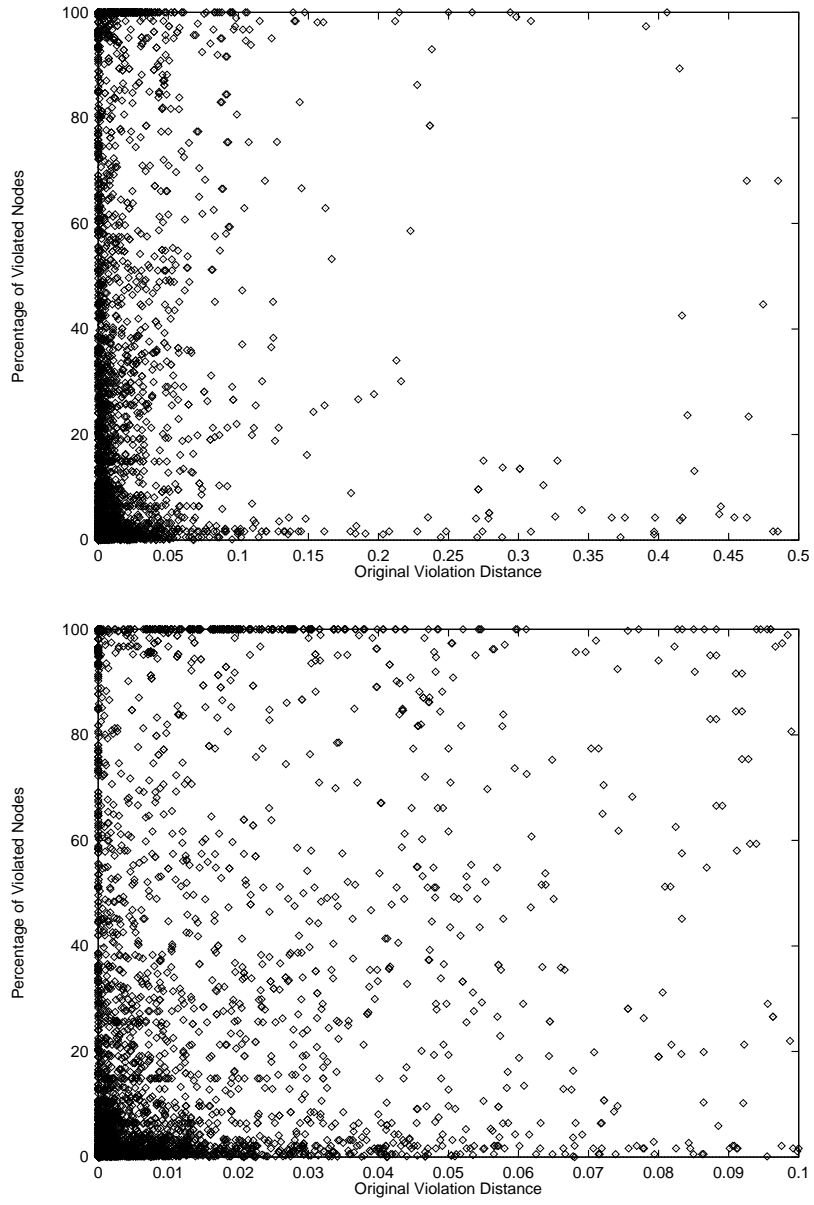


Figure 5.2: Relationship Between γ and d for Test Instances

Table 5.4: Relationship Between γ and d – Estimate of β_1 and its Variance

Name	# Cuts	β_1	$\sigma(\beta_1)$
dcmulti	21	-23.21	47.56
dsbmip	5	-33012.15	51064.65
fiber	236	267.53	84.55
gesa3	117	-33.85	42.74
l152lav	49	55.10	25.29
lseu	218	-19.65	61.81
mod011	111	-10623.42	28372.07
modglob	456	-67.90	60.01
p0282	300	39.09	11.54
p0548	196	63.50	32.75
p2756	467	102.59	20.87
pp08a	143	500.52	116.37
rout	1075	78.66	47.70
set1ch	262	-3.07	70.17
vpm2	131	240.87	39.00

done for d . The results are shown in Table 5.5.

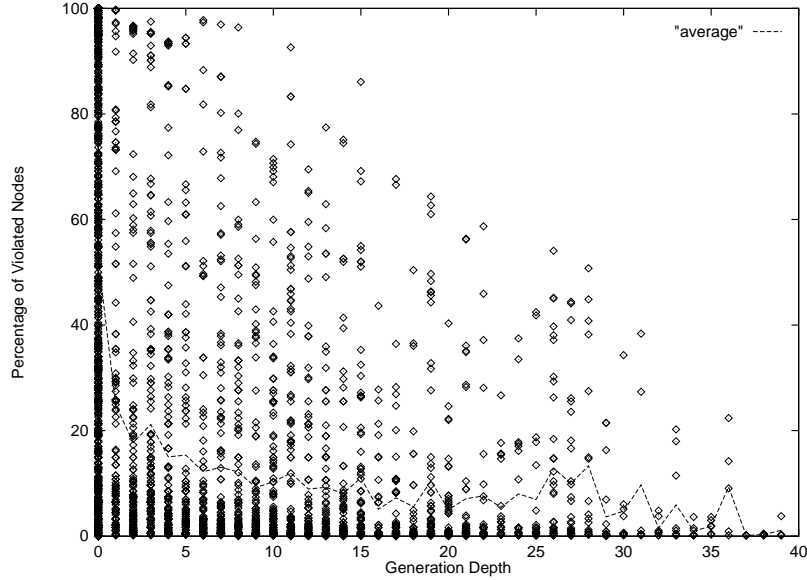


Figure 5.3: Relationship Between γ and D for Test Instances

In every case, β_1 is negative, as we would expect. The smaller the depth in the branch and bound tree at which the cut was generated, the more likely it is to be violated by a large percentage of nodes. Why are cutting planes generated near the top of the branch and bound tree more useful than others? One potential explanation of this phenomenon is that the nodes near the top of the branch and bound tree have a larger number of descendant nodes. If this is true, then the usefulness of a cut generated at node N^i at the node N^j depends on whether or not N^i is an ancestor of N^j . For a cut, consider the numbers γ_d and γ_n , which are the percentage of descendant nodes for which the cut is violated and the percentage of non-descendant nodes for which the cut is violated, respectively. We conjecture that for a given cut $\gamma_d > \gamma_n$. Figure 5.4 shows the percentage of time $\gamma_d > \gamma_n$ as

Table 5.5: Relationship Between γ and D – Estimate of β_1 and its Variance

Name	# Cuts	β_1	$\sigma(\beta_1)$
dcmulti	21	-0.39	0.31
dsbmip	5	-0.30	0.65
fiber	236	-5.68	0.83
gesa3	117	-1.00	0.56
l152lav	49	-0.0097	0.034
lseu	218	-2.00	0.25
mod011	111	-1.66	0.35
modglob	456	-2.95	0.37
p0282	300	-0.52	0.18
p0548	196	-2.11	0.33
p2756	467	-2.19	0.14
pp08a	143	-4.14	0.35
rout	1075	-0.26	0.05
vpm2	131	-1.36	0.24

a function of the the generation depth D of the cut. Clearly, cuts generated are most useful at descendant nodes.

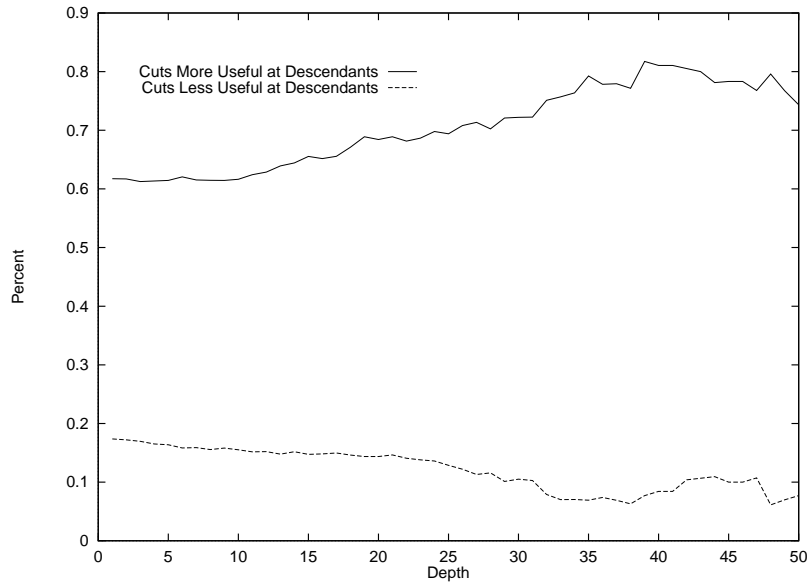


Figure 5.4: The Percentage of Time in Which Cuts Are More Useful at Descendant Nodes and Nondescendant Nodes as a Function of D

From this experiment, two conclusions are drawn. First, cutting planes are more likely to be useful at descendants of the node at which the cut was generated. Second, this relationship depends little on the depth in the tree at which the cut was generated. Nodes near the top of the branch and bound tree are likely to have the most descendants, so cuts generated at these nodes are likely to be the most useful.

5.2 Issues in Cut Sharing

Just as is the case in sharing pseudocost information, the performance of the branch and cut algorithm can be improved by sharing cuts in one of two ways. First, the number of nodes evaluated could be reduced. Second, the time required to process a node could be reduced.

There are two possible reasons why sharing cuts among the processors could reduce the number of nodes in the branch and bound tree. First, since cuts generated at different processors are generated for different LP solutions occurring at possibly distant locations in the branch and bound tree, sharing cuts among the processors may have the effect of building a “complementary” set of cuts to tighten the linear programming relaxation. Second, cut separation is a heuristic procedure. Therefore, an inequality found at one processor may cut off the fractional solution at another processor, even if the separation routines there fail to find a cutting plane. Both of these ways in which the number of nodes might be reduced fall under the category of achieving Goal I of an information sharing scheme.

By sharing cuts, we also might reduce the total time required to process a node. Specifically, for cutting planes that require a significant amount of time to generate, we could hope to use communication to eliminate the need to generate these “expensive” cuts at many processors. By sharing cuts in this matter, we meet Goal II of an information sharing scheme.

We wish to improve our chances of accomplishing Goals I and II while minimizing the sacrifice made in achieving the other goals. We would like to accomplish useful sharing of cuts while minimizing the number of messages passed, balancing the memory requirements of the processors, and doing this in a way that does not

consume a large number of CPU cycles that could better be spent evaluating nodes of the branch and bound tree.

In the next sections, we describe a framework in which to test ideas for the sharing of cuts in a parallel branch and cut algorithm.

5.2.1 Cut Pools

Our framework for cut sharing consists of a number of cut pools located on various processors. Each of these cut pools serves a distinct purpose and has a set of algorithms that operate on it. On each processor P_i , there are three pools – $loadset_i$, $cutcache_i$, and $prefetchpool_i$. In addition, another pool, called the *serverpool*, is maintained by a special process P_S . We now briefly describe the functionality and rationale behind each pool.

Loadset_i

Loadset_i consists of all cuts loaded into the LP solver’s memory at processor P_i . We would like to be able to incorporate as many useful cuts into the LP as possible.

Cutcache_i

The main purpose of *cutcache_i* is to serve as a cache for cuts that once appeared in *loadset_i* and have been removed. We have seen in our sequential study that deleting unnecessary cuts from the linear program and storing them in a cut pool is very useful for the sequential algorithm, and there is no reason to believe that the situation would be different in a parallel environment.

In fact, in a parallel algorithm, the passing of nodes due to quality balancing

increases the likelihood of jumping from one region of the search tree to another. As shown by the experiment of Section 5.1.5, cuts are more likely to be useful if used in the region in which they were generated. Therefore, we presume that more cuts will be removed from the the active set in a parallel algorithm than in a sequential one, so having *cutcache_i* available to store these cuts is very important.

Prefetchpool_i

Prefetchpool_i consists of cuts generated at a processor P_j , $j \neq i$ that were passed to processor P_i . It is not guaranteed that the cuts in *prefetchpool_i* will ever be added to *loadset_i*.

Serverpool

The *serverpool* holds cuts that are generated at the various processors and can be queried by processors to find violated cuts.

Figure 5.2.1 is a schematic of the various cut pools and how cuts might be passed among the pools. The dashed arrows in the figure indicate that cuts *may* be passed between the processors depending on the type of cut sharing scheme being implemented. Clearly, a wide variety of schemes can be employed by using these cut pools. If more than one cut pool is to be used by a cut sharing scheme, then the pools are searched in the following order: *cutcache_i*, *prefetchpool_i*, *serverpool*. Cuts from at most one pool are returned from a search. A more detailed discussion of cut sharing using the pools is delayed until Section 5.2.3.

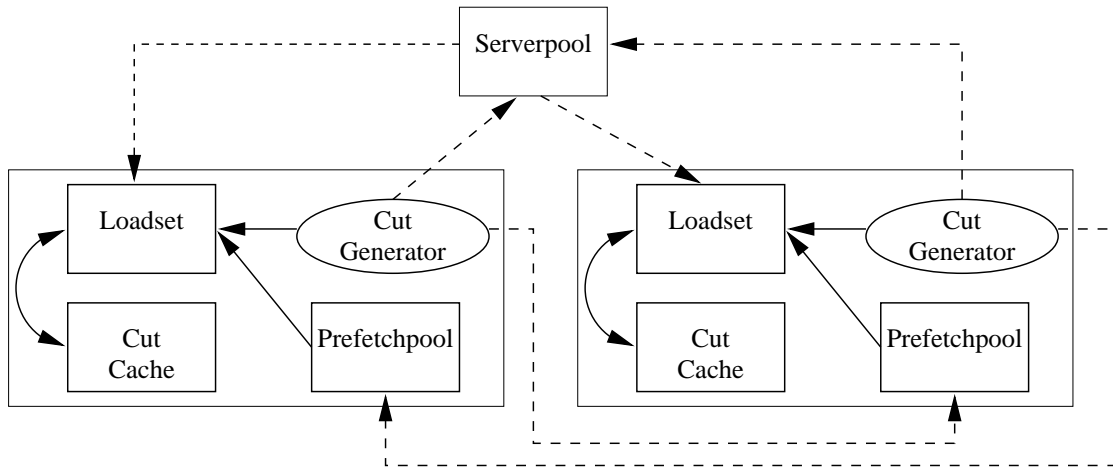


Figure 5.5: Cut Pools

5.2.2 Algorithmic Issues

There are three basic operations that must be performed on each cut pool. First, a cut can be added to a cut pool. Second, given a fractional LP solution, we need to be able to identify whether or not a pool contains violated cuts. Third, the size of the cut pool must be controlled. We call these operations *add*, *match*, and *purge* respectively. In some cases the operations are different for different cut pools.

Add

A cut is added to $loadset_i$ through a call to the linear programming library. For all other types of cut pools, a cut is added to the pool by simply adding it to the data structure holding the inequalities. We would like to avoid adding duplicate inequalities to the linear programming relaxation. If duplicate inequalities are

added, the resulting linear programming relaxation becomes both degenerate and ill-conditioned.

Since cuts are returned from at most one pool in a round of cut generation, one way to ensure that duplicate cuts are never added to $loadset_i$ is to ensure that duplicate cuts are never added to $prefetchpool_i$ or $serverpool$. To efficiently identify duplicate cuts, a hashing scheme is used. When a cut is generated, a hash value for the cut is computed using a multiplicative hash function [71]. Before a cut is added to $prefetchpool_i$ or $serverpool$, its hash value is compared to the existing cuts in the pool. Since the addition of cuts is not required for the correctness of the algorithm, checking if two cuts with equal hash values are indeed duplicates is not done – they are assumed to be duplicates. This is not necessarily true, in which case a possibly useful cut is discarded. However, the implementation is far more efficient, particularly in cut sharing schemes where many duplicate cuts are generated.

Match

Clearly, there is no need to find a violated inequality in $loadset_i$. Finding violated inequalities in the other pools amounts to evaluating each inequality at the given fractional linear programming solution and determining whether or not the inequality is satisfied. Matching the violated inequalities quickly could help us achieve Goal II in our cut sharing system. Making the match operation efficient is particularly important for $serverpool$, since $serverpool$ serves many match requests, and a processor waits idle until a response to a match request is received.

The matching operation may be made efficient by designing an efficient algorithm to find violated inequalities and by keeping the size of the cut pool small.

Keeping only “necessary” cuts in a cut pool is done by the purge operation, and we will discuss purging in the next section.

One idea for speeding up the match operation is to keep the cuts ordered so that inequalities likely to be violated are first in the list. From our study in Section 5.1.5, we saw that cuts generated near the top of the branch and bound tree were more likely to be violated by subsequent nodes. A logical data structure for storing the cuts in a pool is to have a list of ordered “buckets” – one for each cut generation depth D . Searching for violated inequalities in order from the lowest bucket to highest bucket has two positive (and related) effects. First, the match operation time is reduced. Second, if a limited number of cuts are desired, performing the match in this way returns the “best” cuts from the pool. Note that the buckets need not be defined in terms of the generation depth – any other ranking of the importance of a cutting plane could be used. Defining the buckets in terms of generation depth is convenient since determining appropriate bucket sizes is not an issue.

Even with an intelligent implementation scheme, it is often the case that the entire ordered list of cuts is checked for violation anyway. This point merits more explanation, as it also brings up another point about the monotonicity of the parallel branch and cut procedure.

During the branch and bound process, we would like the value of the global upper bound $z_U \equiv \max_{i \in \mathcal{L}} z_U^i$ to always decrease. In order to ensure that this happens, all cuts present in the linear programming formulation at a parent node must also be present in the children. Therefore, in sequential algorithms, generally *all* violated inequalities that have at one time been removed from the active formulation, but are violated by the current fractional point are added to the active

formulation. In our case, this means adding all violated inequalities from *cutcache_i*, so the entire list of inequalities must be checked for violation. In our parallel algorithm, when nodes are passed among processors for quantity or quality balancing, a monotonic decrease cannot be ensured unless the “active” cuts at a node are passed as well. Keeping track of the active cuts for each node in the branch and bound tree and passing the cuts along with the nodes requires too much overhead to be practical.

In the case of *prefetchpool_i* and *serverpool*, we wish to return a number of violated inequalities, since our experiments in Section 5.1 demonstrated that adding multiple cuts to the LP relaxation at one time is computationally effective. Since we search for a number of inequalities violated by the given fractional point, often the entire cut pool is searched, again diminishing the usefulness of keeping the cuts ordered.

The main algorithmic way in which the match operation could be made more efficient is by not checking all of the cuts in the pool for violation. An obvious way in which to accomplish this is to only check the first few buckets for violated inequalities. In the case of *serverpool*, there is an alternative way to limit the number of cuts checked for violation. Certainly, if a cut has already been passed to a processor, it is quite unlikely that the solution passed from that processor would ever be violated the cut again. Thus, by simply keeping track of which cuts have been passed to which processors, a number of checks for violation could be avoided at *serverpool*.

A second strategy for selecting cuts not to match is to keep track of the average distance of the cut to each fractional point for which it was ever checked for violation. (By convention, the distance will be positive if the inequality is violated).

This measure gives a natural ranking to the importance of cuts – the larger the distance, the more likely a cut is to be violated by a fractional solution. However, we saw in Section 5.1.5 that distance may be a weak measure of the importance of a cut.

In the case of *cutcache_i* and *prefetchpool_i*, once a cut is matched and added to the LP relaxation, then it is deleted from the pool. This helps keep the sizes of these pools small.

Purge

We wish to delete cuts from *loadset_i* for the same reason in a parallel algorithm as we do in a sequential one – to ease the solution of the linear programming relaxation. We use the strategy for purging cuts from *loadset_i* described in Section 5.1 – If the dual variable corresponding to the cut has been zero for η consecutive rounds, it is purged (and moved to *cutcache_i*).

For the cut pools, the purpose of purging the cuts is twofold. Purging the cut pool decreases the time required to search through the cut pool, helping achieve Goal II – improving the latency of access to cuts. Also, purging the cut pool may be required due to the limited amount of memory at a processor. This can be seen as attempting to achieve Goal IV of an information sharing scheme. We offer two strategies for purging the cut pools – a reactive strategy and a proactive strategy.

The reactive purging strategy works as follows. Each of the cut pools is given a maximum size. If the cut pool reaches this maximum size, then a certain fraction of the “worst” cuts are deleted from the cut pool. In defining what it means for one cut to be “worse” than another, all of the preceding discussions about the relative importance of cuts can be applied. For example, the average distance

measure introduced in the previous section, the original violation distance d , or the generation depth D could be used to decide which cuts to purge.

The proactive purging strategy for cut pools $cutcache_i$, $prefetchpool_i$, and $serverpool$ works much in the same way as for $loadset_i$. For a given solution, the dual variable for a cut in one of these pools is equal to zero if the cut is not violated, and it is non-zero if the cut is violated. Thus, if a cut has not been violated (or its dual variable has been zero) for η consecutive rounds, then the cut is deleted from the pool. Note that in order to implement this proactive strategy, all cuts in a cut pool must be checked for violation each time a match operation is performed.

5.2.3 Cut Sharing using Pools

We now describe how to implement general fetch and prefetch information sharing schemes using the cut pools we have defined. Since cutting planes and cut management are effective tools in a sequential branch and cut algorithm, we also provide this facility in our parallel algorithm. Thus, all cut sharing schemes we consider use the pools $loadset_i$ and $cutcache_i$.

Master-Worker

To implement the master-worker cut sharing scheme, we will use $serverpool$. When a processor P_i generates a cut, it is added to $loadset_i$ and also passed to P_S where it is placed in $serverpool$. In order to find a violated cut, a processor P_i first queries $cutcache_i$. If no violated inequalities are found, $serverpool$ is queried. If no violated inequalities are found in $serverpool$, then the processor will attempt

to generate a violated inequality by solving the separation problem.

The largest advantage of this cut sharing scheme is that useful cuts can be shared by passing a minimal number of unnecessary messages. The biggest drawback of this scheme is that processor P_i waits idle while waiting for a cut to return from P_S . It therefore strives to achieve Goals I and III of an information sharing scheme at the expense of Goal II. Also, all cuts are stored only in one location, so a memory imbalance may occur, which contradicts our Goal IV of an information sharing system.

