

**HIGH PERFORMANCE COMPUTING
ALGORITHMS FOR DISCRETE OPTIMIZATION**

A Dissertation
Presented to
The Academic Faculty

By

Lluís-Miquel Munguía Conejero

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science and Engineering

Georgia Institute of Technology

December 2017

Copyright © Lluís-Miquel Munguía Conejero 2017

**HIGH PERFORMANCE COMPUTING
ALGORITHMS FOR DISCRETE OPTIMIZATION**

Approved by:

Professor David A. Bader, Advisor
Committee Chair
School of Computational Science
and Engineering
Georgia Institute of Technology

Professor Shabbir Ahmed
School of Industrial and Systems
Engineering
Georgia Institute of Technology

Professor Bistra Dilkina
School of Computational Science
and Engineering
Georgia Institute of Technology

Professor George L. Nemhauser
School of Industrial and Systems
Engineering
Georgia Institute of Technology

Professor Richard Vuduc
School of Computational Science
and Engineering
Georgia Institute of Technology

Date Approved: October 30, 2017

To my dear parents Amelia and José Ángel and my sister Aitana, for your support and love. You will always be in my heart, no matter how many oceans we are apart.

ACKNOWLEDGEMENTS

I would like to thank my advisor David Bader for sharing his passion for graph algorithms, and for his constant support, advice, and encouragement during my time at Georgia Tech. I have learned many important lessons under his guidance. I am also grateful for the freedom provided, which allowed me to study the research questions I enjoyed.

The deepest of my appreciations to professors George L. Nemhauser and Shabbir Ahmed; for bringing me to the world of Integer Programming, and sharing the frustrations and joys found along the road. I am deeply honored to have had the opportunity of working with such talented leaders in the field of Mathematical Optimization.

I am also thankful to the other members of the thesis advisory committee. Bistra Dilkina, for her thoughtful suggestions and fruitful discussions. Richard Vuduc, for his invaluable teachings since the moment I arrived at Georgia Tech as an exchange student six years ago. My work would also not have been possible without the extended support of Exxon-Mobil, and the advice of Yufen Shao, Dimitri J. Papageorgiou, and Myun-Seok Cheon.

I would also like to extend my gratitude to my mentors and friends: Deepak Rajan, Yuji Shinano, Oded Green, Xing Liu, Rodolfo Carvajal, Richard Roberts, and Josep Miquel Jornet for their guidance in the diverse facets needed to succeed as a PhD student. Thanks to my fellow Ph.D. students, many of whom I've had the pleasure of working and sharing laughs with: Amelia Musselman, Elias Khalil, Adam McLaughlin, Anita Zakrzewska, Eisha Nathan, Marat Dukhan, Jiajia Li, and Patrick Flick. And to all of my other friends, for having all the patience in the world with me and my experiments, and for preventing graduate school from swallowing me whole: Jordi Riera, Kenneth Stewart, Carlos Espinosa, Lauren Argabrite, Ira Kaplan, Molly Ward, Katie Merryman, and many others.

Last but not least, I would like to express my gratitude to my parents and sister for supporting me throughout my entire life, and fueling the pursuit of my goals with never-ending encouragement.

TABLE OF CONTENTS

Acknowledgments	v
List of Tables	xii
List of Figures	xiv
Chapter 1: Introduction	1
1.1 Mixed Integer Programming	1
1.2 Branch-and-Bound	2
1.2.1 Preprocessing	3
1.2.2 Cutting planes	4
1.2.3 Branching	4
1.2.4 Search strategy	5
1.2.5 Primal heuristics	6
1.3 The need for parallel algorithms in High Performance Computing	7
1.3.1 Parallel algorithms for MIP optimization	7
1.4 Outline of the dissertation	8
Chapter 2: A Parallel Local Search Framework for the Fixed-Charge Multi-commodity Network Flow Problem	12
2.1 Introduction	12

2.2	Problem description	15
2.3	Local search methodology	17
2.3.1	The local search mechanism	18
2.3.2	Partitioning the subproblem sequence	19
2.3.3	The solution recombination procedure	21
2.3.4	Obtaining a first feasible solution	23
2.4	Parallel implementation	23
2.5	Experimental results	28
2.5.1	C instance set performance results	30
2.5.2	GT instances	32
2.5.3	Scaling results	34
2.5.4	Load balancing	36
2.5.5	Partitioning the subproblem sequence	37
2.5.6	The parallel solution recombination	39
2.6	Conclusions	40

**Chapter 3: Alternating Criteria Search: A Parallel Large Neighborhood Search
Algorithm for Mixed Integer Programs 42**

3.1	Introduction	42
3.1.1	Parallel computing applied to Integer Optimization	43
3.2	Parallel Alternating Criteria Search	45
3.2.1	Parallelization of Alternating Criteria Search	48
3.2.2	Finding a starting point	51
3.2.3	The variable fixing scheme	52

3.2.4	Framing Alternating Criteria Search within an exact algorithm . . .	54
3.3	Experimental results	54
3.3.1	Automating the choice of parameters	55
3.3.2	Evaluation of primal solution quality	56
3.3.3	Framing Alternating Criteria Search within an exact algorithm . . .	61
3.3.4	The impact of nondeterminism	63
3.3.5	Additional performance tests: scalability and parallel efficiency . .	65
3.4	Conclusions	69
Chapter 4:	Rapid Prototyping of Effective Parallel Primal Heuristics for Do-	
	main Specific MIPs: An Application to the Maritime Inventory Rout-	
	ing Problem	71
4.1	Introduction	71
4.2	Related work	72
4.3	Algorithm	74
4.3.1	A time-space discretization of MIRPs	74
4.3.2	Applying Parallel Alternating Criteria Search to MIRP instances . .	78
4.4	Experimental results	85
4.4.1	Tuning of parameters and nondeterminism	86
4.4.2	Group 1 MIRP instances	87
4.4.3	Group 2 MIRP instances	90
4.4.4	Indications for specializing Parallel Alternating Criteria Search . . .	93
4.5	Conclusions	94

Chapter 5: PIPS-SBB: A Parallel Distributed-Memory Branch-and-Bound Algorithm for Stochastic Mixed-Integer Programs	95
5.1 Introduction	95
5.1.1 Contributions and overview	98
5.1.2 Related work: Solving SMIPs using decomposition schemes	100
5.2 PIPS-SBB: A specialized parallel distributed-memory branch-and-bound solver for large-scale stochastic MIP problems	102
5.2.1 Branch-and-Bound	103
5.2.2 Parallelism in the LP relaxation	105
5.2.3 Parallel data distribution	106
5.3 Implementation	106
5.3.1 Software architecture of PIPS-SBB	107
5.3.2 Parallelism in structure-aware algorithms	109
5.4 Experimental Results	112
5.4.1 Scaling experiments	114
5.4.2 Overall performance	116
5.4.3 Specialized structure-aware algorithms	122
5.5 Conclusions and future directions	124
Chapter 6: Parallel PIPS-SBB: Multi-Level Parallelism for Stochastic Mixed-Integer Programs	127
6.1 Introduction	127
6.1.1 Parallel Branch-and-Bound	128
6.1.2 Contributions and outline	130

6.2	PIPS-PSBB: A decentralized lightweight parallel framework for Branch-and-Bound	132
6.2.1	The philosophy behind fine-grained node rebalancing	134
6.2.2	Decentralized node exchange	136
6.2.3	Processor distribution	138
6.3	Parallelizing PIPS-SBB with UG	139
6.3.1	Processor distribution within UG	142
6.3.2	Supervised workload coordination mechanism	142
6.4	Experimental results	143
6.4.1	Scaling performance: Branch-and-Bound parallelization	144
6.4.2	Scaling performance: effects of multi-level parallelism	147
6.4.3	Overall performance and comparison to CPLEX	150
6.5	Conclusions	152
Chapter 7: Conclusions		154
7.1	Main contributions	154
7.2	Future research areas	157
Appendix A: Nomenclature used in the Maritime Inventory Routing Problem formulations		159
A.1	Indices and sets	159
A.2	Problem data	160
A.3	Problem decision variables	160
References		175

Vita 176

LIST OF TABLES

2.1	C instance set optimization results	31
2.2	GT instance set optimization results	33
2.3	Time required to reach certain gap with respect to the best found primal solution and the best lower bound	35
2.4	C instance set: performance comparison between commodity assignation schemes	38
2.5	GT instance set: performance comparison between commodity assignation schemes	39
2.6	Performance comparison of solution recombination schemes	40
3.1	Primal solution performance comparison	62
3.2	Optimality performance comparison	63
3.3	Performance comparison of non-deterministic approaches	65
3.4	Selected instances for Scaling experiments	66
4.1	Group 1 performance comparison of non-deterministic approaches	89
4.2	Group 2 performance comparison of non-deterministic approaches	93
5.1	SSLP instance set results	119
5.2	SSLPRep instance set results	119
5.3	DCAP instance set results	122

5.4	SMKP instance set results	123
5.5	Comparison specialized stochastic and general structure heuristics	124
6.1	Performance comparison for all SSLP instances	151
6.2	Version-to-version performance comparison of PIPS-SBB solvers	151

LIST OF FIGURES

2.1	Parallel decomposition of a sequential local search procedure. Local searches are distributed in a total of P sequences. As a result, P feasible solutions are obtained and recombined in an improved solution.	18
2.2	Subproblem partitioning shown on a small example with three commodities. A subproblem is defined for each commodity, where the flow highlighted in red is optimized and the remaining flow is fixed.	20
2.3	Connection graph generated from a feasible solution. Commodities are represented with vertices, and there is an arc between two vertices if their corresponding commodities share an arc in the solution.	21
2.4	The solution recombination step is shown on a small example with three commodities. A subproblem is defined for a set of input solutions, where the flow in the arcs highlighted in red is optimized while the remaining arcs fixed.	22
2.5	Schematic depiction of parallel systems with different memory configurations. System (a) features a unified memory space, which is accessible by every parallel core simultaneously. In contrast, system (b) has a segmented memory space, which is distributed between the different parallel cores. . .	24
2.6	Parallel hybrid memory framework. The process described in Figure 2.1 is expanded to accommodate the parallel execution in distributed-memory systems.	26
2.7	Potential load imbalance causes of the implementation.	28
2.8	Parameter sensitivity analysis for C and GT instances. Parameter configurations are specified with two numbers X-Y, where X refers to the local search time limit and Y to the solution recombination time limit.	29

2.9	Scaling results for a selected test instance with 500 vertices, 3000 arcs and varying commodities. Each plot depicts the improvement in optimality GAP with respect to the best lower bound as a function of time. The executions shown in each problem differ in the number of parallel cores used.	34
2.10	Overall processor utilization results for a selected test instance with 500 vertices, 3000 arcs and 200 commodities. Each data set corresponds to a different time limit parameter configuration, where the first number refers to the local search time limit and the second refers to the solution recombination time limit.	37
3.1	High level depiction of the sequential heuristic	46
3.2	Transition to a high quality feasible solution	48
3.3	Example depicting a feasibility improvement iteration for a 0-1 knapsack sample instance	50
3.4	Primal gap function $p(t)$ and primal integral $P(t)$ of the solutions provided by CPLEX and PACS for the (a)(d)rail03, (b)shs1023 and (c)sing245 problems.	57
3.5	Improvement and Deterioration of primal gap w.r.t. CPLEX for all instances	58
3.6	Improvement and Deterioration of primal gap in hard and open instances w.r.t. CPLEX	59
3.7	Improvement and Deterioration in primal integral w.r.t. CPLEX for (a)all and (b)hard and open instances	60
3.8	Improvement and Deterioration of optimality gap w.r.t. CPLEX for (a)all and (b)hard and open instances	62
3.9	Performance profiles for all instances in the Reoptimize set. In each of the charts multiple lines are drawn for the best, mean, and worst executions. Charts show the performance for different cutoffs: (a)(d)180s, (b)(e)600s, and (c)(f)3600s	64
3.10	Parallel strong-scaling results	67
3.11	Parallel synchronization overhead in terms of average processor utilization for different parameter configurations	68

3.12	Performance tradeoffs shown by the starting heuristic, in which the time to reach the initial solution is contraposed to its quality as a function of the variable fixing parameter ρ for the (a)in, (b)rail03, (c)shs1023 and (d)sing245 problems.	69
4.1	Time-space horizon modeling of MIRP	75
4.2	Application of Parallel Alternating Criteria Search to a group 1 MIRP instance, with 4 loading ports, 9 discharging ports, 17 vessels, and a horizon of 60 timesteps. (a) Depicts the evolution of the infeasibility of the incumbent, and objective value of found solutions as a function of time. (b) Shows a breakdown of the infeasibility found in the incumbent by constraint type as a function of time.	79
4.3	The port span of a vessel v between two time steps t_1 and t_2 in a solution x is the set of ports traversed by v during the time window in x	83
4.4	(a)Average primal gap and (b)average primal integral for group 1 instances .	87
4.5	(a) Percentage of instances for which a feasible solution is found. (b) Percentage of instances for which MIRPpac finds a feasible solution, broken down by instance subclass	88
4.6	(a)Percentage of instances for which a feasible solution is found. (b)Percentage of instances for which MIRPpac finds a feasible solution, broken down by instance subclass	90
4.7	(a) Average primal gap and (b) average primal integral for all group 2 instances.	91
4.8	Average primal gap and average primal integral for (a) easy, (b) medium, and (c) hard group2 instances.	92
5.1	(a) Schematic depiction of a parallel system with a distributed-memory configuration. It has a segmented memory space, which is distributed among different processes. (b) Data parallelism in PIPS-SBB.	107
5.2	Class diagram of PIPS-SBB. Classes are shown as boxes with the top section as the name of the class. Interactions between classes are depicted as lines between classes, and indicate multiplicity indicators at each end, for example (1..*) representing “one or more”.	108

5.3	(a) Strong scaling performance results of PIPS-S. (b) Strong scaling throughput results of PIPS-SBB.	115
5.4	Strong scaling performance results of PIPS-SBB	117
6.1	Example depicting the coarse-grained distribution of subtrees among the parallel processors at a given point in the parallel exploration. Knowledge of feasible solution at black-colored node could have fathomed many other nodes on many processors.	133
6.2	PIPS-PSBB: Example depicting the fine-grained distribution of subtrees among the parallel processes at a given point in the parallel exploration. The frontier and the active nodes in the frontier are also depicted.	135
6.3	Steps of the node exchange: Node estimates (in circles) are gathered using all-to-all communication and sorted at each worker. When each worker has determined (in parallel) which nodes need to be exchanged, actual node information (in squares) is redistributed using point-to-point communication.	137
6.4	Processor distribution used in PIPS-PSBB: Each row corresponds to the communicator of a single PIPS-SBB solver. Each column corresponds to the communicator used by processes from different PIPS-SBB solvers that own the same data.	140
6.5	Design of UG and ug[PIPS-SBB, MPI]	140
6.6	Processor distribution used in ug[PIPS-SBB,MPI]	142
6.7	Scaling performance of PIPS-PSBB and ug[PIPS-SBB,MPI] when solving <i>sslp_15_45_5</i> : Speedup (time to optimality), branch and bound nodes processed, parallel communication overhead, and proportion of redundant nodes	145
6.8	Communication overhead for PIPS-PSBB and ug[PIPS-SBB,MPI] when solving <i>sslp_15_45_5</i> broken down by stage in the optimization process . . .	146
6.9	Performance of PIPS-PSBB for different communication frequencies λ . ($x - y$) denotes the minimum (x) and the maximum (y) number of solver iterations before communication is attempted.	147
6.10	Scaling performance of PIPS-PSBB when solving <i>sslp_10_50_500</i> : Time to solve the LP relaxation, communication overhead, number of nodes processed, and communication overhead at ramp-up.	148

6.11 Scaling performance of ug[PIPS-SBB,MPI] when solving *sslp_10_50_500*:
Time to solve the LP relaxation, communication overhead, number of nodes
processed, and communication overhead at ramp-up. 149

SUMMARY

This thesis concerns the application of High Performance Computing to Discrete Optimization, and the development of massively parallel algorithms designed to accelerate the solving process of Mixed-Integer Programs (MIPs).

We begin by presenting a portfolio of scalable parallel primal heuristics, which focus on providing the end-user with high quality feasible solutions to any MIP program quickly. In some cases, we show our algorithms to be several orders of magnitude more effective than current state-of-the-art approaches. The first of the contributions in this category is a specialized primal heuristic for the Fixed Charge Multicommodity Network Flow problem. The presented computational experiments back the superior effectiveness of our method at finding substantially better primal solutions when compared to state-of-the-art commercial MIP solvers, even when the latter are allowed five times as much time.

We further generalize the introduced notions and develop Parallel Alternating Criteria Search: a general-purpose parallel primal method prepared for handling any unstructured MIP. We show how the combination of parallelism and simple large neighborhood search schemes can provide a powerful tool for generating high quality solutions for any given problem. Our parallel method is able to produce competitive or better and faster results for more than 90% of the tested instances against CPLEX. Parallel Alternating Criteria Search becomes especially useful in the context of large instances and time-sensitive optimization problems, where traditional branch-and-bound methods may not be able to provide competitive upper bounds and attaining feasibility may be challenging. The modular nature of Parallel Alternating Criteria Search can provide an excellent platform for rapid prototyping parallel domain-specific heuristics. We show this particular achievement on the Maritime Inventory Routing Problem, a complex optimization problem that combines network flows with inventory management. In the presented results, tailored versions of our parallel algorithm are able to significantly outperform domain-specific state-of-the-art heuristics and

parallel MIP solvers alike.

The second half of this thesis is dedicated to the introduction of PIPS-SBB, a parallel distributed-memory solver for deterministic equivalent two-stage stochastic MIPs (sMIPs). The newly introduced solver features multiple levels of nested parallelism. It is also designed with data parallelism in mind, allowing the problem data to be partitioned across multiple distributed-memory machines. We then present two Branch-and-Bound parallelizations extending the already parallel solver. We investigate the effects of leveraging multiple levels of parallelism and their part in improving the scaling performance beyond thousands of cores. We also compare our algorithms against a distributed-memory implementation of a commercial MIP solver. The latter proves to be the best performer at small problem scales. However, the specialized nature of the methods present in PIPS-SBB-based solvers allow them to be the best performers in large SMIP instances.

The direct product of this thesis is a set of algorithms ready to be used in massively parallel systems to quickly find high quality solutions to any MIP problem. The presented works ultimately increase our understanding of the use of parallelism in the context of Discrete Optimization and its important part in improving the effectiveness and performance of its algorithms.

CHAPTER 1

INTRODUCTION

1.1 Mixed Integer Programming

Mixed Integer Programming (MIP) [1] modeling tools and algorithms allows one to formulate and solve a large variety of planning and operational problems in energy, transportation, manufacturing, finance, health, military, and any other domain where decisions are made. MIPs address particular optimization problems in which a linear function of variables subject to linear constraints is minimized or maximized. Additionally, the domain of a subset of decision variables is reduced to integer values only. We formally define a MIP as:

$$\min\{c^t x \mid Ax = b, l \leq x \leq u, x_i \in \mathbb{Z}, \forall i \in \mathcal{I}\} \quad (\text{MIP})$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $\mathcal{I} \subseteq \{1, \dots, n\}$ is the subset of integer variable indices. The decision vector x is bounded by $l \in \mathbb{R}^n$ and $u \in \mathbb{R}^n$.

Integer variables often have the crucial function of representing on-off binary logical decisions, such as choosing a specific routing for a fleet of transportation vessels or a particular assignment in a job schedule. This freedom of expressiveness, however, comes at a great computational cost. MIPs are NP-hard combinatorial optimization problems, and most approaches for solving them are based on the systematic enumeration of all candidate solutions. When facing this complexity, high quality feasible solutions have become a valuable asset in the MIP solving process, and constructing them has been one of the main focuses of research for the past two decades. While feasible solutions can help to speed the optimization, they are most valuable for large-scale and time-sensitive instances, where proving optimality may be computationally intractable.

1.2 Branch-and-Bound

The branch-and-bound algorithm [2] is the most commonly used method for solving MIPs to optimality. It is a search algorithm that systematically partitions the problem into smaller problems and searches the solution space using a dynamically generated tree data structure called the branch-and-bound tree, in order to enumerate feasible integer solutions. In LP-relaxation based branch-and-bound, the quality and promise of subproblems is evaluated by solving an LP relaxation formed by relaxing all integrality constraints. The optimization concludes once all subproblems are fathomed or solved to optimality. Furthermore, since the branch-and-bound algorithm provides upper and lower bounds on the optimal solution, it can also be used to provide a guarantee on the quality of the best solution found, if terminated early.

Algorithm 1 LP-Based Branch-and-Bound

```
 $UB = \infty$   
 $LB = -\infty$   
priority queue  $Q = \emptyset$   
Add root subproblem  $r$  of original MIP problem to  $Q$   
while  $Q \neq \emptyset$  and  $LB < UB$  do  
    remove subproblem  $p$  from top of  $Q$   
    Solve LP relaxation of  $p$ , process  $p$ , bound  $p$   
     $UB_p =$  best solution found within  $p$   
     $LB_p =$  lower bound of  $p$ ,  $\infty$  if infeasible,  $-\infty$  if unbounded  
     $UB = \min\{UB, UB_p\}$   
    if  $LB_p < UB$  then  
        Partition  $p$  into a set  $\{s_0, \dots, s_k\}$  of subproblems, each with a lower bound defined to be  $LB_p$   
        Add  $\{s_0, \dots, s_k\}$  to  $Q$   
    else  
        Continue ▷ If  $LB_p \geq UB$ , problem  $p$  is discarded (fathomed by bound dominance)  
    end if  
     $LB =$  minimum lower bound among all open subproblems in  $Q$   
end while  
if  $LB < UB$  then  
    return infeasible  
else  
    return optimal solution  
end if
```

Despite being an exact algorithm for solving MIPs, a naïve branch-and-bound implementation as described in Algorithm 1 is a computationally infeasible algorithm due to

the exponential growth of the search space. State-of-the-art MIP solvers build upon this branch-and-bound scheme and enhance it with many additional techniques, which prune the space in order to make MIP optimization practical. Developing each of these techniques is not straightforward either, given that they usually involve solving additional NP-hard problems. Primal heuristics [3, 4] are algorithms, usually without a guarantee of success, that focus on finding high quality solutions in order to improve the upper bound. Additionally, pre-processing [5, 6] and cutting planes [7] are techniques focused on strengthening the LP relaxation, and overall reducing the search space. Other components that are equally important are the branching rules (how to partition the problem), and how the priority queue is sorted. The search strategy can have a big impact on how fast high-quality solutions are found and determine greatly the size of the tree required to solve a problem to optimality. We now describe each of these advancements in detail.

The contribution of all these algorithmic components is paramount for maximizing the performance of any MIP solving algorithm. However, their inclusion has also turned the latter into highly complex pieces of software with a multitude of abstraction layers. Using pre-processing, cutting planes, and heuristics comes at an expensive cost, as they often involve solving auxiliary NP-hard problems, which may be almost as hard as the original problem. Thus, their correct tuning is important for finding the best trade-off between the benefits they may provide and the computational overhead they could potentially cause.

1.2.1 Preprocessing

A significant upgrade to standard branch-and-bound comes from MIP model preprocessing. Preprocessing is a technique that aims at removing redundant information of a MIP in order to reduce its complexity. The preprocessing techniques described by Savelsbergh [5] and Achterberg [6] include the removal of redundant constraints and variables, bound tightening, and the identification of special substructures, such as cliques. LP relaxations of simplified models are usually easier to solve than their full-size counterparts. In addition,

the potential number of variable branchings may also be reduced if redundant variables are also eliminated.

1.2.2 Cutting planes

Cutting planes are one of the major improvements for branch-and-bound. They are additional linear inequalities [7] that help eliminate feasible regions of the problem, without discarding any integer solutions. As a result, cutting planes deliver stronger LP relaxations, reducing the number of branching decisions required to achieve optimality. Cutting planes are obtained by performing linear combinations of constraints and rounding. Many cuts can be generated, and the challenge is to know which ones to generate and how to manage them, which remains an NP-complete problem. Knapsack cuts [8], Lifted Cover inequalities [9], Flow Cover inequalities [10], Gomory Mixed Integer cuts [11], Lift and Project cuts [12], Strong Chvatal Gomory cuts [13], and Mixed Integer Rounding cuts [14] are among the most effective cutting planes.

The advent of cutting planes brought branch-and-cut, a modification of standard branch-and-bound, in which cutting planes are used to tighten the LP relaxations in addition of branching. Branch-and-cut (Algorithm 2) requires the solving of the additional separation problem, which entails finding a valid inequality that will cut away the part of the feasible region containing the fractional solution.

1.2.3 Branching

The effectiveness of a MIP branch-and-bound tree search algorithm also depends on tree creation algorithms (branching rules) that determine how to partition the feasible space. Branching rules intend to find the optimal way to partition the problem so that a minimal number of subproblem resolutions are required in order to prove optimality. Examples of widely used branching rules include the most infeasible branching, pseudocost branching [15, 16], strong branching, and an intermediate approach called reliability branch-

Algorithm 2 LP-Based Branch-and-Cut

```
 $UB = \infty$   
 $LB = -\infty$   
priority queue  $Q = \emptyset$   
Add root subproblem  $r$  of original MIP problem to  $Q$   
while  $Q \neq \emptyset$  and  $LB < UB$  do  
  remove subproblem  $p$  from top of  $Q$   
  while termination conditions are not met do  
     $x = \text{Solve LP relaxation of } p, \text{ process } p, \text{ bound } p$   
    Search valid inequalities that are violated by  $x$  and add them to the LP relaxation  
  end while  
   $UB_p = \text{best solution found within } p$   
   $LB_p = \text{lower bound of } p$   
   $UB = \min\{UB, UB_p\}$   
  if  $LB_p < UB$  then  
    Partition  $p$  into a set  $\{s_0, \dots, s_k\}$  of subproblems, each with a lower bound defined to be  $LB_p$   
    Add  $\{s_0, \dots, s_k\}$  to  $Q$   
  else  
    Continue ▷ If  $LB_p \geq UB$ , problem  $p$  is discarded (fathomed by bound dominance)  
  end if  
   $LB = \text{minimum lower bound among all open subproblems in } Q$   
end while  
if  $LB < UB$  then  
  return infeasible  
else  
  return optimal solution  
end if
```

ing [17]. Linderoth [18] and Achterberg [17] perform computational comparisons of the different options.

1.2.4 Search strategy

Another important aspect directly affecting the convergence to an optimal solution is the order in which subproblems are processed. Two of the most known criteria are named *BestBound* and *Diving*. The first prioritizes the processing of subproblems with the tightest lower bounds with the objective of improving the global lower bound as fast as possible. *Diving* seeks to process subproblems deep in the tree, which are more likely to allow the discovery of improved solutions. Most modern MIP solvers incorporate a careful balance of both approaches, or additional rules that sit in the middle of the spectrum, such as the one introduced in [19]. In [18], a computational study with an overview of multiple

search strategies is presented.

1.2.5 Primal heuristics

Primal heuristics are methods aimed at finding good solutions in a short period of time. In branch-and-bound their execution takes places within the processing of certain nodes. They play an important part, especially in providing solutions early in the search. In addition to providing valid upper bounds, high quality feasible solutions can help fathom part of the solution space during the search.

One of the most prominent primal heuristics is the feasibility pump [20], which is a widely used algorithm for finding feasible solutions at the beginning of the search. It consists of an iterative heuristic that combines linear program (LP) relaxations and roundings to enforce LP and integer feasibility until a feasible solution is found.

Other important heuristics, such as RENS [21](relaxation enforced neighborhood search), RINS [22](relaxation induced neighborhood search) or evolutionary algorithms [23] are based on Large Neighborhood Search (LNS) ideas. LNS heuristics entail solving restricted sub-MIPs derived from the original problem. Due to the imposed limitations, these sub-problems are much easier to solve in comparison, which allows the finding of solution improvements faster. Each of the mentioned LNS heuristics differ in how these sub-MIP restrictions are defined.

A vast catalogue of primal heuristics has been proposed for MIPs. All of them can vary dramatically in complexity and algorithmic design. LNS-based heuristics are some of the most computationally expensive heuristics, since they require the optimization of subMIPs. At the same time, the design of primal heuristics is often based on the MIP practitioner's intuition, and they usually do not offer a theoretical guarantee that better solutions will be found.

1.3 The need for parallel algorithms in High Performance Computing

Since the end of Dennard scaling in CPUs, little performance and energy-efficiency gains have come by the hand of traditional CPU design. Processor clock speeds cannot be increased using current chip manufacturing processes, and this has forced manufacturers to design computer architectures with increasing parallelism in order to improve the theoretical peak CPU performance. Realizing a nontrivial fraction of these theoretical performance improvements requires algorithms that leverage parallelism effectively. This trend has only been exacerbated for high-performance computing (HPC) systems used in large-scale scientific and engineering applications, with recent HPC systems enabling million-process parallelism, and future systems enabling even larger process counts and a peak performance nearing the exaflop. High performance algorithm design demands decomposable approaches to problem solving and scalable solutions.

1.3.1 Parallel algorithms for MIP optimization

The algorithmic improvements presented in this section help to reduce the number of necessary operations required for solving a MIP problem to optimality. On the other hand, the introduction of parallelism seeks to harness the power of modern processor architectures in order to speed up the processing of these operations. Developments in both facets are independent, but they both contribute to improving the effectiveness of MIP solving.

In the field of discrete optimization, Bader et al. [24] and Koch et al. [25] discuss potential applications of parallelism.

Branch-and-bound presents a natural avenue of parallelization, since the solving of the different subproblems is independent and, therefore, can be processed in parallel. Most parallel MIP solvers in existence incorporate this kind of parallelization. Despite the apparent abundance of parallelism, modern implementations must consider and overcome many practical challenges in order to achieve a high degree of parallel efficiency. Ralphs et

al. [26] discuss these challenges in depth, and provide an extensive overview of the different parallel branch-and-bound implementations presented to date.

In addition to the parallelization of the tree search, there are many computationally expensive components within branch-and-bound that could benefit greatly from parallelization, such as branching or the LP relaxation. In this thesis, for example, we present multiple variations of parallel primal heuristics, which can be offloaded to auxiliary processors running independently from the main optimization. Alternatively, parallelism could benefit cutting plane generation, where multiple processors could be employed for solving multiple separation problems, which could yield a tighter strengthening. Little work has been done for speeding up these auxiliary components, and remains an open field of research.

1.4 Outline of the dissertation

In this dissertation we present multiple approaches in which MIP solving can benefit from parallelism. We present a portfolio of parallel heuristics, which look at improving the effectiveness of the solver from the primal point of view. We then transition to the improvement of the lower bound, with the introduction of alternative parallelizations of branch-and-bound.

This dissertation is organized as follows: chapter 2 introduces a parallel local search approach for obtaining high quality solutions to the Fixed Charge Multicommodity Network Flow problem (FCMNF), a classic optimization problem used in many applications, including circuit design and transportation networks. The approach proceeds by improving a given feasible solution by solving restricted instances of the problem where flows of certain commodities are fixed to those in the solution while the other commodities are locally optimized. We derive multiple independent local search neighborhoods from an arc-based mixed integer programming (MIP) formulation of the problem which are explored in parallel. Our scalable parallel implementation takes advantage of the hybrid memory architecture in modern platforms and the effectiveness of MIP solvers in solving small problems

instances. Computational experiments on FCMNF instances from the literature demonstrate the competitiveness of our approach against state of the art MIP solvers and other heuristic methods. The presented method proves to be more effective at finding substantially better primal solutions when compared to state-of-the-art commercial MIP solvers, even when the latter are allowed five times as much time. This chapter is based on the work presented in Munguía et al. [27].

In chapter 3, we generalize the notions introduced in the previous chapter in Alternating Criteria Search: a parallel large neighborhood search framework for finding high quality primal solutions for general mixed-integer programs (MIPs). The approach simultaneously solves a large number of sub-MIPs with the dual objective of reducing infeasibility and optimizing with respect to the original objective. Both goals are achieved by solving restricted versions of two auxiliary MIPs, where subsets of the variables are fixed. In contrast to prior approaches, ours does not require a feasible starting solution. We leverage parallelism to perform multiple searches simultaneously, with the objective of increasing the effectiveness of our heuristic. We computationally compare the proposed framework with a state-of-the-art MIP solver in terms of solution quality, scalability, reproducibility, and parallel efficiency. Results show the efficacy of our approach in finding high quality solutions quickly both as a standalone primal heuristic and when used in conjunction with an exact algorithm. Our parallel method is able to produce competitive or better and faster results for more than 90% of the tested instances against CPLEX. This chapter is based on the work presented in Munguía et al. [28].

In chapter 4, we focus on how to quickly develop effective parallel primal heuristics tailored for a specific problem domain. We use the Maritime Inventory Routing Problem (MIRP) as an example, an important application of MIP to real world problems. Our approach entails specializing Parallel Alternating Criteria Search to better fit the problem structure found in MIRPs. Parallel Alternating Criteria search's efficacy relies on the combination of computer parallelism and Large Neighborhood Searches. Comparisons against

state-of-the-art problem specific heuristics and commercial MIP solvers show the effectiveness of our approach, and how the modular nature of Parallel Alternating Criteria Search can provide an excellent platform for rapid prototyping of parallel domain-specific heuristics. In the presented results, tailored versions of our parallel algorithm are able to significantly outperform domain-specific state-of-the-art heuristics and parallel MIP solvers alike.

Chapter 5 is dedicated to the introduction of PIPS-SBB, a parallel distributed-memory solver for deterministic equivalent two-stage stochastic MIPs (sMIPs). SMIPs deal with optimization under uncertainty at many levels of the decision-making process. When solved as extensive formulation MIPs, problem instances can exceed available memory on a single workstation. To overcome this limitation, we present a MIP solver that takes advantage of parallelism at multiple levels of the optimization process. We show promising results on the SIPLIB benchmark by combining methods known for accelerating branch-and-bound methods with new ideas that leverage the structure of SMIPs. This chapter is based on the work presented in Munguía et al. [29].

In chapter 6, we present two parallelizations of branch-and-bound for PIPS-SBB, extending the already parallel solver. In the first of the proposed frameworks, PIPS-PSBB, the coordination and load-balancing of the different optimization workers is done in a decentralized fashion. This new framework is designed to ensure all available cores are processing the most promising parts of the branch-and-bound tree. The second, ug[PIPS-SBB,MPI], is a parallel implementation using the Ubiquity Generator (UG), a universal framework for parallelizing the branch-and-bound tree search that has been successfully applied to other MIP solvers. We show the effects of leveraging multiple levels of parallelism in potentially improving scaling performance beyond thousands of cores, and illustrate our ideas by implementing both frameworks as extensions of PIPS-SBB. We also compare our algorithms against the distributed-memory implementation of the state-of-the-art commercial solver CPLEX. The latter proves to be the best performer at small problem scales.

However, the specialized nature of the methods present in PIPS-SBB-based solvers allow them to outperform CPLEX in large SMIP instances.

Finally, in chapter 7 we summarize our contributions, and provide some possible future research directions.

CHAPTER 2

A PARALLEL LOCAL SEARCH FRAMEWORK FOR THE FIXED-CHARGE MULTICOMMODITY NETWORK FLOW PROBLEM

2.1 Introduction

The Fixed-Charge Multicommodity Network Flow (FCMNF) problem is a classic optimization problem arising in numerous applications. Given a directed capacitated network and a set of commodities, the objective is to route every commodity from its origin to destination through the network so as to minimize the total cost. The cost associated with an arc is the sum of a fixed cost derived from its use and a variable cost proportional to the flow going through it. The total cost is derived from the sum of all arc costs.

The FCMNF problem was proven to be NP-Hard [30]. In practice, realistic sized instances of the FCMNF problem are extremely difficult to solve to optimality. Consequently a variety of heuristic approaches and integer programming techniques have been developed and proven to be effective means to achieve high quality solutions quickly. We introduce a local search heuristic framework for the FCMNF problem that is explicitly designed for both parallel shared-memory systems and distributed-memory systems. Our method finds competitive solutions by exploring a large number of local search neighborhoods concurrently. Given a feasible solution s , the local searches proceed by solving restricted instances of the problem where flows of certain commodities are fixed to those in the solution s while that of the other commodities are optimized. We take advantage of a state-of-the-art Mixed Integer Programming (MIP) solver to drive these local searches.

Recent works have introduced successful heuristic methods for obtaining high quality solutions. Most common heuristics consist of embedding a problem-specific mechanism for improving solutions in the context of a metaheuristic search framework. Ghamlouche

et al. [31] identify cycles in the network as a heuristic strategy for finding alternative flow routes. The same methodology is used in further works in combination with machine learning techniques in order to improve and guide the local search [32]. Chouman et al. [33] use a similar approach to identify arc-balanced cycles in combination with a Tabu Search. A different heuristic approach is presented by Yaghini et al. [34], where the authors define local search neighborhoods based on simplex pivots in the context of a simulated annealing framework. Other meta-heuristic frameworks for the FCMNF problem are based on Evolutionary algorithms [35] and Scatter Search procedures [36, 37, 38]. The latter works were developed more than a decade ago and differ in the generation of the original population, and the mechanisms used for solution improvement and recombination.

Heuristic strategies can also be used in the context of an exhaustive search framework. An example is the local branching technique introduced by Fischetti et al. [39]. Their method uses linear inequalities to branch on smaller subproblems, which are solved by a black-box MIP solver. Examples of applications of local branching for the FCMNF problem are studied by Rodríguez-Martín et al. [40]. The efficacy of the previously cited work resides in the use of heuristics algorithms in combination with exact mixed-integer programming techniques.

Katayama et al. [41] develop a column generation, path-based formulation enhanced by strong inequalities in conjunction with an arc capacity scaling approach. In [42], the same scheme is improved by using local branching ideas to polish the solutions obtained through arc capacity scaling strategies.

Despite providing high quality solutions quickly, heuristic methods cannot provide optimality certificates because of their exclusive focus on primal solutions. Hewitt et al. [43] introduce an algorithm that provides lower bounds on the optimal solution in addition to primal solution improvements. Such improvements are found by solving strategically restricted MIP subproblems while tighter lower bounds are found with mathematical programming approaches. In further work, their approach combines the use of restricted MIPs

in the context of a branch-and-price framework that also provides a performance guarantee upon completion [44]. The authors take advantage of parallelism to solve the pricing problems and restrictions.

Parallelizations of large neighborhood search algorithms have been successfully implemented in other applications such as the LNG inventory routing problem [45]. To our knowledge, parallel computing remains a relatively unexplored field for the FCMNF problem. Crainic et al. [46] propose an asynchronous parallel Tabu Search where every processor communicates with a centralized solution pool. They introduce and test several communication policies as well as strategies for handling the exchanged information. In [47], special emphasis is put on the control of the information diffusion between the different processors. The authors present a multilevel parallel local search algorithm that employs parallel cycle-based Tabu Searches defined by sets of fixed arcs. Their approach differs greatly from ours in many aspects. These include the solution improvement method used, the fact that our method has a solution recombination step, the arrangement and synchronization of parallel resources, the communication protocol, and the information exchanged between processors. Crainic et al. [48] provide a comprehensive literature review on the application of parallelism in meta-heuristics. Our contribution is a highly scalable parallel algorithm specifically designed to find quality primal solutions of large-scale FCMNF problems. Many algorithmic enhancements are combined in order to attain competitive levels of parallel performance: a novel parallel decomposition procedure based on the problem structure, a highly parallelizable local search scheme, and a tiered parallel procedure that is able to combine large numbers of partial solutions quickly. Solution crossover methods such as the one used in our approach have already been introduced and discussed previously [3, 23]. In contrast to these works, we introduce a parallelization of the method that enables the recombination of a large number of solutions simultaneously.

We present experimental results that show the effectiveness of our parallel local search approach. For the instances in the C problem set [49], our method identifies primal solu-

tions that are within an average optimality GAP of 0.58% with respect to the best known lower bound in an average time of 152 seconds per instance. We also test our parallel algorithm against the GT problem set [43], which contains substantially bigger instances. Our method takes less than 200 seconds on average to obtain a better solution than the best one found by CPLEX running for 5 hours. We are able to identify considerably better solutions in more time. In addition, we present parallel scalability and load balancing performance results, which show that our novel implementation is able to take advantage of a large number of parallel processors to effectively reduce computation times in a load balanced execution.

The remainder of the chapter is structured as follows. Section 2.2 presents an arc based MIP formulation of the FCMNF problem. Sections 2.3 and 2.4 provide a detailed description of our local search methodology and its parallel implementation on hybrid-memory parallel architectures, respectively. Section 2.5 presents computational experiments and results on standard instances from the literature. Finally, Section 2.6 provides some concluding remarks.

2.2 Problem description

Our local search approach is based on an arc-based MIP formulation of the FCMNF problem, which is described as follows. Let $G = (V, A)$ be a directed network, where V is the set of vertices and A the set of arcs. Let K be a set of commodities to be routed through G . Each commodity $k \in K$ is specified by a source vertex $s_k \in V$, a destination $t_k \in V$ and a quantity q_k of flow to be routed. Each arc $(i, j) \in A$ has an associated fixed cost f_{ij} that is imposed only when commodities are routed through it. Arcs also have a variable cost c_{ij} that is proportional to the flow traversing it and a maximum flow capacity u_{ij} . The problem consists of finding a routing for every commodity in K such that the arc capacities are respected and the costs are minimized. Let the flow variable x_{ij}^k denote the proportion of commodity $k \in K$ that is routed through the arc $(i, j) \in A$. In addition to the flow

variables, we also introduce the binary variables y_{ij} , which reflect whether each arc (i, j) is used. The FCMNF problem can then be formulated as the following MIP:

$$\min_{x,y} \sum_{k \in K} \sum_{(i,j) \in A} c_{ij} q_k x_{ij}^k + \sum_{(i,j) \in A} f_{ij} y_{ij} \quad (2.1)$$

subject to:

$$\sum_{(i,j) \in A} x_{ij}^k - \sum_{(j,i) \in A} x_{ji}^k = d_i^k \quad \forall i \in V, \forall k \in K \quad (2.2)$$

$$\sum_{k \in K} q_k x_{ij}^k \leq u_{ij} y_{ij} \quad \forall (i, j) \in A \quad (2.3)$$

$$y_{ij} \in \{0, 1\} \quad \forall (i, j) \in A \quad (2.4)$$

$$0 \leq x_{ij}^k \leq 1 \quad \forall (i, j) \in A, \forall k \in K. \quad (2.5)$$

Restriction (2.2) ensures the conservation of flow. The flow differential for a vertex i and a commodity k is expressed by d_i^k , which is defined as:

$$d_i^k = \begin{cases} 1 & \text{if } i = s_k \\ -1 & \text{if } i = d_k \\ 0 & \text{otherwise.} \end{cases}$$

The coupling constraints (2.3) guarantee that the flow through each arc does not exceed the arc capacity. The capacity restrictions have a two-fold function, as they also ensure that the fixed cost is imposed when an arc is used. All commodity flow variables relative to the same arc are aggregated in the same constraint. A tighter and stronger LP relaxation can

be obtained by introducing a set of $|A| \cdot |K|$ independent constraints:

$$x_{ij}^k \leq y_{ij} \quad \forall (i, j) \in A, \forall k \in K. \quad (2.6)$$

These are redundant with respect to (2.3). We choose not to include them in our model due to performance issues resulting from their large number.

We consider two problem variations that differ in whether each commodity may be split through multiple paths or not. In the formulation above, the flow variables are continuous, as specified in constraint (2.5). Alternatively, each variable x_{ij}^k is binary and (2.5) is replaced by:

$$x_{ij}^k \in \{0, 1\} \quad \forall (i, j) \in A, \forall k \in K. \quad (2.7)$$

The techniques described in this chapter are compatible with both problem variants.

2.3 Local search methodology

In the proposed local search scheme, an initial primal solution to an FCMNF instance is improved iteratively by sequentially applying heuristic local searches. In each heuristic local search, solutions are improved by solving a smaller, tractable MIP subproblem that is derived from the original instance. Such reduction in the problem size is obtained by fixing a chosen subset of the variables to the corresponding values of the previously obtained feasible solution. The selection of variables is such that arc sharing is encouraged to reduce costs. New improved solutions replace the primal incumbent after each iteration, and the scheme is repeated.

We parallelize this procedure by partitioning the sequence of local searches to an arbitrary number of independent sequences, each of which can be explored in parallel. To achieve this decomposition, we introduce a local search partitioning mechanism, which determines the work to be performed by each of the parallel processors. Potentially, each independent subproblem sequence can produce an improved primal solution. The final step

in the algorithm combines the solution improvements found in parallel into a single feasible solution using a recombination scheme. This parallel procedure may be repeated an arbitrary number of times by using the obtained solution as an input for the next iteration. We depict this parallel local search procedure in Figure 2.1. Next, we describe each component of the overall method in detail.

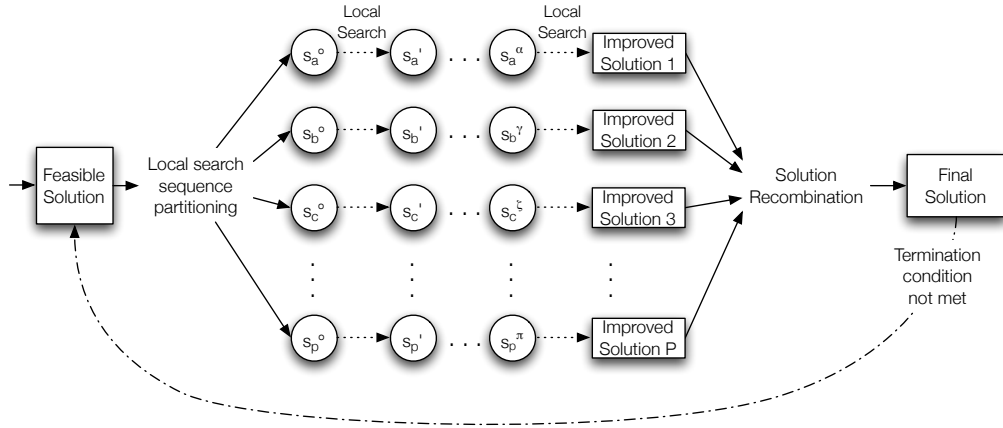


Figure 2.1: Parallel decomposition of a sequential local search procedure. Local searches are distributed in a total of P sequences. As a result, P feasible solutions are obtained and recombined in an improved solution.

2.3.1 The local search mechanism

At the most basic level of the algorithm, improvements in solutions are found by solving restricted instances of the original problem, in which flows of certain commodities are fixed. Given a commodity c and a feasible solution s , we define the set of adjacent commodities $Adj(c)$ as the group of commodities that share flow at least in one arc with c in s , i.e., $Adj(c) = \{c' \in K | \exists (i, j) \in A \text{ s.t. } x_{(i,j)}^c > 0 \text{ and } x_{(i,j)}^{c'} > 0\}$. A local search neighborhood is then defined in the form of a new MIP subproblem, where the flow of the adjacent commodities and c are free and the remaining flow x is taken from s and fixed. Simultaneously, the arc use variables y are also free for arcs that are not used by the remaining flow. A pseudo-code description of the local search procedure is given in Algorithm 3.

Depending on the selected commodity, the resulting subset of fixed variables may vary

Algorithm 3 Local neighborhood search

Input: Feasible solution s and commodity c from a FCMNF instance

Output: Feasible solution to the FCMNF instance

```
function LOCALSEARCH(Solution  $s$ , Commodity  $c$ )  
  Compute  $Adj(c)$  based on  $s$   
  for all commodities  $k$  not in  $Adj(c)$  do  
    Fix the flow variables  $x_{ij}^k$  to the values in  $s$ ,  $\forall (i, j) \in A$   
  end for  
  SOLVESUBPROBLEM()  
  return BESTSOLUTION()  
end function
```

in size. Commodities with a large number of adjacent commodities produce difficult MIP subproblems due to the small number of variable fixings. In contrast, commodities with little flow interaction may yield excessively restricted local searches. Disparities in the instance size can be reduced by establishing a threshold on the number of variables that can be fixed in addition to an optimization time limit.

2.3.2 Partitioning the subproblem sequence

Given the above local search neighborhood definition and a feasible solution to a FCMNF instance, we can define as many different variable fixings as commodities in the instance. A small example is shown in Figure 2.2. Each derived MIP subproblem is characterized by a specific commodity and its adjacent flow and can be optimized in parallel.

Consider a set of local searches to be explored, each of which is identified by a specific commodity. As a first step towards parallelization, we require such work to be decomposed into a set of disjoint local search subsets. Work partitions that yield load-balanced optimizations are highly desirable. As an additional requirement, we incentivate that commodities with heavy flow interaction should be placed in the same subset. With this specific grouping, we would expect each local search subset to correspond to a highly interrelated subset of the variables.

The partitioning problem can be transformed into a graph partitioning problem as follows. A new weighted graph $G^s = (V^s, A^s)$, called the connection graph, is determined

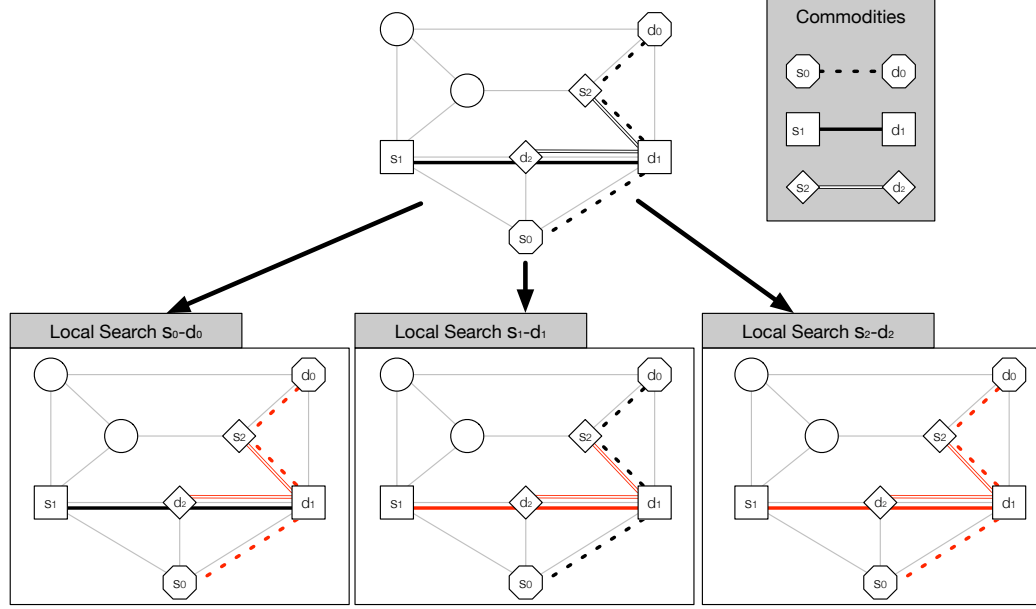


Figure 2.2: Subproblem partitioning shown on a small example with three commodities. A subproblem is defined for each commodity, where the flow highlighted in red is optimized and the remaining flow is fixed.

from a feasible solution s , the set of arcs A and the set of commodities K . In G^s , the set K represents the vertices: $V^s = K$. $A^s = \{(u, v) | u, v \in V^s \text{ and } \exists(i, j) \in A \text{ s.t. } x_{(i,j)}^u > 0 \text{ and } x_{(i,j)}^v > 0\}$, i.e., there is an arc (u, v) in $G^s(V^s, A^s)$ if and only if commodities u and v share flow in an arc in s . In addition, we specify the weight of an edge $(u, v) \in A^s$ to be the number of shared arcs. An example is depicted in Figure 2.3.

Thus, the division of commodities among P parallel processors can be translated into a graph partitioning problem of the connection graph, where the cut between the P different subsets is minimized. The connection graph partitioning can be accomplished with minimal computational efforts by specialized graph partitioning algorithms, including the Kernighan-Lin method [50], which are available in graph partitioning libraries such as Metis [51].

Algorithm 4 Solution recombination algorithm

Input: Feasible solution set S

Output: Feasible solution to the FCMNF instance with an accumulation of the improvements.

function SOLUTIONRECOMBINATION(*Solution set* S)

for $(i, j) \in \text{Arcs}$ **do**

if (i, j) is not used in any $s \in S$ **then**

 Fix variable y_{ij} to 0

end if

end for

Add all $s \in S$ as starting solutions

SOLVESUBPROBLEM()

return BESTSOLUTION()

end function

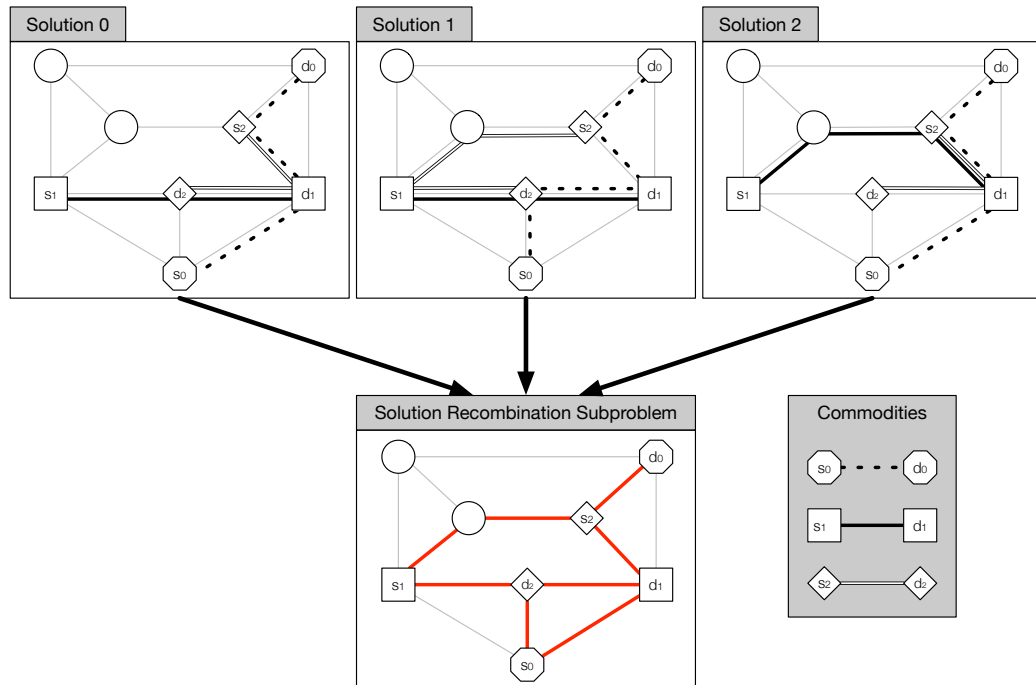


Figure 2.4: The solution recombination step is shown on a small example with three commodities. A subproblem is defined for a set of input solutions, where the flow in the arcs highlighted in red is optimized while the remaining arcs fixed.

Similarly to our heuristic local search scheme, the subset of fixed variables in each

solution recombination may vary in size. Solutions in the earlier stages of the computation may incorporate many changes in the routing, resulting in a difficult MIP subproblem due to the small number of arc fixings. The opposite effect may be obtained if little improvement is found during the local search phase. We resolve the differences in the problem size by specifying an optimization time limit.

2.3.4 Obtaining a first feasible solution

The parallel method presented in this section relies on an original feasible solution to improve upon. A starting solution is produced by solving a relaxation of the original problem, which is obtained by removing the fixed costs from the objective function computation. The fixed costs are a major complicating factor in solving the FCMNF problem. When they are omitted we have an LP if the flow can be split and an IP otherwise. We use these simplified models to obtain preliminary primal solutions that satisfy the flow restrictions, although the solutions are far from being optimal. In our experience, however, the quality of the first feasible solution has proven to be unimportant because great progress is always achieved in the initial iterations of the scheme.

2.4 Parallel implementation

In this section, we present more details of the algorithm implementation. For parallel scalability, our scheme is designed for hybrid memory systems that combine both distributed-memory systems as well as parallel shared-memory machines. Throughout the section, we refer to a computing node (or processor) as a set of multiple CPU cores that share a single, unified memory subsystem as shown in Figure 2.5a (*shared-memory system*). For scalability, parallel execution may take place across several computing nodes concurrently. In this case, the memory space is segmented and distributed between the individual processors as depicted in Figure 2.5b (*distributed-memory system*). As such, a collection of parallel processors constitute a parallel distributed-memory system. Efficient implementations require

algorithm design techniques tailored for each memory environment. An important distinction in the implementation is found in the synchronization of parallel components. Parallel distributed-memory systems usually rely on synchronous message passing techniques such as MPI [52] to perform communications between processors. In contrast, communication between the parallel cores in a single computing node is much simplified because memory is shared.

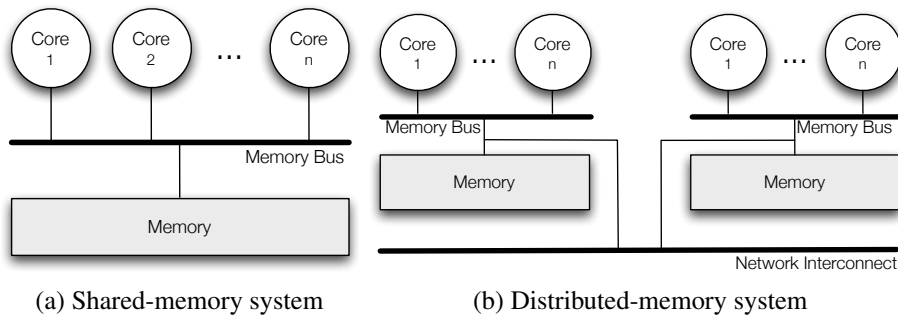


Figure 2.5: Schematic depiction of parallel systems with different memory configurations. System (a) features a unified memory space, which is accessible by every parallel core simultaneously. In contrast, system (b) has a segmented memory space, which is distributed between the different parallel cores.

We start describing our parallel scheme by its implementation in a single shared-memory parallel processor as shown in Algorithm 5. Consider a processor with C parallel processor cores and a starting solution s that is initially given to each of them. Each core proceeds to improve its local solution in parallel by resolving a different sequence of MIP subproblems and accumulating the improvements found in the optimization process. Given that the parallel cores share the same memory space, the work partitioning can be arranged dynamically. In order to do so, we employ a shared data structure which holds the subproblems that every core has access to. When a subproblem is solved, it is removed from the shared set. In this fashion, subproblems are distributed dynamically among the parallel cores such that the routing of each commodity is optimized by exactly one of them.

Each variable fixing yields a smaller integer problem as previously described, which is then solved using a black-box MIP solver. As a product of the parallel local search,

Algorithm 5 Parallel local search iteration

Input: Feasible solution s from a FCMNF instance, set of commodities K

Output: Improved feasible solution to the FCMNF instance

```
function PARALLELLOCALSEARCH(Solution  $s$ , Commodity set  $K$ )  
  for every thread  $t_i \in C$  in parallel do  
    Initialize  $Sol_{t_i}$  as a copy of  $s$   
    while there exists commodity  $k \in K$  do  
      Remove  $k$  from  $K$   
       $newSol = \text{LOCALSEARCH}(Sol_{t_i}, k)$   
      if  $newSol$  represents improvement over  $Sol_{t_i}$  then  
         $Sol_{t_i} = newSol$   
      end if  
    end while  
  end for  
  return SOLUTIONRECOMBINATION( $Sol$ )  
end function
```

improvements in the routing are accumulated in at most C different solutions. We employ our solution recombination step to combine them to a single feasible solution, which may be used as input to the next iteration. To ensure full system utilization, we may assume the number of commodities given by $|K|$ to be greater than the overall number of parallel cores P . When P is bigger than $|K|$, only $|K|$ parallel cores are used during the local search phase. The algorithm is designed such that each parallel core may improve its local solution copy by sequentially solving multiple MIP subproblems and replacing the solution when improvements are found.

We adapt our implementation to a hybrid memory parallel system by augmenting Algorithm 5 in a two-layered scheme, where each level is dedicated to a different level of parallelism. The first layer is responsible for the local search partitioning between the different distributed-memory processors by the use of MPI. The second execution tier takes place in the shared-memory setting of each parallel processor and is responsible for the actual local search exploration. It is in this inner execution level where the MIP subproblems are collectively resolved by the parallel cores in each processor.

Figure 2.6 describes the interactions between each tier and the workflow of the execution. As a first step, the set of MIP subproblems is partitioned and distributed among the

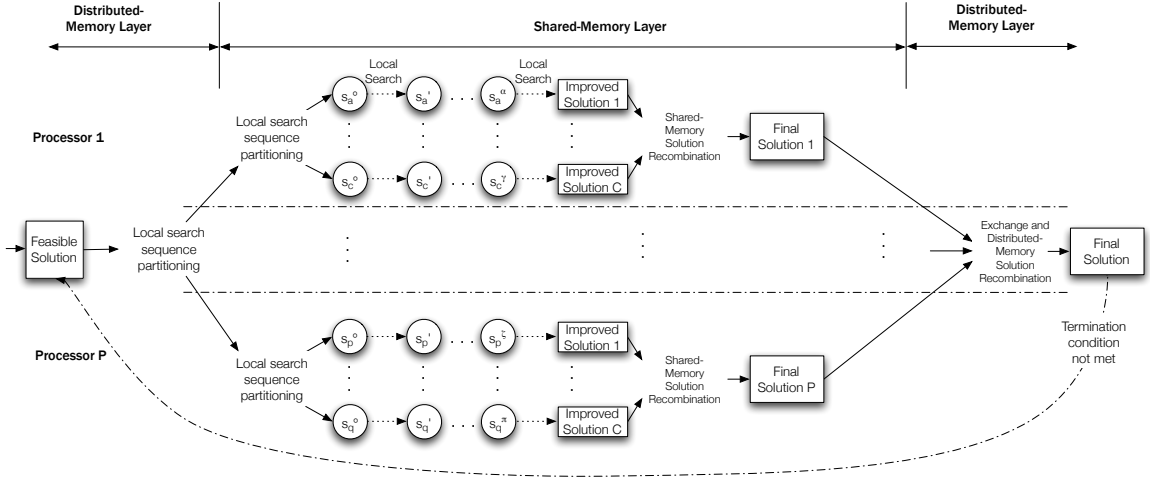


Figure 2.6: Parallel hybrid memory framework. The process described in Figure 2.1 is expanded to accommodate the parallel execution in distributed-memory systems.

processors. After the partitioning, each computing node proceeds to find improvements in the primal solution by the same procedure as shown in Algorithm 5. The execution returns to the distributed-memory context, where an all-to-all solution exchange communication is performed in order to combine the improvements found by the different processors. In Algorithm 6 we present a pseudo-code description of the distributed-memory execution layer.

Algorithm 6 Distributed-memory parallel local search framework

Input: Feasible solution s from a FCMNF instance, set of commodities K

Output: Improved feasible solution to the FCMNF instance

function DISTRIBUTEDLOCALSEARCH($Solution\ s$)

while termination criteria is not met **do**

 Let $Part$ be a partition of the commodity set K in P subsets based on s

for all processors P_i in Parallel **do**

$Solution_{P_i} = \text{PARALLELLOCALSEARCH}(s, Part_{P_i})$

$AllProcSolutions = \text{ALL-TO-ALLEXCHANGE}(Solution_{P_i})$

end for

$s = \text{SOLUTIONRECOMBINATION}(AllProcSolutions)$

end while

return s

end function

A distributed-memory implementation raises the issue of scalability. The effectiveness of the recombination step, for instance, is dependent on the input size. If the number of

input solutions is small and all of them are highly similar, the resulting recombination MIP may be small and relatively easy to solve. However, a large number of input solutions may produce a very small number of fixings and, therefore, a problem that may be hard to optimize quickly. Thus, its scalability may be limited.

By splitting the recombination process into two consecutive phases, a large number of input solutions can be accumulated while maintaining a large number of fixings. An additional benefit lies in a better utilization of the parallelism, since each processor performs the first phase of the recombination independently.

In addition to scalability, load balancing is another component of parallel efficiency. Load balancing refers to the uniform distribution of work among parallel processors. Due to the synchronous nature of our algorithm, achieving an even load balance is essential in order to maximize the throughput of our parallel computations. Figure 2.7 depicts all the potential load imbalance pitfalls of our parallel implementation. First, the parallel cores within a processor may compute each of the local search sequences unevenly. Additionally, load imbalance could be aggravated by differences in the solving time of the first solution recombination. Both factors could affect the synchronization between processors, potentially delaying the second phase of the solution recombination. Its prevention depends on how the work is partitioned and the time limit parameters that decide the granularity of the local searches as well as the recombination times. Further experiments presented in the next section will determine the degree of load imbalance of our approach.

Another positive aspect of the synchronous implementation is an efficient handling of the communications between parallel processors. Overall, only two communication steps are required every iteration. A communication of the work partitions is performed prior to the local search, as well as an all-to-all exchange of solutions between each of the recombination phases. In both cases, collective communication primitives can be used in order to reduce overhead.

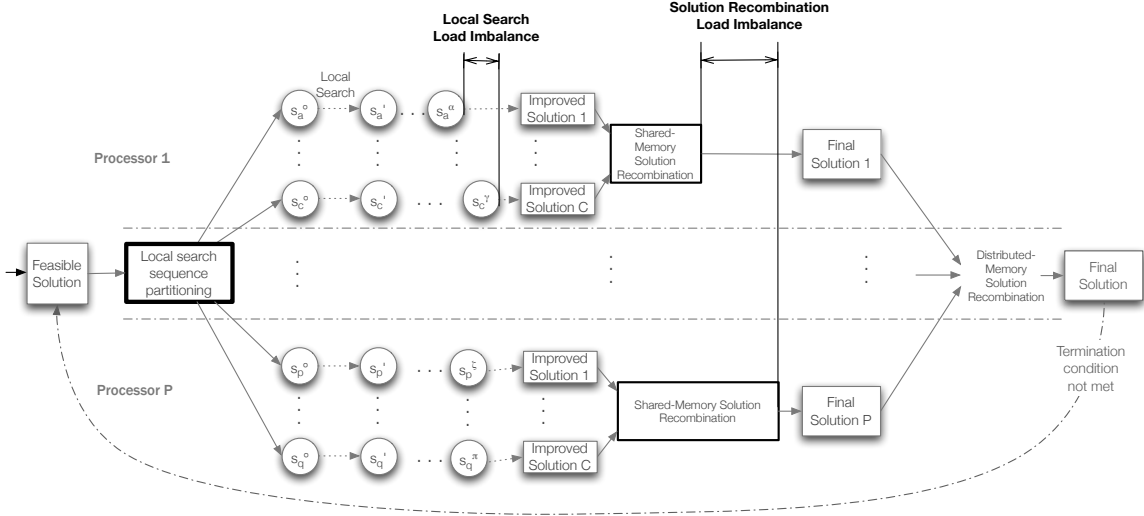


Figure 2.7: Potential load imbalance causes of the implementation.