Broadcast-Prefetch

To implement the broadcast-prefetch cut sharing scheme, *prefetchpool_i* is used. When a cut is generated at a processor P_i , it is added to *loadset_i* and then passed to all P_j , $j \neq i$, where it is placed in *prefetchpool_j*. In order to find a violated inequality, processor P_i first queries *cutcache_i*, then *prefetchpool_i*. If no cuts are matched from either of these sources, P_i then attempts to generate a new violated inequality by solving the separation problem.

The main advantage of the broadcast prefetch sharing scheme over the master worker scheme is that the time required to match a cut is reduced since all the cuts are stored locally on the processor. However, in order to accomplish this, a large number of messages must be passed. It therefore strives to achieve Goals I and II of an information sharing scheme at the expense of Goal III. Placing a cut in more than one *prefetchpool_i* means duplicating the cut, increasing the overall storage requirements for cut information – which goes against Goal IV of our information sharing scheme.

Selective-Prefetch

The selective-prefetch sharing scheme is very similar to the broadcast prefetch sharing scheme. The difference is that when a cut is generated by a processor P_i , it is passed only to a selected number of other processors P_j for inclusion in the *prefetchpool* _{j} . Thus, the number of messages is fewer than that of the broadcast-prefetch sharing scheme. The main issue that arises when implementing this scheme is to decide to which other processors $P_j, j \neq i$ should processor P_i send a cut? The strategy for deciding to which processors to pass cuts may be as simple as choosing a random subset of the processors.

Another strategy might be to pass only the cuts that seem “important enough” to all of the processors, and not to pass the unimportant cuts. The decision about what makes a cut important could be made based on *a priori* information. For example, the experiments in Section 5.1.5 have demonstrated statistics that may help us decide on the importance of a particular cut. Alternatively, the decision of whether or not to pass a cut could be a run time decision. For example, we consider the following strategies.

- If a cut has been binding for a large number of rounds, then it could be deemed important enough to pass to other processors.
- A “representative sample” of LP solutions could be kept at each processor, and if a generated inequality cuts off “enough” of the LP solutions, it could be passed to the other processors.

Certainly these ideas need a more precise definition. The investigation of effective strategies for performing selective prefetch will be a focus of the computational experiments.

For the selective prefetch sharing scheme, it is very likely that many duplicate cuts will be generated and passed to cut pools. Therefore, it is imperative that a hashing scheme to identify duplicate cuts is used.

No Sharing

Even with a reasonable cut sharing scheme in place, it may be more beneficial not to share cuts among the processors at all. Whether or not sharing will be beneficial depends on the degree to which the sharing scheme can meet Goals I and II.

For example, if solving the separation problem is a very fast procedure, then sharing cuts will not reduce the time required to find a cut and Goal II will not be met. The separation procedure used to find violated inequalities attempts to maximize the absolute violation of the given fractional point. Therefore, it may be that cuts found by the separation problem are “better” than cuts found by the match procedure of a cut pool. In this case, Goal I of an information sharing scheme would not be met. A similar point arose in Section 4.4.1 in dealing with whether or not to share pseudocost information. In both cases, information generated for the current node or operating point *might* be better than information generated for a different operating point and shared among the processors.

In Section 5.1.5 we saw that cutting planes are the most useful at nodes which are descendants of the node at which the cut was generated. If few nodes are passed due to quality and quantity balancing, then most descendant nodes of a node at which a cut was generated will exist on the same processor. Perhaps it is sufficient to exploit a cut only at one processor. This is yet another reason why a “no sharing” scheme might be the most efficient. Also from the conclusions of Section 5.1.5, one sees the interaction between cut sharing and node selection. Namely, node selection

strategies with a more “depth-first” flavor are to be preferred for a parallel branch and cut algorithm. Search algorithms of this type are apt to explore entire subtrees at a time, where information generated locally can be used to the highest degree possible without having to resort to sharing the information among the processors.

Adaptive Sharing

In Section 5.1.1, we saw that a cut and branch strategy, where cutting planes are added only at the root node of the branch and bound tree can be very effective. Other researchers have also seen this behavior [29] [101]. Also, we saw in Section 5.1.5 that nodes generated near the top of the branch and bound tree tend to be more important than others. Interpreting these results in the context of a parallel algorithm leads to the conclusion that it may be best not to share cutting planes or that cutting planes should be shared only very early in the branch and cut procedure.

A simple adaptive scheme would allow for cutting planes to be shared only for a specified number of nodes of the branch and bound tree. A more complex adaptive scheme could measure the time and success rate of obtaining a violated cutting plane from both the cut pools and the cut generation procedure. Once these statistics are collected, the algorithm could decide the “best” manner in which to attempt to find violated inequalities. In fact, the default PARINO implementation has a simple adaptive mechanism. If after the first 200 nodes, cut generation or matching has not been successful at least 1% of the time, then cut generation and matching are deactivated for the remainder of the search.

Client-Server and Buffered Prefetch

The other two general categories of information sharing schemes – client-server and buffered-prefetch cannot be implemented by using only the cut pools we have introduced. We have chosen to adopt the approach of first experimenting with simple implementations, then evaluating the results to judge whether or not a more complex implementation is required. Also, we feel that the selective prefetch and adaptive sharing schemes presented in the previous section should allow for useful sharing of cut information while minimizing the overhead required to perform the sharing.

The client-server cut sharing scheme could be implemented by creating more than one *serverpool*. To reiterate, this would have the effect of reducing the contention effects at processor P_S in the master-worker scheme, which would reduce the latency of cut access. The disadvantages are that fewer processors are evaluating nodes of the branch and bound tree, and that a scheme would perhaps need to be developed to share cuts among the servers.

The buffered prefetch sharing scheme could be accomplished by having an additional cut pool *cutbuffer_i* on each of the processors P_i to hold cuts. Once enough cuts were collected at a processor, then the entire *cutbuffer_i* could be passed to the other processors P_j , $j \neq i$. This would reduce the number of messages passed (Goal III), but the useful sharing of cuts would be reduced (Goal I). Also, it may be the case that numerous duplicate cuts would be generated, so a hashing scheme (similar to that required in the selective prefetch sharing scheme) would be required. Also, the question of what would be an effective buffer size is an issue that must be tackled.

5.3 Implementing Distributed Cut Management in PARINO

To implement our distributed cut management scheme in PARINO, we create new entities to perform the necessary tasks. Associated with each worker is a *cut manager* entity. Requests to match cuts and add cuts are made from the worker entity to the cut manager entities. Depending on the type of cut sharing strategy in use, the cut manager decides the sequence of pools to search for a violated inequality and to which pools to add the cut. The object oriented nature of the PARINO implementation makes it a simple matter to implement different cut sharing strategies. The C++ class corresponding to the entity for each cut manager has the boolean functions `match_cache()`, `match_prefetch()`, `match_server()`, `add_prefetch()`, and `add_server()` that return `TRUE` if cuts are to be added or matched from the corresponding pool. In addition, each cut manager entity class has a boolean function `add_server(cut)` if *cut* is to be added to *serverpool*, and a function `add_prefetch(cut, pools)` which in the argument *pools* returns the *prefetchpool_i* to which *cut* is to be added. All the cut sharing schemes discussed in this section can be implemented by simply changing these functions.

There is a *server pool manager* entity that is responsible for overseeing the operation of the *serverpool*. On which processor should the server pool manager entity be placed? There are two obvious alternatives. First, it could be placed on the processor containing the distributor entity. Recall from our discussion about general information sharing schemes that we would like to implement the scheme in the least “disruptive” way possible. Placing the server pool manager entity on the processor with the distributor means that more processors are available

to evaluate nodes, and the information sharing scheme is less disruptive. The drawback to this entity placement that the latency of access to cuts is increased, so Goal II is sacrificed. The server pool manager entity could receive its own processor, reducing the latency but then the information sharing scheme become more disruptive. The entity-FSM structure of PARINO makes it a simple matter to change the location of the server pool manager entity.

Figure 5.6 is a picture of PARINO with the cut management entities added. The server pool manager entity may or may not exist on the same processor as the distributor, primer, and logger entities, which is what the curved dashed lines are meant to represent.

Recall from our discussion in Section 5.2.2 that the `add`, `match`, and `purge` operations are performed differently for different cut pools. Due to the object oriented nature of PARINO, it is simple to change the behavior of a cut pool by redefining virtual functions `add()`, `match()`, and `purge()` for the pools in the cut manager entity classes.

5.4 Computational Results

The goal of this section is to test the various ideas about cut sharing presented in this chapter. First, an experiment evaluating three basic approaches to cut sharing is performed. Drawing conclusions from the results of this experiment, we implement and test other, more advanced cut sharing ideas.

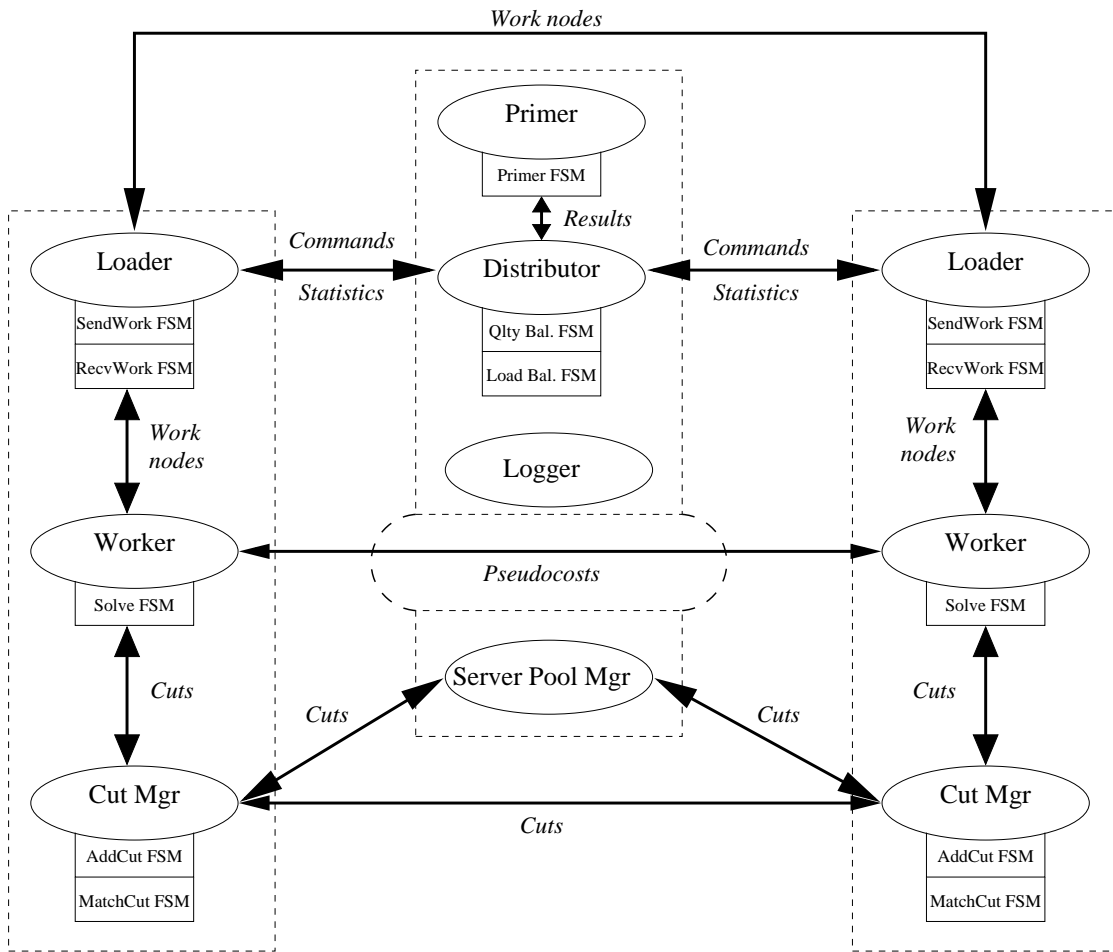


Figure 5.6: PARINO System Architecture

5.4.1 Default Algorithm

For all experiments, unless otherwise stated, we use the PARINO settings listed in Figure 4.6. Figure 5.7 lists the default PARINO cut management settings. For all experiments in this section, each instance/setting combination was solved three times and machines in the **beetle** cluster were used to perform the experiments.

- A skip factor (see Section 5.1.1) of $\kappa = 1$ is used.
- At most $\lambda = 20$ cuts are added to the linear programming relaxation per round.
- A tailing-off percentage of $\alpha = 0.05\%$ is used.
- Weak cuts are deleted as described in Section 5.1.4 with $\eta = 25$.
- The reactive cut purging strategy (see Section 5.2.2) is used. Maximum cut pool sizes of 2000 for *cutcache_i* and *prefetchpool_i*; and a maximum size of 5000 for *serverpool*. When purging, cuts are ranked based on the generation depth D .
- The server pool manager entity resides on the same processor as the distributor.
- Cut generation and cut matching are deactivated if after the first 200 nodes, the success rate of finding a cut is less than 1%.

Figure 5.7: Default PARINO Cut Management Settings

5.4.2 Test Suite

To test the cut sharing schemes we propose, we use the instances listed in Table 5.1. In addition, the MIPLIB instances *gesa2_o*, *gesa3_o*, *pp08a*, and *set1ch* were added

to the test suite and the instance *modglob* was removed. *Modglob* was removed because the addition of cutting planes often introduced numerical instability which made it difficult to measure the relative effectiveness of different sharing schemes. Table 5.6 lists the number of each type of inequality found by MINTO (v3.0) using the default sequential cutting plane strategy listed in Figure 5.1.

Table 5.6: Additional Instances Used for Cut Sharing Study

Name	Knapsack Covers	Flow Covers
gesa2_o	13	124
gesa3_o	12	77
pp08a	0	224
set1ch	0	283

5.4.3 Basic Cut Sharing Paradigms

The first experiment run was to test the basic cut sharing schemes. For each instance in our test set, and on configurations of 4, 8, and 16 processors, PARINO was run using a master-worker cut sharing scheme, a broadcast-prefetch sharing scheme, and where cuts were not shared among the processors. The full results of this experiment are given in Tables D.1 – D.6. For instances in which optimality was not proved, the average final gap to the optimal solution is reported.

To help interpret the results, we break up the instances into two categories – instances solved in one hour and those not solved in one hour. Table 5.7 shows a summary of the performance of the cut sharing schemes on the solved instances.

The heading NN denotes the total number of nodes needed to solve the instances, and the heading t denotes the total time required to solve the instances.

Table 5.7: Summary of Basic Cut Sharing Experiment – Solved Instances

p	Broadcast-Prefetch			Master-Worker			No Sharing		
	Avg. Rank	NN	t	Avg. Rank	NN	t	Avg. Rank	NN	t
4	2	633880	12782	2.33	686903	12707	1.67	656296	12805
8	1.67	714696	6036	2.67	761983	6487	1.67	672673	5809
16	1.67	704273	3078	2.83	720182	3380	1.5	715810	3118

Generally speaking, the performance of the schemes does not seem to be wildly different on the problem instances. Only in three cases – *blend2* on 8 and 16 processors and *l152lav* on 16 processors – are the average times of the sharing schemes for a particular instance larger than one standard deviation apart.

A slight trend evident in Table 5.7 is that as the number of processors increases, the master-worker sharing scheme gets relatively worse. The contention effects of the increased number of processors asking for cuts is the cause of the decreased performance. For example, the average time to receive a cut from the cut server for *blend2* increases from 9 milliseconds on four processors, to 14 milliseconds on eight processors, to 91 milliseconds on 16 processors.

The relative performance of the cut sharing methods on the hard instances in the test suite is more interesting. Table 5.8 summarizes the performance of the cut sharing methods on the instances not solvable in one hour.

We see two general trends in the results summarized in Table 5.8. First, as the

Table 5.8: Performance of Basic Cut Sharing Experiment – Unsolved Instances

	Broadcast Prefetch		Master-Worker		No Sharing	
p	Avg. Rank	NN	Avg. Rank	NN	Avg. Rank	NN
4	1.83	913320	1.67	920269	1.67	930277
8	2.0	2391552	1.83	2601954	1.67	2368534
16	2.33	4856140	1.67	5717950	1.5	6140285

number of processors increases, we evaluate more nodes in the same computation time when not sharing cuts than in either of the cases where we do share cuts. This is an indication that the overhead of sharing cuts is larger than the time to generate the cuts. Inspection of the results for each instance reveals that this is not *always* the case. In Table 5.9 we list the average cut generation time and the average waiting time for a cut in the master-worker sharing scheme on sixteen processors for all instances in which the times were markedly different.

Table 5.9: Cut Generation and Waiting Times for Various Instances

Instance	Avg. Gen Time (ms)	Avg. Wait Time (ms)
mitre	2.11	9.49
l152lav	9.36	36.15
blend2	7.53	91.31
cap6000	4778.07	1910.27
mod011	247.54	20.31
harp2	2.00	10.18

The larger number of nodes listed in Table 5.8 for the no-sharing scheme is largely due to the instance *harp2*, where the generation time is significantly smaller than the waiting time. Table 5.9 indicates that in some cases, we are able to reduce the time required to evaluate a node (achieving Goal II of an information sharing scheme) through a simple mechanism like the master-worker sharing paradigm. We are encouraged by these results for two reasons. First, with more sophisticated sharing schemes, it may be possible to consistently achieve Goal II of an information sharing scheme. Second, for cut classes that require a significant amount more time to generate, for example lift-and-project cuts [7], achieving Goal II seems quite likely.

Another general trend seen in Table 5.8 is that as the number of processors increases, the broadcast-prefetch sharing scheme’s overall performance seems to be suffering – likely from the reduced number of nodes it is able to evaluate. This is not a surprising result, since the broadcast-prefetch scheme is not scalable. The master-worker scheme seems to be performing as well as the no sharing scheme on sixteen processors – even while evaluating fewer nodes. This is one indication that the sharing of cuts might be useful in accomplishing Goal I of an information sharing scheme – that the cuts generated at one processor are indeed useful at another processor.

Even though we are able to spot some trends in Table 5.8, the overall results are not wildly different among the cut sharing paradigms. One reason why the performance is roughly equal is PARINO’s mechanism for deactivating cut generation and cut matching. In the instances *pp08a* and *set1ch*, cut sharing and generation were turned off after relatively few nodes, so the performance of all three sharing mechanisms is roughly equal. We will now point out interesting behavior of the

various sharing schemes on an instance by instance basis.

On the instances *mod011* and *harp2* sharing seems to generally be useful – a smaller optimality gap is proved when cuts are shared as opposed to when cuts are not shared. This is an indication that Goal I of an information sharing scheme is being met. The performance of the broadcast-prefetch sharing scheme on *cap6000* is quite inferior to the other sharing schemes. To explain this behavior, one needs to look no further than the amount of time spent packing and unpacking messages by each processor on which a worker exists. The average time spent packing and unpacking messages in the broadcast-prefetch sharing scheme is 924 seconds on four processors, 1465 seconds on eight processors, and 2044 seconds on 16 processors. More than *half* the total computation time is spent simply packing and unpacking messages. Why is this the case? The success rate of matching a cut from *prefetchpool_i* is on average around 20%, while the success rate of generating a cut after not being able to match a cut from any pool is usually over 80%. The effect of this in the broadcast-prefetch sharing scheme is that an *enormous* number of cuts are being passed around, and the workers have little time to do anything else except to pack outgoing cuts and unpack incoming cuts.

The instance *mitre* is a somewhat pathological example. The no-sharing scheme consistently evaluates many fewer nodes than the broadcast-prefetch and master-worker sharing schemes in the same amount of computation time. One possible explanation of this phenomenon is that the generation time for a cutting plane exceeds the time required to share the cut among the processors. Looking at Table 5.9 reveals that this is *not* the case. The real reason is that when cuts are generated locally, then many nodes at the top of the branch and bound tree are infeasible. Therefore, many processors must sit idle since there are not enough

active nodes in the branch and bound tree. The provable optimality gap does not suffer by the computation of fewer nodes.

Our overall conclusion from this experiment is that sharing cuts among the processors *can* be a useful technique – we have seen instances in which both Goal I and Goal II of an information sharing scheme are achieved. However, care must be taken so that the overhead to perform the sharing is not too high. In the next section we investigate the performance of some of the advanced cut sharing techniques that we have introduced.

5.4.4 Improving on the Basic Cut Sharing Paradigms

Selective-Prefetch

In this section, we report on two selective-prefetch schemes for sharing cutting planes and compare their performance to the broadcast-prefetch scheme. One of the simple ways to decrease the communication overhead associated with the broadcast-prefetch sharing scheme is to not send all cuts to all processors. The simplest way to accomplish this is to simply send each generated cut to a random number of processors. We experimented with this simple scheme on the instances in our test suite. Each cut that was generated at processor i was sent to processor j with probability $1/2$. In general, for a scalable scheme on p processors, the cut could be randomly sent with probability $1/p$.

Since the results in Section 5.1.5 indicated that cuts generated near the top of the branch and bound tree would be the most useful, we also experimented with a selective-prefetch sharing scheme where a cut was passed to all other processors if its generation depth D was less than $D_o = 10$.

Tables D.7 and D.8 show the full results of an experiment comparing the random, depth-based, and broadcast-prefetch sharing schemes. For the solved instances in the test suite, the type of prefetch sharing scheme has little impact on performance. For each instance, the average solution times for the various sharing methods were within one standard deviation of each other. For the difficult instances, Table 5.10 shows the average performance ranking of the different prefetch sharing systems.

Table 5.10: Average Performance Ranking of Prefetch Sharing Schemes

Strategy	Avg. Rank
Broadcast	2.0
Random	2.5
Depth-Based	1.5

Broadcasting only cuts that are generated at depth less than ten in the branch and bound tree is generally the most effective method. This is especially the case on problems where a large number of cuts are generated, like *cap6000* and *harp2*. We will now briefly discuss other interesting behavior seen when solving specific instances.

Not surprisingly, the performance on *cap6000* is greatly improved by sending fewer cuts between processors. In this case the average time spent packing and unpacking messages on a processor is reduced from 2044 seconds with the broadcast-prefetch scheme to 1572 seconds for the random prefetch scheme and 1703 seconds for the depth-based prefetch.

In *pp08a*, the average number of nodes evaluated is roughly the same for all

the prefetch sharing schemes. However, the average final gap after one hour is the highest when cuts are broadcast. This is a counterintuitive result, since reducing the number of cuts passed should also weaken the linear programming relaxations at the nodes, leading to higher optimality gaps if a comparable number of nodes is evaluated.

For the instance *mod011*, we have seen that it is generally useful to share as much cutting plane information as possible. Therefore the broadcast-prefetch sharing schemes outperforms the other prefetch sharing schemes. However, the selective-prefetch sharing scheme’s performance is only slightly inferior to that of the broadcast-prefetch scheme.

Exploiting the Skip Factor

Recall from the experiment in Section 5.1.1 that it may not be the most computationally effective strategy to generate cuts at every node of the branch and bound tree. When designing a parallel cut management scheme, this fact can be exploited in order to minimize the negative effects of cut sharing. We give two alternatives. First, the contention effects arising from a the master-worker cut sharing scheme can be eliminated if every worker queries the cut server only every κ nodes. We experimented with this scheme using values of $\kappa = 10$ and $\kappa = 100$ on sixteen processors. Tables D.9, D.10, and D.11 show the full results of this experiment compared to the case $\kappa = 1$. Again, we break the instances into classes depending on whether or not the problem could be solved to optimality in one hour. Table 5.11 summarizes the results of the experiment on the instances solved in less than one hour.

As expected, more nodes are evaluated as κ increases. The increase in number

Table 5.11: Summary of Using Skip Factor in Master-Worker Sharing Scheme
Experiment – Solved Instances

κ	Avg. Rank	NN	t
1	2.0	715810	3118
10	2.2	1185743	5238
100	1.8	1358550	4283

of nodes does not seem to seriously deteriorate the overall performance. Most of the difference in nodes and time are accounted for by the two instances *gesa2_0* and *vpm2*.

Table 5.12 summarizes the experiment for the instances not solved in one hour. For these instances, it seems that increasing κ has a negative effect on performance.

Table 5.12: Summary of Using Skip Factor in Master-Worker Sharing Scheme
Experiment – Unsolved Instances

κ	Avg. Rank	NN
1	1.5	5097544
10	2.0	5346128
100	2.5	5730244

The most interesting result of this experiment is that by increasing the skip factor the instances *cap6000* and *harp2* can be solved to optimality in some cases. These instances are not included in the rankings in Tables 5.11 or 5.11. (See Table D.10 for the specific results). These are also two of the instances in the test suite for which the most inequalities are found.

A second natural way in which to exploit the skip factor in a parallel algorithm is in using a broadcast-prefetch cut sharing scheme. In this case, since cuts will be generated only every κ nodes, fewer messages will be passed between the processors. Table 5.2 from our sequential study verifies this fact. Tables D.12, D.13, and D.14 show the full results of an experiment testing this idea for the cases $\kappa = 10$ and $\kappa = 100$. The computational results indicate that the overall effect on the broadcast-prefetch scheme when introducing a skip factor is nearly exactly the same as when introducing a skip factor to the master-worker sharing scheme.

Tables D.15, D.16, and D.17 show the results of varying the skip factor when solving instances where cutting plane information is not shared among the processors. It is again interesting to note that except for the instances *cap6000* and *harp2*, instances are usually solved more efficiently with a skip factor $\kappa = 1$ than with a skip factor of $\kappa = 10$ or $\kappa = 100$.

From the result of our experiment on varying the skip factor in a parallel branch and cut algorithm we can conclude that a skip factor $\kappa > 1$ is useful when a large number of cuts are generated. Otherwise, $\kappa = 1$ is the best choice. In Section 5.1.1, we saw that $\kappa = 1$ was generally *not* the best choice of a skip factor in a sequential cutting plane algorithm.

This concludes our experiments on the effective use of cutting planes. We have not tested all the cut sharing ideas presented in this chapter, but from the tests we have performed, we can make the following observations and recommendations about sharing cuts in a parallel branch and cut algorithm.

- By sharing cuts among processors, both the time required to process a node and the total number of nodes required to solve the problem can be reduced.

- Cutting planes generated near the top of the branch and bound tree should be shared among all processors.
- For instances in which a small number of cuts are generated, it is best to generate cuts at every node of the branch and bound tree. For instances where a large number of cuts are generated, a skip factor larger than one is appropriate.

CHAPTER 6

Solving Set Partitioning Problems in Parallel

In this chapter, we discuss the parallel implementation of a specialized algorithm to solve the set partitioning problem (SPP):

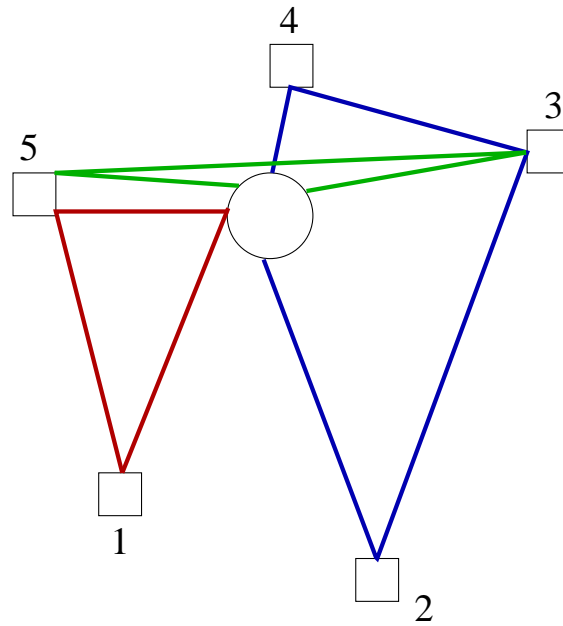
$$\min\{c^T x : Ax = 1, x \in \{0, 1\}^n\},$$

where $A \in \{0, 1\}^{m \times n}$ and $x \in \{0, 1\}^n$. In the first section, we give an example of how set partitioning problems arise in practical applications. Second, we give a strategy for parallelizing the sequential algorithm of Section 2.2 and our motivation for choosing such a strategy. The third section deals with control issues and implementation details of the algorithm. The fourth section describes PSP, a code that implements our parallel computing ideas for solving the set partitioning problem. The final section provides computational evidence that our approach is efficient.

6.1 Routing

The vehicle routing problem is to find a set of routes beginning and ending at a depot such that each member of a set of customers is served. To formulate the vehicle routing problem as a SPP, there is a variable for every route and a row for every customer. The columns represent the feasible routes where $a_{ij} = 1$ if customer i is served by route j , and $a_{ij} = 0$ otherwise. Figure 6.1 gives a small example of this construction. In the figure, the routes are depicted and the columns corresponding to the routes in a set partitioning formulation are shown.

This may not seem like an efficient way to model the vehicle routing problem. However, a route for a vehicle might need to satisfy a number of complex constraints such as time windows on arrival and departure from a customer, capacity constraints, and precedence constraints. Incorporating the rules that define feasible routes into a more compact model would be nearly impossible. Another situation where a set partitioning formulation is the most appropriate is in the crew scheduling problem. The “routes” in crew scheduling problems are called pairings. Legal pairings must conform to complex FAA restrictions, and the cost associated with a pairing is a nonlinear function of the flying time of the pairing, a minimum guaranteed cost, and the total length of the pairing. A pairing-enumeration and set partitioning approach is the manner in which many major airlines schedule their crews [2].



	x_1	x_2	x_3	\dots	
Customer 1 :	1	0	0	\vdots	= 1
Customer 2 :	0	1	0	\vdots	= 1
Customer 3 :	0	1	1	\vdots	= 1
Customer 4 :	0	1	0	\vdots	= 1
Customer 5 :	1	0	1	\vdots	= 1

Figure 6.1: A Vehicle Routing Problem and its Set Partitioning Formulation

6.2 Parallelizing the Sequential Algorithm

Recall from our discussion in Section 2.2 that our procedure to solve the SPP is a heuristic based on evaluating the root node of the branch and bound tree. Clearly, we cannot make use of parallelism in the same way as we did for general MIP; we must decompose the problem into smaller units of computation. Again, this is a main reason why we study the SPP – the parallelization issues are different than for general MIP.

The basic algorithm consists of seven computing tasks:

- duplicate column removal,
- dominant row identification,
- solving linear programs,
- performing heuristics,
- reduced cost fixing,
- probing, and
- cut generation.

Our approach is to use parallelism to help us with each of these tasks individually. Thus, the parallel procedure contains some inherent synchronization as it steps from task to task. Conceptually, the algorithm contains a number of worker processes that perform the algorithmic components and a controlling process that oversees the synchronization and overall flow of the algorithm. We mention this here in order to aid the discussion of the parallelization of the algorithm. A more

detailed discussion of the implementation is delayed until Section 6.4. In the next sections, we explain how each of the computing tasks are parallelized.

6.2.1 Duplicate Column Removal

The task of duplicate column removal amounts to computing hash values for all columns, finding duplicate hash values, and verifying whether or not duplicate hash values indeed correspond to duplicate columns.

To compute the hash values for the columns and verify duplicate columns, a domain decomposition approach to parallelization is taken. A random hash function over the rows (as described in Section 2.2.1) is computed and passed to the worker processors at the time of their initialization. When duplicate columns are to be identified, the controller determines a beginning and ending index b_i and e_i for each worker process W_i . Let n be the number of columns in the formulation and p be the number of worker processes. The simple work division scheme of

$$\begin{aligned} b_1 &= 1, \\ e_i &= \lfloor in/p \rfloor \quad \forall i = 1, \dots, p-1, \\ b_i &= e_i + 1 \quad \forall i = 2, \dots, p, \\ e_p &= n, \end{aligned}$$

is sufficient to evenly divide the work among the processors.

The workers receive the indices b_i and e_i and determine hash values $h(A_j), \forall b_i \leq j \leq e_i$. The hash values are passed back to the controller process, which collects the values and determines the duplicates. The work of verifying that duplicate hash values are really duplicate columns is also divided among the worker processors in

a manner similar to that of computing the hash values. Once the controller receives the information about which columns are actually duplicates, the information is broadcast to the worker processors, so that the representation of the problem remains current on all of the processors.

Ignoring overhead, if t is the number of non-zeroes of A , and we have p worker processes, we would expect to take time $O(t/p + n \log n)$ in order to identify potential duplicate columns. This complexity could be reduced by performing the sorting operation needed to find duplicate hash values in parallel. Duplicate column identification takes up a very small percentage of the overall time of the algorithm, so we assume that the complexity of implementing a parallel sorting algorithm outweighs the potential benefits.

6.2.2 Dominant Row Identification

Dominant row identification is another natural task to be parallelized using domain decomposition. The rows must be checked pairwise, and we would like to evenly divide the number of comparisons to be made among the p worker processors. We say a row index $k < m$ is *validated* if row A_k is checked for dominance against each row $A_j, m \geq j > k$. For each worker process W_i , we need to determine a set of rows T_i for W_i to validate such that the number of row comparisons is roughly equal for each processor. We choose the sets T_i as $T_1 = \{1, 2, \dots, e_1\}, T_2 = \{e_1 + 1, e_1 + 2, \dots, e_2\}, \dots, T_p = \{e_{p-1} + 1, e_{p-2} + 2, \dots, m\}$. In order to validate the set of rows in T_i , worker W_i must perform $\sum_{k=e_{i-1}+1}^{e_i} (m - k)$ row comparisons. The total number of row comparisons is $m(m - 1)/2$, so each worker should perform $m(m - 1)/2p$ comparisons in order that the work be distributed equally. Thus, e_1

should satisfy

$$\sum_{k=1}^{e_1} (m - k) = \frac{m(m-1)}{2p}.$$

Solving this equation yields

$$e_1 = \left\lfloor m - \frac{1}{2} - \sqrt{\left(1 - \frac{1}{p}\right)m^2 - \left(1 - \frac{1}{p}\right)m + \frac{1}{4}} \right\rfloor.$$

Recursively, and in a similar fashion, we can show that for $1 \leq j \leq p-2$, we would like to choose

$$e_{j+1} = \left\lfloor m - \frac{1}{2} - \sqrt{\left(1 - \frac{1}{p}\right)m^2 - \left(1 + 2e_j - \frac{1}{p}\right)m + \frac{1}{4} + \frac{e_j(e_j + 1)}{2}} \right\rfloor.$$

This simple scheme should prove sufficient to evenly divide the work among the processors. The work of determining the variables to be fixed is left to the controller process, which broadcasts this information to the worker processes in order to keep the problem description consistent.

The dominance relation between rows is transitive. Hence, sharing information about dominant rows among the processors could be beneficial. However, dominant row identification is not a very computationally intensive task, so the overhead required to share dominance information likely outweighs the benefit. We do not share dominant row information among the processors.

Note that removing dominant rows can lead to more duplicate columns. Therefore, the operations of duplicate column removal and row dominance testing are iterated upon until no more problem reductions can be done.

6.2.3 Solving the Linear Program and Finding Feasible Solutions

Once the initial linear programming relaxation of SPP is to be solved, there is an opportunity to decompose the computing task using functional decomposition and to reduce the amount of synchronization required by the algorithm. The primal-dual subproblem simplex method is performed on one processor, while the remaining processors search for a feasible solution using one of the heuristics described in Section 2.2.4.

There are two motivations for breaking the problem up in this way. The first is that the primal-dual subproblem algorithm is not especially suited for parallelism, so it makes sense to use the remaining processors to do the useful work of searching for feasible solutions. (Klabjan [69] has parallelized the primal-dual subproblem simplex method and achieved only small speedups). Second, at each iteration of the primal-dual subproblem simplex method, we obtain information that can be used to help guide the heuristics. We now describe our functional decomposition approach in more detail.

The linear program is solved by the controlling process, and the worker processes perform the heuristics. To begin, each of the worker processes is passed a dual feasible solution and performs the randomized dual-based **Heuristic I** of Section 2.2.4. The solution of the linear program itself is naturally broken into two phases. In the first phase, the primal solution x from the subproblem is not feasible to the linear programming relaxation, and in the second phase it is. After each iteration of the first phase, the feasible dual solution ρ is passed to the workers, and the workers use this new feasible solution as a starting point for the

dual based heuristic. After each iteration of the second phase, the set of columns K from the iteration is passed to one of the workers who is performing the dual based heuristic. This worker stops the dual based heuristic and performs the primal based **Heuristic III** of Section 2.2.4 by solving the integer program over the set of columns K .

All feasible solutions found by the heuristics are passed to the controlling process, which broadcasts new solution values to the worker processes. Improved solution information can be useful to the heuristic processes. For example, the primal based **Heuristic III** is often speeded up dramatically by only searching for a solution better than a given value.

If a worker finishes **Heuristic III** before the solution of the initial LP is complete, it returns to performing the dual-based **Heuristic I**. The dual feasible solution ρ found at each iteration of the primal-dual subproblem simplex method is stored for use as a starting point to **Heuristic II**. The use of **Heuristic II** will be discussed in more detail in Section 6.3.1. Only the initial linear programming relaxation is solved by the primal-dual subproblem simplex method. All other linear programs are solved using the dual simplex method.

6.2.4 Reduced Cost Fixing

Determining the variables that can be fixed based on reduced costs is very similar to computing the hash values used in determining duplicate columns. However, in order to perform reduced cost fixing in parallel, the vector of dual variables from the linear programming solution would need to be broadcast to the worker processes each time reduced cost fixing was to be performed. We therefore presume that the potential benefit from performing the reduced cost fixing in parallel outweighs the

communication cost required. The controller process determines which variables can be fixed based on their reduced costs and then broadcasts this information to the worker processes.

6.2.5 Probing

Probing is another natural task to parallelize using a domain decomposition approach. The indices of the variables on which to probe are simply divided among the worker processes. An important difference between probing and the other tasks we have parallelized by domain decomposition is that the implications found at one processor during probing can be very useful to the probing operation at other processors. By sharing implication information among the processors, we may be able to reduce the number of rounds required to produce all implication information, and therefore achieve superlinear speedup in probing. If we are only performing one round of probing (we probe on each variable exactly once), then during the round, we wouldn't expect to see a superlinear speedup in performing this work. Instead, the positive effect would manifest itself as finding more implications and fixing more variables in parallel than if the round of probing were done sequentially.

During probing, implications are found very often and constitute a small amount of information. The clear method for sharing this type of information is to use a buffered-prefetch approach. When probing on a variable, implications are stored in a buffer. The implications in the buffer are broadcast to all other processors when probing on the variable is complete.

In addition to the synergetic effects of sharing implications, we obtain the normal speedup effects from decomposing the work of the problem. Since probing

is generally the most time consuming task in the sequential algorithm, we expect that the time saved by doing probing in parallel will generally be quite significant.

6.2.6 Cut Generation

Of all the techniques mentioned so far, cut generation seems the least natural to decompose. Fortunately, our job of decomposing cut generation is made easier by the separation heuristics used to find violated inequalities. Both the heuristic used for clique inequality separation and the heuristic used for odd cycle inequality separation start with a given node $v \in V(CG_F)$ and perform a limited search for violated inequalities starting from this vertex. Hence, we may use a domain decomposition approach to finding violated inequalities by dividing the nodes from which the heuristics start their search for violated inequalities. Each processor gets an equal number of vertices of $V(CG_F)$ from which to begin a search for violated inequalities.

Note that the same violated inequality may be found multiple times, even from different starting points for the heuristic. Thus, a hashing scheme, similar to that used to eliminate multiple cuts in the case of general mixed integer programming is used.

6.3 Computational and Control Issues

6.3.1 Algorithm Flow

Now that all the basic components have been described and strategies for parallelizing the components have been explained, we can state the order in which the

components are performed. Figure 6.2 is a flowchart of our heuristic for solving the set partitioning problem.

As previously mentioned, the duplicate column removal and row dominance techniques can potentially be iterated upon many times. The *Reduction* box in the algorithm denotes an entire sequence of column removal and row dominance operations until both of these operations fail to reduce the problem size. The problem reduction techniques are executed in parallel as discussed in Sections 6.2.1 and 6.2.2.

The *P-D LP & Heuristics* box denotes the simultaneous execution of the primal-dual subproblem simplex method and heuristics as explained in Section 6.2.3. *RCF* represents reduced cost fixing, and *LP* denotes the dual simplex method.

We think of the heuristic as being broken into two phases. In the first phase, we are interested in obtaining rough upper and lower bounds on the optimal solution by solving the initial linear programming relaxation and by finding feasible solutions. In the second phase, we are interested in refining the upper and lower bounds and in reducing the problem size through the use of probing, cutting planes, and more sophisticated heuristics.

Probing is a time consuming operation, and we would like to probe on as small a problem as possible. Therefore, before we enter the second phase of our procedure, if we have found no feasible solution (and hence can fix no variables based on their reduced cost), we perform the operations denoted by the *Heuristics* box in Figure 6.2. In this step, we use parallelism in a functional decomposition manner. Some processors perform the dual-based **Heuristic II**, and some processors perform the primal based **Heuristic III**. Which processors perform which heuristic and the information used to start the heuristics will be discussed in Sections 6.3.2

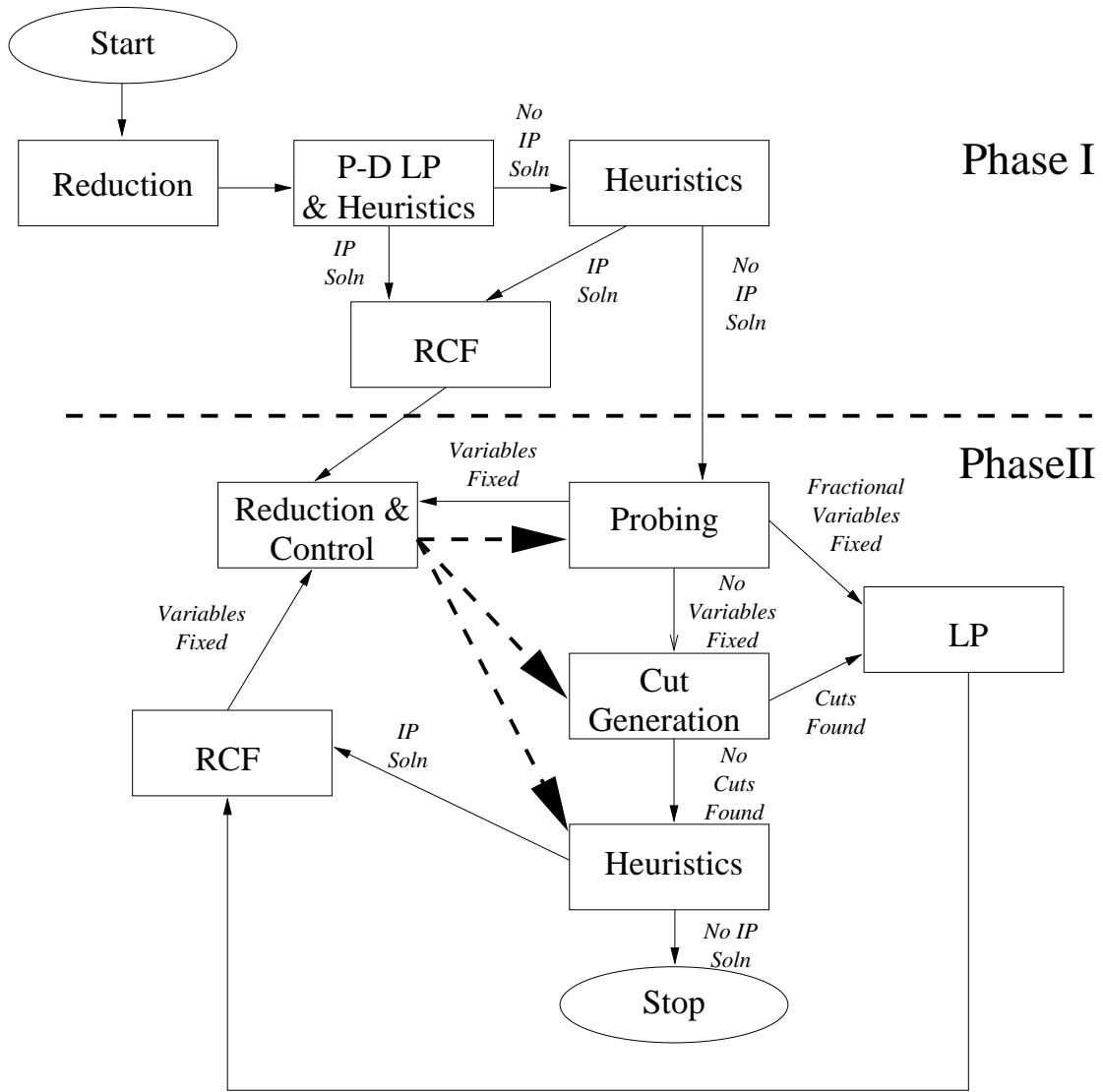


Figure 6.2: Set Partitioning Heuristic

and 6.3.3.