2.5 Experimental results

In this subsection, we study the performance of our parallel local search in terms of solution quality, parallel scalability and load balance. Our framework is implemented in C++ and uses CPLEX 12.4 as the MIP solver. Our tests were performed on an 8-node computing cluster, with each node having two Intel Xeon X5650 6-core processors, 24 GB of RAM memory and a Red Hat Enterprise Linux distribution. Unless otherwise noted, CPLEX was set to its default configuration whenever it was used for comparison purposes. CPLEX is configured with 12 threads, as it can take advantage of parallelism in shared-memory machines. We test the competitiveness of our parallel method by comparing its performance against previous heuristic approaches presented in the literature. For this purpose, two FCMNF problem instance sets are used, the C instances [49] and the GT instances [53] used in [43]. In order to assess the scalability of our parallel method, we test the performance under different processor configurations.

The choice of the local search parameters such as the search time limit can have a significant impact on the effectiveness of the method. An ideal combination is highly dependent on the instance size, the number of arcs and the number of commodities. To cope with this

variety of choices, we test several parameter configurations and choose the best performing one in average. For each problem class, 8 representative instances were selected and sampled with a collection of parameters. In Figure 2.8, we report the performance results for both the C instances and the GT instances in terms of the average optimality GAP. Each algorithm sample had a time limit of 100 seconds and 300 seconds for the C and the GT instances respectively. For the remaining set of experiments in the C instance set, the time limits were set to 10 seconds for each local search and 20 seconds for solution recombination. When solving the GT instances, the local search time limit was set to 20 seconds and 50 seconds were allowed for the recombination process.

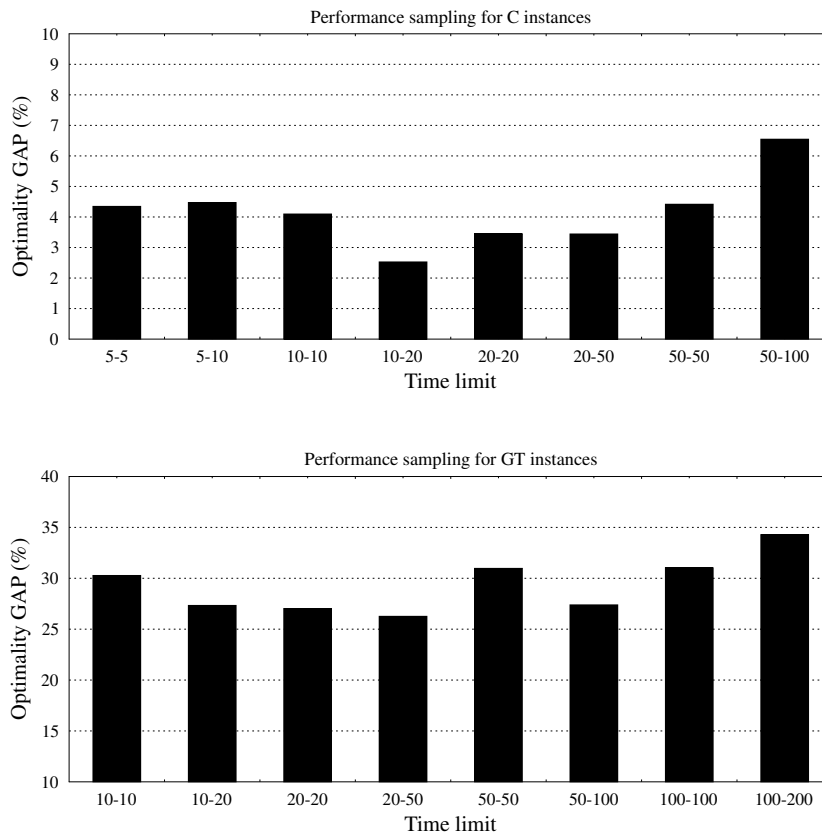


Figure 2.8: Parameter sensitivity analysis for C and GT instances. Parameter configurations are specified with two numbers X-Y, where X refers to the local search time limit and Y to the solution recombination time limit.

2.5.1 C instance set performance results

The C instance set is comprised of 37 FCMNF medium-sized instances. They have networks with 20 or 30 vertices, a number of arcs ranging from 230 to 700 and a number of commodities ranging from 10 to 400. The configuration of each problem instance is specified with the tuple $\{Nvertices, NArcs, NCommodities, F/V, T/L\}$, where F indicates that the instance’s fixed costs are predominant in relation to the variable costs (V otherwise). T characterizes a tightly capacitated problem instance and L denotes loose arc capacities. In order to evaluate the quality of the solutions obtained with our scheme, we compare it against the results presented in prior publications and default CPLEX on the problem variant where the flow routing can be split between different paths. Specifically, results are compared with those reported in the IP Search scheme (*IPSearch*) by Hewitt et al. [43], two sets of results obtained with the capacity-scaling combined scheme (*Comb1* and *Comb2*) by Katayama [42], and CPLEX with a time limit of 1 hour. We refer to the results obtained with our Parallel Local Search as *ParLS*.

The performance results over the C instance set is shown in Table 2.1, where the best lower bound for each instance is reported followed by the best primal solution found by each method. The best values are denoted in bold. When a value is optimal, it is marked with an asterisk. We specify the time required to reach the best solution as reported originally by the authors. In order to eliminate the discrepancies between computer systems, we also report the normalized CPU times for *IPSearch*, *Comb1*, and *Comb2* according to the CPU performance metrics in Passmark[54]. Normalized times are calculated as $T_{norm} = \frac{T_{orig} \cdot S_{orig}}{S_{norm}}$, where T_{orig} is the original time reported by the authors, while S_{norm} ¹ and S_{orig} ²³ are the CPU scores of the processors used in our experiments and the other authors in the comparison respectively. The lower bounds for each problem were either

¹ $S_{norm} = 7605$ (Intel Xeon X5650)

² $S_{Comb1} = S_{Comb2} = 8262$ (Intel Core i7-2600)

³ $S_{IPSearch} = 4452$ (Intel Xeon E5520)

Table 2.1: C instance set optimization results

Problem	Primal solution value				Time to best solution (s)				Normalized Time to best solution (s)							
	LB / Opt	IPSearch	Comb1	Comb2	CPLEX	ParLS	IPSearch	Comb1	Comb2	CPLEX	ParLS	IPSearch	Comb1	Comb2	CPLEX	ParLS
100/400/010/VL	28423*	28423*	28426	28423*	28423*	28486	35	0	2	1	3	20	0	2	1	3
100/400/010/FL	23949*	23949*	24459	23949*	24022	24022	9	23	106	112	1	5	25	115	112	1
100/400/010/FT	63066	65885	68410	64207	64143	64207	813	12	2736	3600	53	476	13	2974	3600	53
100/400/030/VT	384802*	384836	384809	384802*	384802*	384802*	330	2	1503	680	42	193	2	1634	680	42
100/400/030/FL	49018*	49694	49588	49018*	49018*	49018*	886	167	864	3600	29	519	182	939	3600	29
100/400/030/FT	132129	141365	142191	138152	138587	136861	888	12	11028	3600	79	520	13	11986	3600	79
20/230/040/VL	423848*	424385	423848*	423848*	424075	424075	4	0	1	1	2	2	0	1	1	2
20/230/040/VT	371475*	371779	371906	371475*	371573	371573	41	1	3	1	1	24	1	3	1	1
20/230/040/FT	643036*	643187	643649	643036*	643036*	643036*	45	1	15	5	10	26	1	16	5	10
20/230/200/VL	94213*	95097	94218	94213*	94218	94213*	822	104	3060	3600	123	481	113	3326	3600	123
20/230/200/FL	137642*	141253	137702	137642*	137854	137642*	691	254	4409	3600	144	405	276	4792	3600	144
20/230/200/VT	97914*	99410	97968	97914*	97914*	97914*	821	68	2161	606	52	481	74	2349	606	52
20/230/200/FT	135863	140273	136265	136031	136144	135867	156	263	4538	3600	240	91	286	4932	3600	240
20/300/040/VL	429398*	429398*	429398*	429398*	429398*	429398*	19	0	1	1	1	11	0	1	1	1
20/300/040/FL	586077*	586077*	587512	586077*	586077*	586077*	29	1	8	3	16	17	1	9	3	16
20/300/040/VT	464509*	464509*	464509*	464509*	464509*	464509*	24	1	4	1	23	14	1	4	1	23
20/300/040/FT	604198*	604198*	604198*	604198*	604198*	604198*	68	1	4	1	3	40	1	4	1	3
20/300/200/VL	74753	75319	74840	74811	74929	74811	802	101	5048	3600	361	470	110	5487	3600	361
20/300/200/FL	113862	117543	115801	115748	115541	115580	686	350	7769	3600	250	402	380	8444	3600	250
20/300/200/VT	74991*	76198	74995	74991*	74991*	74991*	388	65	2158	791	58	227	71	2345	791	58
20/300/200/FT	106672	110344	107315	107315	107102	107102	396	258	3798	3600	122	232	280	4128	3600	122
30/520/100/VL	53958*	54113	53976	53958*	53958*	53978	218	7	673	721	20	128	8	731	721	20
30/520/100/FL	93570	94388	94201	93967	93967	93967	226	150	3266	3600	83	132	163	3550	3600	83
30/520/100/VT	52046*	52174	52248	52046*	52046*	52046*	455	7	2004	3600	32	266	8	2178	3600	32
30/520/100/FT	96260	98883	97833	97385	97107	97862	815	4992	4802	3600	158	477	5426	5219	3600	158
30/520/400/VL	147790	154218	149486	149423	149242	149677	394	95	5917	3600	542	231	103	6431	3600	542
30/520/400/FL	114641*	114922	114641*	114641*	114641*	114641*	750	1184	3387	3600	463	439	1287	3681	3600	463
30/520/400/VT	150685	154606	152630	152576	153005	154137	621	54	3726	3600	461	364	59	4050	3600	461
30/520/400/FT	47603*	47612	47603*	47603*	47603*	47603*	466	358	5149	3600	288	273	389	5596	3600	288
30/700/100/VL	59558*	60700	60067	59558*	60066	60058	32	4	64	18	179	19	4	70	18	179
30/700/100/FL	45872*	46046	46070	45872*	45872*	45879	741	228	2888	3600	111	434	248	3139	3600	111
30/700/100/VT	54904*	55609	55164	54904*	54904*	54904*	371	9	5559	3600	258	217	10	6042	3600	258
30/700/400/VL	97189	98718	97901	97875	97914	98090	387	26	4456	3600	173	227	28	4843	3600	173
30/700/400/FL	131690	152576	134723	134620	135892	136250	222	466	4727	3600	243	130	506	5138	3600	243
30/700/400/VT	94508	96168	95267	95250	95293	95651	860	2115	7312	3600	223	504	2299	7947	3600	223
30/700/400/FT	128243	131629	129910	129910	130140	131104	365	1570	6376	3600	374	214	1706	6930	3600	374
Average GAP and time		1.73	0.89	0.47	0.51	0.58	408	456	3180	2317	152	239	497	3457	2317	152

obtained from the literature or found by CPLEX with a 12-hour limit. The reported GAP values are relative to the optimal solution or to the best lower bound. It is computed as $GAP = \frac{P_{sol} - L_B}{P_{sol}} \cdot 100$, where P_{sol} is the solution to each instance and L_B its lower bound.

ParLS finds an optimal solution in 15 out of 37 instances. In comparison to the incumbents reported in the literature, better or equally good solutions are found in 20 cases. When we consider all 37 instances, we obtain solutions that are within an average optimality GAP of 0.58%. The convergence to such solutions is obtained very quickly, averaging 152 seconds per instance. Comparing our results to previous research is a difficult task due to the diversity in the experimental conditions and the differences in the hardware and software. However, our parallel method identifies quality solutions that are competitive with the ones reported in prior work and requires much less time to achieve them. The incumbents reported in *Comb2* and *CPLEX* are results that represent an improvement over those obtained with our approach. But the solution times are larger by a factor of 20 and 15 respectively.

2.5.2 GT instances

The GT set is comprised of 24 FCMNF instances, which range in the number of arcs from 2000 to 3000 and have 50 to 200 commodities. When an arc-based formulation is considered, the instance sizes range between 102000 and 603000 variables. They are additionally presented in two versions, whether the problem instances are tightly capacitated (F-T) or loosely capacitated (F-L). We compare the performance of our parallel local search (reported as *ParLS*) with the IP Search scheme from Hewitt et al. [43] (*IPSearch*), both with a time limit of one hour. We also compare the best results obtained by CPLEX with a time limit of 5 hours (*CPLX*). We tested several CPLEX emphasis configurations, including optimality and feasibility, and found the default configuration to be the best performer. In addition to the default setting, we include a comparison against the solution polishing heuristic of CPLEX. In the *CPLXSP* setting, CPLEX is allowed 1 hour of optimization

time and 4 hours of solution polishing. CPLEX is also used to determine the lower bound on every instance. Results are detailed in Table 2.2, where the best found primal solutions found are given, as well as the time required by the parallel local search to improve the best solution found by the other three methods. The reported GAP values are calculated with respect to the best found lower bound.

Table 2.2: GT instance set optimization results

Problem	LB	Primal solution value				Optimality GAP				Time to improve solution (s)		
		CPLX	CPLXSP	IP Search	ParLS	CPLX	CPLXSP	IP Search	ParLS	CPLX	CPLXSP	IPSearch
F.T,500,2000,50	4326550	5038580	5100186	4949780	4892012	14.13	15.17	12.59	11.56	114	100	178
F.T,500,2000,100	6368730	7592260	7381313	7619670	7273916	16.12	13.72	16.42	12.44	78	366	75
F.T,500,2000,150	7208800	8640390	9083303	8807650	8014986	16.57	20.64	18.15	10.06	317	155	234
F.T,500,2000,200	8845440	11858000	11213371	11893100	10617796	25.41	21.12	25.63	16.69	257	463	257
F.T,500,2500,50	3927990	4585510	4448739	4600200	4406080	14.34	11.71	14.61	10.85	72	120	72
F.T,500,2500,100	5330490	6942260	6559397	6953660	6365848	23.22	18.74	23.34	16.26	134	297	134
F.T,500,2500,150	5930530	8094410	7978909	7571640	7037860	26.73	25.67	21.67	15.73	216	302	488
F.T,500,2500,200	8327720	11963100	11911900	11452900	10727261	30.39	30.09	27.29	22.37	312	313	396
F.T,500,3000,50	3529370	4333310	4069239	4262350	4035362	18.55	13.27	17.20	12.54	99	1188	166
F.T,500,3000,100	5442880	7164410	7046750	7186810	6634387	24.03	22.76	24.27	17.96	229	262	214
F.T,500,3000,150	6236240	8773910	8172602	8709390	7517445	28.92	23.69	28.40	17.04	150	257	155
F.T,500,3000,200	7283080	11236600	11354647	10390700	9751002	35.18	35.86	29.91	25.31	308	259	510
F.L,500,2000,50	3432140	3882110	3726114	3823610	3722839	11.59	7.89	10.24	7.81	136	1022	196
F.L,500,2000,100	5497770	6706100	6404834	6453880	6005177	18.02	14.16	14.81	8.45	146	300	271
F.L,500,2000,150	6750150	8205000	7886028	8081600	7510651	17.73	14.40	16.48	10.13	198	351	211
F.L,500,2000,200	8031600	10181700	10376103	9828350	9338097	21.12	22.60	18.28	13.99	424	375	592
F.L,500,2500,50	3176040	3818440	3507652	3612030	3491664	16.82	9.45	12.07	9.04	90	1223	213
F.L,500,2500,100	5062110	6893490	6187629	6400140	5909401	26.57	18.19	20.91	14.34	133	1076	303
F.L,500,2500,150	6542600	10022900	9520783	9089920	8138918	34.72	31.28	28.02	19.61	155	196	319
F.L,500,2500,200	7717740	11937300	11566824	10099200	9788913	35.35	33.28	23.58	21.16	384	483	1976
F.L,500,3000,50	2958630	3668660	3492641	3457280	3369303	19.35	15.29	14.42	12.19	110	187	254
F.L,500,3000,100	4855420	6692780	6187593	6015950	5773133	27.45	21.53	19.29	15.90	178	377	613
F.L,500,3000,150	6031650	9378030	9479082	8919720	7741294	35.68	36.37	32.38	22.08	223	196	254
F.L,500,3000,200	6722660	11240900	11291918	10040000	9195115	40.19	40.46	33.04	26.89	264	264	691
Average value						24.09	21.56	20.96	15.43	196	422	365

The results demonstrate the considerable difficulty in solving the GT instance set, as CPLEX is only able to achieve an average optimality GAP of 24.09% after 5 hours of execution. A big part of the challenge resides in the complexity of obtaining tight lower bounds due to the weakness of the arc-based formulation. The solution polishing heuristic is generally more effective than the default CPLEX configuration, even though its advantage is diminished in instances with a high commodity count. In selected instances with 50 commodities, *CPLXSP* provides solutions of similar quality as *ParLS*. However, *ParLS* achieves them in substantially less time. In comparison to all three methods, our parallel local search scheme finds better solutions for every instance. On average, it requires less than 200 seconds to improve the best solution found by CPLEX running for 5 hours. Improvements become more noticeable in the instances with more commodities

because these benefit more from parallelism and are more challenging for CPLEX.

2.5.3 Scaling results

One of the primary goals of our approach is to exploit a large degree of parallelism. We rely on the concurrent exploration of a large number of local searches to find competitive solutions faster. The next set of experiments is aimed at showing the effectiveness and benefits of the application of parallelism.

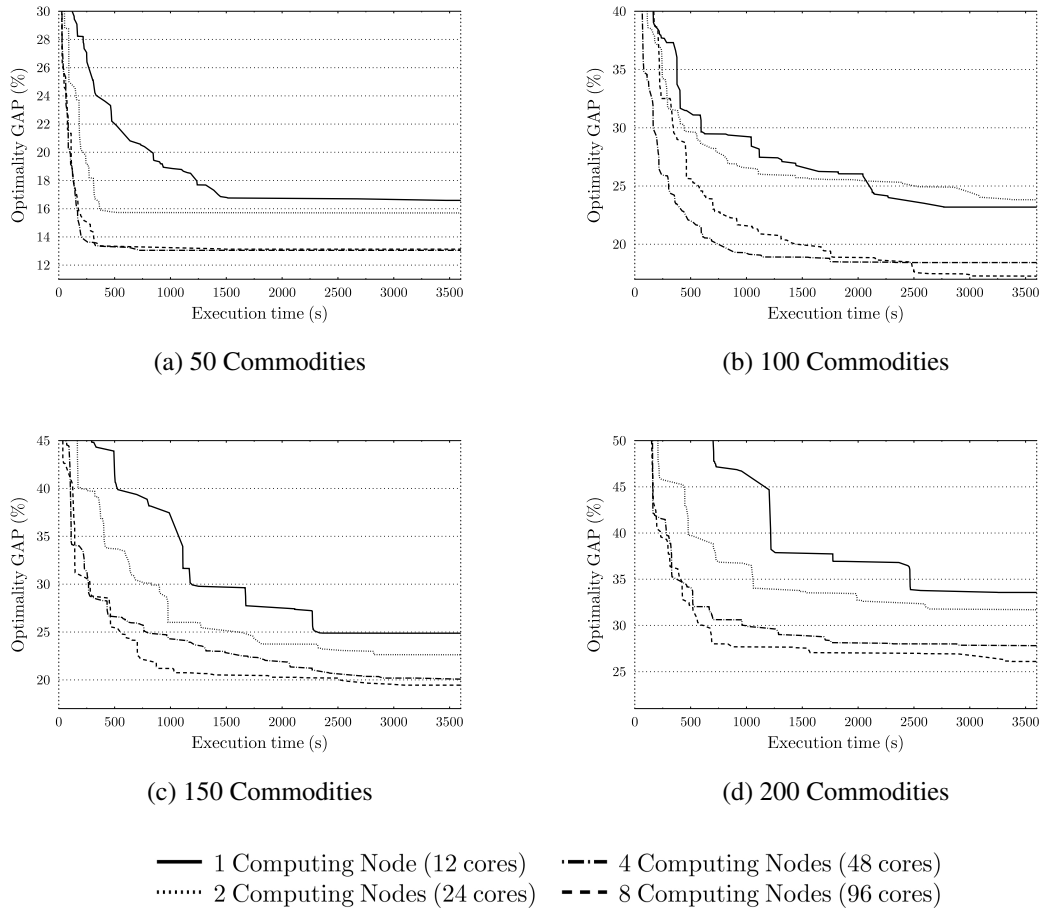


Figure 2.9: Scaling results for a selected test instance with 500 vertices, 3000 arcs and varying commodities. Each plot depicts the improvement in optimality GAP with respect to the best lower bound as a function of time. The executions shown in each problem differ in the number of parallel cores used.

In Figure 2.9, scaling results are shown for a representative set of instances. Since our

approach parallelizes over the set of commodities, we test a variety of instances with a number of commodities ranging from 50 to 200. For each problem, we report performance results for several processor configurations, ranging from executions on one processor (12 parallel cores) to eight (96 parallel cores). The same results are specified in terms of time in Table 2.3, where we show the time required to reach different GAP values with respect to the best solution found for each instance as well as with respect to the lower bound.

Table 2.3: Time required to reach certain gap with respect to the best found primal solution and the best lower bound

Problem	Number of computing nodes	Time required to reach GAP with respect to best solution (s)					Time required to reach GAP with respect to best lower bound (s)				
		20%	10%	5%	1%	0%	35%	30%	25%	20%	15%
F.T,500,3000,50	1 (12 cores)	137	638	1388	–	–	18	137	315	845	–
F.T,500,3000,50	2 (24 cores)	48	184	317	–	–	10	51	146	240	–
F.T,500,3000,50	4 (48 cores)	29	85	144	227	725	9	29	62	104	182
F.T,500,3000,50	8 (96 cores)	28	78	161	314	–	3	28	48	111	282
F.T,500,3000,100	1 (12 cores)	378	2124	–	–	–	378	594	2124	–	–
F.T,500,3000,100	2 (24 cores)	285	2069	–	–	–	247	443	2529	–	–
F.T,500,3000,100	4 (48 cores)	126	300	596	–	–	81	165	303	886	–
F.T,500,3000,100	8 (96 cores)	223	517	1057	2490	3506	221	348	577	1667	–
F.T,500,3000,150	1 (12 cores)	1110	2112	–	–	–	1110	1196	2336	–	–
F.T,500,3000,150	2 (24 cores)	406	978	2325	–	–	406	825	1637	–	–
F.T,500,3000,150	4 (48 cores)	111	433	1303	2902	–	111	267	790	–	–
F.T,500,3000,150	8 (96 cores)	143	462	703	2210	3589	143	281	563	2508	–
F.T,500,3000,200	1 (12 cores)	1216	–	–	–	–	2465	–	–	–	–
F.T,500,3000,200	2 (24 cores)	478	1951	–	–	–	1060	–	–	–	–
F.T,500,3000,200	4 (48 cores)	281	518	1128	–	–	455	999	–	–	–
F.T,500,3000,200	8 (96 cores)	197	427	674	3108	3379	423	623	–	–	–

Overall, parallelism is beneficial and substantially better solutions are achieved by using a larger number of processors. However, little improvement is observed when the number of processors exceeds or equals the commodity count. This is the case for the instances with 50 and 100 commodities, which are comparatively easier than instances with similarly sized networks and a larger number of commodities. The impact of parallelism differs from instance to instance due to their variability and the heuristic nature of our approach, including the eventuality that not every local search may yield improvements at every iteration. Instances with more commodities show better scalability, as more parallelism is exploited and there exists more opportunities for solution improvements.

2.5.4 Load balancing

Load balancing refers to the uniform distribution of work between parallel processors. We characterize the total execution time of a specific processor P_i as the sum of the useful computation time TC_{P_i} , the communication time TX_{P_i} and the idle time TI_{P_i} . We define the communication time as the time spent performing communication between processors, whereas the idle time TI_{P_i} accounts for the idle time spent by a processor on synchronization or waiting for other processors to finish their computations. Then, we define the utilization of a processor P_i as the ratio of useful computation time over the total execution time:

$$U(P_i) = \frac{TC_{P_i}}{TC_{P_i} + TX_{P_i} + TI_{P_i}}$$

Figure 2.10 displays the average core utilization for a representative FCMNF instance under different processor configurations and different time limit parameters. Each parameter configuration is specified with two time limits, where the first number corresponds to the local search time limit and the second refers to the solution recombination time limit. We show that average processor utilizations remain very high through all the tested combinations. When a single computing node (12 cores) is used, the shared-memory dynamic work allocation mechanism proves to be an effective means of load balance, as we achieve a utilization as high as 98%. The utilization also decreases with higher processor counts due to the requirement of more static partitions. Allowing more time for each local search also has a detrimental effect on processor utilization. This is due to the fact that the processors that end their work prematurely will have increased idle time penalties. Communication time is shown to be a small fraction of the overall compute time, which confirms our algorithm to be compute-bound.

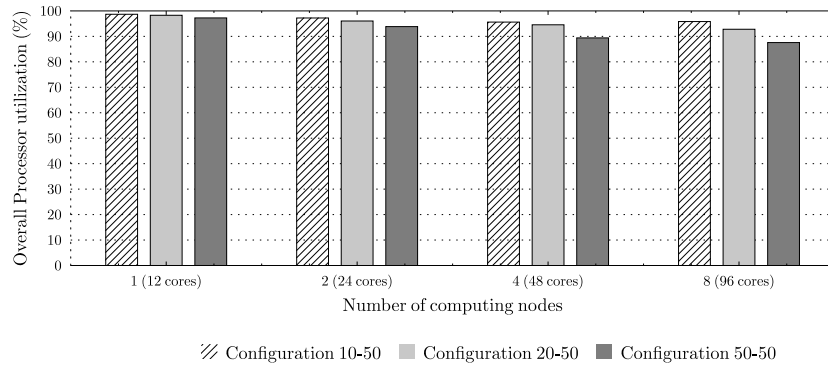


Figure 2.10: Overall processor utilization results for a selected test instance with 500 vertices, 3000 arcs and 200 commodities. Each data set corresponds to a different time limit parameter configuration, where the first number refers to the local search time limit and the second refers to the solution recombination time limit.

2.5.5 Partitioning the subproblem sequence

An important component of the parallel algorithm consists in partitioning the work among the parallel processors. In Section 2.3.2, we introduced the use of a graph partitioning problem to determine the distribution of local searches among the processors. By this transformation, the problem of finding a set of commodity partitions is effectively equivalent to the problem of partitioning a connection graph such that the cut is minimized. Our hope is that, by enforcing tightly connected commodities to be assigned together, better solutions will be achieved.

In order to evaluate the impact of this algorithm phase in the overall solution quality, we compare its performance against a random assignation of commodities. In Table 2.4, a performance comparison is shown for the C instance problem set. We specify two sets of executions performed under the same conditions and parameters. The time limit was set to be the time to best solution by *ParLS* shown in Table 2.1.

On the C instance problem set, the random subproblem assignation shows a slower solution improvement. When the same time limit is considered, results are generally worse or equivalent except for one of the smallest instances. On average, it achieves an optimality

Table 2.4: C instance set: performance comparison between commodity assignment schemes

Problem	LB / Opt	Connection Graph	Random Assignment
100/400/010/VL	28423*	28486	28430
100/400/010/FL	23949*	24022	24022
100/400/010/FT	63066	64207	64492
100/400/030/VT	384802*	384802*	384802*
100/400/030/FL	49018*	49018*	49232
100/400/030/FT	132129	136861	138550
20/230/040/VL	423848*	424075	424075
20/230/040/VT	371475*	371573	371573
20/230/040/FT	643036*	643036*	643036*
20/230/200/VL	94213*	94213*	94227
20/230/200/FL	137642*	137642*	138282
20/230/200/VT	97914*	97914*	98666
20/230/200/FT	135863	135867	136241
20/300/040/VL	429398*	429398*	429398*
20/300/040/FL	586077*	586077*	590964
20/300/040/VT	464509*	464509*	464549
20/300/040/FT	604198*	604198*	604198*
20/300/200/VL	74753	74811	75220
20/300/200/FL	113862	115580	116796
20/300/200/VT	74991*	74991*	75807
20/300/200/FT	106672	107102	108289
30/520/100/VL	53958*	53978	54004
30/520/100/FL	93570	93967	94409
30/520/100/VT	52046*	52046*	52046*
30/520/100/FT	96260	97862	98041
30/520/400/VL	112735	112787	113346
30/520/400/FL	147790	149677	150616
30/520/400/VT	114641*	114641*	114729
30/520/400/FT	150685	154137	156555
30/700/100/VL	47603*	47603*	47603*
30/700/100/FL	59958*	60058	60390
30/700/100/VT	45872*	45879	46040
30/700/100/FT	54904*	54904*	55059
30/700/400/VL	97189	98090	99369
30/700/400/FL	131690	136257	142080
30/700/400/VT	94508	95651	96708
30/700/400/FT	128243	131104	131153
Average GAP		0.51	1.08

GAP of 1.08% in comparison to the 0.51% obtained by our original connection graph partitioning. In addition, the optimal solution is only achieved in 6 instances whereas our proposed scheme reaches optimality in 15.

Performance results on the GT instance problem set with a time limit of 1 hour are presented in Table 2.5. The original algorithm with connection graph partitioning obtains better solutions in all instances but two. These are two of the smallest instances in the set, with 50 commodities. The impact on the performance varies substantially from instance to instance, ranging from an almost identical performance (0.1% GAP difference in the F_L,500,2000,50 problem) to a jump of 2.65% in the case of the F_T,500,3000,150 instance.

Table 2.5: GT instance set: performance comparison between commodity assignment schemes

Problem	LB / Opt	Connection Graph	Random Assignment
F_T,500,2000,50	4326550	4892012	4914193
F_T,500,2000,100	6368730	7273916	7294598
F_T,500,2000,150	7208800	8014986	8155082
F_T,500,2000,200	8845440	10617796	10859566
F_T,500,2500,50	3927990	4406080	4378135
F_T,500,2500,100	5330490	6365848	6474697
F_T,500,2500,150	5930530	7037860	7199470
F_T,500,2500,200	8327720	10727261	10883127
F_T,500,3000,50	3529370	4035362	4056824
F_T,500,3000,100	5442880	6634387	6774373
F_T,500,3000,150	6236240	7517445	7765078
F_T,500,3000,200	7283080	9751002	9997125
F_L,500,2000,50	3432140	3722839	3726822
F_L,500,2000,100	5497770	6005177	6048962
F_L,500,2000,150	6750150	7510651	7587028
F_L,500,2000,200	8031600	9338097	9341089
F_L,500,2500,50	3176040	3491664	3510419
F_L,500,2500,100	5062110	5909401	5935047
F_L,500,2500,150	6542600	8138918	8350529
F_L,500,2500,200	7717740	9788913	9959787
F_L,500,3000,50	2958630	3369303	3364152
F_L,500,3000,100	4855420	5773133	5874379
F_L,500,3000,150	6031650	7741294	7947452
F_L,500,3000,200	6722660	9195115	9527525
Average GAP		15.43	16.53

2.5.6 The parallel solution recombination

In the last stage of the algorithm, the local search improvements are accumulated into one solution. As described in Section 2.3.3, a new MIP subproblem is formulated, where the arc variables that are not used in any of the input solutions are fixed to zero. For parallel scalability, the solution recombination process is split in two structured phases.

In Table 2.6, we compare the effectiveness of our scheme against a single-step recombination of solutions extracted from a number of representative problem instances. For each of the tested instances, the best 96 solutions (one per parallel core) found after certain time limit are selected and used as an input for the recombination. Each of the schemes is allowed 100 seconds of optimization. In the case of the two-step recombination, the time limit is equally distributed with 50 seconds for each phase.

We report improvements as percentages relative to the objective of the best input solution. It is computed as $I = \frac{BI_{sol} - F_{sol}}{BI_{sol}} \cdot 100$, where BI_{sol} is the best objective value among

Table 2.6: Performance comparison of solution recombination schemes

Problem	Allowed Time	Parallel recombination		Single recombination	
		Improvement	Fixed Arcs	Improvement	Fixed Arcs
F_T,500,2500,50	300	0.19	2303	0.00	1930
F_T,500,2500,50	900	0.20	2250	0.00	1754
F_T,500,2500,50	1800	0.20	2251	0.00	1754
F_T,500,2500,100	300	0.47	2283	0.00	1800
F_T,500,2500,100	900	0.95	2240	0.00	1779
F_T,500,2500,100	1800	0.22	2224	0.00	1711
F_T,500,2500,150	300	0.17	2062	0.00	1311
F_T,500,2500,150	900	0.99	2145	0.00	1566
F_T,500,2500,150	1800	0.35	2117	0.00	1569
F_T,500,2500,200	300	3.16	2181	0.17	1492
F_T,500,2500,200	900	0.33	2177	0.00	1534
F_T,500,2500,200	1800	0.01	2180	0.00	1520

the 96 input solutions and F_{sol} is the objective value of the final recombined solution. In addition, the number of fixed arcs is also reported. In the case of the two-phase scheme, an average of the arc fixings of all the recombinations is provided.