The main control decision to be made is represented by the three large dashed arrows in Figure 6.2. In order to help make the decision of which component to perform at this point, two statistics are kept: the percentage of variables fixed since the last time the algorithm did a round of probing (ζ) and the number of consecutive rounds of cut generation (θ). Figure 6.3 is a flowchart of the inner workings of the *Reduction & Control* box in Figure 6.2. The rationale of the control decision is as follows. During preliminary testing, it was observed that if the number of variables fixed by probing or reduced cost fixing was small, then the further implications gathered by doing additional rounds of probing was also small. Since probing is costly, a round is performed only if significant benefit will be attained (i.e. ζ is large). If not probing, then the decision of whether to add cuts or to perform a heuristic must be made. The question to answer here is whether the upper bound on the problem or lower bound on the problem is likely to be improved. Initially, if variables are fixed based on their reduced cost, then it is assumed that the upper bound from the feasible solution is good and we decide to try to improve the lower bound through the addition of cutting planes. If after a number of consecutive rounds of cut generation, we are unable to fix a significant percentage of the variables, we decide to try and improve the upper bound by performing more heuristics.

As mentioned in Section 6.2.5, parallelism can help to speed up the probing operation a great deal. Thus, we may wish to probe more often in a parallel algorithm than in a sequential one. The amount of probing can be easily increased by reducing the percentage of variables that must be fixed before another round of probing is performed.

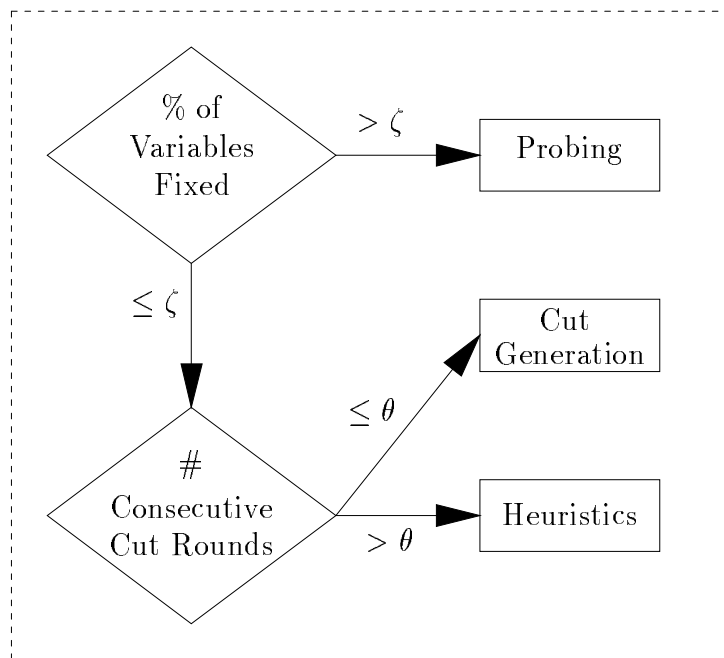


Figure 6.3: Main Control Decision

6.3.2 Heuristics

When we come to the point of the procedure denoted by the *Heuristics* box in Figure 6.2, there is another choice to be made. We must decide the number of each type of heuristic to run.

Computational testing quickly revealed that **Heuristic I** was a very fast procedure, but rarely yielded good solutions to difficult problems. Therefore, **Heuristic I** is not performed in the *Heuristics* component of the procedure, only while the initial linear programming relaxation is being solved.

We adopted the following simple adaptive scheme for determining the number of instances of **Heuristic II** and **Heuristic III** to run. If the best solution so far was found by either of the dual-based heuristics (**Heuristic I** or **Heuristic II**), then 2/3 of the worker processes perform **Heuristic II** and 1/3 perform the primal based **Heuristic III**. Conversely, if the best solution so far was found by **Heuristic III**, then 2/3 of the worker processes perform this heuristic and the remaining 1/3 perform **Heuristic II**. If no feasible solution exists, then 1/2 of the worker processes perform **Heuristic II** and 1/2 perform **Heuristic III**.

The appropriate choice of the parameters κ and δ in Equation (2.3) to guide **Heuristic II** is an important and difficult one. In general, the smaller the value of κ , the less likely the algorithm is to find a feasible solution, but solutions found will be of high quality. Wedelin [118] suggests performing a quick “sweep” for an appropriate value of κ , where a feasible solution is found, and then a more thorough search for smaller and smaller values of κ . We use a similar updating procedure for κ . An appropriate value of δ in Equation (2.3) was found to be far less crucial to the heuristic’s effectiveness. A value of $\delta = 0.001$ is used.

6.3.3 Choosing Columns

If the procedure reaches the point denoted by the *Heuristics* box in Figure 6.2, then the sets of columns K from iterations of the primal-dual subproblem simplex method were insufficient for the primal based heuristic to find a good solution. We therefore must increase the size of this set. In this section, we describe our strategy for performing this task.

We have two goals in choosing sets of columns for processors. First, we would like to choose sets of columns that are likely to contain good feasible integer solutions. Second, we would like to ensure that the different processors that are to perform the primal heuristic get sufficiently different sets of columns. We have adopted the following column choice strategy.

In order to improve the chances that the set of columns we choose will contain a feasible solution, we would like to have each row covered “enough”. Let T_i be the set of columns for which there is a “1” entry in the i th row of A . Define $\Phi(s, T_i)$ to be a set of s randomly chosen columns of T_i , and for a set of columns \hat{K} , let $\tau_{\hat{K}}(i)$ be the number of “1” entries in the i th row of the matrix created from the columns of \hat{K} . Given a starting row index i_0 , a target covering number r , a set of columns \hat{K} , and a target size R , Algorithm 6.1 is a procedure for increasing the number of columns in \hat{K} to R .

Algorithm 6.1 ColumnChoice(i_0, r, \hat{K}, R)

0. **Initialize.** $i = i_0$
 1. **Increase Set.** If $\tau_{\hat{K}}(i) < r$, $\hat{K} = \hat{K} \cup \Phi(\min(r - \tau_{\hat{K}}(i), |T_i|), T_i)$.
 2. **Complete?** If $|\hat{K}| \geq R$, stop. Else $i = i + 1$. If $i > m$, $i = 1$. Go to 1.
-

For each processor that is to perform the primal heuristic, a different value of

i_0 is chosen and `ColumnChoice`($i_0, \alpha n, K, \Lambda|K|$) is called, where α is the density of the matrix A , K is the set of columns from the last iteration of the primal-dual subproblem simplex method, and Λ is a constant greater than one

The time spent performing the heuristic phase is limited to a maximum time of T_H . If a primal heuristic procedure initialized with the set of columns K_0 finishes in a time less than T_H and is unsuccessful in finding an improved integer feasible solution, a new primal heuristic procedure is started with a set of columns generated from `ColumnChoice`($i_0, \alpha n, K_0, \Lambda|K_0|$).

If the set of columns passed to the primal heuristic is the entire set of columns remaining in the problem, then upon completion of the primal heuristic, we can conclude that the solution returned is optimal. If in time T_H none of the heuristic procedures are able to find an improved solution, then the procedure stops.

6.4 PSP – A Parallel Code for Set Partitioning

In this section, we describe PSP, a program for solving set partitioning problems on message passing computers. The implementation of PSP is based on the concepts we have presented in this chapter, and PSP's design is flexible enough to allow for easy experimentation with these concepts. Like PARINO, PSP uses the entity-FSM structure of NAYLAK. There are two types of entities in PSP– *controller* entities and *worker* entities. The controller entity is run on one processor, and worker entities are run on the remaining processors.

From the discussion Section 6.2 of how the heuristic is parallelized, the duties of each entity should be clear. The controller entity has one FSM that is responsible of managing the overall flow of the heuristic and solving the lower bounding linear

programs. The worker entity has FSMs to perform the majority of the work of the heuristic: checking duplicate columns, checking dominant rows, performing primal heuristics, probing, and generating cuts.

For our heuristic procedure to be effective, it is quite important that the operations at the worker entities are interruptible. For example, when performing heuristics, once one processor finds a good feasible solution (one where $\zeta\%$ of the remaining variables are fixed), there is little need to wait for the other heuristics to complete. The actions of the worker entities are performed by FSMs whose states consist of sufficiently fine grained operations in NAYLAK, so it is a simple matter to interrupt them if a good feasible solution is found.

Like PARINO, PSP interacts with the linear and integer programming software through the C++ class library LPSOLVER. PSP is written in C++ and contains over 12,000 lines of code. The code was compiled using the GNU g++ compiler, version 2.8.1, with the optimization level `-O3`. In these experiments, we have used the PVM message passing library, version 3.3.11 for communication [45].

6.5 Computational Experiments

6.5.1 A Test Suite of Problems

The problems on which we will test PSP are taken from a variety of real world applications. The first set of instances is the famous test suite of crew scheduling problems of Hoffman and Padberg [62]. The second source of instances arose in the context of handicapped bus scheduling in Berlin [19]. A third set of instances are set partitioning formulations of some capacitated vehicle routing instances [107].

A final instance arises in a two-phase approach to solving an inventory routing problem [21].

Table 6.1 shows some statistics for the instances in our test suite. For a row i of $A \in \{0, 1\}^{m \times n}$, τ_i denotes the number of “1” entries in row i , and v_j denotes the number of “1” entries in column j . The symbol σ is meant to denote the standard deviation of the values $\tau_1, \tau_2, \dots, \tau_m$ or v_1, v_2, \dots, v_n . The difficulty of solving an instance of SPP seems to have a weak dependence on the number of columns n , but the number of rows m in an instance is a good indicator of its difficulty. Also, we conjecture that the values of $\sigma(\tau)/m$ and $\sigma(v)/n$ are related to the difficulty of a set partitioning instance. These values have an impact on the number of fractional extreme points of the underlying polyhedron. The smaller the value, the more difficult the problem is to solve. See Ryan and Falkner [109] for more details. To give an indication of the difficulty of the instances in our test suite, in Table 6.1 we also show how the mixed integer programming package CPLEXv4.0 [28] fares in solving the instances on an IBM RS/6000 Model 390. The largest instances were not run due to a lack of memory on the machine. On the instance *v1617*, CPLEX exhausted the memory resources of the machine. In Table 6.1, the instances above the first line are crew scheduling instances, above the second line are bus scheduling instances, above the third line are vehicle routing instances, and the final instance comes from an inventory routing problem.

We see some evidence to support our conjecture relating statistics from the matrix A to the difficulty of solving a SPP instance. First, among the airline crew scheduling instances, the instances with the most rows and smallest values of the variance in covering numbers are the most difficult to solve. Second, a number of the the bus scheduling instances have very small variance among the

Table 6.1: Characteristics of Set Partitioning Problem Test Instances

Name	m	n	$\alpha(\%)$	$100\sigma(\tau)/n$	$100\sigma(v)/m$	CPLEX Time	CPLEX Gap (%)
hp1	61	118607	13.96	11.18	2.36	0:35	0.0
air04	823	8904	1.00	0.65	0.27	4:00:00	0.39
air05	426	7195	1.70	1.17	0.45	4:00:00	0.05
nw04	36	87842	20.22	13.68	3.53	25:36	0.0
us01-3	145	300000	8.94	10.39	1.50	> 10 hours	∞
us01-4	145	400000	8.95	10.42	1.48	–	–
us01-5	145	500000	8.92	10.44	1.48	–	–
t0415	1518	7524	0.44	1.01	0.42	> 30 hours	∞
t1716	467	56865	0.94	0.0082	0.42	> 30 hours	∞
t1717	551	73885	0.80	0.0058	0.36	> 30 hours	∞
v1617	1619	113655	0.23	0.42	0.070	Memory	1.37%
eil33-2	32	4516	30.62	14.50	8.34	4:00:00	18.1%
eilA76-2	75	25175	12.15	7.49	2.76	4:00:00	∞
eilA101	100	5073	11.04	5.30	3.59	4:00:00	30.1%
eilB76-2	75	19349	9.35	7.10	2.34	4:00:00	0.25
eilB101-2	100	53444	10.81	8.34	3.05	4:00:00	∞
prax	39	20315	12.40	8.16	1.64	4:00:00	0.15

covering numbers and a significant number of rows. These instances appear to be intractable by standard integer programming methods.

6.5.2 Speedup and Performance

In this section, we demonstrate the effectiveness of PSP on the instances in our test suite. The parallel heuristic for solving SPP is much more synchronized than our implementation of the parallel branch and cut algorithm. Therefore, the variation between different runs on the same instance is very small, and we can avoid running each instance multiple times in order to reduce the chances of randomness skewing our results. Figure 6.4 shows the settings used in our computational experiments. Machines in the **beetle** cluster were used to perform all experiments in this section.

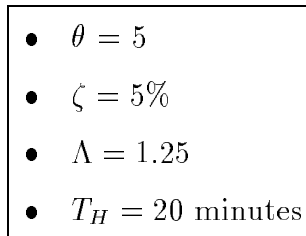
- 
- $\theta = 5$
 - $\zeta = 5\%$
 - $\Lambda = 1.25$
 - $T_H = 20$ minutes

Figure 6.4: Settings for PSP

The PSP code was able to find the optimal solution and prove it optimal in eight of the sixteen instances. Table 6.2 shows the time required to solve instances on 2, 4, 8, and 16 processors, as well as the “relative efficiency” (see Equation (4.7)).

From Table 6.2, we see that using parallelism can be beneficial, but only on instances requiring over 1000 seconds to solve. Also note the greatly improved

Name	p	Time (sec.)	\hat{E}
air04	2	2201	1.00
air04	4	1774	0.62
air04	8	997	0.32
air04	16	1102	0.13
air05	2	1331	1.00
air05	4	618	0.72
air05	8	426	0.45
air05	16	381	0.23
eilB76-2	2	672	1.00
eilB76-2	4	536	0.42
eilB76-2	8	690	0.14
eilB76-2	16	727	0.06
nw04	2	1035	1.00
nw04	4	440	0.78
nw04	8	337	0.44
nw04	16	281	0.25
prax	2	1317	1.00
prax	4	575	0.76
prax	8	287	0.66
prax	16	212	0.41
us01-3	2	1013	1.00
us01-3	4	639	0.53
us01-3	8	544	0.27
us01-3	16	350	0.20
us01-4	2	475	1.00
us01-4	4	403	0.39
us01-4	8	294	0.23
us01-4	16	378	0.08
us01-5	2	449	1.00
us01-5	4	371	0.40
us01-5	8	427	0.15
us01-5	16	401	0.07

Table 6.2: PSP Performance on Solved Instances

performance of the specialized algorithm over directly solving the integer program using CPLEX in Table 6.1. The “us” crew scheduling instances are very easy despite their size. For *us01-5* nearly half of the computation time is spent in simply reading the problem and initializing the data structures. For instances where parallelism is useful, it is interesting to see in which section of the algorithm taking advantage of parallelism has the greatest effect. To that end, consider Table 6.3, where the time in various components of the algorithm is reported.

Name	p	Time (sec.)	Prep. Time	Probe Time	Cut Time	LP Time	Heur Time
air05	2	1331	2	11	20	13	1279
air05	4	618	3	8	30	14	559
air05	8	426	4	2	15	13	381
air05	16	381	1	1	15	14	334
nw04	2	1035	4	848	73	6	72
nw04	4	440	5	301	18	4	70
nw04	8	337	7	174	31	8	70
nw04	16	281	2	102	37	7	98
us01-3	2	1013	17	652	7	12	224
us01-3	4	639	25	236	7	18	152
us01-3	8	544	33	156	12	17	152
us01-3	16	350	8	60	8	12	88
prax	2	1317	3	1153	113	14	28
prax	4	575	4	439	49	11	52
prax	8	287	4	191	46	11	24
prax	16	212	1	119	40	11	24

Table 6.3: Percentage of Time in Solution Phases

Table 6.3 clearly shows the benefit of performing probing in parallel. In some cases, the amount of time spent performing heuristics can also be reduced because a good solution is found more quickly. We do not see positive effects from

parallelizing our simple preprocessing techniques. In order to obtain benefit in performing these operations in parallel, larger instances must be considered.

Table 6.4 shows the performance of PSP on instances for which it cannot prove the optimality of its solution. The column heading *Best Known Gap* refers to the percentage gap between the solution found by our procedure z_{HEUR} and the best known solution to the problem $z_{BEST} : 100(z_{BEST} - z_{HEUR})/z_{HEUR}$. The column heading *Provable Gap* refers to the value $100(z_{HEUR} - z_{LP})/z_{HEUR}$, where z_{LP} is the solution to the final linear programming relaxation in the our SPP heuristic.

From Table 6.4, we conclude that increasing the number of processors can help find better solutions. This is due to the fact that more and different heuristics are run. In some cases (*eilB101-2* and *t0415* for example), increasing the number of processors can also help improve the strength of the formulation. In this case, more cutting planes are found with the addition of extra processors. In nearly all the unsolvable cases, the heuristics we use in the procedure failed to find a feasible solution close to the best known or optimal value. The notable exception to this is *t0415* where our code improves upon the value of the best known solution to the problem. If a better solution was found, then the problem reduction procedures would be much more effective. From this we conclude that the lower bounding and problem reduction procedures in PSP work very well, but more emphasis should be placed on procedures for finding good feasible solutions.

Name	p	Time (sec.)	Best Known Gap (%)	Provable Gap (%)
eil33-2	2	1403	6.7689	18.2453
eil33-2	4	1385	6.7689	18.1733
eil33-2	8	737	6.7689	18.1441
eil33-2	16	1327	6.7689	18.1526
eilA101	2	2659	58.1046	63.6674
eilA101	4	2780	16.1146	26.9180
eilA101	8	2705	16.1146	25.6422
eilA101	16	2783	16.1146	25.9306
eilA76-2	2	3602	16.9713	18.5922
eilA76-2	4	3604	9.8082	11.0640
eilA76-2	8	3602	15.6717	17.0121
eilA76-2	16	2616	7.3278	8.7123
eilB101-2	2	10566	32.0273	37.7385
eilB101-2	4	5625	13.1511	20.4483
eilB101-2	8	3718	13.1511	20.1678
eilB101-2	16	3637	13.1511	19.7695
t0415	2	3602	-0.0001	8.3037
t0415	4	3602	-0.0001	8.2823
t0415	8	3602	-0.0001	8.2823
t0415	16	3603	-0.0001	8.2632
t1716	2	3048	No Soln.	No Soln.
t1716	4	3300	81.3672	85.9722
t1716	8	3602	81.3672	85.9723
t1716	16	3602	81.3672	85.9724
t1717	2	3603	No Soln	No Soln
t1717	4	3603	83.4433	87.9299
t1717	8	3602	83.4433	87.9333
t1717	16	3602	83.4433	87.9324
v1617	2	3602	9.5024	9.5300
v1617	4	3430	2.7050	2.8264
v1617	8	3423	2.7050	2.7576
v1617	16	3372	2.7050	2.7805

Table 6.4: PSP Performance on Unsolved Instances

CHAPTER 7

Conclusions

7.1 Contributions

This is a computationally oriented thesis where we have investigated many algorithmic and implementation issues arising in parallel solution approaches to mixed integer programming problems.

An important concept in designing a parallel optimization algorithm for a distributed memory computer architecture is that of *globally useful information*. Beneficial sharing of global information allows the parallel computer to be used to its greatest capacity, and may result in superlinear speedup. Conversely, the power of parallel computing can be diminished if information is not shared in a sensible manner. Thus, examining the ways in which globally useful information may be shared is an important issue. In Chapter 3, we develop a categorization of information sharing schemes, and we state advantages and disadvantages of each category. The manner in which information is best shared depends on the underlying importance of the information.

In Chapter 4, we consider the parallelization of a linear programming based branch and bound algorithm for solving mixed integer programs. A careful study

of the importance of pseudocost information and search strategies was undertaken in a sequential setting. Leveraging on the knowledge gained from our sequential experiments, we discuss good schemes for pseudocost management and node selection in a parallel algorithm. Through a series of computational experiments, we show that for small problem instances, pseudocost information need not be shared among the processors. For large instances, pseudocosts should be initialized in parallel and all initialization information should be broadcast to other processors. In terms of node selection, we show that depth first oriented strategies are to be preferred in a parallel algorithm.

Chapter 5 deals with parallel branch and cut. Similar to Chapter 4, a systematic investigation of the importance of cutting planes and of the algorithmic issues in cut management was performed in a sequential setting. We show that cutting planes are most useful for the subtree rooted at the node at which the cuts are generated, which gives additional justification for more depth first oriented search strategies in parallel algorithms. We show that sharing cutting plane information among processors can often be beneficial, and for moderate levels of parallelism, a well implemented fully centralized scheme can be an effective means of performing the sharing. From the results of our sequential study and through experimentation, we show that only cutting planes generated near the top of the branch and bound tree need to be shared among the processors in order to have an effective branch and cut algorithm. This is an important conclusion for designing scalable sharing schemes for massively parallel computer architectures.

In Chapter 6, we describe a parallelization approach to a linear-programming based heuristic for solving SPP. We show how to reap the benefits of parallelism in our algorithm in a number of ways. The operation of probing can be performed

much more expeditiously in parallel, and in addition the sharing of implication information can lead to synergetic, superlinear-speedup-like effects. Secondly, the increased number of processors at our disposal gives the ability to employ a wide assortment of heuristics, which in turn leads to better solutions in a fixed amount of time.

7.2 Future Research

The research described in this thesis has led to a better understanding of the basic issues in parallelizing linear programming based algorithms for solving discrete optimization problems. In the process, a number of interesting questions have evolved.

Number of descendants

Is it possible to estimate the number of descendant nodes that will be created for a node in the active set? For example, the number of fractional variables in the solution to the linear programming relaxation could give some indication as to the number of descendants. Being able to do so would impact two aspects of a parallel branch and cut algorithm. First, quantity balancing can make use of this information to pass the number of nodes necessary to truly balance the work between processors. Second, cut sharing can use this information to “intelligently” share cuts generated at nodes that will have a large number of descendants.

Value of cutting planes

Is it possible to estimate the value of a cutting plane or a set of cutting planes? We have observed that the sharing of cutting planes often tightens the linear programming relaxation. Are better separation heuristics for lifted cover inequalities

required, or are the shared cuts “complementary” in some sense? If shared cuts create a complementary set that tightens the formulation, then are there ways to determine a good *set* of cuts to add to the linear programming relaxation in a branch and cut algorithm? At the heart of this issue is how to measure the quality of a single cutting plane and of a set of cutting planes with respect to how they enhance the solution process.

Adaptive Schemes

In our research, we have found that different strategies perform best on different instances. Can the knowledge gained from this study be used to develop adaptive schemes, where the algorithm configures itself at runtime? For example, the decision of whether or not to perform parallel pseudocost initialization and sharing can be based on the time required to solve the initial linear programming relaxation. Statistics such as cut generation depth, cut generation and cut waiting times, and cut generation and cut matching success rates can be used to guide an adaptive cut sharing scheme. For algorithms to solve set partitioning problems, we can consider an adaptive column choice scheme in which the solution of the integer program returns information useful to make the next column choice for a primal heuristic.

Parallel Branch-and-Price Algorithms

With the knowledge gained by developing implementations of a parallel branch and cut algorithm, the next logical step is to design and implement a parallel branch and price approach.

APPENDIX A

Tables of Sequential Branch and Bound Experimental Results

Table A.1: Comparison of Pseudocost Initialization Methods

Problem	Initialization Method	Nodes	Final Gap	Sol. Time(sec.)
air04	Obj. Coef.	2618	0.159%	3600
air04	Computed	0	XXX	3600
air04	Computed Fractional	195	0%	2351
air04	Averaged	1588	0.950%	3600
arki001	Obj. Coef.	31915	0.0155%	3600
arki001	Computed	30739	0.00350%	3600
arki001	Computed Fractional	38583	0.00454%	3600
arki001	Averaged	36086	0.0101%	3600
bell3a	Obj. Coef.	30926	0%	99
bell3a	Computed	28627	0%	96
bell3a	Computed Fractional	28319	0%	97
bell3a	Averaged	30864	0%	98
bell5	Obj. Coef.	120000	0.387%	XXX
bell5	Computed	21365	0%	66
bell5	Computed Fractional	14013	0%	45
bell5	Averaged	64361	0%	240
gesa2	Obj. Coef.	32815	0.0572%	3600
gesa2	Computed	34681	0.0233%	3600
gesa2	Computed Fractional	40885	0.00814%	3600
gesa2	Averaged	32951	0.0226%	3600

Problem	Initialization Method	Nodes	Final Gap	Sol. Time(sec.)
harp2	Obj. Coef.	4900	0.185%	3600
harp2	Computed	11145	7.75e-06%	3600
harp2	Computed Fractional	10695	9.02e-06%	3600
harp2	Averaged	6476	0.148%	3600
l152lav	Obj. Coef.	7481	0%	481
l152lav	Computed	1573	0%	241
l152lav	Computed Fractional	1511	0%	133
l152lav	Averaged	9039	0%	533
mod011	Obj. Coef.	414	5.15%	3600
mod011	Computed	372	6.09%	3600
mod011	Computed Fractional	418	5.17%	3600
mod011	Averaged	432	5.37%	3600
pp08a	Obj. Coef.	85000	5.47%	XXX
pp08a	Computed	85000	5.20%	XXX
pp08a	Computed Fractional	83000	5.10%	XXX
pp08a	Averaged	85000	5.73%	XXX
qiu	Obj. Coef.	3614	112%	3600
qiu	Computed	4116	131%	3600
qiu	Computed Fractional	4310	113%	3600
qiu	Averaged	3717	91.3%	3600
qnet1	Obj. Coef.	30000	3.60%	XXX
qnet1	Computed	73	0%	118
qnet1	Compute Fractional	59	0%	20
qnet1	Averaged	4441	0%	511
rgn	Obj. Coef.	1953	0%	20
rgn	Computed	3465	0%	40
rgn	Computed Fractional	2907	0%	30
rgn	Averaged	3201	0%	37
stein45	Obj. Coef.	54841	0%	2540
stein45	Computed	61283	0%	2553
stein45	Computed Fractional	60315	0%	2283
stein45	Averaged	61933	0%	3008
vpm2	Obj. Coef.	14453	0%	435

Problem	Initialization Method	Nodes	Final Gap	Sol. Time(sec.)
vpm2	Computed	9375	0%	215
vpm2	Computed Fractional	9431	0%	210
vpm2	Averaged	16793	0%	486

Table A.2: Comparison of Pseudocost Update Methods

Problem	Update Method	Nodes	Final Gap (%)	Sol. Time(sec.)
air04	Average	1986	0.104%	3600
air04	First	1523	0.458%	3600
air04	Last	1932	0.105%	3600
arki001	Average	37821	0.00456%	3600
arki001	First	49037	0.00366%	3600
arki001	Last	39512	0.00288%	3600
bell3a	Average	28319	0%	100
bell3a	First	29841	0%	108
bell3a	Last	28293	0%	99
bell5	Average	14013	0%	45
bell5	First	15941	0%	51
bell5	Last	119421	0%	403
gesa2	Average	39190	0.00882%	3600
gesa2	First	37147	0.0155%	3600
gesa2	Last	35682	0.0181%	3600
harp2	Average	10110	9.51e-06%	3600
harp2	First	9734	8.35e-03%	3600
harp2	Last	9910	8.85e-06%	3600
l152lav	Average	1511	0%	138
l152lav	First	1639	0%	141
l152lav	Last	1323	0%	125
mod011	Average	407	5.21%	3600
mod011	First	416	5.64%	3600
mod011	Last	403	5.11%	3600
pp08a	Average	101574	4.76%	3600
pp08a	First	103054	5.46%	3600
pp08a	Last	97446	4.9%	3600
qiu	Average	4172	114%	3600
qiu	First	4844	181%	3600
qiu	Last	4062	124%	3600
qnet1	Average	57	0%	20
qnet1	First	57	0%	20

Problem	Update Method	Nodes	Final Gap (%)	Sol. Time(sec.)
qnet1	Last	57	0%	20
rgn	Average	2907	0%	32
rgn	First	3639	0%	41
rgn	Last	2823	0%	32
stein45	Average	60315	0%	2286
stein45	First	66552	5%	3600
stein45	Last	64079	0%	2463
vpm2	Average	9431	0%	216
vpm2	First	12517	0%	270
vpm2	Last	10689	0%	258

Table A.3: Comparison of Combination of Up and Down Pseudocost Information

Problem	(α_1, α_2)	Nodes	Final Gap (%)	Sol. Time(sec.)
air04	(10, 1)	147	0%	2722
air04	(2, 1)	167	0%	2604
air04	(1, 1)	189	0%	2598
air04	(1, 0)	203	0%	2857
air04	(1, 2)	221	0%	2691
air04	(1, 10)	257	0%	2727
arki001	(10, 1)	25385	0.00327%	3600
arki001	(2, 1)	21096	0.00238%	3600
arki001	(1, 1)	22158	0.00264%	3600
arki001	(1, 0)	15625	0.0049%	3600
arki001	(1, 2)	22002	0.00313%	3600
arki001	(1, 10)	20059	0.00501%	3600
bell3a	(10, 1)	28449	0%	110
bell3a	(2, 1)	28419	0%	109
bell3a	(1, 1)	28321	0%	109
bell3a	(1, 0)	28559	0%	111
bell3a	(1, 2)	28307	0%	109
bell3a	(1, 10)	28445	0%	109
bell5	(10, 1)	202893	0%	771
bell5	(2, 1)	33491	0%	121
bell5	(1, 1)	17527	0%	63
bell5	(1, 0)	281761	0%	1056
bell5	(1, 2)	16599	0%	60
bell5	(1, 10)	17153	0%	62
gesa2	(10, 1)	33999	0.00278%	3600
gesa2	(2, 1)	33646	0.00397%	3600
gesa2	(1, 1)	31788	0.00724%	3600
gesa2	(1, 0)	26629	0.189%	3600
gesa2	(1, 2)	29858	0.0116%	3600
gesa2	(1, 10)	29423	0.0116%	3600
harp2	(10, 1)	12020	7.09e-06%	3600
harp2	(2, 1)	9084	7.46e-06%	3600

Problem	(α_1, α_2)	Nodes	Final Gap (%)	Sol. Time(sec.)
harp2	(1, 1)	5704	3.04e-02%	3600
harp2	(1, 0)	3486	2.21e-02%	3600
harp2	(1, 2)	5729	2.89e-02%	3600
harp2	(1, 10)	5382	2.76e-02%	3600
l152lav	(10, 1)	209	0%	65
l152lav	(2, 1)	313	0%	74
l152lav	(1, 1)	475	0%	103
l152lav	(1, 0)	407	0%	93
l152lav	(1, 2)	1005	0%	167
l152lav	(1, 10)	1061	0%	172
mod011	(10, 1)	382	4.09%	3600
mod011	(2, 1)	394	4.39%	3600
mod011	(1, 1)	373	5.17%	3600
mod011	(1, 0)	375	4.39%	3600
mod011	(1, 2)	397	5.56%	3600
mod011	(1, 10)	396	5.65%	3600
pp08a	(10, 1)	65737	6.66%	3600
pp08a	(2, 1)	71116	5.54%	3600
pp08a	(1, 1)	69241	5.84%	3600
pp08a	(1, 0)	66081	7.81%	3600
pp08a	(1, 2)	69356	6.33%	3600
pp08a	(1, 10)	69515	6.92%	3600
qiu	(10, 1)	3707	105%	3600
qiu	(2, 1)	3775	110%	3600
qiu	(1, 1)	3786	114%	3600
qiu	(1, 0)	3534	117%	3600
qiu	(1, 2)	3770	121%	3600
qiu	(1, 10)	4000	127%	3600
qnet1	(10, 1)	147	0%	44
qnet1	(2, 1)	73	0%	25
qnet1	(1, 1)	57	0%	22
qnet1	(1, 0)	309	0%	71
qnet1	(1, 2)	57	0%	21

Problem	(α_1, α_2)	Nodes	Final Gap (%)	Sol. Time(sec.)
qnet1	(1, 10)	61	0%	21
rgn	(10, 1)	2101	0%	25
rgn	(2, 1)	2137	0%	25
rgn	(1, 1)	2305	0%	27
rgn	(1, 0)	2387	0%	29
rgn	(1, 2)	2327	0%	27
rgn	(1, 10)	2421	0%	28
stein45	(10, 1)	50877	0%	2743
stein45	(2, 1)	47241	0%	2386
stein45	(1, 1)	46749	0%	2404
stein45	(1, 0)	57411	0%	2991
stein45	(1, 2)	48655	0%	2515
stein45	(1, 10)	49269	0%	2547
vpm2	(10, 1)	7939	0%	206
vpm2	(2, 1)	7143	0%	183
vpm2	(1, 1)	8113	0%	206
vpm2	(1, 0)	10365	0%	301
vpm2	(1, 2)	9061	0%	233
vpm2	(1, 10)	9859	0%	266

Table A.4: Comparison of Node Selection Methods

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
10teams	N1	450	1071	0%	0%
10teams	N2	3311	3600	(N/A)	(N/A)
10teams	N3	240	611	0%	0%
10teams	N4	4570	3600	0.649%	1.398%
10teams	N5	290	677	0%	0%
10teams	N6	490	1011	0%	0%
air03	N1	3	62	0%	0%
air03	N2	3	62	0%	0%
air03	N3	3	62	0%	0%
air03	N4	3	62	0%	0%
air03	N5	3	62	0%	0%
air03	N6	3	62	0%	0%
air04	N1	701	3475	0%	0%
air04	N2	1505	3355	0%	0%
air04	N3	684	3600	0%	0.740%
air04	N4	891	3601	0.00178%	0.742%
air04	N5	862	3600	0.00178%	0.232%
air04	N6	499	3605	0.203%	0.554%
air05	N1	520	3600	3.58%	4.031%
air05	N2	2193	2778	0%	0%
air05	N3	1284	2689	0%	0%
air05	N4	1390	2962	0%	0%
air05	N5	1501	2730	0%	0%
air05	N6	2193	2780	0%	0%
bell3a	N1	38421	153	0%	0%
bell3a	N2	23755	102	0%	0%
bell3a	N3	173626	742	0%	0%
bell3a	N4	148294	641	0%	0%
bell3a	N5	39292	360	0%	0%
bell3a	N6	38712	288	0%	0%
bell5	N1	23683	90	0%	0%
bell5	N2	1098601	3600	0.846%	1.502%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
bell5	N3	635000	XXX	0.109%	4.097%
bell5	N4	553000	XXX	0.189%	0.856%
bell5	N5	40723	339	0%	0%
bell5	N6	153075	3600	0%	0.019%
blend2	N1	2365	292	0%	0%
blend2	N2	31424	1674	0%	0%
blend2	N3	9788	578	0%	0%
blend2	N4	3509	161	0%	0%
blend2	N5	4041	441	0%	0%
blend2	N6	3872	296	0%	0%
cap6000	N1	4099	1851	0%	0%
cap6000	N2	21171	3301	0%	0%
cap6000	N3	8884	2328	0%	0%
cap6000	N4	8758	2294	0%	0%
cap6000	N5	6860	3284	0%	0%
cap6000	N6	11403	3077	0%	0%
danooint	N1	561	3600	1.264%	5.330%
danooint	N2	4521	3600	1.264%	5.784%
danooint	N3	477	3600	1.264%	5.729%
danooint	N4	1058	3600	1.264%	5.729%
danooint	N5	1656	3600	0%	4.191%
danooint	N6	1798	3600	0.918%	5.123%
dcmulti	N1	967	38	0%	0%
dcmulti	N2	5563	120	0%	0%
dcmulti	N3	3906	96	0%	0%
dcmulti	N4	3659	85	0%	0%
dcmulti	N5	1176	36	0%	0%
dcmulti	N6	5283	117	0%	0%
dsbmip	N1	150	55	0%	0%
dsbmip	N2	1290	219	0%	0%
dsbmip	N3	40	25	0%	0%
dsbmip	N4	129	55	0%	0%
dsbmip	N5	150	69	0%	0%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
dsbmip	N6	321	118	0%	0%
egout	N1	3	0	0%	0%
egout	N2	3	0	0%	0%
egout	N3	3	0	0%	0%
egout	N4	3	0	0%	0%
egout	N5	3	0	0%	0%
egout	N6	3	0	0%	0%
enigma	N1	6887	112	0%	0%
enigma	N2	4657	33	0%	0%
enigma	N3	518	11	0%	0%
enigma	N4	2480	42	0%	0%
enigma	N5	1080	23	0%	0%
enigma	N6	949	22	0%	0%
fast0507	N1	32	3600	5.747%	6.313%
fast0507	N2	54	3600	6.322%	6.948%
fast0507	N3	33	3600	10.345%	10.220%
fast0507	N4	33	3600	9.770%	9.750%
fast0507	N5	40	3600	10.345%	10.214%
fast0507	N6	37	3600	10.345%	10.219%
fiber	N1	35	13	0%	0%
fiber	N2	105	24	0%	0%
fiber	N3	58	17	0%	0%
fiber	N4	79	20	0%	0%
fiber	N5	105	24	0%	0%
fiber	N6	105	24	0%	0%
fixnet6	N1	79	15	0%	0%
fixnet6	N2	71	10	0%	0%
fixnet6	N3	71	13	0%	0%
fixnet6	N4	70	13	0%	0%
fixnet6	N5	86	15	0%	0%
fixnet6	N6	71	13	0%	0%
flugpl	N1	4305	8	0%	0%
flugpl	N2	4091	5	0%	0%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
flugpl	N3	3551	5	0%	0%
flugpl	N4	3426	5	0%	0%
flugpl	N5	3511	7	0%	0%
flugpl	N6	3665	7	0%	0%
gen	N1	3	2	0%	0%
gen	N2	3	2	0%	0%
gen	N3	3	2	0%	0%
gen	N4	3	2	0%	0%
gen	N5	3	2	0%	0%
gen	N6	3	2	0%	0%
gesa2	N1	23169	3600	0.316%	0.329%
gesa2	N2	50309	3600	0.489%	1.331%
gesa2	N3	40983	3600	0%	0.559%
gesa2	N4	43730	3600	0.079%	0.927%
gesa2	N5	21116	3600	0.316%	0.331%
gesa2	N6	23062	3600	0.306%	0.329%
gesa2_o	N1	22388	3600	0.220%	0.229%
gesa2_o	N2	49978	3600	0.002%	0.950%
gesa2_o	N3	41847	3600	0.009%	0.698%
gesa2_o	N4	45317	3600	0.008%	0.956%
gesa2_o	N5	21227	3600	0.101%	0.115%
gesa2_o	N6	27676	3600	0.014%	0.031%
gesa3	N1	839	142	0%	0%
gesa3	N2	4205	362	0%	0%
gesa3	N3	928	110	0%	0%
gesa3	N4	1031	114	0%	0%
gesa3	N5	726	123	0%	0%
gesa3	N6	4205	362	0%	0%
gesa3_o	N1	971	142	0%	0%
gesa3_o	N2	1041	116	0%	0%
gesa3_o	N3	973	116	0%	0%
gesa3_o	N4	895	112	0%	0%
gesa3_o	N5	1041	117	0%	0%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
gesa3_o	N6	1041	117	0%	0%
gt2	N1	318	3	0%	0%
gt2	N2	776554	3600	111.471%	69.928%
gt2	N3	413	4	0%	0%
gt2	N4	110	1	0%	0%
gt2	N5	202	3	0%	0%
gt2	N6	202	3	0%	0%
harp2	N1	2550	3600	0.404%	0.453%
harp2	N2	21397	3600	0.116%	0.549%
harp2	N3	13455	3600	0%	0.432%
harp2	N4	14240	3600	0.173%	0.606%
harp2	N5	4164	3600	0.389%	0.458%
harp2	N6	4277	3600	0.270%	0.349%
khb05250	N1	15	3	0%	0%
khb05250	N2	15	3	0%	0%
khb05250	N3	15	3	0%	0%
khb05250	N4	15	3	0%	0%
khb05250	N5	15	3	0%	0%
khb05250	N6	15	3	0%	0%
l152lav	N1	337	111	0%	0%
l152lav	N2	402	94	0%	0%
l152lav	N3	637	169	0%	0%
l152lav	N4	434	135	0%	0%
l152lav	N5	964	191	0%	0%
l152lav	N6	402	94	0%	0%
lseu	N1	105	3	0%	0%
lseu	N2	105	2	0%	0%
lseu	N3	115	2	0%	0%
lseu	N4	159	4	0%	0%
lseu	N5	89	2	0%	0%
lseu	N6	105	2	0%	0%
misc03	N1	371	12	0%	0%
misc03	N2	387	9	0%	0%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
misc03	N3	517	16	0%	0%
misc03	N4	385	11	0%	0%
misc03	N5	401	12	0%	0%
misc03	N6	387	9	0%	0%
misc06	N1	53	9	0%	0%
misc06	N2	298	17	0%	0%
misc06	N3	60	7	0%	0%
misc06	N4	61	7	0%	0%
misc06	N5	56	8	0%	0%
misc06	N6	73	10	0%	0%
misc07	N1	39657	3449	0%	0%
misc07	N2	46249	1693	0%	0%
misc07	N3	49055	2846	0%	0%
misc07	N4	44669	2270	0%	0%
misc07	N5	41715	2598	0%	0%
misc07	N6	26613	1175	0%	0%
mitre	N1	38	221	0%	0%
mitre	N2	40	214	0%	0%
mitre	N3	35	209	0%	0%
mitre	N4	35	209	0%	0%
mitre	N5	49	224	0%	0%
mitre	N6	40	215	0%	0%
mod008	N1	101	9	0%	0%
mod008	N2	105	8	0%	0%
mod008	N3	105	8	0%	0%
mod008	N4	105	8	0%	0%
mod008	N5	105	8	0%	0%
mod008	N6	105	8	0%	0%
mod010	N1	17	26	0%	0%
mod010	N2	37	25	0%	0%
mod010	N3	45	46	0%	0%
mod010	N4	43	36	0%	0%
mod010	N5	37	25	0%	0%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
mod010	N6	37	25	0%	0%
mod011	N1	347	3600	0.666%	5.339%
mod011	N2	1594	3600	1.796%	14.880%
mod011	N3	337	3600	0.826%	8.226%
mod011	N4	474	3600	0.916%	9.029%
mod011	N5	422	3600	1.603%	6.304%
mod011	N6	651	3600	0.600%	6.867%
modglob	N1	569	96	0%	0%
modglob	N2	99287	3600	2.15%	3.131%
modglob	N3	357	30	0%	0%
modglob	N4	1089	76	0%	0%
modglob	N5	703	138	0%	0%
modglob	N6	1174	204	0%	0%
noswot	N1	66336	3600	4.651%	4.878%
noswot	N2	287126	3600	4.651%	4.878%
noswot	N3	189713	3600	4.651%	4.878%
noswot	N4	239744	3600	4.651%	4.878%
noswot	N5	50852	3600	4.651%	4.878%
noswot	N6	54369	3600	4.651%	4.878%
p0033	N1	7	0	0%	0%
p0033	N2	14	0	0%	0%
p0033	N3	7	0	0%	0%
p0033	N4	7	0	0%	0%
p0033	N5	14	0	0%	0%
p0033	N6	14	0	0%	0%
p0201	N1	87	8	0%	0%
p0201	N2	127	8	0%	0%
p0201	N3	95	8	0%	0%
p0201	N4	89	6	0%	0%
p0201	N5	87	6	0%	0%
p0201	N6	127	8	0%	0%
p0282	N1	33	5	0%	0%
p0282	N2	35	4	0%	0%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
p0282	N3	33	4	0%	0%
p0282	N4	31	5	0%	0%
p0282	N5	35	4	0%	0%
p0282	N6	35	4	0%	0%
p0548	N1	5	1	0%	0%
p0548	N2	50	4	0%	0%
p0548	N3	9	1	0%	0%
p0548	N4	9	2	0%	0%
p0548	N5	12	2	0%	0%
p0548	N6	16	2	0%	0%
p2756	N1	39	34	0%	0%
p2756	N2	431	103	0%	0%
p2756	N3	66	44	0%	0%
p2756	N4	146	49	0%	0%
p2756	N5	59	47	0%	0%
p2756	N6	251	87	0%	0%
pk1	N1	60336	3600	0%	34.923%
pk1	N2	99208	3600	36.364%	100%
pk1	N3	52472	3600	0%	82.855%
pk1	N4	53292	3600	18.182%	100%
pk1	N5	41121	3600	9.091%	46.882%
pk1	N6	45924	3600	0%	40.443%
pp08a	N1	64136	3600	3.265%	8.274%
pp08a	N2	183600	3600	0.408%	25.737%
pp08a	N3	77151	3600	0.408%	14.066%
pp08a	N4	115009	3600	0%	19.628%
pp08a	N5	64694	3600	1.224%	8.466%
pp08a	N6	128733	3600	1.361%	11.909%
pp08aCUTS	N1	55400	3600	3.129%	8.314%
pp08aCUTS	N2	144271	3600	2.177%	27.023%
pp08aCUTS	N3	73080	3600	0%	13.221%
pp08aCUTS	N4	98183	3600	0%	19.628%
pp08aCUTS	N5	59422	3600	0.136%	8.033%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
pp08aCUTS	N6	70251	3600	1.497%	10.197%
qiu	N1	3201	3600	0%	115.106%
qiu	N2	19331	3600	151.747%	1123.264%
qiu	N3	10177	3600	0%	379.090%
qiu	N4	11213	3481	0%	0%
qiu	N5	4040	3600	0%	112.739%
qiu	N6	12235	2988	0%	0%
qnet1	N1	73	35	0%	0%
qnet1	N2	89	24	0%	0%
qnet1	N3	75	27	0%	0%
qnet1	N4	104	27	0%	0%
qnet1	N5	89	24	0%	0%
qnet1	N6	89	24	0%	0%
qnet1_o	N1	291	43	0%	0%
qnet1_o	N2	285	24	0%	0%
qnet1_o	N3	321	42	0%	0%
qnet1_o	N4	298	32	0%	0%
qnet1_o	N5	374	45	0%	0%
qnet1_o	N6	285	24	0%	0%
rentacar	N1	21	102	0%	0%
rentacar	N2	47	116	0%	0%
rentacar	N3	17	78	0%	0%
rentacar	N4	17	77	0%	0%
rentacar	N5	22	112	0%	0%
rentacar	N6	39	123	0%	0%
rgn	N1	2189	29	0%	0%
rgn	N2	2053	20	0%	0%
rgn	N3	2119	28	0%	0%
rgn	N4	2051	25	0%	0%
rgn	N5	2143	27	0%	0%
rgn	N6	2187	27	0%	0%
rout	N1	2940	3600	1.462%	5.555%
rout	N2	33620	3600	23.625%	26.294%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
rout	N3	3504	3600	8.161%	14.751%
rout	N4	4776	3600	0%	7.794%
rout	N5	4446	3600	4.422%	8.420%
rout	N6	4652	3600	3.780%	7.695%
set1ch	N1	12311	3600	(N/A)	(N/A)
set1ch	N2	64360	3600	10.826%	32.728%
set1ch	N3	11962	3600	(N/A)	(N/A)
set1ch	N4	11964	3600	5.140%	25.390%
set1ch	N5	12181	3600	8.245%	24.968%
set1ch	N6	12331	3600	7.130%	24.310%
seymour	N1	342	3600	(N/A)	(N/A)
seymour	N2	2956	3600	1.418%	5.863%
seymour	N3	307	3600	(N/A)	(N/A)
seymour	N4	271	3600	(N/A)	(N/A)
seymour	N5	300	3600	(N/A)	(N/A)
seymour	N6	333	3600	(N/A)	(N/A)
stein27	N1	3389	42	0%	0%
stein27	N2	3361	21	0%	0%
stein27	N3	3341	34	0%	0%
stein27	N4	3185	29	0%	0%
stein27	N5	3351	25	0%	0%
stein27	N6	3325	21	0%	0%
stein45	N1	48569	2837	0%	0%
stein45	N2	51188	947	0%	0%
stein45	N3	57461	1592	0%	0%
stein45	N4	49401	1068	0%	0%
stein45	N5	49609	1019	0%	0%
stein45	N6	49663	1065	0%	0%
vpml	N1	59	1	0%	0%
vpml	N2	61	1	0%	0%
vpml	N3	57	1	0%	0%
vpml	N4	91	2	0%	0%
vpml	N5	61	1	0%	0%