Through all tested instances, the two-phase scheme proves to be a more effective strategy for recombining a large number of solutions. Improvements are achieved because the input is partitioned, and therefore a high level of fixings can be maintained. That is not the case for a single-phase scheme, where the number of fixings is significantly lower. As a result, improvements can't be found in the allowed time except for one instance.

2.6 Conclusions

We propose a scalable parallel approach for the Fixed Charge Multicommodity Network Flow problem that is designed for both shared memory parallel systems and distributed memory systems. By the use of heuristic local searches based on solving restricted MIP subproblems obtained by variable fixings, improvements in the flow routing are found in parallel and are further combined to obtain improved solutions. We rely on the network characteristics of the instances and the given solutions to define core components of the algorithm, such as the work partitioning and the solution recombination mechanism. Computational experiments demonstrate the effectiveness and scalability of our approach, as high-quality solutions are obtained for two problems sets from the literature.

Large sized FCMNF problem instances represent a computational challenge. Commercially available solvers and previous heuristic methods struggle to provide solutions and lower bounds that are within a reasonable optimality gap. It is precisely in the size of these instances where many opportunities to exploit parallelism can be found. We demonstrate the value of parallel computing and heuristic approaches for effectively generating good primal solutions to large FCMNF problem instances. Optimality certificates in the form of lower bounds are still difficult to achieve.

CHAPTER 3
ALTERNATING CRITERIA SEARCH: A PARALLEL LARGE
NEIGHBORHOOD SEARCH ALGORITHM FOR MIXED INTEGER
PROGRAMS

3.1 Introduction

In discrete optimization, high quality feasible solutions are valuable assets in the optimization process, and constructing them has been one of the main focuses of research for the past two decades. Berthold [3] and Fischetti et al. [55] present comprehensive literature reviews on primal heuristics and their applications to MIPs.

Starting heuristics and improvement heuristics are two classes of primal heuristics that differ in whether they require a starting feasible solution or not. The feasibility pump [20, 56] is a widely used starting heuristic for finding feasible solutions to MIP instances quickly. It consists of an iterative algorithm that combines linear program (LP) relaxations and roundings to enforce LP and integer feasibility until a feasible solution is found. Successive works built on the original feasibility pump to improve the solution quality [57] and its overall success rate [58, 59, 60, 61]. Structure-based heuristics [62] and RENS [21](relaxation enforced neighborhood search) are other compelling starting heuristics for finding feasible solutions. RENS belongs to the class of LNS heuristics, which entail solving carefully restricted sub-MIPs derived from the original problem. Their effectiveness relies on the ability to use the full power of a MIP solver to optimize the subproblem. LNS approaches differ in how the search neighborhood is defined. RENS attempts to find the best possible rounding when a fractional solution is given. Based on domain propagation and rounding, shift-and-propagate [63] and ZI rounding [64] are additional successful starting heuristics which are computationally tested in [65]. More algorithms for finding solutions

to MIP instances can be found in the literature [66, 67, 68, 69, 70].

Improvement heuristics, on the other hand, require a feasible starting solution and their focus is to improve its quality with respect to the objective. Simple improvement algorithms include the 1-opt [71] and 2-opt [72] heuristics. A large number of improvement heuristics found in the literature rely on LNS ideas. Successful LNS heuristics include local branching [39] and subsequent LNS heuristics that use local branching [73], RINS [22](relaxation induced neighborhood search) , DINS [74](distance induced neighborhood search), proximity search [75] and evolutionary algorithms [23]. The neighborhoods within these heuristics are usually defined using branch-and-bound information, such as the best available solutions or the LP relaxation at the current tree node. Because of this requirement, they must be executed as part of the node processing routine during the branch-and-bound process. Due to this input dependence, many nodes must be explored before these heuristics become effective at exploring diversified neighborhoods. Thus, high quality upper bound improvements are rarely found early in the search. In order to address this issue, the search neighborhoods used in our heuristic are defined using randomization instead of branch-and-bound dependent information. This allows us to obtain a wide range of diverse search neighborhoods from the beginning of the search, thus increasing the heuristic's effectiveness.

3.1.1 Parallel computing applied to Integer Optimization

Due to recent trends in processor design, parallelism has become ubiquitous in today's computers. With the advent of multi-core CPUs, it has become necessary to rethink most conventional algorithms in order to leverage the benefits of parallel computing. In the field of discrete optimization, Bader et al. [24] and Koch et al. [25] discuss potential applications of parallelism. The most widely used strategy entails exploring the branch-and-bound tree in parallel by solving multiple subproblems simultaneously. Due to its simplicity, most state-of-the-art MIP solvers incorporate this technique, such as CPLEX [76], GUROBI [77], and

ParaSCIP [78]. However, studies have suggested that parallelizing the branch-and-bound search may not scale well to a large number of cores [25]. Alternative parallelizations have been proposed in [79] and [80], where parallelism is used to explore different perturbations of the same problem. Other options include the application of parallelism to cut generation, primal heuristics, preprocessing and branching. To our knowledge, the work of Koc et al. [81] are the only effort to parallelize primal heuristics for general MIPs. In this work, the authors present a parallelization of the Feasibility Pump [20]. Therefore, Parallel Alternating Criteria Search is the first parallel algorithm to combine features from starting and improvement heuristics, offering the possibility of generating starting solutions and improving them with respect to the original objective.

LNSs are some of the most computationally expensive heuristics, since they are based on the optimization of sub-MIPs. A strategy to leverage parallelism is to perform a large number of LNS simultaneously over a diversified set of neighborhoods with the objective of increasing the chances of finding better solutions. To some degree, the parallelization of the branch-and-bound tree already provides this diversification and improvement in performance, since the exploration of multiple nodes in parallel includes the simultaneous execution of multiple heuristics with a diverse set of inputs. Our heuristic builds upon a similar parallelization strategy, and expands it by adding an additional algorithmic step that combines and consolidates the improvements found in parallel.

Parallel Alternating Criteria Search combines parallelism and diversified large neighborhood searches in order to deal with large instances. Our approach is suitable for MIPs belonging to all kinds of applications, since it does not require knowledge or assumptions regarding the underlying structure of the problem. Although most of the algorithmic components present in Parallel Alternating Criteria Search have been introduced in the literature before, we demonstrate their great effectiveness when put together in a parallel algorithm. We find our approach to be competitive or better than CPLEX at finding solutions for more than 90% of the instances in the MIPLIB2010 library [82]. The improvement of the pro-

posed method becomes more pronounced on harder instances. Additionally, we present a parallel scheme that combines the use of Alternating Criteria Search in conjunction with an exact algorithm. Results show that alternative parallelizations of the branch-and-bound process can be more efficient than traditional methods, especially when large-scale instances are considered.

We introduce our primal heuristic in Section 3.2, where we present components that define it and give further details on the parallel implementation. Section 3.3 presents computational experiments and results on standard instances from the literature. Section 3.4 provides some concluding remarks.

3.2 Parallel Alternating Criteria Search

We define a mixed-integer program (MIP) as:

$$\min\{c^t x \mid Ax = b, l \leq x \leq u, x_i \in \mathbb{Z}, \forall i \in \mathcal{I}\} \quad (\text{MIP})$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and $\mathcal{I} \subseteq \{1, \dots, n\}$ is the subset of integer variable indices. The decision vector x is bounded by $l \in \overline{\mathbb{R}}^n$ and $u \in \overline{\mathbb{R}}^n$, where $\overline{\mathbb{R}}$ is the extended set of real numbers $\mathbb{R} \cup \{-\infty, \infty\}$.

Our intent is to satisfy a two-fold objective: to find a feasible starting solution and to improve it with respect to the original objective. We introduce an LNS heuristic, in which two auxiliary MIP subproblems are iteratively solved to attain both goals. The process requires an initial vector, which is not required to be a feasible solution. As seen in Figure 3.1, this vector is improved by solving sub-MIPs, in which a subset of the variables are fixed to its input values.

For linear programs, the feasibility problem is solved via the two-phase Simplex method, in which an auxiliary optimization problem is developed in order to find a feasible starting basis. This approach was introduced for the case of 0-1 MIPs in [83]. In a similar fash-

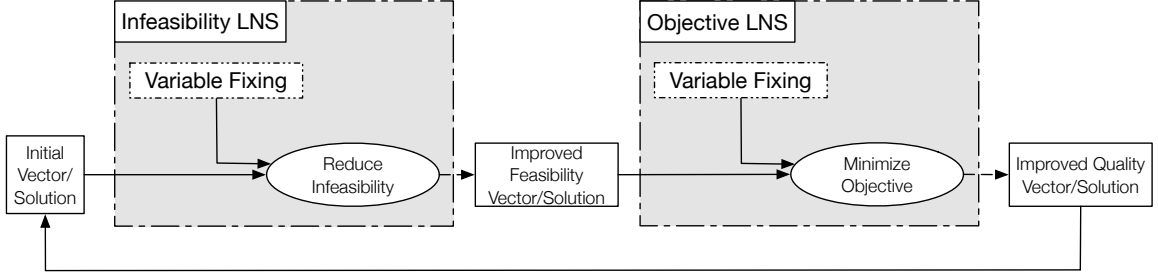


Figure 3.1: High level depiction of the sequential heuristic

ion, the following auxiliary MIP, denoted as FMIP, poses the problem of finding a feasible starting solution as an optimization problem:

$$\min \sum_{i=0}^m \Delta_i^+ + \Delta_i^-$$

s.t.

$$Ax + I_m \Delta^+ - I_m \Delta^- = b$$

$$x_i = \hat{x}_i, \forall i \in \mathcal{F}$$

$$l \leq x \leq u$$

$$x_i \in \mathbb{Z}, \forall i \in \mathcal{I}$$

$$\Delta^+ \geq 0, \Delta^- \geq 0$$

(FMIP)

where I_m is an $m \times m$ identity matrix and Δ^+ , Δ^- are two vectors of continuous variables of size m corresponding to the m constraints. A decision vector is feasible to a MIP if and only if it can be extended to a solution of value 0 to the associated FMIP. Instead of directly solving FMIP, neighborhoods are restricted by fixing a given subset \mathcal{F} of the integer variables to the values of an input vector $[\hat{x}, \hat{\Delta}^+, \hat{\Delta}^-]$. Due to the addition of slack variables, \hat{x} is not required to be a feasible solution vector. However, it must be integer and within the variable bounds in order to preserve the feasibility of the model: $l \leq \hat{x} \leq u$ and $\hat{x}_i \in \mathbb{Z}, \forall i \in \mathcal{I}$.

FMIP ensures that feasibility is preserved under any arbitrary variable fixing scheme.

This represents a departure from most LNS heuristic improvement approaches such as RINS, DINS, local branching, and proximity search, where the choice of variable fixings is tied to the availability of a feasible solution. In the context of our heuristic, variable fixings become a viable tool for reducing the complexity of the problem.

Using a similar approach, we introduce a second auxiliary problem aimed at improving a partially feasible vector $[\hat{x}, \hat{\Delta}^+, \hat{\Delta}^-]$ with respect to the original objective:

$$\begin{aligned}
& \min && c^t x \\
& \text{s.t.} && \\
& && Ax + I_m \Delta^+ - I_m \Delta^- = b \\
& && \sum_{i=0}^m \Delta_i^+ + \Delta_i^- \leq \sum_i \hat{\Delta}_i^+ + \hat{\Delta}_i^- \tag{OMIP} \\
& && x_i = \hat{x}_i, \forall i \in \mathcal{F} \\
& && l \leq x \leq u \\
& && x_i \in \mathbb{Z}, \forall i \in \mathcal{I} \\
& && \Delta^+ \geq 0, \Delta^- \geq 0
\end{aligned}$$

OMIP is a transformation of the original MIP model, in which auxiliary slack variables are introduced in each constraint. Achieving and preserving the feasibility of the incumbent is our primary concern. In order to ensure that the optimal solution to OMIP remains at most as infeasible as the input solution \hat{x} , an additional constraint that limits the amount of slack is added, where the degree of infeasibility is bounded by $\sum_i \hat{\Delta}_i^+ + \hat{\Delta}_i^-$.

By iteratively solving subproblems of both auxiliary MIPs, the heuristic will hopefully converge (although its convergence is not guaranteed) to a high quality feasible solution. By construction, infeasibility decreases monotonically after each iteration. On the other hand, the solution quality may fluctuate with respect to the original objective. Figure 3.2 depicts the expected behavior of the algorithm. A similar approach for achieving feasibility via the FMIP model is presented in [83], although our work differs in several key

aspects. The authors of the aforementioned work limit their scope to 0-1 problems and use local branching to explore FMIP. Our approach accepts general-integer variables and uses variable fixings as a means of exploiting parallelism and to reduce infeasibility. Another differentiating factor of our approach is the transition and use of an auxiliary MIP model, OMIP, to provide improvements with respect to the original objective.

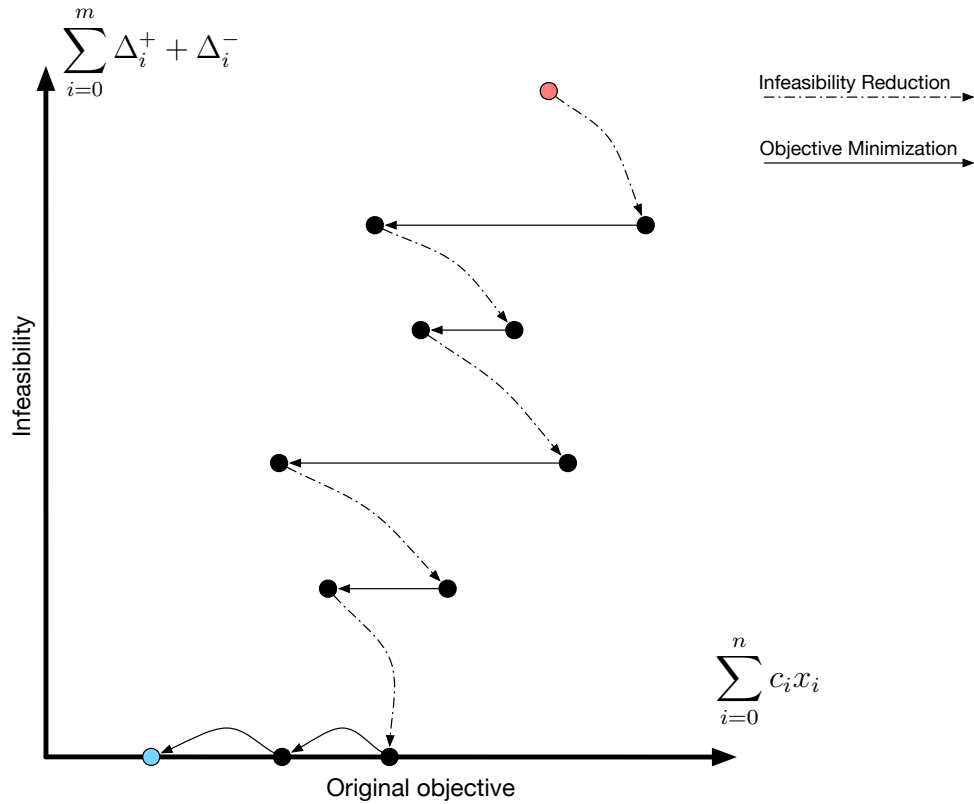


Figure 3.2: Transition to a high quality feasible solution

3.2.1 Parallelization of Alternating Criteria Search

We leverage parallelism by generating a diversified set of large neighborhood searches, which are solved simultaneously. By exploring a large number of different search neighborhoods in parallel, we hope to increase the chances of finding solution improvements, hence speeding up the overall process. After this exploration phase, improvements found in parallel are combined efficiently. For this purpose, an additional search subproblem is generated, in which the variables that have the same value across the different solutions

are fixed. A similar approach has been previously described in the literature [23, 3]. In this context, the solution recombination provides the ability to merge the improvements found in parallel, providing a speedup in the process. A pseudocode version is given in Algorithm 7.

Algorithm 7 Parallel Feasibility Heuristic

Output: Feasible solution \hat{x} if found
initialize $[\hat{x}, \Delta^+, \Delta^-]$ as an integer solution
 $T := numThreads()$
while time limit not reached **do**
 if $\sum_i \Delta_i^+ + \Delta_i^- > 0$ **then**
 for all threads $t_i \in \{0, T - 1\}$ **in parallel do**
 $\mathcal{F}_{t_i} :=$ randomized variable index subset, $\mathcal{F}_{t_i} \subseteq \mathcal{I}$ \triangleright Variable Fixings are diversified
 $[x^{t_i}, \Delta^{+t_i}, \Delta^{-t_i}] := \text{FMIP_LNS}(\mathcal{F}_{t_i}, \hat{x})$ \triangleright FMIP LNS are solved concurrently
 end for
 $\mathcal{U} := \{j \in \mathcal{I} | x_j^{t_i} = x_j^{t_k}, 0 \leq i < k < T\}$
 $[\hat{x}, \Delta^+, \Delta^-] := \text{FMIP_LNS}(\mathcal{U}, x^{t_0})$ \triangleright The recombination step differs in variable fixings
 end if
 $\Delta^{UB} := \sum_i \Delta_i^+ + \Delta_i^-$
 for all threads $t_i \in \{0, T - 1\}$ **in parallel do**
 $\mathcal{F}_{t_i} :=$ randomized variable index subset, $\mathcal{F}_{t_i} \subseteq \mathcal{I}$
 $[x^{t_i}, \Delta^{+t_i}, \Delta^{-t_i}] := \text{OMIP_LNS}(\mathcal{F}_{t_i}, \hat{x}, \Delta^{UB})$
 end for
 $\mathcal{U} := \{j \in \mathcal{I} | x_j^{t_i} = x_j^{t_k}, 0 \leq i < k < T\}$
 $[\hat{x}, \Delta^+, \Delta^-] := \text{OMIP_LNS}(\mathcal{U}, x^{t_0}, \Delta^{UB})$
end while
return $[\hat{x}, \Delta^+, \Delta^-]$

function FMIP_LNS(\mathcal{F}, \hat{x})
 return $\min\{\sum_i \Delta_i^+ + \Delta_i^- | Ax + I_m \Delta^+ - I_m \Delta^- = b, x_j = \hat{x}_j \forall j \in \mathcal{F}, x_j \in \mathbb{Z} \forall j \in \mathcal{I}\}$
end function

function OMIP_LNS($\mathcal{F}, \hat{x}, \hat{\Delta}$)
 return $\min\{c^t x | Ax + I_m \Delta^+ - I_m \Delta^- = b, \sum_i \Delta_i^+ + \Delta_i^- \leq \hat{\Delta}, x_j = \hat{x}_j \forall j \in \mathcal{F}, x_j \in \mathbb{Z} \forall j \in \mathcal{I}\}$
end function

Each parallel processor iteratively generates a set of randomized variable fixings and solves the associated sub-MIP of either FMIP or OMIP until the allowed time limit. Upon termination, all solutions are exchanged and the set \mathcal{U} containing the indices of the variables with identical values across solutions is determined. The solution recombination MIP consists of a subproblem, in which the variables present in \mathcal{U} are fixed. The size of \mathcal{U} is dependent on the similarities between the solutions to be merged. If the set is too large and no improvements can be found, the best solution used in the recombination is returned.

The best solution will be the most feasible or the most optimal, depending on whether a recombination FMIP or OMIP is being optimized. Every solution used as input can be also added as a MIP start, since they remain feasible under the set of variable fixings. Figure 3.3 depicts an example for a simple 0-1 knapsack instance. Firstly, the Feasibility MIP is derived from the original problem instance. Next, two subproblems characterized by different fixings are solved in parallel. In a final step, the variables with coinciding values are fixed and a feasible solution is found.

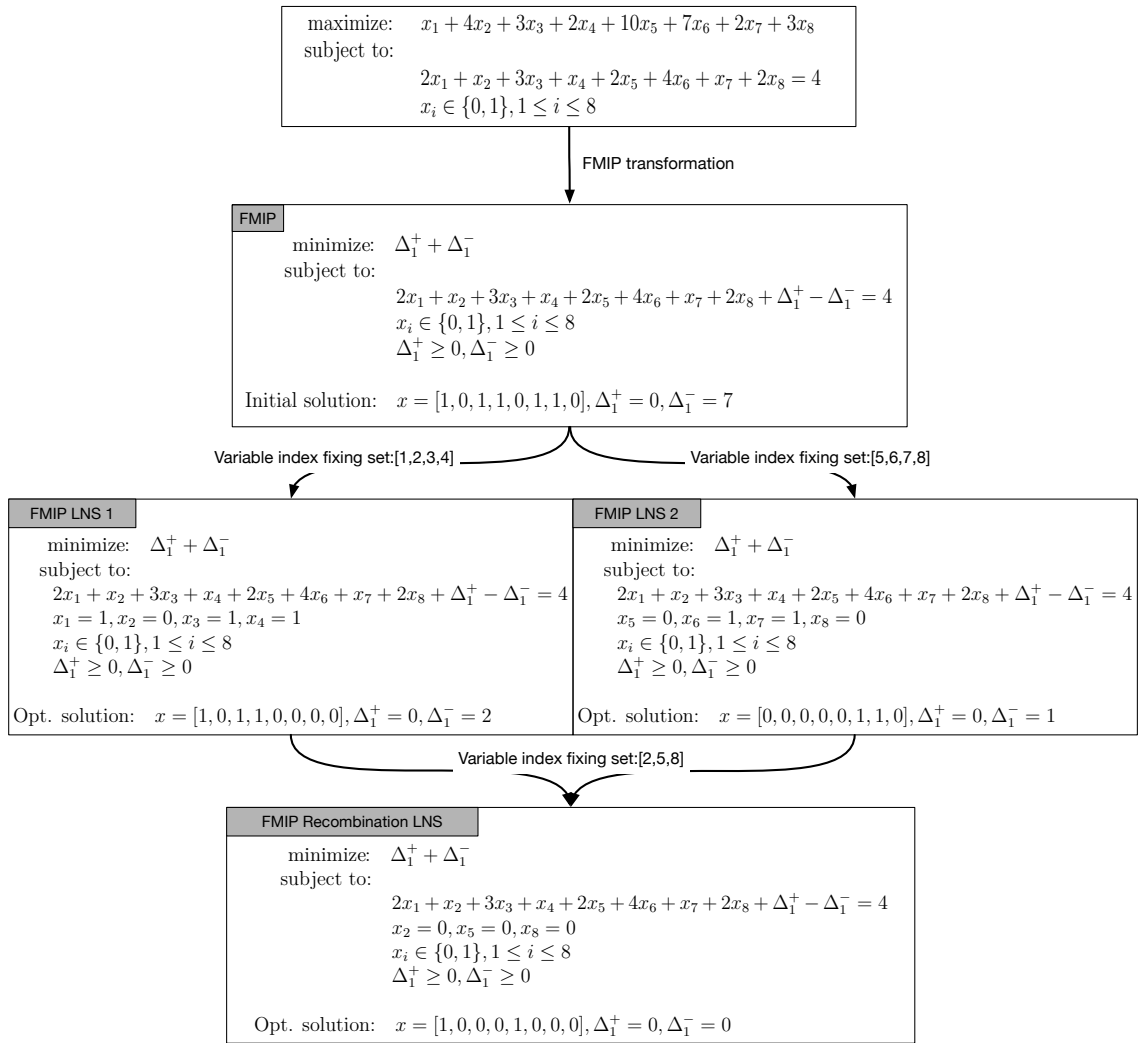


Figure 3.3: Example depicting a feasibility improvement iteration for a 0-1 knapsack sample instance

Distributed-memory parallelism is the main paradigm for large-scale parallel computer

architectures. One of the defining characteristics is the fact that memory is partitioned among parallel processors. As a result, processor synchronization and memory communication must be done via a message passing interface, such as MPI [84]. Processor coordination and communication becomes a problematic element in the algorithm design, inducing occasional overheads. Our approach requires the exchange of solutions between processors before and after each solution recombination. The arrangement is such that every processor communicates its best solution to every other processor during an all-to-all synchronous exchange. Unlike point-to-point communications, all-to-all collective communication primitives take advantage of the underlying network structure to communicate synchronously among a large number of processors in a more efficient manner [85].

3.2.2 Finding a starting point

Alternating Criteria Search only requires a starting vector that is integer feasible and within variable bounds. However, given that feasibility is one of the primary objectives of the heuristic, it is proposed to choose a starting point that is as feasible as possible with respect to the objective function of FMIP. Many strategies can provide a start for the algorithm. A common strategy solves the LP relaxation and rounds every fractional variable to the nearest integer. Since LP relaxations can potentially be very costly to solve, we propose a quick heuristic, Algorithm 8, that tries to minimize the infeasibility of a starting point. Within each iteration, subsets of variables are fixed to random integer values within bounds while the remaining ones are optimized towards feasibility. The algorithm terminates once all integer variables are fixed.

Starting with a sorted list of variables by increasing bound range and an input parameter θ , the algorithm proceeds to fix the top $\theta\%$ of variables to a random integer value within their bounds. It is possible that a variable may have an infinite bound, and therefore, an infinite range. In this case, the infinite bound is replaced by a constant input parameter c_b . For our practical purposes, this was determined to be 10^6 . With sorting, the goal is to drive

Algorithm 8 Starting vector heuristic

Input: Percentage of variables to fix θ , $0 < \theta \leq 100$, Fixed bound constant c_b

Output: Starting integer-feasible vector \hat{x}

```
1:  $V :=$ list of integer variables sorted by increasing bound range  $u - l$ 
2:  $\mathcal{F} := \emptyset$ 
3: while  $\hat{x}$  is not integer feasible and  $\mathcal{F} \neq \mathcal{I}$  do
4:    $\mathcal{K} :=$  top  $\theta$  % of unfixed variables from  $V$ 
5:   for  $k \in \mathcal{K}$  do
6:      $\hat{x}_k :=$ random integer value between  $[\max(l_k, -c_b), \min(u_k, c_b)]$ 
7:   end for
8:    $\mathcal{F} := \mathcal{F} \cup \mathcal{K}$ 
9:    $[x, \Delta^+, \Delta^-] := \min\{\sum_i \Delta_i^+ + \Delta_i^- | Ax + I_m \Delta^+ - I_m \Delta^- = b, x_j = \hat{x}_j \forall j \in \mathcal{F}\}$ 
10:   $\mathcal{Q} :=$  index set of integer variables of  $x$  with integer value
11:   $\hat{x}_q = x_q, \forall q \in \mathcal{Q}$ 
12:   $\mathcal{F} := \mathcal{F} \cup \mathcal{Q}$ 
13: end while
14: return  $\hat{x}$ 
```

binary variables integer first, given that binary decisions force integrality on other variables. Until all integer variables are fixed, the LP relaxation of FMIP is solved in order to optimize the unfixed variables towards feasibility. Consecutively, the variables that become integer are fixed. A minimum of $\theta\%$ variables are fixed at every iteration. Hence, the algorithm will require at most $\lceil \frac{100}{\theta} \rceil$ iterations to converge to a starting solution. The θ parameter controls a tradeoff of difficulty of the LP relaxations against the quality of the starting feasible solution.

3.2.3 The variable fixing scheme

The variable fixing scheme determines the difficulty and the efficacy of each large neighborhood search. A desirable quality of the fixings is that they must not be too restrictive, as very few improvements will be found. At the same time, if not enough variables are fixed, the search space might be too large to find any improvements in a small amount of time. Neighborhood diversification is another key property, since parallel efficiency depends on it. In order to generate a large number of diverse neighborhoods early in the search, we use randomization instead of branch-and-bound information.

Devising a general method for fixing variables may be challenging, since there are many problem structures to consider. Problems range in structure, constraint matrix shape, number, and kinds of variables. Given these requirements, we propose a simple, yet intuitive variable fixing algorithm. It incorporates randomness, in order to satisfy the need for diversity and it allows the fixing of an adjustable number of variables. As shown in Algorithm 9, fixings are determined by selecting a random integer variable x' and fixing a consecutive set of integer variables starting from x' up to a certain cap determined by an input parameter ρ . If the end of \mathcal{I} is reached before enough variables are chosen, the algorithm continues the selection starting from the beginning of the set in a circular way.

Algorithm 9 Variable Fixing Selection Algorithm

Input: Fraction of variables to fix ρ , $0 < \rho < 1$

Output: Set of integer indices \mathcal{F}

- 1: **function** RANDOMFIXINGS(ρ)
 - 2: $i :=$ random element in \mathcal{I}
 - 3: $\mathcal{F} :=$ first $\rho \cdot |\mathcal{I}|$ consecutive integer variable indices starting from i in a circular fashion
 - 4: **return** \mathcal{F}
 - 5: **end function**
-

For any MIP we consider, its formulation is usually structured, and its variables are arranged consecutively depending on their type and their logical role. Consecutive sets of variables often belong to cohesive substructures within a problem. This is the case in network flow and routing problems where flow assignments for a particular entity are formulated successively. Similar properties can be found in formulations for scheduling problems. In our experience, our proposed variable selection often produces an easier subMIP, in which the fixings affect a subset of contiguous substructures and the remaining ones are left unfixed. Due to its simplicity, it is an efficient variable selection strategy. However, any permutation of rows and columns alters its effectiveness as well as the solving process, as discussed in [86].

3.2.4 Framing Alternating Criteria Search within an exact algorithm

In the current parallel branch-and-bound paradigm, threads are focused on solving multiple subproblems in parallel, which has shown poor scalability [25]. We propose an alternative use of the parallel resources by decoupling the search for high quality solutions from the lower bound improvement. Thus, a subset of the threads are allocated to an instance of the branch-and-bound solver focused on improving the lower bound, and our Alternating Criteria Search replaces the traditional primal heuristics. In the process, our parallel heuristic searches for solutions to the entire problem regardless of the variable fixings produced in the branch-and-bound tree, and supplies them to the solver. Both algorithms proceed to run concurrently until a time limit or optimality is reached. Communication between the parallel heuristic and the branch-and-bound is performed via MPI collective communications and new feasible solutions are added back via callbacks.

3.3 Experimental results

In this subsection, we evaluate the performance and behavior of Parallel Alternating Criteria Search (PACS) in terms of solution quality, scalability, reproducibility and parallel efficiency. The framework is implemented in C++, using CPLEX 12.6.1 as a backbone solver. We compare our framework against different configurations of the state-of-the-art general purpose MIP solver CPLEX 12.6.1. Out of the 361 instances from the MIPLIB 2010 library [82], we select as a benchmark those 333 for which feasibility has been proven as a benchmark. MIPLIB classifies such instances by difficulty based on the time required to reach optimality. 206 easy instances are defined as the subset in which optimality has been proven in less than one hour by at least one of the tested MIP solvers. 54 additional instances have also been solved, but not under the previous conditions (hard instances). The remaining 73 unsolved instances are classified as open. All of our computations are performed on an 8-node computing cluster, each with two Intel Xeon X5650 6-core processors

and 24 GB of memory.

3.3.1 Automating the choice of parameters

Three main parameters regulate the difficulty and the solution time of each LNS within the heuristic, and their appropriate selection is crucial for performance. We heuristically calibrate the settings for each instance automatically by executing a single iteration of the algorithm under multiple independent sample configurations and with the same initial solution. Each iteration run is performed in parallel and the best performing one is selected for the full run. The parameter θ regulates the number of variables to be fixed during the initial solution generation process. Depending on the difficulty of the instance, θ is chosen from the set $\{1\%, 5\%, 10\%, 20\%, 50\%, 100\%\}$. θ aside, parameters $[\rho, t]$ determine the percentage of variables to be fixed and the time limit in each LNS. In this case, the configurator chooses a subset of the permutations $\rho \in \{5\%, 10\%, 20\%, 50\%, 75\%, 95\%\}$ and $t \in \{5s, 20s, 50s\}$. A total of 6 initial configurations for θ are tested, in addition to a total of 16 permutations of $[\rho, t]$. The time required for calibration is not counted in the final execution time. The chosen configuration is the one which delivers the largest improvement per unit of time. On average, the calibration process takes 126s due to the fact that the calibration is run in parallel. With the intention of drawing a fair comparison, CPLEX is set to its parallel distributed-memory setting using 96 cores and allowed instance-specific tuning for the same amount of calibration time prior to the search. In our experience, instance-specific tuning certainly helps CPLEX improve its performance in most of the small instances. However, the time allowed seems to be insufficient to correctly tune for a subset of larger instances. In such cases, the parameters chosen result in inferior runs compared to the default setting. We opt to combine all results, and select the best run of CPLEX: either default or tuned. When comparing primal bounds, the default setting is on primal solutions (the emphasis on *hidden feasible solutions* setting). Settings are default otherwise.