Problem	Selection Method	Nodes	Time	Best Sol.	Final Gap
vpm1	N6	61	1	0%	0%
vpm2	N1	9453	278	0%	0%
vpm2	N2	17199	308	0%	0%
vpm2	N3	7460	194	0%	0%
vpm2	N4	11931	312	0%	0%
vpm2	N5	9622	285	0%	0%
vpm2	N6	10279	296	0%	0%

APPENDIX B

Tables of Parallel Branch and Bound Experimental Results

Table B.1: Effect of Pseudocost Buffer Size. 16 Processors. Solved Instances.

Name	Buffer Size	Avg. Nodes	σ	Avg. Time	σ
bell3a	1	54282.7	462.00	95.7	1.75
bell3a	25	53773.7	5.77	41.3	1.73
bell3a	100	53825.0	12.17	38.4	0.35
bell3a	1000	53825.7	33.55	38.3	1.30
bell3a	No Sharing	54076.3	437.11	37.6	0.90
gesa2_o	1	115705.0	8633.62	1340.0	130.77
gesa2_o	25	102847.7	8215.13	1074.0	125.73
gesa2_o	100	103307.7	8697.08	1062.0	118.03
gesa2_o	1000	116654.0	15283.94	1230.0	183.58
gesa2_o	No Sharing	113732.3	19077.01	1097.3	190.48
misc07	1	58723.3	3348.35	299.3	16.50
misc07	25	58731.0	5663.61	244.3	20.03
misc07	100	62632.3	6059.29	260.3	26.58
misc07	1000	61199.3	642.38	258.0	7.81
misc07	No Sharing	52069.3	5413.96	217.3	25.01
p2756	1	1332.0	115.43	92.7	9.74
p2756	25	1696.7	222.42	108.1	12.95
p2756	100	1184.3	89.91	92.9	1.31
p2756	1000	1662.3	132.71	117.3	15.95
p2756	No Sharing	1478.7	66.06	103.4	7.97

Name	Buffer Size	Avg. Nodes	σ	Avg. Time	σ
stein45	1	99810.3	1712.82	305.3	9.45
stein45	25	105239.7	6235.95	186.0	19.16
stein45	100	108301.7	1393.48	186.0	3.00
stein45	1000	109804.0	4287.98	188.0	7.21
stein45	No Sharing	109422.0	2307.39	181.3	3.51
vpm2	1	120957.7	4453.20	439.3	22.05
vpm2	25	109416.0	12075.95	290.7	31.47
vpm2	100	110570.7	21649.66	271.3	54.42
vpm2	1000	103971.3	3838.95	251.0	20.88
vpm2	No Sharing	105275.7	2610.51	239.3	5.13

Table B.2: Effect of Pseudocost Buffer Size. 16 Processors. Instances Sometimes Solved.

Name	Buffer Size	Avg. Nodes	σ	Avg. Time	σ	# Unsolved
air04	1	4052.7	1632.02	2240.0	567.10	0
air04	25	4696.7	2349.32	2446.7	902.79	0
air04	100	6418.3	953.39	3400.0	275.14	1
air04	1000	5899.0	1140.74	3193.3	920.89	2
air04	No Sharing	6638.0	137.54	3693.3	5.77	3
air05	1	10432.7	788.80	3403.3	330.81	1
air05	25	3016.0	209.52	2203.3	20.82	0
air05	100	3959.7	1279.85	2940.0	220.68	0
air05	1000	4608.7	1021.12	3170.0	380.00	1
air05	No Sharing	5560.7	803.94	3610.0	115.33	3

Table B.3: Effect of Pseudocost Buffer Size. 16 Processors. Unsolved Instances.

Name	Buffer Size	Avg. Nodes	σ	Avg. Final Gap (%)	σ
mod011	1	4125.0	253.70	3.6804	0.5139
mod011	25	4405.0	203.66	3.4587	0.4408
mod011	100	4170.3	156.47	4.5445	0.4706
mod011	1000	4162.7	73.20	4.7657	0.3339
mod011	No Sharing	4206.3	111.63	4.8180	0.3078
pp08a	1	656477.3	6162.22	3.3804	0.6239
pp08a	25	874054.0	16123.36	3.6295	0.7440
pp08a	100	799230.7	60151.06	3.1045	0.3586
pp08a	1000	837635.3	40252.80	2.8982	0.2249
pp08a	No Sharing	924975.7	93105.09	3.9244	1.0106
rout	1	93473.0	6785.86	3.9998	0.9607
rout	25	55246.7	9143.63	4.9089	0.5577
rout	100	57347.7	4175.21	6.1280	0.5051
rout	1000	63222.3	12775.98	4.8155	0.3919
rout	No Sharing	101311.0	12161.81	3.4198	0.4549

Table B.4: Comparison of Parallel Pseudocost Initialization with Other Schemes. 16 Processors. Solved Instances.

Name	Scheme	Avg. Nodes	σ	Avg. Time	σ
bell3a	Broadcast	54282.7	462.00	95.7	1.75
bell3a	Parallel Init.	55801.0	3342.91	39.0	4.23
bell3a	No Sharing	54076.3	437.11	37.6	0.90
gesa2_o	Broadcast	115705.0	8633.62	1340.0	130.77
gesa2_o	Parallel Init.	169348.3	31621.44	1516.7	299.56
gesa2_o	No Sharing	113732.3	19077.01	1097.3	190.48
misc07	Broadcast	58723.3	3348.35	299.3	16.50
misc07	Parallel Init.	61426.7	7601.09	242.0	31.58
misc07	No Sharing	52069.3	5413.96	217.3	25.01
p2756	Broadcast	1332.0	115.43	92.7	9.74
p2756	Parallel Init.	1730.0	97.25	108.3	6.35
p2756	No Sharing	1478.7	66.06	103.4	7.97
stein45	Broadcast	99810.3	1712.82	305.3	9.45
stein45	Parallel Init.	109203.7	1523.43	175.0	11.27
stein45	No Sharing	109422.0	2307.39	181.3	3.51
vpm2	Broadcast	120957.7	4453.20	439.3	22.05
vpm2	Parallel Init.	141756.3	20100.10	320.3	41.28
vpm2	No Sharing	105275.7	2610.51	239.3	5.13

Table B.5: Comparison of Parallel Pseudocost Initialization with Other Schemes. 16 Processors. Instances Sometimes Solved.

Name	Scheme	Avg. Nodes	σ	Avg. Time	σ	# Unsolved
air04	Broadcast	4052.7	1632.02	2240.0	567.10	0
air04	Parallel Init.	5496.3	1223.17	3313.3	747.82	2
air04	No Sharing	6638.0	137.54	3693.3	5.77	3
air05	Broadcast	10432.7	788.80	3403.3	330.81	1
air05	Parallel Init.	8072.0	2603.23	3196.7	847.01	2
air05	No Sharing	5560.7	803.94	3610.0	115.33	3

Table B.6: Comparison of Parallel Pseudocost Initialization with Other Schemes. 16 Processors. Unsolved Instances.

Name	Scheme	Avg. Nodes	σ	Avg. Final Gap (%)	σ
mod011	Broadcast	4125.0	253.70	3.6804	0.5139
mod011	Parallel Init.	4385.0	101.39	5.2189	0.0650
mod011	No Sharing	4206.3	111.63	4.8180	0.3078
pp08a	Broadcast	656477.3	6162.22	3.3804	0.6239
pp08a	Parallel Init.	848191.3	71015.90	4.7087	0.5222
pp08a	No Sharing	924975.7	93105.09	3.9244	1.0106
rout	Broadcast	93473.0	6785.86	3.9998	0.9607
rout	Parallel Init.	147245.7	6273.51	1.2095	0.0828
rout	No Sharing	101311.0	12161.81	3.4198	0.4549

Table B.7: Comparison of Parallel Node Selection Methods. 8 Processors.

Name	Scheme	Avg. Nodes	σ	Gap Opt. (%)	Gap Prove (%)	Avg. Time
air04	Backtracking	4113.7	454.48	0.0000	0.0867	3490.00
air04	Best Bound	3044.5	38.89	0.0000	0.3899	3715.00
air04	Best Estimate	2915.7	211.77	0.0006	0.7622	3700.00
air04	Plunging	2109.7	1613.94	0.0166	0.6500	3906.67
air05	Backtracking	3242.0	1116.35	0.0000	0.0000	3080.00
air05	Best Bound	2856.7	324.17	0.2073	0.5849	3710.00
air05	Best Estimate	2244.3	550.65	0.0000	0.0000	2693.33
air05	Plunging	2362.3	442.32	0.0000	0.0000	3093.33
bell5	Backtracking	41995.3	14825.60	0.0000	0.0000	36.83
bell5	Best Bound	30486.7	23632.12	0.0000	0.0000	30.93
bell5	Best Estimate	10004.0	374.73	0.0000	0.0000	14.60
bell5	Plunging	19636.7	9510.53	0.0000	0.0000	18.40
gesa2_o	Backtracking	95269.0	12420.61	0.0000	0.0000	1500.00
gesa2_o	Best Bound	104130.0	6602.81	0.0000	0.0000	1910.00
gesa2_o	Best Estimate	75956.0	4524.08	0.0000	0.0000	1183.33
gesa2_o	Plunging	105986.3	13421.22	0.0000	0.0000	1673.33
misc07	Backtracking	56233.3	7218.87	0.0000	0.0000	368.33
misc07	Best Bound	62568.7	2347.86	0.0000	0.0000	538.67
misc07	Best Estimate	57903.3	10670.94	0.0000	0.0000	427.00
misc07	Plunging	56328.7	6519.24	0.0000	0.0000	373.67
p2756	Backtracking	2669.7	180.10	0.0000	0.0000	292.00
p2756	Best Bound	798.0	145.66	0.0000	0.0000	102.70
p2756	Best Estimate	545.7	146.60	0.0000	0.0000	63.47
p2756	Plunging	3137.0	928.75	0.0000	0.0000	345.33
qiu	Backtracking	82919.0	5412.85	0.0000	32.3154	3720.00
qiu	Best Bound	40662.0	1891.33	0.0000	6.3540	3370.00
qiu	Best Estimate	38799.3	4037.60	0.0000	185.1317	3720.00
qiu	Plunging	67196.3	10537.50	0.0000	16.7628	3436.67
rout	Backtracking	99453.3	1316.91	0.0000	3.6304	3720.00
rout	Best Bound	49571.7	998.54	0.0000	2.8228	3720.00
rout	Best Estimate	105489.3	4846.21	0.0000	8.4083	3720.00

Name	Scheme	Avg. Nodes	σ	Gap Opt. (%)	Gap Prove (%)	Avg. Time
rout	Plunging	115647.0	5157.98	0.0000	3.4011	3720.00
set1ch	Backtracking	258334.0	483.52	5.8482	20.1441	3720.00
set1ch	Best Bound	205178.3	3173.51	17.0950	25.9647	3720.00
set1ch	Best Estimate	183274.0	3084.34	16.8713	33.2503	3720.00
set1ch	Plunging	258196.0	71.36	5.8482	19.7862	3720.00
vpm2	Backtracking	153563.3	19239.64	0.0000	0.0000	740.00
vpm2	Best Bound	114403.7	7527.95	0.0000	0.0000	589.67
vpm2	Best Estimate	121597.3	16279.82	0.0000	0.0000	604.33
vpm2	Plunging	142360.0	13375.86	0.0000	0.0000	639.67

Table B.8: Comparison of Parallel Node Selection Methods. 16 Processors.

Name	Scheme	Avg. Nodes	σ	Gap Opt. (%)	Gap Prove (%)	Avg. Time
air04	Backtracking	7145.7	1618.10	0.0000	0.0238	3283.33
air04	Best Bound	5898.7	2205.68	0.0000	0.2612	3710.00
air04	Best Estimate	7699.0	838.63	0.0000	0.4591	3730.00
air04	Plunging	3075.0	379.01	0.0009	0.9238	3720.00
air05	Backtracking	3866.0	774.21	0.0000	0.0000	2960.00
air05	Best Bound	4443.7	1403.21	0.0000	0.0000	2996.67
air05	Best Estimate	2552.3	385.92	0.0000	0.0000	2176.67
air05	Plunging	3315.7	624.76	0.0000	0.0000	3246.67
bell5	Backtracking	275658.3	409566.63	0.0000	0.0000	129.63
bell5	Best Bound	12241.3	2216.46	0.0000	0.0000	14.50
bell5	Best Estimate	21535.0	20659.97	0.0000	0.0000	22.53
bell5	Plunging	1004473.3	1464005.62	0.0000	0.0000	624.10
gesa2_o	Backtracking	114730.0	917.89	0.0000	0.0000	936.67
gesa2_o	Best Bound	107002.3	10321.36	0.0000	0.0000	1013.00
gesa2_o	Best Estimate	93513.0	3353.02	0.0000	0.0000	737.67
gesa2_o	Plunging	119871.0	8235.34	0.0000	0.0000	908.67
misc07	Backtracking	57305.0	2131.54	0.0000	0.0000	178.00
misc07	Best Bound	55988.7	3566.88	0.0000	0.0000	218.67
misc07	Best Estimate	59517.7	6682.62	0.0000	0.0000	200.67
misc07	Plunging	55301.0	4568.72	0.0000	0.0000	172.33
p2756	Backtracking	3310.0	247.18	0.0000	0.0000	191.67
p2756	Best Bound	1418.0	161.93	0.0000	0.0000	91.23
p2756	Best Estimate	720.7	58.07	0.0000	0.0000	46.73
p2756	Plunging	3295.3	43.89	0.0000	0.0000	178.33
qiu	Backtracking	74834.3	2367.69	0.0000	0.0000	1730.00
qiu	Best Bound	87292.0	22942.79	0.0000	3.5149	3120.00
qiu	Best Estimate	48010.3	4542.55	0.0000	0.0000	1720.00
qiu	Plunging	79021.0	8053.18	0.0000	0.0000	1810.00
rout	Backtracking	232044.0	1617.04	0.0000	3.1409	3720.00
rout	Best Bound	111172.3	7368.18	0.0000	2.6811	3720.00
rout	Best Estimate	205567.3	14852.06	0.0000	2.8427	3350.00

Name	Scheme	Avg. Nodes	σ	Gap Opt. (%)	Gap Prove (%)	Avg. Time
rout	Plunging	252556.3	2589.20	0.0000	4.5082	3700.00
set1ch	Backtracking	532986.7	2890.72	4.9829	19.4318	3720.00
set1ch	Best Bound	424050.0	3761.02	15.1839	24.3018	3720.00
set1ch	Best Estimate	383164.3	1982.36	15.5174	32.3624	3720.00
set1ch	Plunging	543723.0	11537.14	5.5695	19.4457	3720.00
vpm2	Backtracking	133410.7	9336.68	0.0000	0.0000	272.00
vpm2	Best Bound	98748.7	8802.14	0.0000	0.0000	219.00
vpm2	Best Estimate	119639.3	19769.35	0.0000	0.0000	265.00
vpm2	Plunging	144891.0	22666.49	0.0000	0.0000	309.67

APPENDIX C

Tables of Sequential Branch and Cut Experimental Results

If no solution was found, then in the “Final Gap” column, the best upper bound (in italics) is written.

Table C.1: Effect of Varying the Skip Factor

Name	κ	Final Gap	Nodes	Time
blend2	1	0.000%	11400	722.6
blend2	5	0.000%	8244	357.6
blend2	10	0.000%	6044	212.3
blend2	100	0.000%	11971	298.9
blend2	1000	0.000%	11474	254.8
blend2	∞	0.000%	11515	261.6
cap6000	1	0.002%	14001	3600.0
cap6000	5	0.000%	15104	3028.3
cap6000	10	0.000%	16619	2969.4
cap6000	100	0.000%	18453	2697.1
cap6000	1000	0.000%	19456	2780.6
cap6000	∞	0.000%	19740	2813.0
fiber	1	0.000%	201	35.1
fiber	5	0.000%	221	32.0
fiber	10	0.000%	543	50.1
fiber	100	0.000%	791	55.6
fiber	1000	0.000%	871	53.8

Name	κ	Final Gap	Nodes	Time
fiber	∞	0.000%	871	53.8
harp2	1	0.390%	18576	3600.0
harp2	5	0.410%	32563	3600.0
harp2	10	0.510%	30852	3600.0
harp2	100	0.380%	44363	3600.0
harp2	1000	0.420%	23547	3600.0
harp2	∞	0.430%	34589	3600.1
l152lav	1	0.000%	603	139.9
l152lav	5	0.000%	745	156.2
l152lav	10	0.000%	743	154.4
l152lav	100	0.000%	647	133.8
l152lav	1000	0.000%	647	133.9
l152lav	∞	0.000%	647	133.8
mitre	1	<i>-114791</i>	221	3600.0
mitre	5	<i>-114791</i>	285	3600.0
mitre	10	<i>-114791</i>	328	3600.0
mitre	100	<i>-114791</i>	397	3600.0
mitre	1000	<i>-114791</i>	398	3600.0
mitre	∞	<i>-114791</i>	399	3600.0
mod011	1	11.900%	652	3600.0
mod011	5	12.160%	671	3600.0
mod011	10	12.060%	698	3600.0
mod011	100	13.880%	801	3600.0
mod011	1000	13.700%	815	3600.0
mod011	∞	13.230%	891	3600.0
modglob	1	0.000%	1411	107.5
modglob	5	0.000%	1649	107.2
modglob	10	0.000%	2445	144.2
modglob	100	0.000%	1842	94.6
modglob	1000	0.000%	1897	92.4
modglob	∞	0.000%	1681	84.7
p2756	1	0.000%	7007	2718.0
p2756	5	0.000%	12697	3004.4

Name	κ	Final Gap	Nodes	Time
p2756	10	0.000%	9771	1714.9
p2756	100	0.000%	11012	1235.2
p2756	1000	0.000%	18485	1730.0
p2756	∞	0.000%	26665	2288.5
vpm2	1	0.000%	8752	189.6
vpm2	5	0.000%	11604	232.1
vpm2	10	0.000%	17377	351.3
vpm2	100	0.000%	17295	309.6
vpm2	1000	0.000%	21014	363.4
vpm2	∞	0.000%	22042	369.2