3.3.2 Evaluation of primal solution quality

We evaluate the quality of primal solutions in terms of the metrics introduced in [87]. Given a solution x for a MIP with an optimal solution \hat{x} , the primal gap $\gamma(x) \in [0, 1]$ of x is defined as:

$$\gamma(x) = \begin{cases} 0 & \text{if } |c^T \hat{x}| = |c^T x| = 0 \\ 1 & \text{if } c^T \hat{x} \cdot c^T x < 0 \\ \frac{|c^T \hat{x} - c^T x|}{\max\{|c^T \hat{x}|, |c^T x|\}} & \text{else.} \end{cases} \quad (3.1)$$

Given a time limit t_{max} , we define the primal gap function $p : [0, t_{max}] \mapsto [0, 1]$ as:

$$p(t) = \begin{cases} 1 & \text{if no solution is found until point } t \\ \gamma(x(t)) & \text{with } x(t) \text{ being the incumbent solution at point } t, \text{ else.} \end{cases} \quad (3.2)$$

The primal gap function is monotonically decreasing and allows to depict the progress of the optimization towards the optimal solution. The primal integral is defined as:

$$P(t) = \int_{t=0}^T p(t) dt \quad (3.3)$$

The primal integral $P(t)$ captures the notion of how early solutions are found. Both $p(t)$ and $P(t)$ promise to be powerful metrics to evaluate the performance of finding high quality primal solutions. To illustrate their use, we introduce three examples in Figure 3.4, in which the performance of both CPLEX and PACS are plotted with respect to both metrics.

For rail03, PACS is able to find primal solutions of higher quality than those found by CPLEX. This results in a lower primal gap profile, as well as a slower increasing primal integral function. In the second problem, CPLEX finds a better solution at the end of the optimization. Its primal integral function value, however, is higher than the one for PACS because PACS is able to find relatively better solutions earlier in the search. For sing245, both schemes find the optimal solution, as shown by the flat profile of the primal integral.

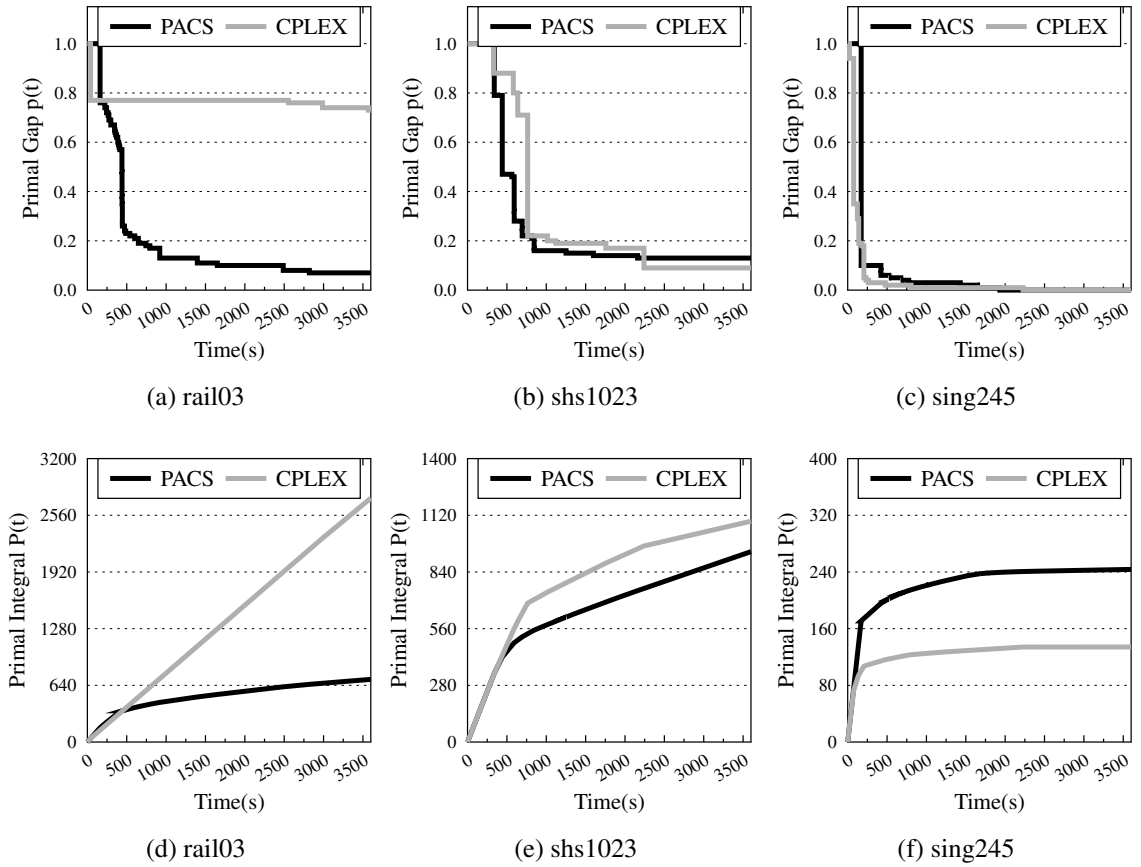


Figure 3.4: Primal gap function $p(t)$ and primal integral $P(t)$ of the solutions provided by CPLEX and PACS for the (a)(d)rail03, (b)shs1023 and (c)sing245 problems.

However, CPLEX does so earlier in the search, thus resulting in a lower integral value.

The following set of results evaluate the quality of the provided solutions for the proposed set of MIPLIB2010 instances. In Figure 3.5, we show a performance profile that illustrates the differences in primal gap output of both schemes. Let $p(t)_{CPX}$ and $p(t)_{PACS}$ be the primal gap functions of CPLEX and our parallel heuristic respectively after a time limit t . We report improvements in terms of the difference:

$$Improvement = (p(t)_{CPX} - p(t)_{PACS}) \cdot 100$$

Figure 3.5 displays the differences between the primal gap functions found by both methods after different time limits. Specifically, each performance curve measures the

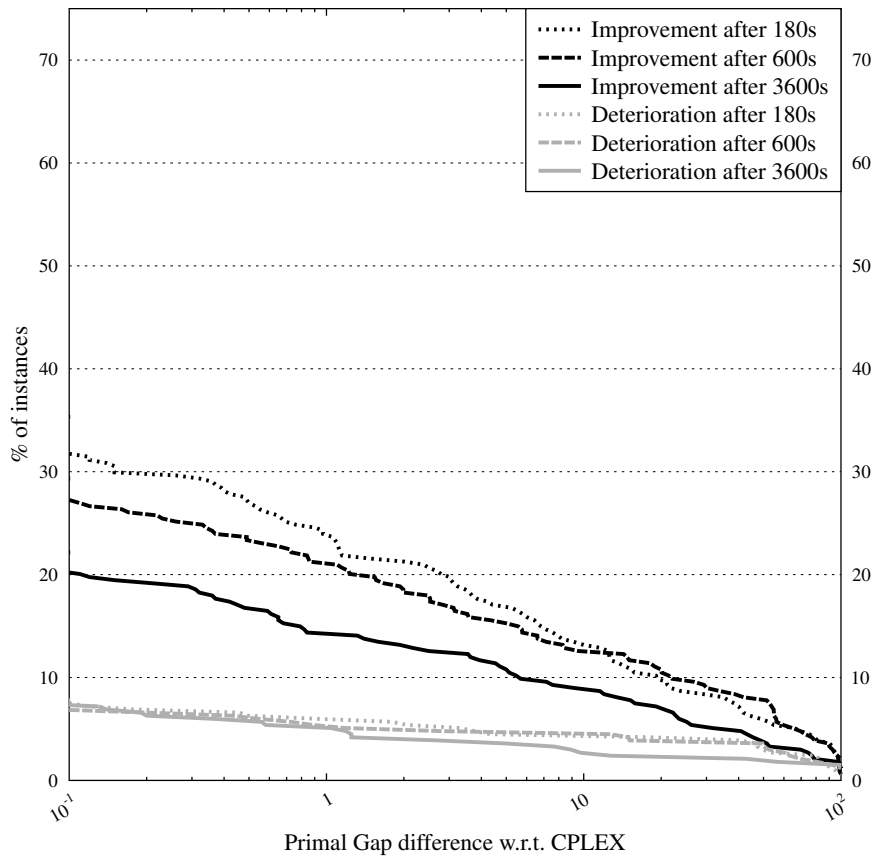


Figure 3.5: Improvement and Deterioration of primal gap w.r.t. CPLEX for all instances

cumulative percentage of instances among the total that reach or surpass certain amounts of difference in favor of PACS (Improvement) or CPLEX (Deterioration). For example, PACS produced solutions with a primal gap that was at least 10% better after 180 seconds for 13% of the problem instances. In contrast, only 4% of the instances yield a worse primal gap of 10% or more in the same amount of time. Both schemes tie or show differences between -0.1% and 0.1% for the remaining percentage of instances.

Results show that our heuristic performs better for a substantial number of instances, whereas worse solutions are found in a relatively smaller subset of cases. One of our primary focuses is the ability to provide high quality solutions early in the search. After 3 minutes of execution, PACS provides solution improvements for 32% of the instances. At

the time limit of one hour, this percentage decreases to 20% after CPLEX neutralizes the advantage for some of the instances. In contrast, worse solutions are only obtained in 7% of the cases.

In Figure 3.6, the comparison is restricted exclusively to hard and open instances. Results show the scalability of our heuristic in hard MIPs, as more than 58% of the instances yield improvements after 3 minutes of execution. A comparison between the 180 and the 600 second profiles indicate that the competitive advantage of PACS is sustained and increased throughout a large portion of the runtime.

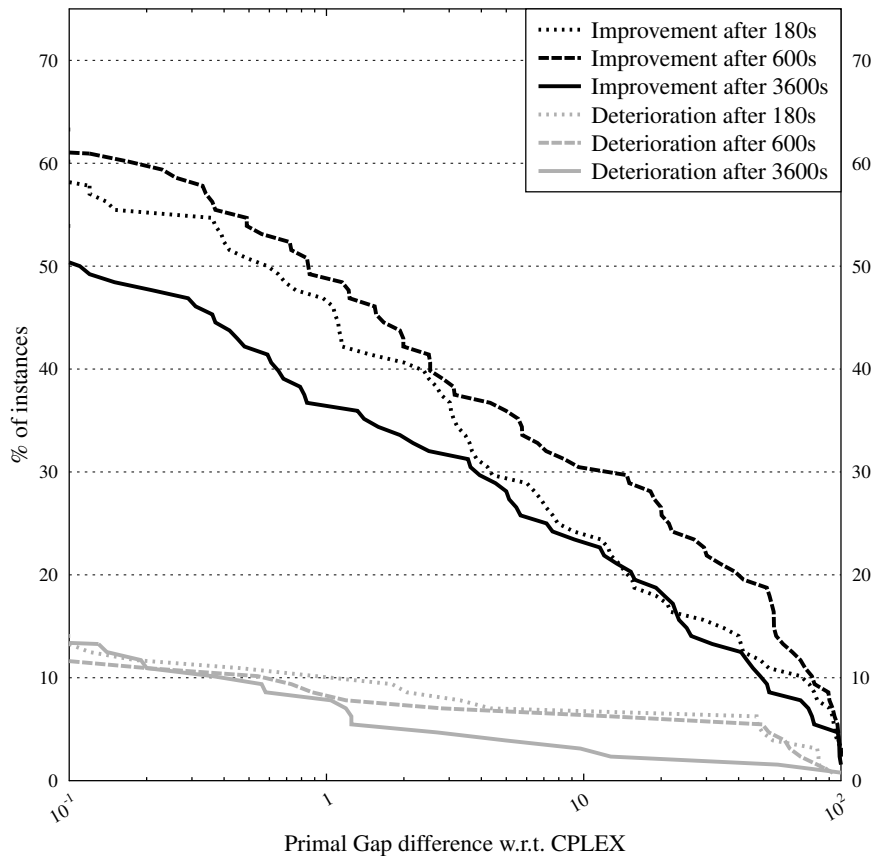


Figure 3.6: Improvement and Deterioration of primal gap in hard and open instances w.r.t. CPLEX

Figure 3.7 depicts a comparison of the primal integral. Given a time t , we define

$P(t)_{\text{CPX}}$ and $P(t)_{\text{PACS}}$ as the primal integrals provided by CPLEX and PACS respectively at time t . For improved readability, we compare the improvements in terms of the scaled difference:

$$\text{Improvement} = \frac{P(t)_{\text{CPX}} - P(t)_{\text{PACS}}}{t} \cdot 100$$

The difference between primal integrals is scaled by t in order to be able to plot profiles belonging to different time cutoffs.

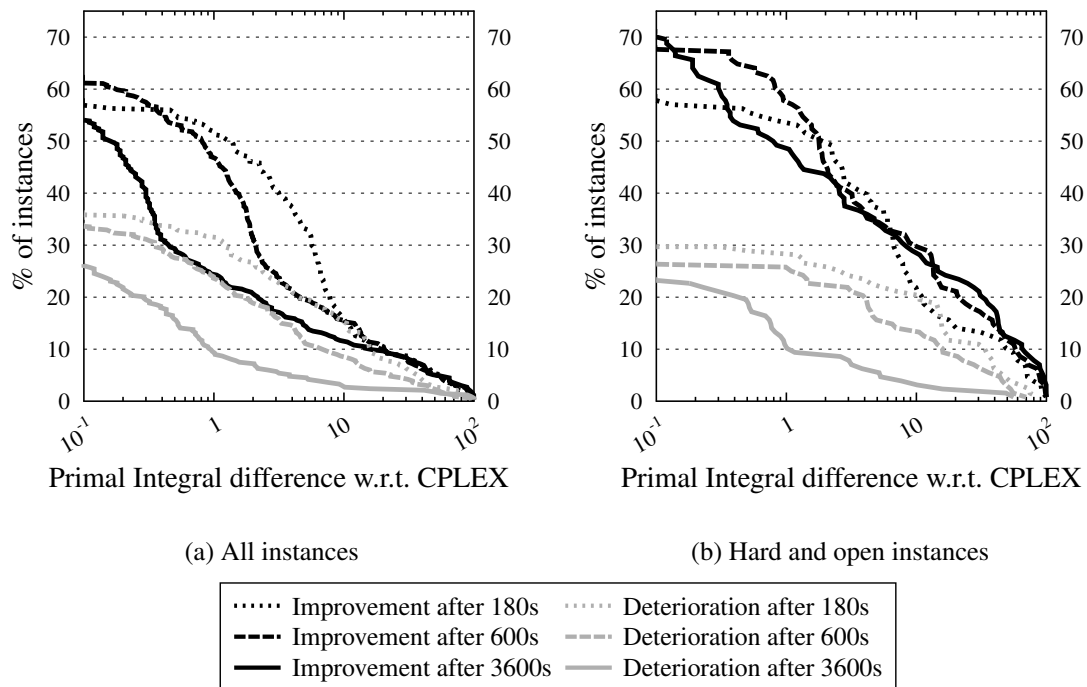


Figure 3.7: Improvement and Deterioration in primal integral w.r.t. CPLEX for (a)all and (b)hard and open instances

The performance profiles indicate that PACS finds solutions earlier in the search, resulting in significantly lower primal integral profiles for a majority of the instances. After 600 seconds, the primal integral values show an improvement for 61% of the instances. The performance profiles shift to the left as time advances, indicating that the advantage of PACS is reduced. When only hard or open instances are considered, performance profiles are clustered together and further to the right, indicating that the advantage of PACS is

sustained through time. After one hour, PACS showed better primal integrals for more than 54% of the instances. The plots suggest that, not only does PACS find better solutions for most of the instances, but it does so faster.

3.3.3 Framing Alternating Criteria Search within an exact algorithm

We test the performance of our heuristic when run in combination with an exact branch-and-bound algorithm. In this hybrid scheme, a fraction of the cores are dedicated to improving the primal incumbent and the rest are committed to computing the lower bound in the tree search.

We give a direct performance comparison between parallel memory-distributed CPLEX using 96 threads, and our combined scheme. In the latter, 84 cores are allocated to the parallel heuristic while the remaining 12 threads are allocated to CPLEX in shared-memory parallel mode. The comparison is in terms of the difference between the optimality gap provided by both algorithms: $Gap_{CPX} - Gap_{PACS}$. For any of the two algorithms X , its optimality gap may be defined in terms of its best found upper and lower bound, UB_X and LB_X :

$$Gap_X = \frac{UB_X - LB_X}{UB_X} \cdot 100$$

Figure 3.8 shows multiple performance profiles for different time cutoffs. Both approaches outperform each other for different instance subsets, but our combined scheme performs better in more instances. After 10 minutes, our algorithm provides a better gap for over 30% of the instances, while the opposite is true for 18% of the instances. We observe a similar shift in performance when only the hard and open instances are considered. In this case, our ability to produce solutions of high quality early in the search allows us to achieve a smaller optimality gap for over 55% of the instances after 10 minutes. At termination, over 14% of the instances show a competitive advantage of over 10% of the gap. In contrast, CPLEX shows better performance for at most 5% of the instances.

prove optimality for a larger number of instances, given that it has more processors dedicated to the branch-and-bound. Because better primal solutions are found by the combined scheme, however, smaller gaps are obtained for more instances in which optimality can't be proven.

Table 3.2: Optimality performance comparison

Time Cutoff(s)	Combined Scheme			CPLEX (Balanced Emphasis)		
	180	600	3600	180	600	3600
Avg. Gap(%)(All Inst.)	22.74	16.58	12.87	25.59	21.44	14.67
Avg. Gap(%)(Hard and Open Inst.)	42.16	33.49	28.32	50.34	45.34	33.76
Instances Solved(All Inst.)	79	93	103	107	114	124
Avg. Time To Opt.(s)(All Inst.)	242			143		

3.3.4 The impact of nondeterminism

Our approach is nondeterministic by nature due to the different time limits and the parallel synchronization required for the solution recombination. We have demonstrated its performance in comparison to CPLEX, which is deterministic in its default setting. When in deterministic mode, most parallel MIP solvers must sacrifice a fraction of the performance in order to ensure a proper repeatability of the results. In order to assess the impact of nondeterminism, we compare the performance of our approach with CPLEX running in Opportunistic mode, which is non-deterministic. In the experiments shown below, we evaluate the consistency of 5 runs for each feasible instance in the Reoptimize set, which is a subset of 63 easy, hard, and open instances from the MIPLIB2010 library.

Figure 3.9 depicts a comparison of the primal gap and primal integral obtained with both methods. For each of the charts shown, multiple lines are depicted, showing the performance when the the best, the mean, and the worst run is selected. When all instances in the Reoptimize set are considered, PACS shows better performance earlier in the search, as reflected in the primal integral. After one hour, PACS shows a better primal integral for

62% of the instances, in the worst execution. CPLEX is competitive in finding solutions at the end of the execution, as reflected by chart 1(c), as it finds better or equal solutions for 75% of the instances, where PACS does so for 80% of them.

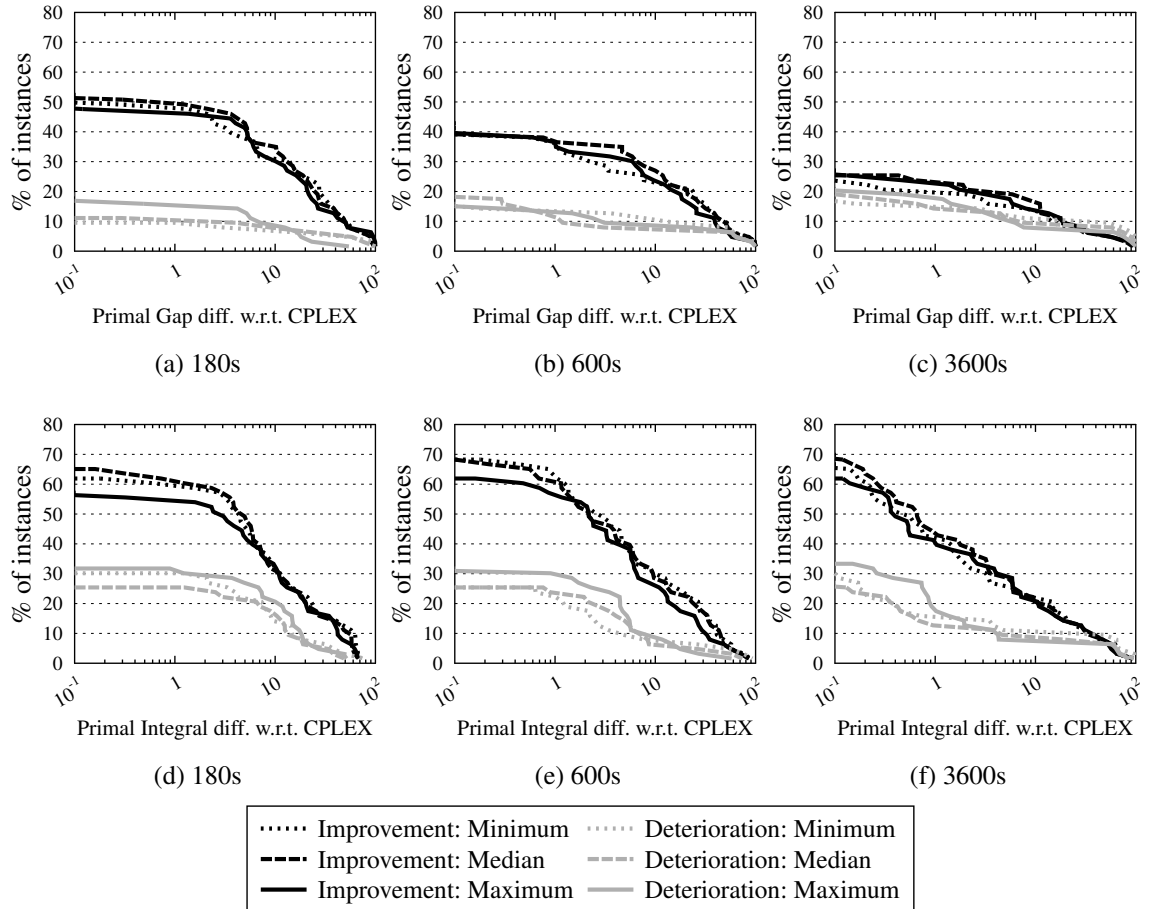


Figure 3.9: Performance profiles for all instances in the Reoptimize set. In each of the charts multiple lines are drawn for the best, mean, and worst executions. Charts show the performance for different cutoffs: (a)(d)180s, (b)(e)600s, and (c)(f)3600s

Table 3.3 reports the statistical results in terms of the average, standard deviation, and median of the primal gap and primal integral for different time cutoffs.

Table 3.3: Performance comparison of non-deterministic approaches

Time Cutoff(s)			Parallel Alternating			CPLEX		
			Criteria Search			(Opportunistic Mode)		
			180	600	3600	180	600	3600
Avg.	P. Gap(%)	All inst.	24.86	17.01	11.47	35.97	22.95	10.58
		H&O inst.	31.77	21.50	12.39	45.73	35.35	16.06
	P. Integral	All inst.	77.20	162.60	528.50	91.11	210.18	648.57
		H&O inst.	88.75	99.29	602.31	103.99	277.88	954.45
Std. Dev.	P. Gap(%)	All inst.	3.16	2.73	1.32	3.23	0.71	1.74
		H&O inst.	3.48	3.78	1.10	0.89	0.95	1.53
	P. Integral	All inst.	5.64	17.35	58.85	3.26	9.10	45.99
		H&O inst.	5.26	20.00	57.00	1.59	4.96	38.28
Min.	P. Gap(%)	All inst.	22.14	14.56	10.50	32.80	21.92	8.86
		H&O inst.	28.33	17.51	11.21	44.77	33.86	14.19
	P. Integral	All inst.	70.98	146.27	479.46	87.54	200.55	591.79
		H&O inst.	83.02	179.23	539.64	101.74	271.08	904.34
Med.	P. Gap(%)	All inst.	24.03	16.59	11.06	35.65	23.13	9.95
		H&O inst.	30.94	21.31	12.22	45.77	35.49	16.02
	P. Integral	All inst.	76.84	160.06	516.95	91.42	208.43	651.69
		H&O inst.	88.67	197.20	600.70	104.18	278.34	957.46
Max.	P. Gap(%)	All inst.	29.25	20.67	13.49	39.69	23.68	12.48
		H&O inst.	36.05	25.96	13.75	46.65	36.36	17.79
	P. Integral	All inst.	84.23	187.19	617.50	95.02	222.05	699.69
		H&O inst.	95.17	227.01	678.11	105.73	283.32	998.57

3.3.5 Additional performance tests: scalability and parallel efficiency

We examine the performance of Alternating Criteria Search from the perspective of scalability and parallel efficiency. Due to the large amount of experiments required to evaluate this set of metrics, we reduce the test bed to a representative subset of the instances chosen randomly. The 15 instances shown in Table 3.4 are selected from all three difficulty categories.

Table 3.4: Selected instances for Scaling experiments

Easy Instances	Hard Instances	Open Instances
a1c1s1	atm20-100	momentum3
bab5	germanrr	nsr8k
csched007	n3-3	pb-simp-nonunif
danoint	rmatr200-p5	t1717
map14	set3-20	ramos3

Parallel scalability

Strong scalability is the ability to increase the algorithm’s performance proportionally when parallel resources are incremented while the problem size remains fixed, and a particularly relevant metric is the speedup to cut off: a comparison of the time required by different processor configurations to reach certain solution quality. Let T_c^p be the time required for the algorithm to reach a cut-off c when p processing nodes are used. We define the parallel speedup to cut off S_{SC} with respect to a baseline configuration B as $S_{SC} = \frac{T_c^B}{T_c^p}$. In Figure 3.10, we show the speedup demonstrated by the heuristic for several processor configurations ranging from a baseline of 12 cores to a total of 96. Among the runs for different processor configurations, the cutoff was determined to be the objective value of the best solution achieved by the worst performing run.

In terms of scaling, our heuristic shows a variable performance dependent on the characteristics of each individual instance. This behavior is expected since increasing the number of simultaneous searches does not guarantee a translation to faster improvements. In general, however, speedups are achieved more consistently as the difficulty of the problem increases. The addition of more cores sometimes exhibits a multiplicative effect. In most small instances, optimality is achieved quickly for all processor configurations, thus resulting in small speedup values.

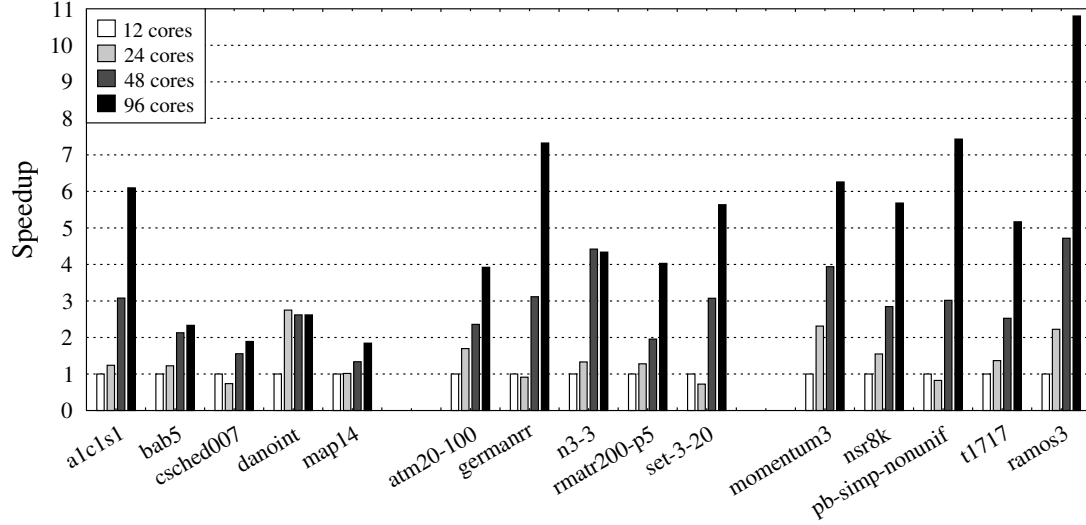


Figure 3.10: Parallel strong-scaling results

Parallel load balancing

Load balancing is the property that measures the degree of uniformity of the work distribution among processors. Given the synchronous nature of our approach, an even difficulty of the subproblems is essential in order to ensure all processors optimize for an equivalent amount of time. In the case of the parallel heuristic, the size of each subproblem is regulated by the number of fixed variables and the imposed search time limit. Hence, a proper calibration of these parameters must ensure an even distribution of the workload.

The load balance of a parallel application can be evaluated as follows: Let the total execution time of a processor P_i be defined as the sum of the time spent performing useful computations (TU_{P_i}), communications (TC_{P_i}) and the synchronization time TS_{P_i} spent waiting for other processors to complete their computations. Then, we characterize the utilization of a processor U_{P_i} as the ratio of useful computation time over the total execution time:

$$U_{P_i} = \frac{TU_{P_i}}{TU_{P_i} + TS_{P_i} + TC_{P_i}}$$

We believe hard instances represent the worst-case scenario, since these are the ones

that require the most computational effort and prolonged optimization times. Figure 3.11 shows the average core utilization for the hard instance momentum3. Performance results are displayed for different processor configurations as well as different time limit parameters. Time limit configurations are denoted as $C_{T_{LNS}-T_R}$, where T_{LNS} is the time allowed for each search and T_R is the time allowed to the recombination step. Results show that processors sustain high utilizations (above 95%) throughout the execution, even when large processor configurations are used. The setting with the shortest solution times remains the most efficient because smaller time penalties are paid due to early terminations.

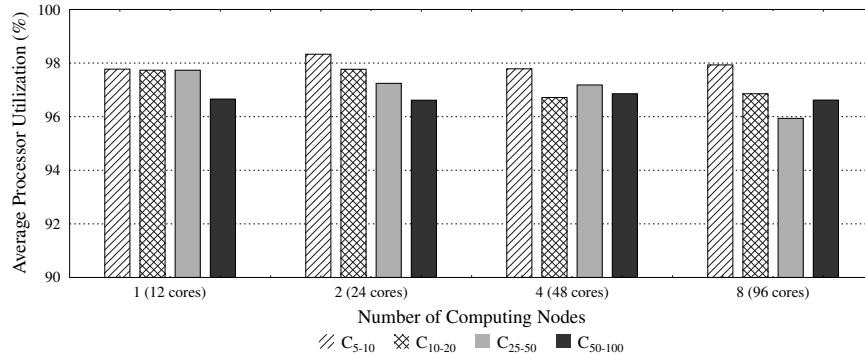


Figure 3.11: Parallel synchronization overhead in terms of average processor utilization for different parameter configurations

The impact of the starting heuristic

In Section 3.2.2, we described a quick heuristic for finding a starting solution suitable for Alternating Criteria Search. The objective of this strategy is to provide a starting point for the algorithm that is as feasible as possible with respect to the objective function of FMIP. An iterative algorithm is proposed, in which successive restricted LP relaxations are solved, the difficulty of which depend on a variable fixing parameter ρ .

Figure 3.12 illustrates the behavior of the starting heuristic for four problem instances, as the variable fixing parameter ρ varies. For the instances shown, the infeasibility of the provided starting solution increases as more variables are randomly fixed per iteration.

However, the algorithm also becomes faster, as a direct consequence of reducing the difficulty of the LP relaxations. For the evaluated test set, a good compromise can usually be found, in which better solutions than the ones provided by random fixings ($\rho = 1$) are found, at the expense of slightly longer runtimes.

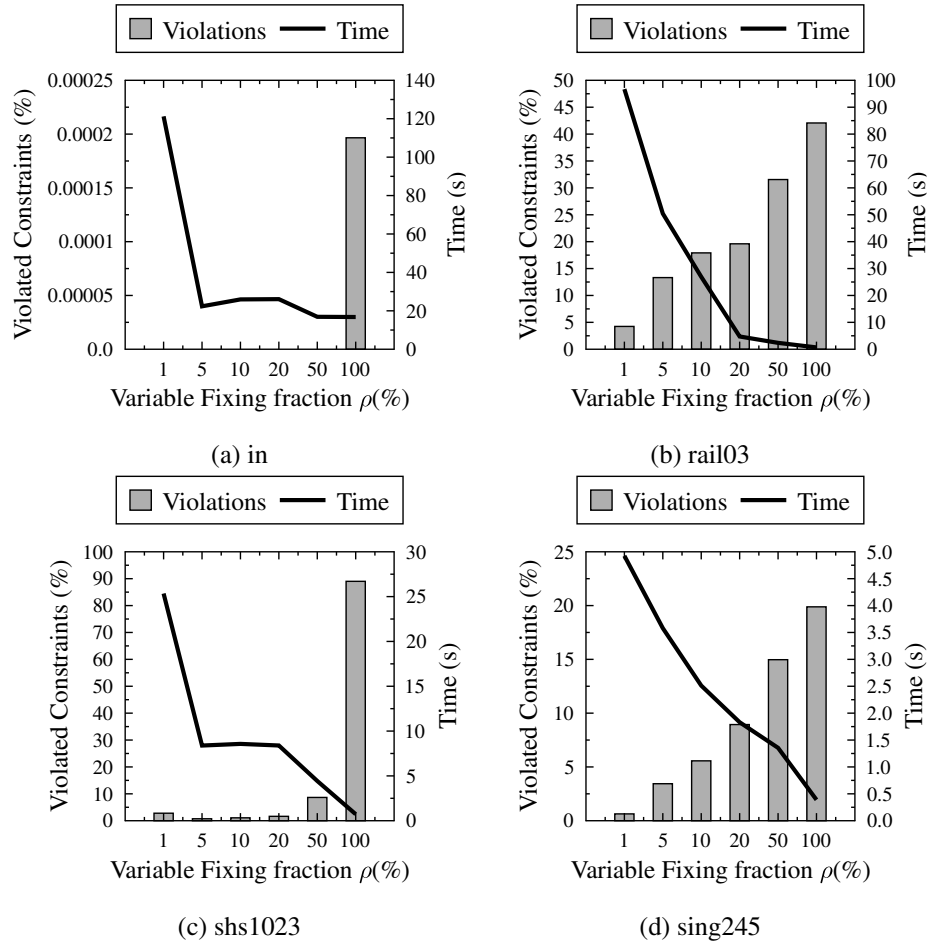


Figure 3.12: Performance tradeoffs shown by the starting heuristic, in which the time to reach the initial solution is contraposed to its quality as a function of the variable fixing parameter ρ for the (a)in, (b)rail03, (c)shs1023 and (d)sing245 problems.