Table C.2: Effect of Adding Multiple Cuts Per Round

Name	Maximum Cuts Per Round	Final Gap	Nodes	Time
blend2	1	0.000%	9716	258.1
blend2	10	0.000%	5914	212.0
blend2	20	0.000%	6044	212.0
blend2	50	0.000%	6044	213.0
blend2	∞	0.000%	6044	212.1
cap6000	1	0.000%	13780	2504.5
cap6000	10	0.000%	16619	2971.1
cap6000	20	0.000%	16619	2968.7
cap6000	50	0.000%	16619	2972.6
cap6000	∞	0.000%	16619	2974.8
fiber	1	0.000%	657	52.6
fiber	10	0.000%	371	41.0
fiber	20	0.000%	543	50.3
fiber	50	0.000%	489	51.3
fiber	∞	0.000%	489	51.0
harp2	1	0.470%	13383	3600.0
harp2	10	0.360%	46056	3600.0
harp2	20	0.510%	30851	3600.0
harp2	50	0.510%	30778	3600.0
harp2	∞	0.510%	30751	3600.0
l152lav	1	0.000%	743	155.2
l152lav	10	0.000%	743	154.5
l152lav	20	0.000%	743	154.3
l152lav	50	0.000%	743	154.5
l152lav	∞	0.000%	743	154.9
mitre	1	<i>-114794</i>	1	3600.0
mitre	10	<i>-114853</i>	275	3600.0
mitre	20	<i>-114791</i>	328	3600.0
mitre	50	<i>-114791</i>	321	3600.0
mitre	∞	<i>-114791</i>	343	3600.0
mod011	1	11.670%	700	3600.0
mod011	10	12.020%	734	3600.0

Name	Maximum Cuts Per Round	Final Gap	Nodes	Time
mod011	20	12.060%	678	3600.0
mod011	50	12.060%	653	3600.0
mod011	∞	12.060%	698	3600.0
modglob	1	0.000%	1766	76.5
modglob	10	0.000%	1601	92.6
modglob	20	0.000%	2445	144.3
modglob	50	0.000%	1303	79.9
modglob	∞	0.000%	849	68.6
p2756	1	0.000%	7680	1188.4
p2756	10	0.000%	7071	1157.0
p2756	20	0.000%	10527	1892.4
p2756	50	0.000%	8404	1429.0
p2756	∞	0.000%	13984	2343.6
vpm2	1	0.000%	10375	201.0
vpm2	10	0.000%	12065	243.4
vpm2	20	0.000%	17377	351.3
vpm2	50	0.000%	18499	362.6
vpm2	∞	0.000%	18499	362.6

Table C.3: Effect of Varying the Tailing Off Percentage

Name	α	Final Gap	Nodes	Time
blend2	0.0%	0.000%	4490	228.1
blend2	0.1%	0.000%	7194	242.1
blend2	0.2%	0.000%	5844	213.1
blend2	0.5%	0.000%	6044	212.7
blend2	1.0%	0.000%	5895	209.3
blend2	2.0%	0.000%	9340	346.2
cap6000	0.0%	0.002%	19933	3600.0
cap6000	0.1%	0.000%	16619	2974.5
cap6000	0.2%	0.000%	16619	2974.0
cap6000	0.5%	0.000%	16619	2975.0
cap6000	1.0%	0.000%	16619	2974.3
cap6000	2.0%	0.000%	16619	2972.9
fiber	0.0%	0.000%	489	51.3
fiber	0.1%	0.000%	489	51.0
fiber	0.2%	0.000%	489	51.1
fiber	0.5%	0.000%	489	51.1
fiber	1.0%	0.000%	489	51.3
fiber	2.0%	0.000%	489	51.0
harp2	0.0%	0.310%	44802	3600.0
harp2	0.1%	0.380%	45925	3600.0
harp2	0.2%	0.370%	21542	3600.0
harp2	0.5%	0.510%	30761	3600.0
harp2	1.0%	0.510%	30750	3600.0
harp2	2.0%	0.510%	30751	3600.0
l152lav	0.0%	0.000%	743	154.7
l152lav	0.1%	0.000%	743	154.8
l152lav	0.2%	0.000%	743	154.5
l152lav	0.5%	0.000%	743	154.6
l152lav	1.0%	0.000%	743	154.6
l152lav	2.0%	0.000%	743	155.6
mitre	0.0%	<i>-114791</i>	343	3600.0
mitre	0.1%	<i>-114791</i>	343	3600.0

Name	α	Final Gap	Nodes	Time
mitre	0.2%	-114791	343	3600.0
mitre	0.5%	-114791	343	3600.0
mitre	1.0%	-114791	343	3600.0
mitre	2.0%	-114791	343	3600.0
mod011	0.0%	12.110%	681	3600.0
mod011	0.1%	12.060%	697	3600.0
mod011	0.2%	12.060%	696	3600.0
mod011	0.5%	12.060%	677	3600.0
mod011	1.0%	12.110%	681	3600.0
mod011	2.0%	12.060%	697	3600.0
modglob	0.0%	0.000%	1926	109.5
modglob	0.1%	0.000%	1862	132.4
modglob	0.2%	0.000%	915	67.7
modglob	0.5%	0.000%	849	68.8
modglob	1.0%	0.000%	849	68.8
modglob	2.0%	0.000%	972	78.1
p2756	0.0%	0.000%	5684	961.8
p2756	0.1%	0.000%	8471	1472.0
p2756	0.2%	0.000%	7578	1297.9
p2756	0.5%	0.000%	14259	2403.9
p2756	1.0%	0.000%	6731	1081.8
p2756	2.0%	0.000%	6414	1028.7
vpm2	0.0%	0.000%	16402	329.0
vpm2	0.1%	0.000%	18499	362.8
vpm2	0.2%	0.000%	18499	362.6
vpm2	0.5%	0.000%	18499	362.2
vpm2	1.0%	0.000%	18499	361.9
vpm2	2.0%	0.000%	18499	363.0

Table C.4: Effect of Deleting Weak Cuts

Name	η	Final Gap	Nodes	Time
blend2	1	0.000%	5859	571.4
blend2	25	0.000%	8839	843.6
blend2	50	0.000%	5165	598.2
blend2	100	0.000%	5293	683.7
blend2	Never	0.000%	5371	859.3
cap6000	1	0.000%	18297	3087.7
cap6000	25	0.000%	16959	2975.2
cap6000	50	0.000%	16619	2971.9
cap6000	100	0.000%	16253	3036.0
cap6000	Never	0.002%	15576	3600.0
fiber	1	0.000%	379	38.8
fiber	25	0.000%	489	51.1
fiber	50	0.000%	489	51.0
fiber	100	0.000%	489	51.3
fiber	Never	0.000%	489	51.1
harp2	1	0.490%	21923	3600.0
harp2	25	0.510%	30782	3600.0
harp2	50	0.510%	30832	3600.0
harp2	100	0.510%	25170	3600.0
harp2	Never	0.530%	14683	3600.0
l152lav	1	0.000%	757	153.9
l152lav	25	0.000%	743	154.5
l152lav	50	0.000%	743	154.6
l152lav	100	0.000%	743	154.5
l152lav	Never	0.000%	743	154.4
mitre	1	<i>-114793</i>	1	3600.0
mitre	25	<i>-114853</i>	250	3600.0
mitre	50	<i>-114791</i>	326	3600.0
mitre	100	<i>-114791</i>	326	3600.0
mitre	Never	<i>-114791</i>	326	3600.0
mod011	1	10.920%	658	3600.0
mod011	25	11.860%	691	3600.0

Name	η	Final Gap	Nodes	Time
mod011	50	12.060%	653	3600.0
mod011	100	12.110%	709	3600.0
mod011	Never	12.060%	694	3600.0
modglob	1	0.000%	2173	93.4
modglob	25	0.000%	1823	105.4
modglob	50	0.000%	2445	144.5
modglob	100	0.000%	2143	130.3
modglob	Never	0.000%	2097	127.5
p2756	1	0.000%	7477	1131.6
p2756	25	0.000%	9729	1637.6
p2756	50	0.000%	9771	1716.2
p2756	100	0.000%	9279	1749.5
p2756	Never	0.000%	8547	1645.4
vpm2	1	0.000%	9783	186.2
vpm2	25	0.000%	26880	506.9
vpm2	50	0.000%	17377	351.4
vpm2	100	0.000%	17457	357.8
vpm2	Never	0.000%	17775	377.5

APPENDIX D

Tables of Parallel Branch and Cut Experimental Results

Table D.1: Comparison of Basic Cut Sharing Strategies. 4 Processors. Solved Instances.

Instance	Sharing Strategy	Avg. Nodes	σ	Avg. Time	σ
blend2	Prefetch	2758.0	654.77	72.4	17.69
blend2	Fetch	2808.7	302.86	73.2	7.60
blend2	No Sharing	2421.0	333.22	66.0	8.97
gesa2_o	Prefetch	85592.0	2479.68	2593.3	41.63
gesa2_o	Fetch	94631.0	6207.68	2510.0	170.59
gesa2_o	No Sharing	94704.7	4398.80	2830.0	266.27
gesa3_o	Prefetch	2169.7	112.53	89.7	3.10
gesa3_o	Fetch	1982.7	430.54	85.0	14.48
gesa3_o	No Sharing	1930.7	238.30	83.6	5.46
l152lav	Prefetch	2078.7	87.37	91.2	7.04
l152lav	Fetch	2267.7	225.02	92.7	9.32
l152lav	No Sharing	2176.0	521.44	90.1	19.27
p2756	Prefetch	443.3	112.88	97.2	24.23
p2756	Fetch	412.3	80.21	99.9	15.16
p2756	No Sharing	490.7	39.72	100.1	5.80
vpm2	Prefetch	118251.7	19350.70	1316.7	332.47
vpm2	Fetch	126865.3	35531.83	1375.0	486.39
vpm2	No Sharing	117042.3	23323.98	1098.7	266.88

Table D.2: Comparison of Basic Cut Sharing Strategies. 4 Processors. Unsolved Instances.

Instance	Sharing Strategy	Avg. Nodes	σ	Avg. Final Gap (%)	σ
cap6000	Prefetch	1665.7	1126.42	0.0015	0.0002
cap6000	Fetch	3071.7	49.44	0.0009	0.0000
cap6000	No Sharing	2924.0	8.19	0.0009	0.0000
harp2	Prefetch	17168.7	7598.46	0.0601	0.0184
harp2	Fetch	8594.0	5724.30	0.1091	0.0299
harp2	No Sharing	12853.7	3684.82	0.0766	0.0288
mitre	Prefetch	410.3	79.20	0.3126	0.0000
mitre	Fetch	448.0	69.31	0.3126	0.0000
mitre	No Sharing	288.7	43.00	0.3138	0.0020
mod011	Prefetch	1202.3	31.82	5.0163	0.1659
mod011	Fetch	1166.0	36.39	4.9432	0.0242
mod011	No Sharing	1221.0	23.64	5.1856	0.4313
pp08a	Prefetch	169346.7	20319.44	5.3843	1.9073
pp08a	Fetch	179116.0	9601.15	4.4132	0.1409
pp08a	No Sharing	178679.3	5061.53	4.6269	0.1235
set1ch	Prefetch	114646.3	116.23	13.1878	0.0029
set1ch	Fetch	114360.7	163.86	13.3094	0.1064
set1ch	No Sharing	114125.7	84.68	13.2550	0.1094

Table D.3: Comparison of Basic Cut Sharing Strategies. 8 Processors. Solved Instances.

Instance	Sharing Strategy	Avg. Nodes	σ	Avg. Time	σ
blend2	Prefetch	2791.3	419.18	33.3	3.42
blend2	Fetch	3409.0	495.70	46.8	3.41
blend2	No Sharing	3024.3	260.75	36.5	2.63
gesa2_o	Prefetch	115727.0	36488.13	1346.7	332.47
gesa2_o	Fetch	122176.7	25104.96	1406.7	267.64
gesa2_o	No Sharing	101821.3	11513.45	1306.7	203.06
gesa3_o	Prefetch	2033.7	248.04	45.5	5.35
gesa3_o	Fetch	2073.3	632.53	48.9	13.75
gesa3_o	No Sharing	1940.0	344.71	45.4	4.80
l152lav	Prefetch	2099.3	1013.19	42.4	13.57
l152lav	Fetch	1433.7	378.00	34.2	5.22
l152lav	No Sharing	2879.0	1011.09	54.5	16.28
p2756	Prefetch	780.0	160.10	79.8	10.94
p2756	Fetch	789.7	197.44	94.3	17.79
p2756	No Sharing	847.3	56.08	71.7	4.22
vpm2	Prefetch	102100.3	3046.30	188.3	10.50
vpm2	Fetch	104535.7	10142.50	199.7	9.07
vpm2	No Sharing	115596.0	11108.92	197.0	23.64

Table D.4: Comparison of Basic Cut Sharing Strategies. 8 Processors. Unsolved Instances.

Instance	Sharing Strategy	Avg. Nodes	σ	Avg. Final Gap (%)	σ
cap6000	Prefetch	1483.7	457.88	0.0013	0.0000
cap6000	Fetch	5153.7	3048.26	0.0008	0.0005
cap6000	No Sharing	6174.7	429.07	0.0008	0.0002
harp2	Prefetch	67996.0	4520.07	0.0332	0.0065
harp2	Fetch	74658.0	7032.93	0.0274	0.0038
harp2	No Sharing	83508.5	18603.27	0.0405	0.0218
mitre	Prefetch	1274.7	113.81	0.2973	0.0058
mitre	Fetch	1022.7	175.15	0.3068	0.0066
mitre	No Sharing	561.7	104.99	0.2993	0.0048
mod011	Prefetch	2850.0	52.37	3.5467	0.0414
mod011	Fetch	2873.7	100.38	3.7338	0.2946
mod011	No Sharing	2906.3	30.11	4.6655	0.1481
pp08a	Prefetch	461822.0	23766.82	4.1377	0.7560
pp08a	Fetch	522654.7	62272.84	3.5827	0.3011
pp08a	No Sharing	463007.0	2293.21	4.0906	0.7949
set1ch	Prefetch	261757.7	314.07	13.0208	0.0405
set1ch	Fetch	260955.3	1079.48	13.0618	0.1274
set1ch	No Sharing	261189.3	144.22	13.0101	0.1013

Table D.5: Comparison of Basic Cut Sharing Strategies. 16 Processors. Solved Instances.

Instance	Sharing Strategy	Avg. Nodes	σ	Avg. Time	σ
blend2	Prefetch	3194.3	1809.78	28.6	2.84
blend2	Fetch	3868.7	800.19	61.2	9.34
blend2	No Sharing	4017.0	1176.47	27.9	5.55
gesa2_o	Prefetch	122577.3	4461.58	659.3	30.44
gesa2_o	Fetch	124902.3	1727.01	700.0	11.27
gesa2_o	No Sharing	112281.7	15438.61	668.7	106.25
gesa3_o	Prefetch	3413.3	1355.37	46.9	15.71
gesa3_o	Fetch	3035.7	1148.94	43.2	11.91
gesa3_o	No Sharing	2965.0	870.06	41.4	9.18
l152lav	Prefetch	2237.3	121.17	31.8	2.16
l152lav	Fetch	2513.3	761.95	40.8	2.95
l152lav	No Sharing	2280.0	736.18	30.6	4.47
p2756	Prefetch	1235.0	447.34	71.2	21.03
p2756	Fetch	1205.0	251.99	82.0	16.71
p2756	No Sharing	1463.7	304.80	73.7	11.41
vpm2	Prefetch	102100.3	3046.30	188.3	10.50
vpm2	Fetch	104535.7	10142.50	199.7	9.07
vpm2	No Sharing	115596.0	11108.92	197.0	23.64

Table D.6: Comparison of Basic Cut Sharing Strategies. 16 Processors. Unsolved Instances.

Instance	Sharing Strategy	Avg. Nodes	σ	Avg. Final Gap (%)	σ
cap6000	Prefetch	3254.3	1395.43	0.0015	0.0002
cap6000	Fetch	6501.7	703.91	0.0009	0.0000
cap6000	No Sharing	16408.7	1166.40	0.0001	0.0000
harp2	Prefetch	207279.7	29948.01	0.0069	0.0068
harp2	Fetch	200300.3	9505.26	0.0002	0.0003
harp2	No Sharing	268192.0	26200.66	0.0023	0.0033
mitre	Prefetch	2662.0	242.28	0.2672	0.0124
mitre	Fetch	2686.0	197.49	0.2686	0.0050
mitre	No Sharing	1082.0	119.78	0.2880	0.0208
mod011	Prefetch	6421.7	238.89	3.2300	0.0850
mod011	Fetch	6394.7	108.91	3.1753	0.5362
mod011	No Sharing	6705.0	130.37	3.2269	0.1167
pp08a	Prefetch	1283496.0	127941.07	4.0382	0.8189
pp08a	Fetch	1149565.3	178986.88	3.2897	1.2173
pp08a	No Sharing	1215627.7	82221.81	3.6662	0.7022
set1ch	Prefetch	543431.7	4528.68	12.5389	0.2326
set1ch	Fetch	540535.3	2939.87	12.9493	0.1270
set1ch	No Sharing	538746.3	3488.83	12.7455	0.1727

Table D.7: Comparison of Prefetch Sharing Schemes. 16 Processors. Solved Instances.

Instance	Sharing Strategy	Avg. Nodes	σ	Avg. Time	σ
blend2	Broadcast	3194.3	1809.78	28.6	2.84
blend2	Random	3651.0	1144.37	32.4	5.57
blend2	Selective	2347.0	342.53	28.1	2.91
gesa2_o	Broadcast	122577.3	4461.58	659.3	30.44
gesa2_o	Random	128985.7	9417.94	766.7	32.62
gesa2_o	Selective	113554.7	16258.24	721.3	144.13
gesa3_o	Broadcast	3413.3	1355.37	46.9	15.71
gesa3_o	Random	2457.7	884.67	37.8	8.21
gesa3_o	Selective	4351.7	715.79	56.9	7.75
l152lav	Broadcast	2237.3	121.17	31.8	2.16
l152lav	Random	1914.7	413.58	29.4	1.37
l152lav	Selective	2089.7	385.76	30.6	3.20
p2756	Broadcast	1235.0	447.34	71.2	21.03
p2756	Random	1286.7	250.63	78.1	16.43
p2756	Selective	1737.7	409.15	98.8	13.10
vpm2	Broadcast	102100.3	3046.30	188.3	10.50
vpm2	Random	112167.3	6590.28	216.7	20.65
vpm2	Selective	104165.0	1968.41	173.7	8.62

Table D.8: Comparison of Prefetch Sharing Schemes. 16 Processors. Unsolved Instances.

Instance	Sharing Strategy	Avg. Nodes	σ	Avg. Final Gap (%)	σ
cap6000	Broadcast	3254.3	1395.43	0.0015	0.0002
cap6000	Random	7223.7	3863.39	0.0012	0.0002
cap6000	Selective	5427.0	548.23	0.0004	0.0002
harp2	Broadcast	207279.7	29948.01	0.0069	0.0068
harp2	Random	157895.0	11751.11	0.0134	0.0226
harp2	Selective	199354.7	8840.43	0.0017	0.0026
mitre	Broadcast	2662.0	242.28	0.2672	0.0124
mitre	Random	3308.7	320.51	0.2770	0.0159
mitre	Selective	1148.0	216.96	0.3005	0.0071
mod011	Broadcast	6421.7	238.89	3.2300	0.0850
mod011	Random	6609.0	147.08	3.9699	0.0040
mod011	Selective	6436.7	40.08	3.5058	0.1780
pp08a	Broadcast	1283496.0	127941.07	4.0382	0.8189
pp08a	Random	1279385.0	168221.67	2.4199	0.8216
pp08a	Selective	1240868.7	25049.58	2.2802	0.3247
set1ch	Broadcast	543431.7	4528.68	12.5389	0.2326
set1ch	Random	544980.0	3505.37	12.5752	0.2562
set1ch	Selective	543178.3	3581.62	12.5131	0.1135

Table D.9: Effect of Varying Skip Factor in Master-Worker Sharing Scheme. 16 Processors. Solved Instances

Name	κ	Avg. Nodes	σ	Avg. Time	σ
blend2	1	3868.7	800.19	61.2	9.34
blend2	10	3677.7	924.45	15.2	5.09
blend2	100	3783.3	1198.72	10.6	3.90
gesa2_o	1	124902.3	1727.01	700.0	11.27
gesa2_o	10	229436.0	17602.45	1340.0	36.06
gesa2_o	100	224048.0	75371.66	1041.0	376.32
gesa3_o	1	3035.7	1148.94	43.2	11.91
gesa3_o	10	5377.3	883.72	52.9	5.05
gesa3_o	100	9632.0	1541.91	64.4	6.36
l152lav	1	2513.3	761.95	40.8	2.95
l152lav	10	2505.3	459.11	37.6	13.77
l152lav	100	3658.7	1661.73	35.5	6.59
p2756	1	1205.0	251.99	82.0	16.71
p2756	10	4070.7	405.20	124.3	11.02
p2756	100	8611.0	2283.23	92.6	12.79
vpm2	1	104535.7	10142.50	199.7	9.07
vpm2	10	150180.7	2198.35	176.0	2.65
vpm2	100	203117.0	7384.97	183.7	10.60

Table D.10: Effect of Varying Skip Factor in Master-Worker Sharing Scheme. 16 Processors. Instances Sometimes Solved.

Name	κ	Avg. Nodes	σ	Avg.Time	σ	# Unsolved
cap6000	1	6501.7	703.91	3620.0	0.0000	3
cap6000	10	26325.3	1926.69	1380.0	147.31	0
cap6000	100	32521.3	1994.46	621.3	150.19	0
harp2	1	200300.3	9505.26	3720.0	0.00	3
harp2	10	550940.3	167981.93	3393.3	436.62	1
harp2	100	653152.5	3655.03	3720.0	0.00	3

Table D.11: Effect of Varting Skip Factor in Master-Worker Sharing Scheme. 16 Processors. Unsolved Instances.

Name	κ	Avg. Nodes	σ	Avg. Final Gap (%)	σ
mitre	1	2686.0	197.49	0.2686	0.0050
mitre	10	5884.3	187.69	0.2455	0.0053
mitre	100	37056.3	1425.58	0.2391	0.0079
mod011	1	6394.7	108.91	3.1753	0.5362
mod011	10	6901.7	143.06	4.9354	0.4448
mod011	100	10661.7	268.99	5.0480	0.2830
pp08a	1	1149565.3	178986.88	3.2897	1.2173
pp08a	10	1260564.7	44195.18	5.0910	1.0947
pp08a	100	1272228.0	29938.73	6.3176	0.9911
set1ch	1	540535.3	2939.87	12.9493	0.1270
set1ch	10	508692.0	7335.50	13.5242	0.2331
set1ch	100	590135.3	6284.75	14.0147	0.1516

Table D.12: Effect of Varying Skip Factor in Broadcast-Distributed Sharing Scheme. 16 Processors. Solved Instances.