3.4 Conclusions

The combination of parallelism and simple large neighborhood search schemes can provide a powerful tool for generating high quality solutions. The heuristic becomes especially useful in the context of large instances and time-sensitive optimization problems, where tra-

ditional branch-and-bound methods may not be able to provide competitive upper bounds and attaining feasibility may be challenging. Our method is a highly versatile tool, as it can be applied to any general MIP as a standalone heuristic or in the context of an exact algorithm. Many algorithmic ideas contribute to the competitiveness of our approach, such as the use of two auxiliary MIP transformations and fixing diversifications as the main source of parallelism.

CHAPTER 4
RAPID PROTOTYPING OF EFFECTIVE PARALLEL PRIMAL HEURISTICS
FOR DOMAIN SPECIFIC MIPS: AN APPLICATION TO THE MARITIME
INVENTORY ROUTING PROBLEM

4.1 Introduction

Currently, global seaborne logistics are the most utilized form of freight transportation and account for the vast majority of the world trade. In most cases, the optimization of the costs related to these shipping operations is vital to their economic viability. As a result, these kind of problems are ideal examples of real-world application of MIPS. The Maritime Inventory Routing Problem (MIRP) is a Mixed-Integer Programming formulation, which models most details present in maritime shipping, such as inventory management at ports and vessels, as well as vessel routing and scheduling. Despite being an extremely flexible and faithful mathematical model, MIRP instances are very challenging to solve to optimality. In most cases, it is challenging to find a single feasible solution.

In this chapter, we illustrate the application of Parallel Alternating Criteria Search to a coherent set of real-world MIP instances. In addition, we show how it is possible to prototype rapidly a specialization of the parallel algorithm to better address the specific structure found in MIRPs. Despite being a parallel heuristic designed for general MIPS, Parallel Alternating Criteria Search is on par with specialized heuristics of the domain in terms of its ability to provide high quality solutions quickly. When specialized to better exploit the internal structure of MIRPs, the same parallel algorithm can be substantially more effective. Improvements in performance are made possible with two main novel contributions: new definitions of MIRP-specific search neighborhoods and a modified objective function.

The remainder of this chapter is organized as follows: Section 4.2 covers the recent

work performed in developing primal heuristics for MIRPs. In Section 4.3 we introduce the mathematical models as well as details of the specialization. Section 4.4 presents computational experiments and results on standard instances from the literature. Finally, Section 4.5 provides some concluding remarks.

4.2 Related work

Our primary focus is on methods for finding primal solutions to inventory routing problems. Papageorgiou et al. [88, 89] provide thorough literature reviews as well as a comprehensive comparison of the state of the art in primal heuristics and exact methods used for solving this class of problems.

Maritime inventory routing problems have been applied to a broad range of applications, such as cement manufacturing [90], calcium carbonate slurry shipping [91], oil supply in stochastic scenarios [92], and routing and inventory management of vacuum gas oil [93]. MIRP instances present a real challenge to most commercial MIP solvers, and it is often challenging to find a feasible solution. Many construction heuristics have been proposed for MIRPs or similarly constructed problems, such as the ones presented in [94, 95, 96]. These are specialized algorithms designed to provide a first feasible solution at the beginning of the optimization. Construction heuristics usually rely on greedy procedures, multi-start local searches or solving restricted subproblems. Additional specialized heuristics for similar LNG routing problems are presented in [97].

A large number of works have proposed solution methods based on some variation of Large Neighborhood Search (LNS). LNS heuristics circumvent the complexity of the original problem by solving derived subproblems obtained by restricting a subset of the variables. A fully featured MIP solver is then used to optimize the subproblem, which delivers a solution valid to the original problem. LNS approaches differ in how the search neighborhood is defined. In the context of MIRP instances, a widely used LNS strategy entails fixing the variables related to a subset of the vessels. Different variations of this

approach are used in [94, 44, 98, 99, 100, 88], and differ in how the subset of vessels is selected.

Song and Furman [100] apply LNS techniques in combination with a branch-and-cut algorithm to find improved solutions to an arc-flow model very similar to the one used in this section. On the same instances, Engineer et al. [101] introduce an alternative column generation formulation, and solve it using branch-cut-and-price in combination with several newly introduced classes of cuts. Hewitt et al. [102] follows by applying a branch-and-price guided search (a method previously used for Fixed-Charge Multicommodity Network Flow problems[44]) in order to find high quality solutions quickly. This approach uses a small amount of parallelism (four processors) to speed up the algorithm.

An alternative decomposition approach very popularly applied to MIRPs is based on rolling horizon techniques. In a rolling horizon heuristic, the planning horizon is subdivided into smaller overlapping subhorizons. Each subdivision can be characterized with a tractable subproblem, which can be consecutively solved in a limited amount of time. Rolling horizon heuristics are introduced in [103, 104, 105, 96]. A variation is the fix-and-relax heuristic proposed by Uggen et al. [106], in which all posterior integer decision variables not included in the subhorizon are relaxed, and left to be continuous. Another significant departure is the approximate dynamic programming approach proposed by Papageorgiou et al. [98], in which the MIRP instance is formulated as a dynamic programming problem and interpreted as a sequence of vessel dispatching problems. Outside of the realm of heuristics, Goel et al. [107] introduce a constraint programming method based on a disjunctive scheduling representation.

Most of the aforementioned works focus on large-scale MIRPs with large planning horizons. Papageorgiou et al. [99] focuses on an operational MIRP with a much smaller horizon, a more detailed model and more challenging from the perspective of feasibility. They overcome the added complexity by designing a two-stage algorithm, in which decisions are first made among loading/discharging regions. In a second step, more detailed

routing decisions are made at the ports within each of the regions.

To the best of our knowledge, the works of Asokan et al. [108] are the only attempt at introducing parallel heuristics for LNG inventory routing problems, which are a special class of MIRPs. In this work, the authors parallelize LNS heuristics introduced in [109, 94]. Parallel MIP solvers such as CPLEX [76], GUROBI [77], and ParaSCIP [78] are the only alternative parallel algorithms currently available. In addition to high quality solutions, MIP solvers also provide a lower bound. Studies have suggested that parallelizing the branch-and-bound search may not scale well to a large number of cores [25], and this behavior is reflected in some of the aforementioned distributed-memory implementations. Another disadvantage is the fact that MIP solvers are general algorithms and they usually do not exploit the underlying network structure that characterizes MIRPs.

4.3 Algorithm

In this section, we introduce the arc-based formulations used in the modeling of MIRPs. Further, we perform a preliminary analysis of the performance of Parallel Alternating Criteria Search when applied to the problems at hand. To conclude, we propose a set of problem-specific modifications in order to further improve the performance of the parallel algorithm.

4.3.1 A time-space discretization of MIRPs

MIRPs model deep-sea vessel routing with inventory tracking at every port-vessel pair throughout a time-space network. A depiction of the model used is shown in Figure 4.1. Given a set \mathcal{T} of time periods and a set of \mathcal{J} of ports, the network consists of a set of nodes $\mathcal{N}_{s,t}$, which symbolize the state of the ports through different time periods. Additionally, a source node n_s and a sink node n_t are used to symbolize the entrance/exit of vessels to the system. A set of directed arcs \mathcal{A} model the travel of vessels between ports. More concretely, each vessel v uses a set of dedicated arcs \mathcal{A}^v . For a particular pair of node n

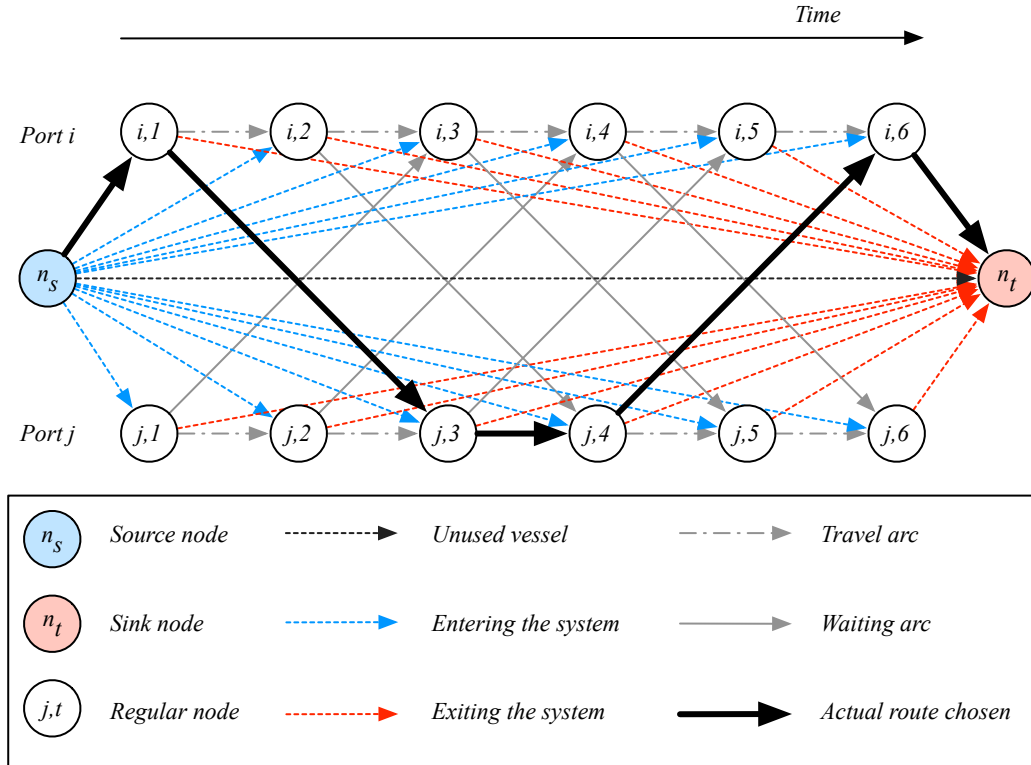


Figure 4.1: Time-space horizon modeling of MIRP

and vessel v , we may define the forward star \mathcal{FS}_n^v as the set of outgoing arcs associated to a vessel v leaving from the node n . Conversely, the reverse star \mathcal{RS}_n^v denotes the set of incoming arcs.

The MIRPLIB library [89] consists of a set of MIRP instances inspired by real-world problems. The library is composed of multiple instance classes, which differ in the modeling detail, and the time horizon outlook. Group 1 instances feature an operational MIRP with planning horizons of 45 and 60 periods, multiple ports per region and split pickups and deliveries. Group 1 instances present a challenge from the perspective of feasibility, and most commercial MIP solvers struggle to find a single feasible solution. In contrast, group 2 instances offer simplified models with planning horizons greater than 60 time periods, but only involving one port per region and never split pickups and deliveries. Feasibility is trivial for group 2 instances, and the challenge is rather to find high quality feasible solutions quickly.

Group 1 instances: the challenge of feasibility

In group 1 instances, the objective is to maximize the revenue obtained when product is delivered to a port and to minimize the expenses incurred by the transportation costs, penalties for buying/selling product to the spot market as well as the delays in their loading/unloading.

$$\max \sum_{n \in \mathcal{N}} \sum_{v \in \mathcal{V}} R_n f_n^v - \sum_{v \in \mathcal{V}} \sum_{a \in \mathcal{A}^v} C_a^v x_a^v - \sum_{v \in \mathcal{V}} \sum_{n \in \mathcal{N}} (t \epsilon_z) z_n^v - \sum_{j \in \mathcal{J}} \sum_{t \in \mathcal{T}} P_{j,t} \alpha_{j,t} \quad (4.1a)$$

$$s.t. \sum_{a \in \mathcal{F} \mathcal{S}_n^v} x_a^v - \sum_{a \in \mathcal{R} \mathcal{S}_n^v} x_a^v = \begin{cases} 1 & \text{if } n = n_s \\ -1 & \text{if } n = n_t \\ 0 & \text{if } n \in \mathcal{N} \end{cases}, \quad \forall n \in \mathcal{N}_{s,t}, \forall v \in \mathcal{V} \quad (4.1b)$$

$$s_{j,t} = s_{j,t-1} + \Delta_j (d_{j,t} - \sum_{v \in \mathcal{V}} f_n^v - \alpha_{j,t}), \quad \forall n = (j,t) \in \mathcal{N} \quad (4.1c)$$

$$s_t^v = s_{t-1}^v + \sum_{n=(j,t) \in \mathcal{N}} \Delta_j f_n^v, \quad \forall t \in \mathcal{T}, \forall v \in \mathcal{V} \quad (4.1d)$$

$$\sum_{v \in \mathcal{V}} z_n^v \leq B_j, \quad \forall n = (j,t) \in \mathcal{N} \quad (4.1e)$$

$$z_n^v \leq \sum_{a \in \mathcal{R} \mathcal{S}_n^v} x_a^v, \quad \forall n = (j,t) \in \mathcal{N}, \forall v \in \mathcal{V} \quad (4.1f)$$

$$s_t^v \geq Q^v x_a^v, \quad \forall v \in \mathcal{V}, \forall a = ((j_1, t), (j_2, t')) \in \mathcal{A}^v : j_1 \in \mathcal{J}^P, j_2 \in \mathcal{J}^C \cup \{n_t\} \quad (4.1g)$$

$$s_t^v \leq Q^v (1 - x_a^v), \quad \forall v \in \mathcal{V}, \forall a = ((j_1, t), (j_2, t')) \in \mathcal{A}^v : j_1 \in \mathcal{J}^C, j_2 \in \mathcal{J}^P \cup \{n_t\} \quad (4.1h)$$

$$\sum_{t \in \mathcal{T}} \alpha_{j,t} \leq \alpha_j^{\max}, \quad \forall j \in \mathcal{J} \quad (4.1i)$$

$$0 \leq \alpha_{j,t} \leq \alpha_{j,t}^{\max}, \quad \forall j \in \mathcal{J}, \forall t \in \mathcal{T} \quad (4.1j)$$

$$F_{j,t}^{\min} z_{j,t}^v \leq f_{j,t}^v \leq F_{j,t}^{\max} z_{j,t}^v, \quad \forall n = (j,t) \in \mathcal{N}, \forall v \in \mathcal{V} \quad (4.1k)$$

$$S_{j,t}^{\min} \leq s_{j,t} \leq S_{j,t}^{\max}, \quad \forall n = (j,t) \in \mathcal{N} \quad (4.1l)$$

$$0 \leq s_t^v \leq Q^v, \quad \forall v \in \mathcal{V}, \forall t \in \mathcal{T} \quad (4.1m)$$

$$x_a^v \in \{0, 1\}, \quad \forall v \in \mathcal{V}, \forall a \in \mathcal{A}^v \quad (4.1n)$$

$$z_n^v \in \{0, 1\}, \quad \forall n = (j,t) \in \mathcal{N}, \forall v \in \mathcal{V} \quad (4.1o)$$

The model features two sets of binary decision variables: $x \in \mathbb{B}^{|\mathcal{A}|}$ and $z \in \mathbb{B}^{|\mathcal{M}| \times |\mathcal{V}|}$. x_a^v takes value 1 if vessel v uses a travel arc a . The binary variable z_n^v indicates whether vessel v loads/discharges at node n . The constraints ensure the coherency of the model as follows: constraints (4.1b) guarantee the conservation of vessel flow for each triplet of port, time, and vessel. Meanwhile, (4.1c) and (4.1d) ensure the inventory is balanced at each vessel and port throughout timesteps and maintained within limits. Constraints (4.1e) guarantee that the number of loads/discharges at a given pair of port and time does not exceed the number of available berths. The coupling constraints (4.1f) ensure a vessel can only load/discharge at a node if it is present. Constraints (4.1g) and (4.1h) require the vessels to travel at capacity from a loading region to a discharging region and empty when traveling in the opposite direction. The amount of product that a port can buy/sell in the spot market is restricted by (4.1i) and (4.1j). A thorough explanation of each of the symbols is provided in the Appendix.

Group 2 instances: the challenge for optimality

Group 2 instances feature long-horizon deterministic routing with a simplified mathematical model. The objective remains to minimize the transportation costs and penalties caused by buying/selling product in/to the spot market. The only integer decision variables are x_a^{vc} , which determine the number of vessels belonging to vessel class vc that take arc a . Similar to group 1 instances, constraints (4.2b) ensure the conservation of flow for each triplet of port, time, and vessel class. Inventory restrictions for each pair of port and time are specified in (4.2c), while constraints (4.2d) ensure that the number of vessels that attempt to load/discharge is limited by the number of berths available. This formulation does not require inventory tracking at vessels, since vessels are required to travel at capacity from a loading region to a discharging region and empty when traveling in the opposite direction. In contrast to group 1 instances, there are no constraints that limit the amount of stockout, although stockout is penalized in the objective.

$$\max \sum_{vc \in \mathcal{VC}} \sum_{a \in \mathcal{A}^{vc}} -C_a^{vc} x_a^{vc} + \sum_{j \in \mathcal{J}} \sum_{t \in \mathcal{T}} -P_{j,t} \alpha_{j,t} \quad (4.2a)$$

$$s.t. \sum_{a \in \mathcal{FS}_n^{vc}} x_a^{vc} - \sum_{a \in \mathcal{RS}_n^{vc}} x_a^{vc} = \begin{cases} 1 & \text{if } n = n_s \\ -1 & \text{if } n = n_t \\ 0 & \text{if } 0 \in \mathcal{N} \end{cases}, \quad \forall n \in \mathcal{N}_{s,t}, \forall vc \in \mathcal{VC} \quad (4.2b)$$

$$s_{j,t} = s_{j,t-1} + \Delta_j (d_{j,t} - \sum_{vc \in \mathcal{VC}} \sum_{a \in \mathcal{FS}_n^{vc,inter}} Q_a^{vc} x_a^{vc} - \alpha_{j,t}), \quad \forall n = (j,t) \in \mathcal{N} \quad (4.2c)$$

$$\sum_{vc \in \mathcal{VC}} \sum_{a \in \mathcal{FS}_n^{vc,inter}} x_a^{vc} \leq B_j, \quad \forall n = (j,t) \in \mathcal{N} \quad (4.2d)$$

$$\alpha_{j,t} \geq 0, \quad \forall n = (j,t) \in \mathcal{N} \quad (4.2e)$$

$$s_{j,t} \in [S_{j,t}^{\min}, S_{j,t}^{\max}], \quad \forall n = (j,t) \in \mathcal{N} \quad (4.2f)$$

$$x_a^{vc} \in \{0, 1\}, \quad \forall vc \in \mathcal{VC}, \forall a \in \mathcal{A}^{vc,inter} \quad (4.2g)$$

$$x_a^{vc} \in \mathbb{Z}_+, \quad \forall vc \in \mathcal{VC}, \forall a \in \mathcal{A}^{vc} \setminus \mathcal{A}^{vc,inter} \quad (4.2h)$$

4.3.2 Applying Parallel Alternating Criteria Search to MIRP instances

As presented in Section 3, Parallel Alternating Criteria Search does not exploit the underlying flow network structure featured in MIRP instances. It is a parallel heuristic designed for general purpose MIPs, and as such, no assumption on the MIP structure of the input problem can be made. Our primary focus is on the application of Parallel Alternating Criteria Search to group 1 instances, since these pose a greater challenge than group 2 from the perspective of feasibility. Our computational experiments presented in Subsection 4.4 indicate that the algorithm struggles considerably to converge to feasible solutions. The reason for this is an apparent stalling of the heuristic, and its inability to repair all infeasibilities. We illustrate the issue in Figure 4.2(a), where the performance of the heuristic is depicted for a particular problem instance. Specifically, we display the number of violated constraints in the incumbent solution as a function of time. The objective value of all the

solutions found by the heuristic are also plotted on the secondary axis. As seen in the chart, the infeasibility of the incumbent solution is reduced significantly at the beginning of the optimization. Meanwhile, the solution quality of the solutions found also converges to a value significantly lower than the best known bound for the problem. Since not all constraints are enforced, it is possible to obtain infeasible solutions with a better objective than the best bound. Parallel Alternating Criteria Search is unable to repair all infeasibilities. Figure 4.2(b) provides further information regarding the nature of the infeasible constraints. We determine that the algorithm fails in solutions featuring a few unsatisfied flow conservation constraints. While we illustrate the issue with a particular example, the same issue can be extended to most of the problems in the set, as detailed in Subsection 4.4.2.

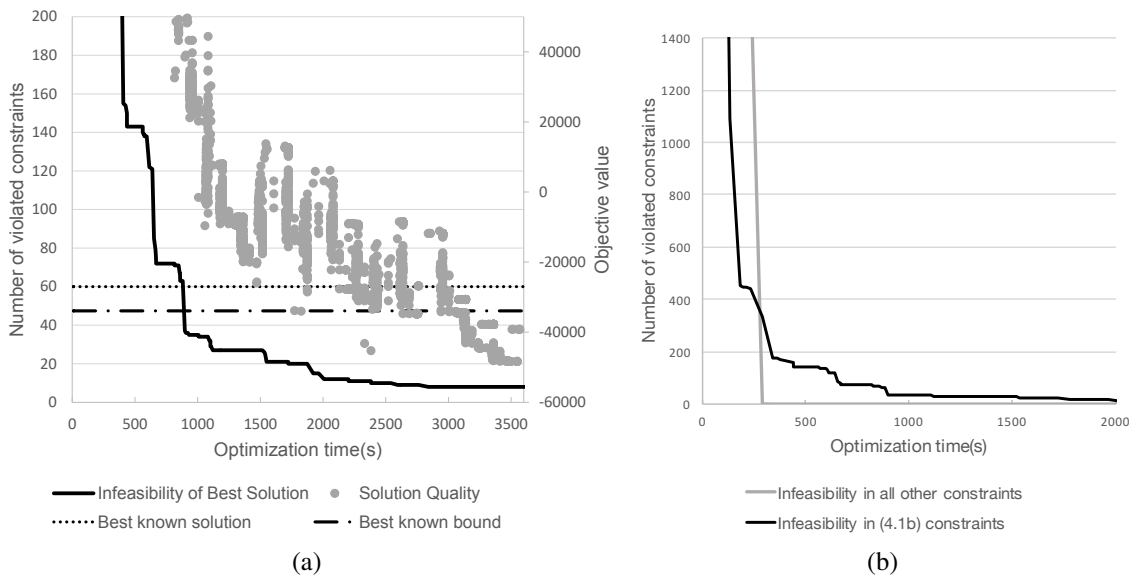


Figure 4.2: Application of Parallel Alternating Criteria Search to a group 1 MIRP instance, with 4 loading ports, 9 discharging ports, 17 vessels, and a horizon of 60 timesteps. (a) Depicts the evolution of the infeasibility of the incumbent, and objective value of found solutions as a function of time. (b) Shows a breakdown of the infeasibility found in the incumbent by constraint type as a function of time.

Tailoring Parallel Alternating Criteria Search to MIRP instances

We discuss two algorithm modifications, which attempt to increase the effectiveness of the algorithm at finding a first feasible solution. The first proposed modification is to the

objective of FMIP, with the intent of preventing infeasibility from accumulating in constraints that may be very hard to repair. Secondly, we introduce MIRP-specific variable fixing schemes, which may be more effective at finding quality solutions than their generic counterparts.

Objective penalizations

Finding feasible solutions for group 1 instances is challenging due to the small tolerances imposed in the inventory constraints of ports and vessels. A feasible vessel schedule must ensure enough product is transported to satisfy the consumption demand at every port and to avoid stockout. Vessels must carry the bulk of the product, since constraints $(4.1i) \cup (4.1j)$ severely limit the amount of product that can be bought or sold on the spot market. This is not the case on group 2 instance. In the latter, no upper bound on the spot market is imposed.

In order to avoid stalling, we propose Group1FMIP, a modification of FMIP in which we establish penalties in the objective for certain Δ variables. The main goal with this measure is to achieve feasibility in the most critical constraints first at the expense of allowing infeasibility to accumulate in a targeted group of constraints, which may be easier to repair in the future. In the spirit of group 2 instances, our intent is to prioritize the satisfiability of the vessel schedule over the inventory management at ports. Thus, we apply a large penalty Λ to all variables except the ones representing the constraints regulating port inventory limits (the set $(4.1i) \cup (4.1j)$). A priori, the generated solutions will provide feasible routing schedules for each vessel at the expense of violating many stock deficiency/excess constraints. As optimization advances, we hope the latter can be gradually fixed by systematically re-adjusting feasible vessel trips. The excess of slack $\alpha_{j,t}$ is also minimized in OMIP, since it is present in the original objective. Therefore, the amount of infeasibility is

minimized in both auxiliary MIP models.

$$\begin{aligned}
& \min \sum_{i=0}^m \Lambda(\Delta_i^+ + \Delta_i^-) + \sum_{i=0}^k \Delta_i^{+\alpha} + \Delta_i^{-\alpha} \\
& \text{s.t.} \\
& Ax + I_m \Delta^+ - I_m \Delta^- = b \\
& A^\alpha x + I_k \Delta^{+\alpha} - I_k \Delta^{-\alpha} = b^\alpha \\
& x_i = \hat{x}_i, \forall i \in \mathcal{F} \\
& l \leq x \leq u \\
& x_i \in \mathbb{Z}, \forall i \in \mathcal{I} \\
& \Delta^+ \geq 0, \Delta^- \geq 0 \\
& \Delta^{+\alpha} \geq 0, \Delta^{-\alpha} \geq 0
\end{aligned} \tag{Group1FMIP}$$

As seen in Group1FMIP, constraints are divided into two separate submatrices. A^α represents the submatrix related to the constraints regulating the excess slack $\alpha_{j,t}$, the set $(4.1i) \cup (4.1j)$. In turn, A contains the submatrix of the remaining constraints. Δ variables associated with the constraints in A are penalized with Λ in the objective, in order to prioritize their satisfiability.

A MIRP-specific variable fixing scheme

In Large Neighborhood Search, the set of variables to be fixed adjusts the difficulty of the subMIP to be optimized, as well as the effectiveness of the MIP solver at finding high quality solutions quickly. Parallel Alternating Criteria Search uses a simple, generic, yet intuitive variable fixing scheme to be able to tackle any kind of problem structure. Greater effectiveness in finding solutions can be achieved by considering the internal structure of the problem in variable fixing. We propose to use two kinds of MIRP-specific variable fixing schemes, which decompose the problems in complementary substructures. The first

is a variation of a vessel decomposition approach, and the latter is a modification of a time-window variable fixing scheme. Both incorporate randomness in order to satisfy the need for parallel diversified search neighborhoods.

As stated in Section 4.2, a prevalent large neighborhood search algorithm in the literature is the so called k-opt search, which entails fixing the variables belonging to all but a subset of vessels. This is a widely used strategy due to its simplicity and effectiveness. We specify ours with two key elements. We incorporate an input parameter ρ , which determines the proportion of vessels to be fixed. In addition, we incorporate randomness in the selection of vessels. Pseudocode is provided in Algorithm 10.

Algorithm 10 Random Vessel selection neighborhood

Input: Fraction of variables to fix $\rho, 0 < \rho < 1$
Output: Set of integer indices \mathcal{F}

- 1: **function** VESSELSELECTIONFIXING(ρ)
- 2: \mathcal{F} =set of all integer variable indices \mathcal{I}
- 3: **while** $|\mathcal{F}| > \rho \cdot |\mathcal{I}|$ **do**
- 4: i := random vessel $i \in \mathcal{V}$
- 5: Remove from \mathcal{F} all variable indices related to vessel i
- 6: **end while**
- 7: **return** \mathcal{F}
- 8: **end function**

While the k-opt search decomposes the problem by vessel, the second strategy we propose seeks to break the problem down by time. The time-window selection scheme involves establishing a time window between two timesteps and fixing all travel arcs outside of it. The generated neighborhood allows the improvement of a solution by modifying the travel schedules of all vessels between the allowed time window. Travel arcs related to all ports and vessels are left unfixed at the same time. Pseudocode is provided in Algorithm 11.

The time-window selection scheme as presented can become terribly ineffective at finding solution improvements if a small parameter ρ is selected, or if the problem features a large number of vessels and ports. In such cases, the produced variable selection may encompass a small time window. Any vessel trip longer than such time window will have a fraction of its active travel variables fixed and won't be rerouted when solving the associ-

Algorithm 11 Random time-window selection neighborhood

Input: Fraction of variables to fix ρ , $0 < \rho < 1$

Output: Set of integer indices \mathcal{F}

```

1: function TIMEWINDOWSELECTIONFIXING( $\rho$ )
2:    $\mathcal{F}$  =set of all integer variable indices  $\mathcal{I}$ 
3:   offset = 0
4:    $t :=$  random time  $t \in \mathcal{T}$ 
5:   while  $|\mathcal{F}| > \rho \cdot |\mathcal{I}|$  do
6:     Remove from  $\mathcal{F}$  all variable indices related to time  $t - \text{offset}$ , for vessels  $v \in \mathcal{V}$  and ports
        $p \in \mathcal{J}$ 
7:     Remove from  $\mathcal{F}$  all variable indices related to time  $t + \text{offset}$ , for vessels  $v \in \mathcal{V}$  and ports
        $p \in \mathcal{J}$ 
8:     offset = offset + 1
9:   end while
10:  return  $\mathcal{F}$ 
11: end function
  
```

ated LNS.

We present a modification of the standard time-window variable selection in Algorithm 12, in which a subset of the ports are fixed for each vessel in order to allow larger time windows. In order to determine which subset of ports remains fixed, we define the auxiliary concept of a port span. The port span of a vessel v between two time steps t_1 and t_2 in a solution x is the set of ports traversed by v during the time window in x . An example depicting multiple examples of port spans is shown in Figure 4.3. By its definition, the produced set contains only the ports visited by v within the time window. The proposed constrained time-window selection scheme incorporates the definition of a port span in order to include only the relevant ports for each vessel in the variable selection,

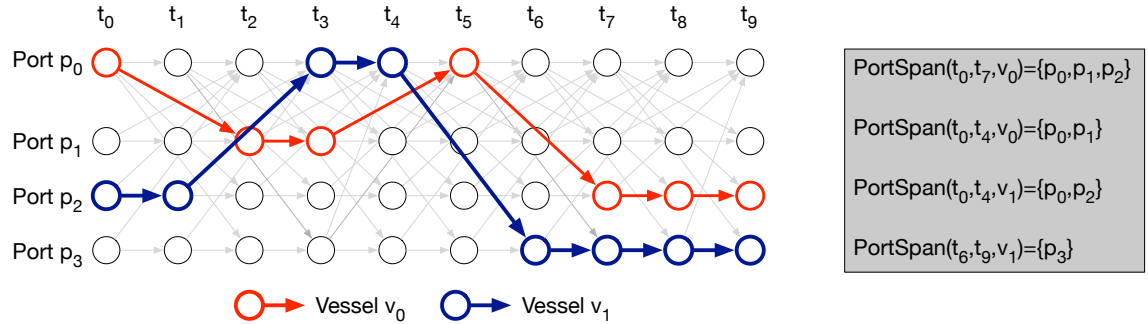


Figure 4.3: The port span of a vessel v between two time steps t_1 and t_2 in a solution x is the set of ports traversed by v during the time window in x .

while the remaining are fixed.

Algorithm 12 Constrained time-window selection neighborhood

Input: Fraction of variables to fix ρ , $0 < \rho < 1$, input solution x

Output: Set of integer indices \mathcal{F}

```

1: function CONSTRAINEDTIMEWINDOWSELECTIONFIXING( $\rho$ )
2:   Generate all pairs  $\langle p, t \rangle$ 
3:   Rank all pairs by inventory excess/deficit  $\alpha_{j,t}$ 
4:   Select pair  $\langle p', t' \rangle$  randomly among the pairs with most excess/deficit
5:    $\text{offset} = 0$ 
6:    $\mathcal{F}$  =set of all integer variable indices  $\mathcal{I}$ 
7:   while  $|\mathcal{F}| > \rho \cdot |\mathcal{I}|$  do
8:      $t_1 := t' - \text{offset}$ 
9:      $t_2 := t' + \text{offset}$ 
10:    for every vessel  $v \in \mathcal{V}$  do
11:       $P := \text{PORTSPAN}(t_1, t_2, v, x)$ 
12:       $P := P \cup p'$ 
13:      for all ports  $p \in P$  do
14:        Remove from  $\mathcal{F}$  all variable indices related to vessel  $i$  and port  $p$  between  $[t_1, t_2]$ 
15:      end for
16:    end for
17:     $\text{offset} := \text{offset} + 1$ 
18:  end while
19:  return  $\mathcal{F}$ 
20: end function
21: function PORTSPAN( $t_1, t_2, v, x$ )
22:    $P := \emptyset$ 
23:   for each  $t \in \{t_1, \dots, t_2\}$  do
24:     for each  $p \in \mathcal{P}$  do
25:       if vessel  $v$  is traversing  $p$  at time  $t$  then
26:          $P := P \cup p$ 
27:       end if
28:     end for
29:   end for
30: end function

```

The proposed algorithm also prioritizes the optimization of the port and time pairs with the largest excess/deficiency of product. For this purpose, all pairs of ports and time $\langle p, t \rangle$ are ranked by the value of their associated $\alpha_{p,t}$ variable. A single tuple $\langle p', t' \rangle$ is selected randomly among the pairs with the largest amount. Then, a time-window is defined using t' as its epicenter. For each vessel v , the algorithm proceeds to unfix the variables within the time window belonging to the port span and p' . The size of the time window is incremented gradually until the desired amount of variables is selected.

With $\langle p', t' \rangle$ as the epicenter of the time-window, we intend to rectify the deficiency

Algorithm 13 Variable fixing selection algorithm

Input: Fraction of variables to fix ρ , $0 < \rho < 1$, input solution x

Output: Set of integer indices \mathcal{F}

```
1: function HYBRIDVARIABLEFIXING( $\rho$ )
2:   Generate random integer  $n$ 
3:   if  $n\%2 = 0$  then
4:      $F :=$  VESSELSELECTIONFIXING( $\rho$ )
5:   else
6:      $F :=$  CONSTRAINEDTIMEWINDOWSELECTIONFIXING( $\rho$ )
7:   end if
8:   return  $\mathcal{F}$ 
9: end function
```

of stock at port p' and time t' by rerouting some vessel to p' before the timestep t' . By using the port span, we force non-relevant ports to remain fixed in hopes of producing an easier subproblem with a larger time-window that contains the high quality solutions.

If feasibility has already been achieved, pairs $\langle p, t \rangle$ are ranked by their absolute contribution to the original objective instead. Modifications in high ranking pairs will hopefully have a more significant impact in improving the solution.

Each of the proposed variable fixing strategies provides a substantially different set of search neighborhoods, which add to the diversity of the approach. We incorporate both by allowing each parallel thread to choose randomly among both strategies, as shown in Algorithm 13.

4.4 Experimental results

In this section, we evaluate the performance and behavior of the MIRP-specific Parallel Alternating Criteria Search (MIRPpacs) when solving problems of the MIRPLIB library [89]. Out of the 100 instances currently present, there are 28 categorized as group 1 instances, while the remaining belong to group 2. MIRPLIB group 2 instances feature a horizon of 120, 180 or 360 periods and are classified in three difficulty categories [88]. An instance is declared easy if at least one commercial MIP solver (in default settings) is able to close more than 90% of the gap (as defined in Section 3.3.2) in half an hour. On the other hand, if no MIP solver is able to close more than 10% of the optimality gap, the instance is labeled

as hard. In total, the library contains 21 easy instances and 26 hard instances. The difficulty of the remaining 25 instances is classified as medium. The difficulty of a problem instance is directly related to the number of ports, vessels, and timesteps it features. Most easy instances contain fewer than 5 discharging ports and 180 time periods, while all but 2 hard instances have more than 8 discharging ports and most of them more than 180 time periods. MIRPpacs is implemented in C++, using CPLEX 12.7.2 as a backbone solver. We compare our framework against the state-of-the-art general purpose MIP solver CPLEX 12.7.2, and the default version of Parallel Alternating Criteria Search (PACS). All of our computations are performed on an 8-node computing cluster, each with two Intel Xeon X5650 6-core processors (96 cores in total) and 24 GB of RAM memory.

We evaluate the quality of primal solutions in terms of the primal gap and primal integral, as described in Section 3.3.2.

4.4.1 Tuning of parameters and nondeterminism

MIRPpacs and PACS require two kinds of input parameters, that regulate the difficulty and the solution time of each LNS within the heuristic. $[\rho, t]$ determine the percentage of variables to be fixed and the LNS time limit. For group 1 instances, the tuple $[0.7, 5]$ is selected. In turn, group 2 instances are solved using $[0.2, 5]$, $[0.5, 5]$ and $[0.7, 5]$, for Easy, Medium and Hard instances respectively. CPLEX is set in its parallel nondeterministic distributed-memory setting in order to utilize all 96 available cores and with a focus on primal solutions (the emphasis on Hidden Feasible solutions setting). Settings are set to default, otherwise. All the compared parallel algorithms are of nondeterministic nature, and we repeat each of the experiments five times. Unless otherwise noted, the performance charts presented in this section display the average among all runs.

4.4.2 Group 1 MIRP instances

The following set of charts and table evaluate the quality of the solutions provided by the different methods when solving group 1 instances. In Figure 4.4(a), we show the evolution of the average primal gap as a function of time. As stated in its definition, a primal gap of 100% is assigned if no solution for a particular instance is found. At first glance, standard PACS is only able to find solutions for a small subset of instances. As a result it scores a comparatively higher average primal gap throughout the optimization. The reason for its poor performance has already been analyzed in section 4.3.2, and it is one of the main motivating factors behind the development of MIRPpacs. CPLEX is a fully fledged MIP solver. As such, it is not afflicted by the same stalling problem and performs slightly better. The algorithmic modifications introduced in MIRPpacs certainly make a significant difference in comparison to its generic counterpart. The specialized heuristic becomes effective at finding high quality solutions for most instances, and this reflects in a much improved average gap right from the beginning of the optimization.

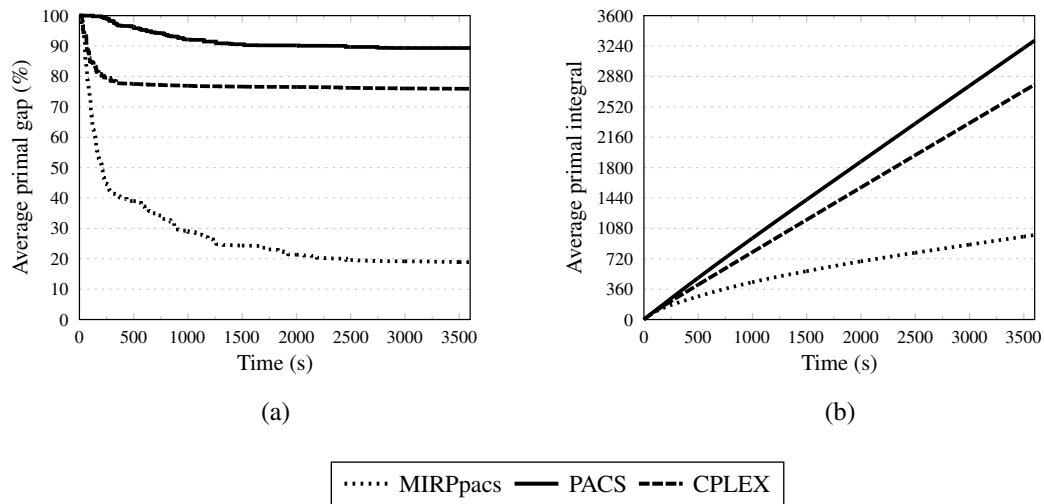


Figure 4.4: (a)Average primal gap and (b)average primal integral for group 1 instances

The same performance differences are reflected in Figure 4.4(b), in which the primal integral is depicted instead. The differences are quite significant after one hour of opti-

mization, as MIRPpacs shows an average primal integral that is 2.5 times smaller.

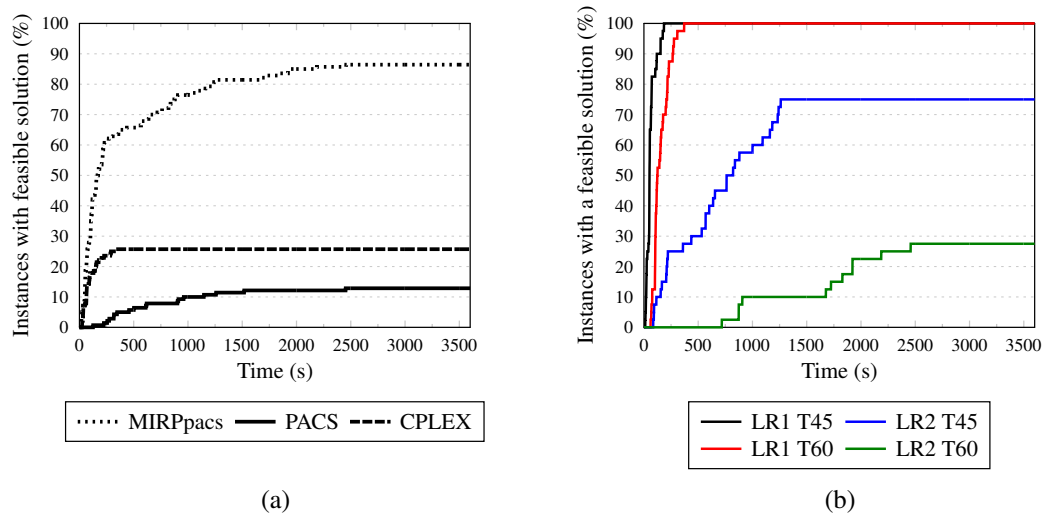


Figure 4.5: (a) Percentage of instances for which a feasible solution is found. (b) Percentage of instances for which MIRPpacs finds a feasible solution, broken down by instance subclass

Figure 4.5(a) plots the percentage of group 1 instances for which a feasible solution is found by each of the compared methods. MIRPpacs is able to find feasible solutions for 85% of the instances, while CPLEX is only able to do so for 25% and standard PACS for 18%. In Figure 4.5(b), the performance of MIRPpacs is shown after group 1 instances are subdivided by time period (45 periods. vs 60) and by number of Loading/Discharging port regions (1 L/D region vs 2). Results show that instances with a single region are significantly easier than their multiple region counterparts, as MIRPpacs is able to find solutions for 100% of the instances in less than 730 seconds. Increasing the number of regions raises the complexity. When 45 period instances are considered, solutions are found for 75% of the cases. Performance drops significantly for instances with two regions and 60 ports.

Table 4.1 provides further details about the variability in the performance of the two best non-deterministic approaches. We analyze the statistical results of the multiple executions in terms of the average, standard deviation, minimum, maximum, and median of the

primal gap and primal integral for different time cutoffs. MIRPpacs shows more variability than CPLEX, especially at early stages of the optimization. However, the relative distance between both diminishes with time. Even when the worst run of MIRPpacs is compared with the best run of CPLEX, the parallel heuristic displays a better primal gap after 180 seconds than the one accomplished by CPLEX after one hour.

Table 4.1: Group 1 performance comparison of non-deterministic approaches

Time Cutoff(s)		MIRP Parallel Alternating Criteria Search			CPLEX (Opportunistic Mode)		
		180	600	3600	180	600	3600
Avg.	P. Gap(%)	53.02	37.10	18.89	80.96	77.36	75.90
	P. Integral	137.16	311.19	1001.33	160.31	489.16	2782.76
	Count (%)	52.14	67.86	86.43	22.14	25.71	25.71
Std dev	Primal Gap	9.51	4.40	3.84	1.74	1.62	2.05
	Primal Integral	10.10	29.63	128.12	1.17	9.06	60.65
	Count (%)	9.94	3.19	1.60	1.60	1.60	1.60
min	Primal Gap	42.10	31.88	13.41	78.44	74.72	72.51
	Primal Integral	125.98	274.80	837.57	158.92	475.78	2679.98
	Count (%)	42.86	64.29	85.71	21.43	25.00	25.00
median	Primal Gap	54.13	37.32	19.45	81.21	77.86	76.58
	Primal Integral	136.45	310.89	1005.48	160.22	490.63	2804.07
	Count (%)	50.00	67.86	85.71	21.43	25.00	25.00
max	Primal Gap	63.24	42.13	22.56	82.32	78.37	77.17
	Primal Integral	149.93	346.80	1158.71	161.75	496.56	2818.57
	Count (%)	64.29	71.43	89.29	25.00	28.57	28.57

In Section 4.3.2, we presented a modification of the standard time-window variable fixing approach. In the proposed variant, a subset of the ports are fixed for each vessel with the objective of allowing larger time windows. In addition, our variable scheme prioritizes optimizing the tuples of port and time with the largest amount of stock deficiency first. In Figure 4.6, we evaluate the impact of the proposed modifications by comparing MIRPpacs to a variant of the heuristic in which the standard time-window variable fixing strategy (Algorithm 11) is used instead.

Our proposed modification allows MIRPpacs to find feasible solutions for 7% more instances. After one hour of optimization the version using the standard time-window displays a primal integral that is 28% larger, and an average primal gap that is 32% higher.

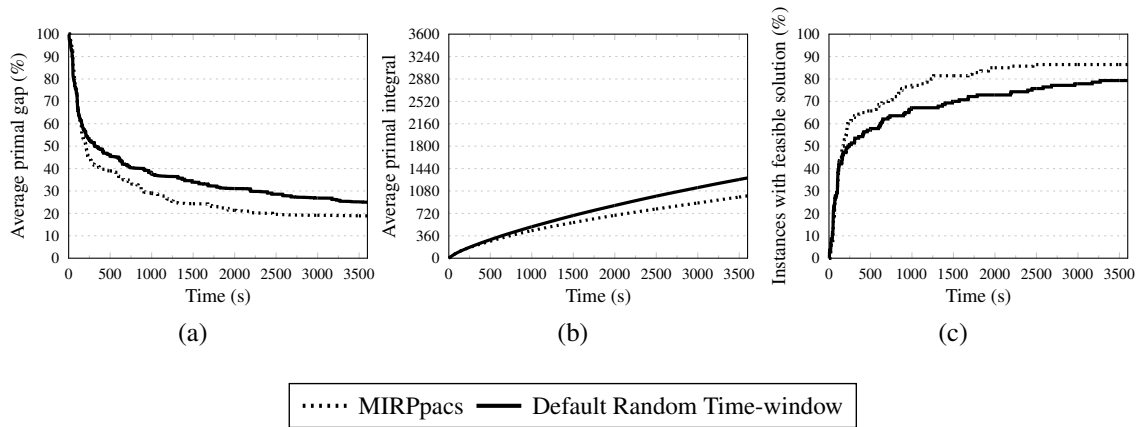


Figure 4.6: (a)Percentage of instances for which a feasible solution is found. (b)Percentage of instances for which MIRPpacs finds a feasible solution, broken down by instance sub-class

4.4.3 Group 2 MIRP instances

We incorporate a state-of-the-art MIRP-specific heuristic to the comparison, such as the rolling horizon heuristic as introduced in Papageorgiou et al. [88]. As explained in Section 4.2, the rolling horizon heuristic is a very common form of time decomposition applied to MIRPs. After the authors in the aforementioned work compared multiple state-of-the-art primal algorithms for MIRPs, the provided heuristic proved to be one of the best performing construction heuristics. While it is a sequential algorithm, it takes advantage of the shared-memory parallelism provided by the underlying MIP solver. The data presented in the following charts has been kindly provided by the authors. The rolling horizon heuristic (RHH) was run using a single computing node with 8 parallel cores. With the purpose of eliminating the discrepancies between computer systems, we also report its performance after normalizing the CPU times according to the performance metrics in Passmark [54]. Normalized times are calculated as $T_{norm} = \frac{T_{orig} \cdot S_{orig}}{S_{norm}}$, where T_{orig} is the original time reported by the authors, while S_{norm}^1 and S_{orig}^2 are the CPU scores of the processors used in our experiments and the other authors in the comparison respectively.

¹ $S_{norm} = 7605$ (Intel Xeon X5650)

² $S_{orig} = 14403$ (Intel Xeon E5-2687W)

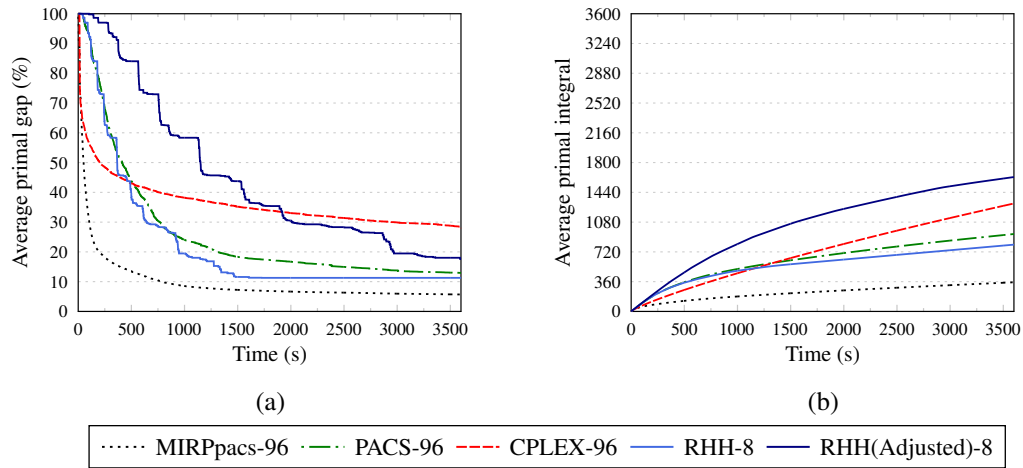


Figure 4.7: (a) Average primal gap and (b) average primal integral for all group 2 instances.

Figure 4.7(a) shows the evolution of the average primal time as a function of time for all compared methods. When all instances are considered, MIRPpacs proves to be the best performing algorithm. It is able to achieve an average primal gap of 20% in less than 168s, while the next algorithm to achieve the same requires 5 times as much time. It also requires less than 600s to find solutions with an average gap of less than 10%. The mark of 10% is not surpassed by any other method. The commendable performance of standard PACS is also worth noting, as it performs similarly to the non-normalized version of the rolling horizon heuristic despite being a heuristic for general purpose MIPs. CPLEX seems to follow a different pattern to the aforementioned heuristics, as it is able to perform on the level of MIRPpacs at the beginning of the optimization. However, it is surpassed by most heuristics after 500s and ends as the worst performing contender after 2000s. The rolling horizon heuristic proves to be a better performer than most parallel methods despite using only 8 cores.

The primal integral is plotted in Figure 4.7(b). Similarly reflected as in the previous plot, MIRPpacs shows a significantly lower average primal integral. Precisely, it is 2.5 times better than the next best contender after 3600s. CPLEX shows a slight advantage at the beginning of the optimization versus PACS and RHH. The advantage is neutralized

after 1200s.

In Figure 4.8, group 2 instances are split by difficulty category. When only small instances are considered, a fully featured MIP solver is able provide high quality solutions after other methods stagnate. CPLEX is able to outperform all methods at the end of the optimization and obtain the best average gap. However, MIRPpacs presents a lower primal integral due to the fact that it converges to high quality solutions much earlier in the search. The advantage of the primal heuristics gradually increases as the problem size grows. RHH shows better performance than PACS for both medium and hard instances, though PACS uses 12 times as many cores. The combination of domain-specific improvements and parallelism allow MIRPpacs to be particularly effective for hard instances in comparison to its general purpose counterpart.

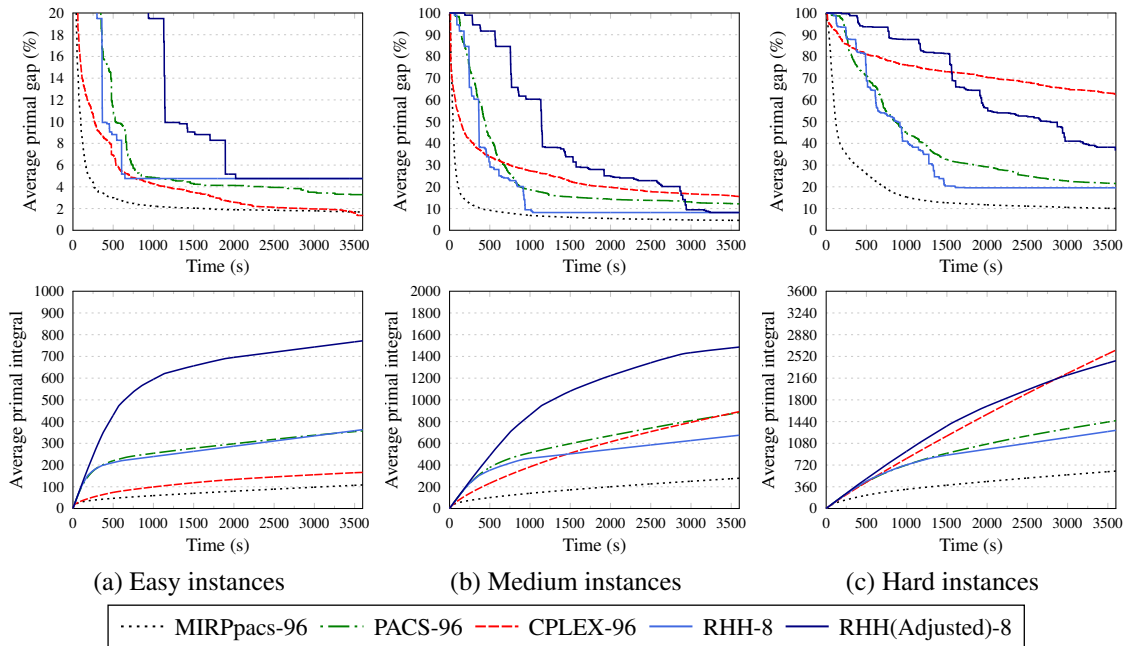


Figure 4.8: Average primal gap and average primal integral for (a) easy, (b) medium, and (c) hard group2 instances.

Table 4.2 provides details regarding the effects of nondeterminism on the variability between experiments. MIRPpacs has a similar variability to the one displayed by CPLEX. The standard deviation on the primal gap decreases as the optimization advances. The trend

seen in group 1 instances is maintained, as MIRPpacs achieves a better average primal gap in less than 180s than CPLEX is able to achieve after 1 hour of optimization. As a result MIRPpacs shows a primal integral that is 3.7 times better in average.

Table 4.2: Group 2 performance comparison of non-deterministic approaches

Time Cutoff(s)		MIRP Parallel Alternating Criteria Search			CPLEX (Opportunistic Mode)		
		180	600	3600	180	600	3600
Avg.	P. Gap(%)	20.75	12.17	5.70	51.19	41.76	27.98
	P. Integral	75.48	140.56	352.36	111.26	302.07	1305.22
Std dev	Primal Gap	3.84	3.71	2.23	3.29	2.56	2.24
	Primal Integral	5.75	18.79	86.46	3.55	12.52	67.22
min	Primal Gap	16.53	8.10	3.45	47.75	38.49	25.08
	Primal Integral	68.77	119.81	263.15	107.54	288.25	1226.04
median	Primal Gap	20.49	11.72	5.28	50.87	41.79	28.14
	Primal Integral	75.39	138.20	339.69	110.98	301.13	1305.10
max	Primal Gap	25.82	17.15	8.95	55.51	44.75	30.56
	Primal Integral	82.97	166.18	476.49	116.09	318.04	1391.54

At the time of this writing, primal heuristics remain the best available option for handling MIRP instances of practical size, as they significantly outperform state-of-the-art MIP solvers in the process. In turn, specialized heuristics will always outperform their general purpose counterparts.

4.4.4 Indications for specializing Parallel Alternating Criteria Search

We conclude this section with further indications on how to specialize Parallel Alternating Criteria Search for any problem at hand. Since solution improvements are driven by LNS, it is paramount to design specialized high quality search neighborhoods in order to maximize the effectiveness of the parallel primal heuristic. A significant quantity of them can be immediately found in the literature for most common MIP applications. All high performing variable selection schemes intend to preserve the integrity and cohesiveness of substructures within the problem, as this is the key for decomposing it effectively. A successful variable fixing must identify which variables must be changed in order to find a

better solution, free them, and leave the remainder fixed.

If feasibility is also a challenge, it is important to establish an effective objective penalization, so that stalling can be avoided. The main goal is to achieve feasibility in the most critical constraints first. As a result, infeasibility is driven to secondary constraints instead, which will be easier to repair in the future. In the case of MIRPs, the first task was to ensure a feasible routing schedule for each vessel, which resulted in numerous stock deficiencies at many ports. However, the latter can be gradually absorbed by systematically rerouting feasible vessel trips.

4.5 Conclusions

Parallel Alternating Criteria Search proves to be an effective framework when solving Maritime Inventory Routing Problems, but it can be significantly improved by tailoring a few of the key components of the algorithm. Firstly, we introduce specific objective penalizations, with the intent of improving the convergence to a first feasible solution. Secondly, we introduce new definitions of MIRP-specific variable fixing schemes, in order to improve the effectiveness of the large neighborhood search. The new specialized parallel heuristic is able to severely outperform state-of-the-art MIP solvers and domain specific heuristics. The advantage increases considerably when solving hard instances featuring long horizon periods, and a large number of ports and vessels.

Parallel Alternating Criteria Search is an excellent platform that the MIP practitioner can rely on as is, or for a rapid prototyping of effective parallel heuristics for any particular MIP domain. In turn, the parallel heuristic is a highly versatile tool that can be applied as a standalone heuristic or in the context of an exact algorithm. We hope this work will motivate other researchers to consider applying parallel computing to their own MIP domains.

CHAPTER 5

PIPS-SBB: A PARALLEL DISTRIBUTED-MEMORY BRANCH-AND-BOUND ALGORITHM FOR STOCHASTIC MIXED-INTEGER PROGRAMS

5.1 Introduction

Stochastic mixed-integer programs (SMIPs) are a generalization of mixed-integer Programs (MIPs) to deal with optimization under uncertainty. Consider the MIP

$$\min_{x \in \mathbb{R}^n} \{c^T x : Ax = b, l \leq x \leq u, x_j \in \mathbb{Z}, \forall j \in I \subseteq [n]\}, \quad (\text{MIP})$$

where $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, and I is the set of integer variable indices. Throughout this chapter, we use $[n]$ to denote the set $\{1, \dots, n\}$. For ease of exposition, we assume that x is bounded below by $l \in \mathbb{R}^n$ and above by $u \in \mathbb{R}^n$. We assume that all problem data belong to the set of rationals; see [110] for complications that arise in the presence of irrational data.

Typically, stochastic optimization problems are formulated as multi-stage optimization problems where some model parameters are random variables (with known probability distributions). In each stage, a decision has to be made, and after each decision is made, one learns the realization of some of the random variables. Usually, the goal is to minimize the expected total cost, where the expectation is over all realizations; see [111] for a detailed discussion.

In this work, we focus on two-stage SMIPs, a common variant in which the optimization problem is subdivided into two stages. For two-stage SMIPs, the first-stage variables determine the set of decisions before the uncertainty takes place. Second-stage variables represent the set of decisions to be taken once the uncertainty is revealed, as recourse to the

decisions taken in the first stage. A two-stage SMIP takes the form

$$\min_{x \in \mathbb{R}^{n_1}} \{c^T x + \mathbb{E}_P[Q(x, \xi)] : Ax = b, l \leq x \leq u, x_j \in \mathbb{Z}, \forall j \in I_1 \subseteq [n_1]\}, \quad (\text{SMIP})$$

where x is the first-stage decision variable, ξ is a random vector with support Ξ with a known probability distribution P , \mathbb{E} is the expectation operator over this probability distribution, and I_1 is the set of first-stage integer variable indices. For a given first-stage solution x , the second-stage optimization problem is of the form

$$Q(x, \xi) = \min_{y \in \mathbb{R}^{n_2}} \{q(\xi)^T y : W(\xi)y = h(\xi) - T(\xi)x, l(\xi) \leq y \leq u(\xi), y_j \in \mathbb{Z}, \forall j \in I_2 \subseteq [n_2]\}, \quad (\text{SS}_\xi^x)$$

where $W(\cdot)$, $h(\cdot)$, $l(\cdot)$, $u(\cdot)$, $q(\cdot)$ and $T(\cdot)$ may depend on ξ , but not on x . We assume that I_2 , the set of second-stage integer variable indices, does not depend on ξ . If (SS_ξ^x) is infeasible, then we set $Q(x, \xi) = \infty$. Throughout this chapter we assume finite support for ξ .

As defined so far, the main computational challenges in solving SMIPs exactly arise from the difficulty in optimizing the expected cost, which is non-convex. However, SMIPs can be approximated via Sample Average Approximation [112], thus transforming the problem into one large MIP as seen in (EXT) (usually called the *extensive formulation*) and enabling the use of traditional MIP approaches. We use $[s]$ to represent the set of possible sampled realizations, also known as scenarios. Suppose scenario i has probability p_i .

such as CPLEX [76], Xpress [113], SCIP [114], or GUROBI [77]. General purpose MIP solvers use an enumerative tree search algorithm known as branch-and-bound in which linear programming (LP) relaxations are solved at each node of the branch-and-bound tree; see Section 5.2.1 for more details. For the remainder of this work, we do not distinguish between SMIPs and their extensive formulations; in general, by “SMIP”, we refer to the extensive formulation in (EXT).

General purpose solvers do not recognize and/or leverage the dual block-angular structure of the extensive formulation. Furthermore, SMIPs present additional computational challenges. For example, SMIPs become harder to solve efficiently as the number of scenarios grows, mainly due to the increase in problem size. In addition to increased solution times, the problem may not fit in memory at all. Shared-memory and distributed-memory versions of branch-and-bound algorithms have been implemented by these state-of-the-art MIP solvers and other MIP solvers such as ParaSCIP ([78, 115]), BLIS ([116, 117]), and PICO [118]. However, all of these efforts focus on parallel branch-and-bound, and not on parallelizing the underlying LP relaxation solved at each node of the branch-and-bound algorithm. Though state-of-the-art MIP solvers are highly optimized when run in sequential mode, prior studies have shown that the branch-and-bound search algorithm does not scale well beyond modest amounts of parallelism [25] and thus these solvers do not leverage recent advances in HPC architectures. Scalability is also limited by memory when optimizing extended formulations. Given that these parallel implementations do not support data distribution, highly detailed approximations are computationally intractable and coarse-grained versions (with fewer scenarios) must be used instead, resulting in lower quality solutions to the original SMIP.

5.1.1 Contributions and overview

In our approach, we leverage the dual block-angular structure of SMIPs to solve LP relaxations at each node of the branch-and-bound tree using a parallel algorithm. The simplex

method is the default LP algorithm, and is known for its limited parallel scalability despite some recent parallelization efforts; see [119] for a review of the challenges involved. On the other hand, scalable parallel implementations of both primal and dual simplex have been recently developed (PIPS-S [120]) to solve LP problems with dual block-angular structure. In our approach, we use the PIPS-S solver to solve the LP relaxations of the extensive formulation (which have dual block-angular structure) at the nodes of the branch-and-bound tree. This allows us to build a distributed-memory branch-and-bound-based solver for Stochastic MIPs called PIPS-SBB (PIPS - Simple Branch-and-Bound)¹. To improve the performance of PIPS-SBB, we also developed new (and computationally efficient) methods for preprocessing, cut generation and heuristics that maintain the dual block-angular structure of SMIPs. These methods also apply to any MIP with dual block-angular structure; we focus on the SMIP case due to its prevalence in the literature and in applications.

The main contributions of PIPS-SBB, a novel branch-and-bound algorithm for general two-stage stochastic mixed-integer programs, are the following.

- PIPS-SBB is the first branch-and-bound algorithm to solve LP relaxations using a distributed-memory simplex algorithm that leverages the structure of SMIPs.
- By distributing SMIP data such that each (A, b_0, T_i, W_i, b_i) tuple for $i \in [s]$ is stored in memory on only one MPI [84] process, PIPS-SBB can leverage the distributed-memory architecture of supercomputers to address more memory to store and solve LP relaxations. In contrast, existing MIP solvers must load the entire extensive formulation in memory on each process that solves LP relaxations; some of these solvers parallelize the branch-and-bound tree search, not the solution of LP relaxations.
- Adapting methods known to accelerate MIP solvers (Presolve [5] and Primal Heuristics [4]) to a distributed-memory setting with dual block-angular data structure, we

¹The name PIPS-SBB deliberately alludes both to COIN-OR's now defunct Sbb (simple branch-and-bound) MIP solver, developed by John Forrest, and PIPS-S. The Cbc solver replaced Sbb in COIN. PIPS abbreviates "Parallel Interior Point Solvers"; the name was chosen prior to the work in [120] on parallel simplex algorithms for solving dual block-angular LPs.

present initial results on benchmark SIPLIB [121] instances that show the effectiveness of our method.

- Building PIPS-SBB as a modular, expandable codebase, we enable easy implementation of features, modifications, and extensions as plug-ins.

The remainder of this chapter proceeds as follows: First, we review related work on SMIP decomposition schemes to conclude Section 5.1. Then, in Section 5.2, we describe branch-and-bound algorithms, and present the main ideas behind PIPS-SBB. In particular, we focus our design choices for data distribution and parallelism. In Section 5.3, we describe the structure of PIPS-SBB in detail, in particular its extensible software implementation. We also describe the special-purpose distributed-memory algorithms implemented in PIPS-SBB that leverage the dual block-angular data structure of stochastic MIPs. We present computational results in Section 5.4 using instances from SIPLIB to illustrate the effectiveness of our algorithms. Finally, we look forward to the future, presenting next steps and ideas in Section 5.5.

5.1.2 Related work: Solving SMIPs using decomposition schemes

Most of the work in the literature avoids solving extensive formulations and alternative problem decompositions are devised instead.