Name	κ	Avg. Nodes	σ	Avg. Time	σ
blend2	1	3194.3	1809.78	28.6	2.84
blend2	10	4291.3	1320.05	18.7	6.58
blend2	100	5109.0	2313.60	10.9	1.99
gesa2_o	1	122577.3	4461.58	659.3	30.44
gesa2_o	10	227573.3	25230.31	1290.0	183.30
gesa2_o	100	254289.3	29357.97	1276.7	228.11
gesa3_o	1	3413.3	1355.37	46.9	15.71
gesa3_o	10	6683.0	3200.34	61.0	13.14
gesa3_o	100	11156.0	4190.60	73.4	22.66
l152lav	1	2237.3	121.17	31.8	2.16
l152lav	10	2667.7	690.82	32.8	5.13
l152lav	100	2412.3	830.67	29.1	2.92
p2756	1	1235.0	447.34	71.2	21.03
p2756	10	4560.0	430.47	121.7	13.80
p2756	100	9866.7	2738.65	98.3	9.27
vpm2	1	102100.3	3046.30	188.3	10.50
vpm2	10	169214.3	10619.57	201.3	13.87
vpm2	100	214894.0	26372.77	197.3	26.73

Table D.13: Effect of Varying Skip Factor in Broadcast-Distributed Sharing Scheme. 16 Processors. Instances Sometimes Solved.

Name	κ	Avg. Nodes	σ	Avg.Time	σ	# Unsolved
cap6000	1	3254.3	1395.43	3620.0	0.00	3
cap6000	10	26134.7	1790.95	2003.3	355.29	0
cap6000	100	34935.3	2783.20	738.7	231.36	0

Table D.14: Effect of Varying Skip Factor in Broadcast-Distributed Sharing Scheme. 16 Processors. Unsolved Instances.

Name	κ	Avg. Nodes	σ	Avg. Final Gap (%)	σ
harp2	1	207279.7	29948.01	0.0069	0.0068
harp2	10	277219.0	35864.07	0.0797	0.0460
harp2	100	690250.0	84936.41	0.0845	0.0138
mitre	1	2662.0	242.28	0.2672	0.0124
mitre	10	5914.3	259.72	0.2350	0.0158
mitre	100	35835.3	1924.54	0.2307	0.0127
mod011	1	6421.7	238.89	3.2300	0.0850
mod011	10	6983.7	150.21	4.9675	0.0284
mod011	100	10348.3	305.01	4.4951	0.4051
pp08a	1	1283496.0	127941.07	4.0382	0.8189
pp08a	10	1302352.7	70959.20	5.1642	0.2770
pp08a	100	1319419.0	14259.66	5.8093	0.1093
set1ch	1	543431.7	4528.68	12.5389	0.2326
set1ch	10	522016.0	7714.68	13.3814	0.1166
set1ch	100	579151.0	5197.06	14.1194	0.0956

Table D.15: Effect of Varying Skip Factor in No Sharing Scheme. 16 Processors. Solved Instances.

Name	κ	Avg. Nodes	σ	Avg. Time	σ
blend2	1	4017.0	1176.47	27.9	5.55
blend2	10	4621.3	1377.87	14.2	1.91
blend2	100	6590.3	752.74	11.2	1.38
gesa2_o	1	112281.7	15438.61	668.7	106.25
gesa2_o	10	180865.0	49486.37	1006.7	398.79
gesa2_o	100	213130.3	35536.06	977.3	262.67
gesa3_o	1	2965.0	870.06	41.4	9.18
gesa3_o	10	5223.7	1869.34	52.1	11.66
gesa3_o	100	12465.3	5136.50	81.7	28.92
l152lav	1	2280.0	736.18	30.6	4.47
l152lav	10	2349.0	666.46	32.5	2.28
l152lav	100	3082.0	993.87	31.7	3.50
p2756	1	1463.7	304.80	73.7	11.41
p2756	10	3855.3	664.31	115.3	8.02
p2756	100	8660.7	1114.96	91.5	9.53
vpm2	1	115596.0	11108.92	197.0	23.64
vpm2	10	167602.0	6103.36	194.3	8.14
vpm2	100	221158.0	7675.59	201.0	6.08

Table D.16: Effect of Varying Skip Factor in No Sharing Scheme. 16 Processors. Instances Sometimes Solved.

Name	κ	Avg. Nodes	σ	Avg. Time	σ	# Unsolved
cap6000	1	16408.7	1166.40	0.0001	0.0000	3
cap6000	10	26633.0	1057.93	1510.0	235.16	0
cap6000	100	33004.3	3233.84	705.0	153.07	0
harp2	1	268192.0	26200.66	3720.0	0.00	3
harp2	10	525392.3	35756.36	3236.7	500.83	1
harp2	100	877460.5	55822.54	3720.0	0.00	3

Table D.17: Effect of Varying Skip Factor in No Sharing Scheme. 16 Processors. Unsolved Instances.

Name	κ	Avg. Nodes	σ	Avg. Final Gap (%)	σ
mitre	1	1082.0	119.78	0.2880	0.0208
mitre	10	6101.7	410.38	0.2342	0.0062
mitre	100	39746.0	2217.14	0.2403	0.0070
mod011	1	6705.0	130.37	3.2269	0.1167
mod011	10	6816.0	209.46	5.1304	0.4086
mod011	100	10502.0	185.78	4.6707	0.4940
pp08a	1	1215627.7	82221.81	3.6662	0.7022
pp08a	10	1109180.0	22739.01	4.6859	0.4036
pp08a	100	1170517.3	13018.38	6.7734	0.3767
set1ch	1	538746.3	3488.83	12.7455	0.1727
set1ch	10	520092.3	5021.42	13.6483	0.5060
set1ch	100	576534.7	448.50	14.2903	0.3796

BIBLIOGRAPHY

- [1] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. AFIPS Press, 1967.
- [2] R. Anbil, E. L. Johnson, and R. Tanga. A global approach to crew-pairing optimization. *IBM Systems Journal*, 31:73–78, 1992.
- [3] R. W. Ashford, P. Connard, and R. C. Daniel. Experiments in solving mixed integer programming problems on a small array of transputers. *Journal of the Operational Research Society*, 43:519–531, 1992.
- [4] A. Atamtürk, G. Nemhauser, and M. W. P. Savelsbergh. Conflict graphs in integer programming. Technical Report LEC-98-03, Georgia Institute of Technology, 1998.
- [5] A. Atamtürk, G. L. Nemhauser, and M. W. P. Savelsbergh. A combined Lagrangian, linear programming, and implication heuristic for large-scale set partitioning problems. *Journal of Heuristics*, 1:247–259, 1995.
- [6] E. Balas. Facets of the knapsack polytope. *Mathematical Programming*, 8:146–164, 1975.
- [7] E. Balas, S. Ceria, and G. Corneujols. A lift-and-project cutting plane algorithm for mixed 0-1 programs. *Mathematical Programming*, 58:295–324, 1993.

- [8] E. Balas, S. Ceria, and G. Cornuejols. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Science*, 42, 1996.
- [9] E. Balas, S. Ceria, M. Dawande, F. Margot, and G. Pataki. OCTANE: A new heuristic for pure 0-1 programs. 1994.
- [10] E. Balas and R. Martin. Pivot and complement: a heuristic for 0-1 programming. *Management Science*, pages 86–96, 1980.
- [11] E. Balas and M. Padberg. Set partitioning: A survey. *SIAM Review*, 18:710–760, 1976.
- [12] P. Bauer, J. T. Linderoth, and M. W. P. Savelsbergh. A branch and cut approach to the cardinality constrained circuit problem. Technical Report TLI-98-04, The Logistics Institute, Georgia Institute of Technology, 1998.
- [13] E. M. L. Beale. Branch and bound methods for mathematical programming systems. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Discrete Optimization II*, pages 201–219. North Holland Publishing Co., 1979.
- [14] M. Bénichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1:76–94, 1971.
- [15] R. Bixby, W. Cook, A. Cox, and E. Lee. Parallel mixed integer programming. Technical Report CRPC-TR95554, Center for Parallel Computation, Rice University, 1995.
- [16] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, 1998.
- [17] R. E. Bixby and E. Lee. Solving a truck dispatching scheduling problem using branch-and-cut. *Operations Research*, 46:355–367, 1994.

- [18] R. L. Boehning, R. M. Butler, and B. E. Gillett. A parallel integer linear programming algorithm. *European Journal of Operational Research*, 34:393–398, 1988.
- [19] R. Borndörfer. *Aspects of Set Packing, Partitioning, and Covering*. PhD thesis, Technischen Universität Berlin, 1997.
- [20] R. Breu and C. A. Burdet. Branch and bound experiments in zero-one programming. *Mathematical Programming*, 2:1–50, 1974.
- [21] A. Campbell, L. Clarke, A. Kleywegt, and M.W.P. Savelsbergh. The inventory routing problem. Technical Report TLI-97-07, The Logistics Institute, Georgia Institute of Technology, 1997.
- [22] T. L. Cannon and K. L. Hoffman. Large-scale 0-1 linear programming on distributed workstations. *Annals of Operations Research*, 22:181–217, 1990.
- [23] E. Cheng and W. H. Cunningham. Wheel inequalities for stable set polytopes. *Mathematical Programming*, 77:389–421, 1997.
- [24] D. P. Christman. Programming the connection machine. Master’s thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1983.
- [25] V. Chvátal. A greedy heuristics for the set covering problem. *Mathematics of Operations Research*, 4:233–235, 1979.
- [26] V. Chvátal. *Linear Programming*. W. H. Freeman and Co., New York, 1983.
- [27] C. Cordier, H. Marchand, R. Laundry, and L. A. Wolsey. *bc-opt* a branch-and-cut code for mixed integer programs. Discussion Paper 9778, CORE, 1997.
- [28] CPLEX Optimization, Inc. *Using the CPLEX Callable Library*, 1995.

- [29] H. Crowder, E. L. Johnson, and M. W. Padberg. Solving large scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.
- [30] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM-SIGMOD Conference on the Management of Data*, pages 257–266, May 1993.
- [31] R. J. Dakin. A tree search algorithm for mixed programming problems. *Computer Journal*, 8:250–255, 1965.
- [32] A. de Bruin, G. A. P. Kindervater, and H. W. J. M. Trienekens. Asynchronous parallel branch and bound and anomalies. Report EUR-CS-95-05, Erasmus University, Rotterdam, 1995.
- [33] J. J. Dongarra, H. Meuer, and E. Strohmaier. The TOP500 report 1995. *Supercomputer*, 12, 1996. Special Edition.
- [34] N. J. Driebeek. An algorithm for the solution of mixed integer programming problems. *Management Science*, 12:576–587, 1966.
- [35] S. Dutt. Five large MIP problems with up to more than half a million nodes solved on the IBM SP-2 using 128 processors. <http://www-mount.ee.umn.edu/~dutt/mip-sp2-1.html>.
- [36] S. Dutt and N. R. Mahapatra. Parallel A* algorithms and their performance on hypercube multiprocessors. In *Proceedings of the Seventh International Parallel Processing Symposium*, pages 797–803, April 1993.
- [37] J. Eckstein. Parallel branch-and-bound algorithms for general mixed integer programming on the CM-5. *SIAM Journal on Optimization*, 4:794–814, 1994.
- [38] J. Eckstein. Parallel branch-and-bound methods for mixed integer programming. *SIAM News*, 27:12–15, 1994.

- [39] J. Eckstein. Distributed versus centralized control for parallel branch and bound: Mixed integer programming on the CM-5. *Computational Optimization and Applications*, 7:199–220, 1997.
- [40] J. Eckstein. How much communication does parallel branch and bound need. *INFORMS Journal on Computing*, 9:15–29, 1997.
- [41] M. Fisher and P. Kedia. Optimal solution of set covering / partitioning problems using dual heuristics. *Management Science*, 36:674–688, 1990.
- [42] J. J. H. Forrest, J. P. H. Hirst, and J. A. Tomlin. Practical solution of large scale mixed integer programming problems with UMPIRE. *Management Science*, 20:736–773, 1974.
- [43] I. Foster. *Designing and Building Parallel Programs : Concepts and Tools for Parallel Software Engineering*. Addison Wesley, 1995.
- [44] J. M. Gauthier and G. Ribière. Experiments in mixed-integer linear programming using pseudocosts. *Mathematical Programming*, 12:26–47, 1977.
- [45] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 Users's Guide and Reference Manual*. Oak Ridge National Laboratory, Oak Ridge, TN 37831-6367, September 1994.
- [46] B. Gendron and T. G. Crainic. Parallel branch and bound algorithms: Survey and synthesis. *Operations Research*, 42:1042–1066, 1994.
- [47] M. Goemans and D. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems. *Journal of the ACM*, 42:1115–1145, 1995.
- [48] R. E. Gomory. Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Monthly*, 64:275–278, 1958.

- [49] W. Gropp and E. Lusk. *User's Guide for mpich, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Labs, 1996. Available as Tech Report ANL/MCS-TM-ANL-96-6.
- [50] Z. Gu. *Lifted Cover Inequalities for 0-1 and Mixed 0-1 Integer Programs*. PhD thesis, Georgia Institute of Technology, 1995.
- [51] Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Cover inequalities for 0-1 linear programs: Computation. *INFORMS Journal on Computing*, 1994. To appear.
- [52] Z. Gu, G. L. Nemhauser, and M. W. P. Savelsbergh. Lifted flow covers for mixed 0-1 integer programs. Technical Report TLI-96-05, Georgia Institute of Technology, 1995.
- [53] Z. Gu, G.L. Nemhauser, and M.W.P. Savelsbergh. Sequence independent lifting. Technical Report TLI-95-08, Georgia Institute of Technology, 1995.
- [54] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.
- [55] J. L. Gustafson, G. R. Montry, and R. E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9:609–638, 1988.
- [56] M. T. Hajian, L. Hai, and G. Mitra. A distributed processing algorithm for solving integer programs using a cluster of workstations. *Parallel Computing Applications*, 1994. Submitted.
- [57] P. L. Hammer, E. L. Johnson, and U. N. Peled. Facets of regular 0-1 polytopes. *Mathematical Programming*, 8:179–206, 1975.

- [58] F. Harche and G. L. Thompson. Column subtraction algorithm: An exact method for solving weighted set covering, packing and partitioning problems. *Computers & Operations Research*, 21:689–705, 1994.
- [59] R. Hempel. The MPI Standard for Message Passing. In Wolfgang Gentzsch and Uwe Harms, editors, *High-performance computing and networking: international conference and exhibition, Munich, Germany, April 18–20, 1994: proceedings*, volume 797 of *Lecture notes in computer science*, pages 247–252, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1994. Springer-Verlag.
- [60] W. D. Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.
- [61] J. P. H. Hirst. Features required in branch and bound algorithms for (0-1) mixed integer linear programming. Privately circulated manuscript, December 1969.
- [62] K. Hoffman and M. Padberg. Solving airline crew-scheduling problems by branch-and-cut. *Management Science*, 39:667–682, 1993.
- [63] D. Homeister. Efficient implementation of parallel branch and cut. In *Proceedings of the Fifth SIAM Conference on Optimization*, May 1996.
- [64] J. Hu and E. L. Johnson. Computational results with a primal-dual subproblem simplex method. Submitted, 1998.
- [65] International Business Machines. *IBM Parallel Environment for AIX : Hitchhiker's Guide*, version 2, release 3 edition, 1997.
- [66] M. Jünger and P. Störmer. Solving large scale traveling salesman problems with parallel branch-and-cut. Technical Report 95.191, Universität zu Köln, Zentrum für Paralleles Rechnen, 1995.

- [67] R. M. Karp and Y. Zhang. Randomized parallel algorithms for backtrack search and branch and bound computation. *Journal of the ACM*, 40:765–789, 1993.
- [68] P. Keleher, A. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, 1994.
- [69] D. Klabjan, E. L. Johnson, and G. L. Nemhauser. Solving large scale linear programs: Parallel primal dual algorithm. Submitted, 1999.
- [70] D. Klabjan, G. L. Nemhauser, and C. A. Tovey. The complexity of cover inequality separation. Technical Report 95-06, The Logistics Insitute, Georgia Tech, 1995.
- [71] D. E. Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973.
- [72] V. Kumar, P.S. Gopalkrishnan, and L. Kanal, editors. *Parallel Algorithms for Machine Intelligence and Vision*. Springer Verlag, New York, 1990.
- [73] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 22:379–391, September 1994.
- [74] V. Kumar, K. Ramesh, and V. N. Rao. Parallel best-first search of state-space graphs: A summary of results. In *Proceedings of the 1988 National Conference on Artificial Intelligence*, pages 122–127, August 1988.
- [75] V. Kumar and V. N. Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16:501–519, 1987.
- [76] L. Ladányi. A parallel branch-and-cut-and-price framework for mip. Presented at the SIAM Annual Meeting, Toronto, Canada, July 1998.

- [77] T.H. Lai and S. Sahni. Anomalies in parallel branch and bound algorithms. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 183–190, 1983.
- [78] A. Land and S. Powell. Computer codes for problems of integer programming. In P. L. Hammer, E. L. Johnson, and B. H. Korte, editors, *Discrete Optimization II*, pages 221–269. North Holland Publishing Co., 1979.
- [79] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.
- [80] R. Laundry. Implementation of parallel branch and bound algorithms in XPRESS-MP. Dash Internal Report no. DA/TR-101, January 1998.
- [81] L. Lettovský. *Airline Recovery: An Optimization Approach*. PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, 1997.
- [82] D. Levine. *A Parallel Genetic Algorithm for the Set Partitioning Problem*. PhD thesis, Illinois Institute of Technology, 1994.
- [83] T. G. Lewis and H. El-Rewini. *Introduction to Parallel Computing*. Prentice Hall, Englewood Cliffs, NJ, 1992.
- [84] J. T. Linderoth. LPSOLVER, a transparent interface to linear and integer programming software, 1998. In preparation.
- [85] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies in mixed integer programming. *INFORMS Journal on Computing*, 1998. To appear.
- [86] J. D. C. Little, K. G. Murty, D. W. Sweeney, and C. Karel. An algorithm for the traveling salesman problem. *Operations Research*, 21:972–989, 1963.

- [87] R. Lüling and B. Monien. Load balancing for distributed branch and bound algorithms. In *Proceedings of the International Parallel Processing Symposium*, pages 543–549, Beverly Hills, CA, 1992.
- [88] International Business Machines. Livermore and IBM in \$93 million deal to build world’s fastest supercomputer, July 1996. Press release, available at <http://www.rs6000.ibm.com/resource/features/1996/fast.friends/livermore.html>.
- [89] N.R. Mahapatra and S. Dutt. New anticipatory load balancing strategies for scalable parallel best-first search. In *American Mathematical Society’s DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, volume 22, pages 197–232, 1995.
- [90] D. L. Miller and J. F. Pekny. Results from a parallel branch and bound algorithm for the asymmetric travelling salesman problem. *Operations Research Letters*, 8:129–135, 1989.
- [91] G. Mitra. Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming*, 4:155–170, 1973.
- [92] J. Mohan. Experience with two parallel programs solving the traveling salesman problem. In *Proceedings of the IEEE Conference on Parallel Processing*, pages 191–193, 1983.
- [93] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite file system. *ACM Transactions on Computer Systems*, 6:134–154, February 1988.
- [94] G. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

- [95] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15:47–58, 1994.
- [96] G. L. Nemhauser and L. E. Trotter Jr. Properties of vertex packing and independence system polyhedra. *Mathematical Programming*, 6:48–61, 1974.
- [97] M. Padberg. On the facial structure of set packing polyhedra. *Mathematical Programming*, 5:199–215, 1973.
- [98] M. Padberg. Perfect zero-one matrices. *Mathematical Programming*, 6:180–196, 1974.
- [99] M. Padberg, T. J. Van Roy, and L. Wolsey. Valid linear inequalities for fixed charge problems. *Operations Research*, 33:842–861, 1985.
- [100] M. W. Padberg and G. Rinaldi. A branch and cut algorithm for the solution of large scale traveling salesman problems. *SIAM Review*, 33:60–100, 1991.
- [101] G. Pataki. A new computational study of lift-and-project cuts. Presented at the 16th International Symposium on Mathematical Programming, Lausanne, Switzerland, 1997.
- [102] K. Perumalla and K. Schwan. CoGEnt – A distributed active object framework. Manuscript under development, December 1996.
- [103] E. Pruul, G. L. Nemhauser, and R. A. Rushmeier. Branch and bound and parallel computation: A historical note. *Operations Research Letters*, 7:65–69, 1988.
- [104] T. K. Ralphs. *Parallel Branch and Cut for Vehicle Routing*. PhD thesis, Cornell University, 1995.
- [105] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, New York, 1990.

- [106] V. J. Rayward-Smith, S. A. Rush, and G. P. McKeown. Efficiency considerations in the implementation of parallel branch-and-bound. *Annals of Operations Research*, 43:123–145, 1993.
- [107] G. Reinelt. TSPLIB - a traveling salesman problem library. *ORSA Journal on Computing*, 3:376–384, 1991.
- [108] T. J. Van Roy and L. A. Wolsey. Solving mixed integer 0-1 programs by automatic reformulation. *Operations Research*, 35:45–57, 1987.
- [109] D. M. Ryan and J. C. Falkner. On the integer properties of scheduling set partitioning models. *European Journal of Operational Research*, 35:442–456, 1988.
- [110] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.
- [111] A. J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11:7–21, December 1978.
- [112] T. H. C. Smith and G. L. Thompson. Parallel implementation of the column subtraction algorithm. *Parallel Computing*, 21:63–74, 1995.
- [113] X. H. Sun and J. L. Gustafson. Toward a better parallel performance metric. *Parallel Computing*, 17:1093–1109, 1991.
- [114] Thinking Machines Corporation, Cambridge, MA. *The Connection Machine CM-5 Technical Summary*, 1992.
- [115] H. W. J. M. Trienekens and A. de Bruin. Towards a taxonomy of parallel branch and bound algorithms. Report EUR-CS-92-01, Erasmus University, Rotterdam, 1992.
- [116] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

- [117] S. Tschöke, R. Lüling, and B. Monien. Solving the traveling salesman problem with a distributed branch-and-bound algorithm on a 1024 processor network. In *Proceedings of the 9th International Parallel Programming Symposium*, pages 182–189, Santa Barbara, CA, 1995.
- [118] D. Wedelin. An algorithm for large scale 0-1 integer programming with applications to airline crew scheduling. *Annals of Operations Research*, 57:283–301, 1995.
- [119] L. A. Wolsey. Faces for a linear inequality in 0-1 variables. *Mathematical Programming*, 8:165–178, 1975.