One approach is to derive convex approximations of the expected second-stage cost within an iterative algorithm, such as Benders' decomposition. Such iterative schemes are collectively known as stage-wise decomposition schemes; see [122] and [123] for a detailed survey. Among its strengths, stage-wise decomposition can leverage the dual block-angular problem structure. However, these schemes are computationally impractical due to their slow convergence. Furthermore, they typically cannot handle the most general case where both stages are mixed-integer. To our knowledge, the only stage-wise scheme without variable type limitations has been presented by [124]. In this work, the authors present a

novel convergent generalization of Benders' algorithm, albeit in a theoretical context. In contrast, our scheme is practical and general (applicable to any two-stage SMIP) as it allows binary, continuous and discrete variables in both stages. Other works based on stage-wise decomposition schemes such as [125, 126, 127] present more limitations.

Alternatively, one can rely on scenario-based decomposition schemes. In such schemes, stages are decoupled by the replication of first-stage variables for each scenario. Then, non-anticipativity constraints are used to ensure these first-stage variables remain identical across scenarios. The decomposition is achieved by relaxing such constraints. Many heuristic scenario-wise decomposition strategies have been developed for stochastic mixed-integer programs. For example, Progressive Hedging iteratively solves single-scenario relaxations until convergence. Progressive Hedging has been used as a successful primal heuristic [128] and to obtain lower bounds [129]. To our knowledge, the first exact scenario-wise decomposition scheme was presented by [130], in which the authors solve Lagrangian duals at each node of a branch-and-bound procedure. In [131], the authors expanded the same work by presenting a parallel algorithm, which showed potential for parallel speedup, as well as some barriers to scalability. However, the authors only attempt to solve a single relaxation and do not address the challenges of incorporating such schemes within a branch-and-bound framework. In combination with Progressive Hedging, a recent parallel implementation of the dual decomposition scheme looks highly promising [132]. Other scenario-based decomposition schemes with variable type limitations include [133], where cover inequalities are used to solve SMIPs with pure binary first-stage variables.

Additional implementations of branch-and-bound algorithms using decomposition-based relaxations such as Generic Column Generation (GCG) [134], BapCod [135] and Dip [136] are typically not competitive when compared to LP-relaxation based MIP solvers since they rely on relaxations that cannot be warm-started within a branch-and-bound scheme.

There exists very few parallel software libraries that model and solve mixed-integer stochastic programs, with the exception of PySP [128]. In [137], the authors introduce DSP

- a parallel implementation based on a Lagrangian scheme in combination with Benders-type cuts. In [138], the authors introduce a parallel implementation of Benders decomposition for stochastic MIPs with first-stage integer variables. There are many sequential implementations of stage-wise and scenario-wise decomposition schemes; a list of software is maintained at [139].

5.2 PIPS-SBB: A specialized parallel distributed-memory branch-and-bound solver for large-scale stochastic MIP problems

PIPS-SBB is a parallel branch-and-bound framework for MIPs that feature a dual block-angular structure, such as the extensive formulation (EXT). This dual block-angular structure offers opportunities for parallelism at many levels of the optimization process, which will eventually enable PIPS-SBB to solve significantly larger extensive formulations than existing technologies. Exploiting these opportunities for parallelism also has the potential to reduce significantly computation times. In this work, we leverage this dual block-angular structure to induce task and data parallelism by distributing MIP data across multiple processors, and solving LP relaxations in parallel using PIPS-S² within the MIP infrastructure provided by PIPS-SBB.

In this section, we overview the main ingredients of PIPS-SBB. First, in Section 5.2.1, we overview the branch-and-bound algorithm. Then, in Section 5.2.2, we discuss how to expose parallelism in solving the LP relaxation. This discussion segues into a description of MIP data distribution in Section 5.2.3. We defer discussion of structure-aware MIP infrastructure details such as developing branching rules, primal heuristics, and presolve to Section 5.3.

²Aside from reductions, the algorithms presented in [120] operate on second-stage blocks of dual block-angular LPs independently.

5.2.1 Branch-and-Bound

In branch-and-bound [2], a mixed-integer program (MIP) is solved to optimality by systematically partitioning and searching the solution space using a tree data structure called a branch-and-bound tree to enumerate feasible integer solutions. In LP-relaxation based branch-and-bound, an LP relaxation (formed by relaxing all integrality constraints) is solved at each node of this tree. The objective value of the solution to the resulting LP relaxation (*fractional solution*) provides a lower bound on the MIP solution value. If an optimal solution of the LP relaxation is *integer feasible*, then this point is also a feasible solution to the MIP. The objective value of this feasible solution provides an upper bound on the MIP optimal solution value. However, if the calculated LP relaxation solution is not integer feasible, either the corresponding node is deleted (pruned) or the node is divided into two or more nodes with the use of additional inequalities. The former step is *bounding*, one of the main steps of the branch-and-bound algorithm, and can be done if the LP relaxation solution objective value is larger than the best upper bound (from the objective value of all integer feasible solutions found so far). The latter step is *branching*, in which inequalities are added to eliminate the fractional solution (which is LP-feasible but not integer feasible) and divide the solution space such that no feasible solutions to (MIP) are cut.

During the search process, let L be the current best lower bound and let U be the current best upper bound (also the objective value of the current best integer-feasible solution). Progress in the branch-and-bound algorithm is measured in terms of the *relative gap*, defined by

$$\frac{U - L}{10^{-10} + |U|}, \quad (\text{RelGap})$$

as in CPLEX. The branch-and-bound algorithm terminates when the relative gap is less than a given tolerance, or when there are no nodes remaining in the branch-and-bound tree.

State-of-the-art MIP solvers build upon this branch and bound scheme and enhance it

with many additional algorithmic practices to improve its performance, primarily by focusing on improving the upper and lower bounds. Primal heuristics ([3, 4]) are essential for finding high quality integer feasible solutions (better upper bounds) early in the search and reducing the solution space by pruning. Better lower bounds are obtained by developing stronger formulations. One method for strengthening formulations adds cutting planes (inequalities) [7] that strengthen the LP relaxation by eliminating parts of its feasible space without eliminating any integer feasible solutions. Another method for strengthening formulations is pre-processing [5], in which additional information about the problem structure can be derived from the constraints, potentially improving coefficients, eliminating redundant constraints, tightening variable bounds, and even fixing the value of some of the variables. The effectiveness of a MIP branch-and-bound tree search algorithm also depends on tree creation algorithms (branching rules) that determine how to partition the feasible space [17]. State-of-the-art MIP solvers are highly optimized in all these aspects, representing over two decades of research [140]. The current version of PIPS-SBB contains a subset of these methods; we plan on implementing more in the future.

Beyond established methods for MIPs, the structure of extensive formulations enables additional algorithmic improvements. The two-stage hierarchical organization in (EXT) suggests that first-stage information may be more important than second-stage information, as first-stage variables may affect multiple scenarios simultaneously, while the impact of second-stage variables is restricted to a single scenario. For this reason, branching rules and primal heuristics in PIPS-SBB prioritize first-stage variables over second-stage variables; examples that illustrate this prioritization are presented in section 5.3.2. Leveraging the dual block-angular problem structure is a critical feature of PIPS-SBB. In Section 5.4.3, we see how specialized branching rules and heuristics allow to reduce the relative gap faster.

5.2.2 Parallelism in the LP relaxation

At the very heart of a branch-and-bound algorithm, the LP relaxation provides a lower bound on the best MIP solution at every node of the branch-and-bound tree. Given its central importance, it is essential that LP relaxations are solved as efficiently as possible. The decomposable nature of the extensive formulation for stochastic MIPs incentivizes the use of interior-point methods to speed up solution of LP relaxations. These algorithms are highly parallelizable, as shown recently ([141, 142, 143]). Despite their scalability and ability to tackle big problem instances, interior-point methods are not typically used in a branch-and-bound algorithm because these methods warm-start less efficiently (requiring about half as many interior-point iterations [144]) than simplex methods (sometimes requiring only a few pivot iterations). The enumerative nature of branch-and-bound makes warm-starting crucial for performance. For that specific reason, branch-and-bound algorithms typically favor the simplex algorithm to solve LP relaxations, since this algorithm can be warm-started for an LP relaxation from the optimal solution of its parent node.

Even though this is an area of active research [119], parallel simplex implementations have been unable to outperform substantively an efficient modern sequential simplex solver for general, unstructured LPs. However, it is possible to develop parallel algorithms that exploit the dual block-angular structure exhibited by stochastic LPs in the extensive form and outperform efficient modern sequential simplex solvers. PIPS-S [120] implements such an algorithm. PIPS-SBB builds upon PIPS-S and uses it as its core LP solver. Thus, PIPS-SBB is able to exploit parallelism in the LP relaxation of every branch-and-bound node. Exploiting parallelism within each node of the branch-and-bound tree is a novel departure from general purpose MIP solvers, which reserve parallelism to solving LP relaxations of multiple branch-and-bound nodes simultaneously.

5.2.3 Parallel data distribution

For scalability, PIPS-SBB is designed for distributed-memory parallel computer architectures. Distributed-memory paradigms assume the addressable memory space is segmented and distributed among individual processes, as depicted in Figure 5.1a. Due to this decentralized memory space, communications libraries such as MPI are required in order to coordinate among processes. PIPS-SBB uses only MPI collectives to communicate efficiently among processes, both in the branch-and-bound algorithm and while solving the LP relaxations.

In conjunction with a distributed-memory parallel simplex solver, data is also distributed across processes. PIPS-SBB distributes the data representing each scenario to different processes while first-stage information is replicated. In other words, W_i, T_i, q_i and b_i are allocated on a single process for each $i \in [s]$, while c, A and b_0 are replicated on all processes; see Figure 5.1b. This data distribution can be scaled to as many processes as scenarios specified in the input problem, which enables PIPS-SBB to solve large SMIPs that would not otherwise fit in memory. Every component of PIPS-SBB conforms to this data distribution policy, including PIPS-S. This data distribution policy also extends to other data stored by PIPS-SBB, such as cutting planes, variable bound updates (as a result of branching), and LP warm-start information.

5.3 Implementation

In this subsection, we present the structure of PIPS-SBB in detail, describing both its software architecture and its features. Section 5.3.1 discusses the main software components of PIPS-SBB and their concerns, including where users can add their own functionality, such as branching rules. Section 5.3.2 discusses how existing MIP algorithms are adapted to versions that are dual block-angular structure-aware, exploiting this structure to expose parallelism and increase performance.

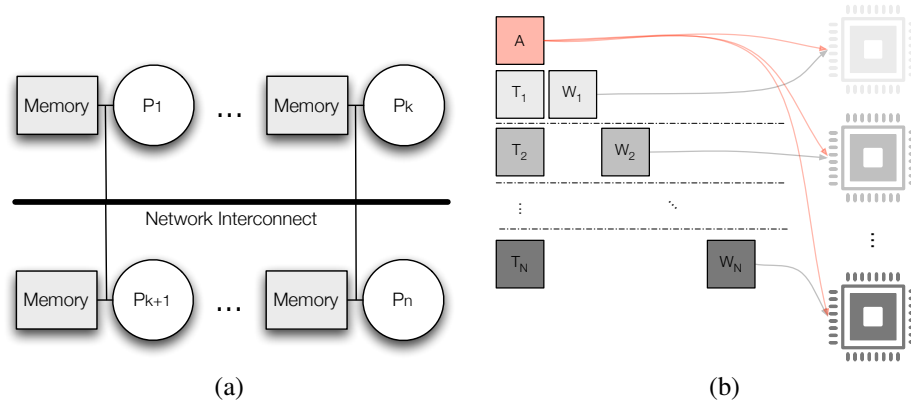


Figure 5.1: (a) Schematic depiction of a parallel system with a distributed-memory configuration. It has a segmented memory space, which is distributed among different processes. (b) Data parallelism in PIPS-SBB.

5.3.1 Software architecture of PIPS-SBB

PIPS-SBB is written in C++ and is designed to provide users with a flexible parallel framework suitable for solving any mixed-integer program with dual block-angular structure, which includes all two-stage stochastic mixed-integer programs. PIPS-SBB uses COINUTILS as an auxiliary library for much of its basic functionality.

PIPS-SBB code is distributed in two main software components, presented in Figure 5.2 using a schematic UML representation of the components as well as the interactions between them. The Solver Component manages the distributed problem data. It includes all algorithms that must directly access the distributed data, such as the presolver and the interface to the PIPS-S simplex solver. The Search State Component coordinates the branch-and-bound tree search and contains the current branch-and-bound algorithm state. It includes tree search algorithms such as branching rules and primal heuristics, and tree node data such as warm-start and branching information.

The problem formulation is read, stored and managed within the Solver Component of PIPS-SBB. The `BBSMPSSolver` class is one of the most critical components, as it acts as a proxy for outer software abstractions that may require LP relaxations or access to the solution pool. Due to its ubiquitous access from other classes, it is imple-

mented as a singleton class (represented by the ¹ next to the class name in Figure 5.2). The associated `BBSMPSResolver` class performs all data presolving operations and `PIPSSInterface` serves as the interface for the LP relaxation solver.

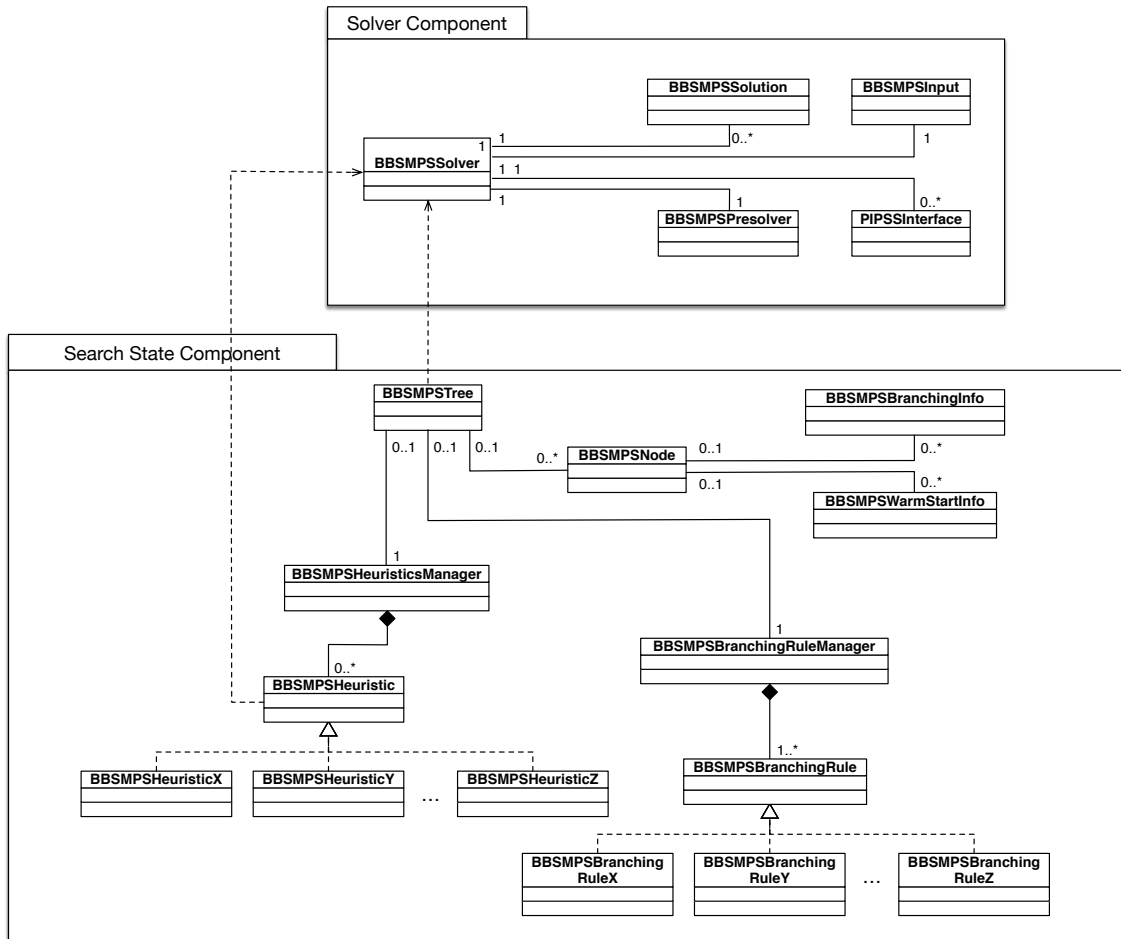


Figure 5.2: Class diagram of PIPS-SBB. Classes are shown as boxes with the top section as the name of the class. Interactions between classes are depicted as lines between classes, and indicate multiplicity indicators at each end, for example (1..*) representing “one or more”.

In the Search State Component, the `BBSMPSSTree` stores the collection of open nodes and controls the branch-and-bound tree search through a set of optimization managers. Currently, we have implemented PIPS-SBB Managers that provide users with a flexible and extendable framework to coordinate the execution of primal heuristics and branching rules within the search process. For instance, these managers enable users to determine which

primal heuristics are executed and the frequency with which they do so. Under the manager paradigm, the creation of additional heuristics and branching rules becomes a simple process, consisting of generating a new class by extending the generic `BBSMPSHeuristic` or `BBSMPSBranchingRule` and registering it in the appropriate manager. Observe from Figure 5.2 that there may be no `BBSMPSHeuristic` defined (0.*), but there must be at least one `BBSMPSBranchingRule` defined (1.*). In future versions of PIPS-SBB, other managers will coordinate the execution of other features, such as cutting-plane algorithms and tree search strategies.

In designing the architecture of PIPS-SBB, care is taken to minimize the memory footprint, allowing PIPS-SBB to solve large problem instances. For example, information related to relaxations such as LP warm-start information and branching decisions are stored incrementally with respect to the parent problem. Without incremental storage, storing tree information would quickly exhaust available memory, as is the case for the state-of-the-art solver CPLEX when solving certain problem instances; see Section 5.4.2.

5.3.2 Parallelism in structure-aware algorithms

The current version of PIPS-SBB features branching rules, primal heuristics, and presolve, all of which are designed and adapted to leverage dual block-angular problem structure and parallelism. There are two major design assumptions. First, every algorithm within PIPS-SBB must conform to the data distribution imposed in Section 5.2.3. Second, this data representation must remain distributed throughout the entire algorithm, and thus every MPI process is responsible for performing all operations on the data it owns. We will subsequently see in Section 5.4.3 that the specializations which leverage the dual block-angular problem structure significantly improve the performance of PIPS-SBB, allowing it to outperform state-of-the-art general purpose MIP solvers in some instances. In particular, these algorithmic specializations prioritize decisions on first-stage variables and constraints over second-stage variables and constraints. Algorithms 14, 15, and 16 show examples on

how these design assumptions are maintained in PIPS-SBB and the algorithms specialized to leverage the dual block-angular structure. For ease of exposition, in these examples we assume that each MPI process owns one scenario.

Branching Rules

The current version of PIPS-SBB features three specialized versions of branching rules: minimum infeasible index branching, most infeasible branching, and a more complex pseudo-cost branching. We refer the reader to [17] for a review of these and other branching rules for general purpose MIPs. Algorithm 14 illustrates the specialized most infeasible branching rule. It proceeds by identifying the most infeasible first-stage variable and returning its index if one is found. If no such variable is found, it searches in parallel for the most infeasible second-stage variable in each scenario. An all-to-all reduction is then required in order to find the most infeasible second-stage variable among all scenarios and communicate it to all processes.

Primal Heuristics

In addition, PIPS-SBB incorporates ten specialized versions of primal heuristics, ranging from simple rounding and diving schemes to more computationally expensive large neighborhood search schemes such as RENS. We refer the reader to [3] for a survey of primal heuristics for general purpose MIPs. Algorithm 15 shows the specialized diving strategy for finding feasible integer solutions, where an input fractional solution is iteratively rounded and bounded. After a variable rounding takes place, the LP relaxation is re-optimized. Once all first-stage variables become integer, each MPI process independently rounds one locally owned second-stage variable in each iteration. The procedure terminates when an integer solution is found or when the fixings render the LP relaxation infeasible.

Algorithm 14 PIPS-SBB Most infeasible branching rule

function BLOCKANGULARMOSTINFEASIBLEBRANCHING(x, y, comm) \triangleright Input: Integer infeasible LP-feasible solution, MPI communicator $scen := -1$ \triangleright Scenario number to be branched on; -1 is sentinel value for first-stage
if $I_1 \cap F \neq \emptyset$ **then** $\triangleright F$ is the index set of all fractional-valued first-stage variables
 return [$scen, \arg \max_j \{|x_j - \lfloor x_j \rfloor\}, j \in I_1$] \triangleright Return first-stage “scenario”, and variable index
end if $idx_i := -1$ \triangleright In this line and the following, -1 is a sentinel indicating integer feasibility
 $frac_i := -1$
 $scen := i$ \triangleright Scenario i owned by process (MPI rank) i
if $I_2 \cap F^i \neq \emptyset$ **then** $\triangleright F^i$ is the index set of all fractional second-stage variables of scenario i
 $idx_i := \arg \max_j \{|y_{i,j} - \lfloor y_{i,j} \rfloor\}, j \in I_2$
 $frac_i := |y_{i,idx_i} - \lfloor y_{i,idx_i} \rfloor$
end if \triangleright All-to-all reduce process number of maximum second-stage fractional valueMPI_Allreduce($[frac_i, scen]$, MPI_IN_PLACE, 1, MPI_DOUBLE_INT, MPI_MAXLOC, comm) \triangleright Broadcast index idx_i of maximum fractional value on process $scen$ to all processesMPI_Bcast(idx_i , MPI_IN_PLACE, 1, MPI_INT, $scen$, comm)**return** [$scen, idx_i$] \triangleright Returns scenario number and index of variable to branch on \triangleright If idx_i is -1, solution is integer-feasible**end function**

Presolving

MIP presolve in PIPS-SBB implements a specialized version of Savelsbergh [5, Section 1], with the exception of deleting redundant constraints. In order to accommodate the distributed nature of MIP problem data in PIPS-SBB, the presolve algorithm operates as in Algorithm 16. While presolve continues to modify the MIP, presolve first updates first-stage variable bounds, updates first-stage constraints, and assesses MIP feasibility. Then, because second-stage data is distributed by scenario, presolve processes second-stage constraints in parallel. This step updates first- and second-stage variable bounds, updates second-stage constraints, and assesses MIP feasibility. Since information on first-stage variable bounds and MIP feasibility may be different on different MPI processes due to preprocessing second-stage constraints in distributed fashion, this information must be synchronized across all processes via appropriate all-to-all reductions, as depicted in Algorithm 16.

Algorithm 15 PIPS-SBB Simple Parallel Diving Heuristic

function BLOCKANGULARPARALLELDIVINGHEURISTIC(x, y, comm)
 \triangleright Input: Integer infeasible LP-feasible solution, MPI communicator

Using x , compute F , the index set of all fractional-valued first stage variables

while $I_1 \cap F \neq \emptyset$ **do**
 $idx := \arg \max_j \{|x_j - \lceil x_j \rceil|, j \in I_1\}$ $\triangleright idx$ is index of most fractional variable in I_1
 Fix x_{idx} to the nearest integer value by modifying bounds: $l_{idx} = u_{idx} = \lceil x_{idx} \rceil$
 Solve LP relaxation of modified problem \triangleright This LP relaxation is solved in parallel using PIPS-S
 if LP relaxation infeasible **then**
 return Failure
 else
 $[x, y] :=$ optimal value of the LP relaxation
 end if
 Recalculate F using new x , index $idx \notin F$ since x_{idx} is fixed to integral value
end while
Fix all first-stage variables x by modifying bounds: $l = u = x$ \triangleright All x are currently integer-valued

\triangleright Scenario i owned by process (MPI rank) i
 \triangleright Compute total number of fractional-valued second-stage variables over all processes
Using y_i , compute F^i , the index set of all fractional-valued second-stage variables of scenario i
MPI_Allreduce($|I_2 \cap F^i|$, totalFracVars, 1, MPI_INT, MPI_SUM, comm)

while totalFracVars $>$ 0 **do**
 if $I_2 \cap F^i \neq \emptyset$ **then**
 $idx_i := \arg \max_j \{|y_{i,j} - \lceil y_{i,j} \rceil|, j \in I_2 \cap F^i\}$ $\triangleright idx_i$ is index of most fractional variable in $I_2 \cap F^i$
 Fix y_{i,idx_i} to the nearest integer value by modifying bounds: $l_{i,idx_i} = u_{i,idx_i} = \lceil y_{i,idx_i} \rceil$
 end if
 Solve LP relaxation of modified problem \triangleright This LP relaxation is solved in parallel using PIPS-S
 if LP relaxation infeasible **then**
 return Failure
 else
 $[x, y] :=$ optimal value of the LP relaxation $\triangleright x$ does not change in this line; it has been fixed
 end if
 Recalculate F^i using new y_i , index $idx_i \notin F^i$ since y_{i,idx_i} is fixed to integral value
 \triangleright Recompute total number of fractional-valued second-stage variables over all processes
 MPI_Allreduce($|I_2 \cap F^i|$, totalFracVars, 1, MPI_INT, MPI_SUM, comm)
end while

return $[x, y]$ \triangleright Returns integer feasible solution
end function

5.4 Experimental Results

We illustrate the performance of PIPS-SBB using instances from SIPLIB [121], a testbed of stochastic mixed-integer programs. In particular, we solve instances from the following test suites: Stochastic Server Location Problem (SSLP), Stochastic Server Location Problem Replication (SSLPRep), Stochastic Multiple Knapsack Problem (SMKP), and Dynamic CAPacity acquisition and allocation under uncertainty (DCAP). All our computations were performed on the Sierra Cluster at Lawrence Livermore National Laboratory. This cluster consists of 1,944 nodes, with nodes connected using InfiniBand QDR interconnects. Each individual node consists of 2 Intel 6-core Xeon X5660 processors and 24GB of memory.

Algorithm 16 PIPS-SBB Presolve

```
function BLOCKANGULARPRESOLVE( $A, T_i, W_i, b_0, b_i, I_1, I_2, l, u, l_i, u_i, \text{comm}$ )  
   $\triangleright$  Input: Coefficient matrices, right-hand side vectors, index sets, variable bounds, MPI communicator  
  
  while  $True$  do  
    [ $\text{isFeasible}, \text{isMIPchanged1}, A, b_0, l, u$ ] := FIRSTSTAGEPRESOLVE( $A, b_0, I_1, l, u$ )  
    if  $\text{isFeasible}$  is  $False$  then  $\triangleright$  Feasibility information is stored in a boolean variable  
       $\text{isFeasible}$   
      return MIP is infeasible  
    end if  
  
     $\triangleright$  Scenario  $i$  owned by process (MPI rank)  $i$   
    [ $\text{isFeasible}, \text{isMIPchanged2}, A, T_i, W_i, b_0, b_i, l, u, l_i, u_i$ ] :=  
      SECONDSTAGEPRESOLVE( $A, T_i, W_i, b_0, b_i, l, u, l_i, u_i$ )  
  
     $\triangleright$  Synchronize infeasibility information.  
    MPI_Allreduce( $\text{isFeasible}, \text{MPI\_IN\_PLACE}, 1, \text{MPI\_INT}, \text{MPI\_LAND}, \text{comm}$ )  
    if  $\text{isFeasible}$  is  $false$  then  
      return MIP is infeasible  
    end if  
  
     $\triangleright$  Synchronize whether MIP was modified  
     $\text{isMIPchanged} := \text{isMIPchanged1}$  or  $\text{isMIPchanged2}$   
    MPI_Allreduce( $\text{isMIPchanged}, \text{MPI\_IN\_PLACE}, 1, \text{MPI\_INT}, \text{MPI\_LOR}, \text{comm}$ )  
    if  $\text{isMIPchanged}$  is  $False$  then  $\triangleright$  Exit loop if MIP was not modified  
       $Break$   
    end if  
  
     $\triangleright$  Synchronize upper and bounds on  $x$ , which may be tighter from second-stage presolve  
    MPI_Allreduce( $u, \text{MPI\_IN\_PLACE}, n, \text{MPI\_DOUBLE}, \text{MPI\_MIN}, \text{comm}$ )  
    MPI_Allreduce( $l, \text{MPI\_IN\_PLACE}, n, \text{MPI\_DOUBLE}, \text{MPI\_MAX}, \text{comm}$ )  
  end while  
  
  return  $A, T_i, W_i, b_0, b_i, l, u, l_i, u_i$   $\triangleright$  Returns modified coefficient matrix  
end function
```

In all PIPS-SBB experiments, we bind 1 MPI process per core to ensure that cores are not over-subscribed with multiple processes. For our experiments, we built PIPS-SBB with MVAPICH2 version 1.7.

We note that PIPS-SBB can solve any stochastic MIP whose single scenario data can fit in the memory available to a single MPI process. On the other hand, general purpose solvers must be able to load the entire extensive formulation in the memory available to a single node. To be able to compare PIPS-SBB against the general purpose MIP solver CPLEX 12.6.2 running on a single node and using 12 threads (1 per processor), we chose

instances from SIPLIB for which the extensive formulation can be loaded in memory on a single node. Even then, CPLEX ran out of memory on a few instances due to a rapid growth in the branch-and-bound tree size.

5.4.1 Scaling experiments

First, we present results that demonstrate the scaling performance of PIPS-SBB. In particular, we show that PIPS-SBB scales as well as PIPS-S, the underlying distributed-memory LP solver. Note that both PIPS-S and PIPS-SBB can use no more MPI processes than the number of scenarios. We present scaling results on instances from SSLP [145], since this test set contains the largest variation in the number of scenarios, with instances ranging from 5 to 2000 scenarios. The SSLP instances model server location problems using a pure binary first-stage and mixed-binary second-stage. They are written in the form $sslp.m.n.s$, where m is the number of potential server locations, n is the number of potential clients, and s is the number of scenarios.

To measure the strong scaling performance of PIPS-S, we calculate the speedup in solving the LP relaxation at the root node of the PIPS-SBB branch-and-bound tree of the instances $sslp.10.50.*$. Defining speedup (a function of the number of cores N) as the ratio of (LP relaxation solution time with N cores) to (LP relaxation solution time with 5 cores), we see in Figure 5.3a that for the smaller instances, such as $sslp.10.50.100$, the speedup peaks at 10 cores. On the other hand, PIPS-S scales up to 25-50 cores for the large instances. In particular, it strong scales at 90% efficiency up to 10 cores for $sslp.10.50.2000$, and then strong scaling efficiency drops off quickly, with speedup peaking at 50 cores. This speedup curve is comparable to [120, Table 3], where the authors note that PIPS-S scaling depends on the relative sizes of the second- and first-stage coefficient matrices (T and A).

Based on these results, the current algorithms in PIPS-SBB will not strong scale to a large number of cores. Some opportunities for exposing additional parallelism are proposed in Section 5.5. Since PIPS-SBB solves an LP relaxation for each node of the branch-and-

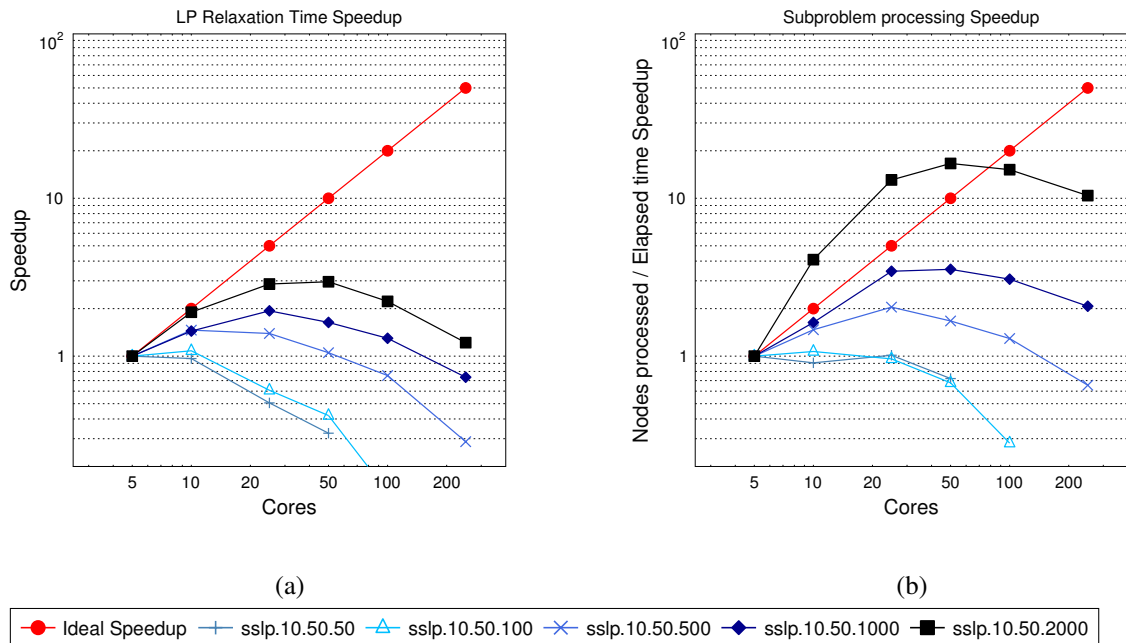


Figure 5.3: (a) Strong scaling performance results of PIPS-S. (b) Strong scaling throughput results of PIPS-SBB.

bound tree, one possible metric is its *throughput*, or the number of branch-and-bound nodes it can process per unit time. PIPS-SBB speedup (a function of the number of cores N) is therefore measured as the ratio of (Number of branch-and-bound Nodes Processed per second with N cores) to (Number of branch-and-bound Nodes Processed per second with 5 cores). For this experiment, we turned off all the computationally expensive branching rules and primal heuristics, tuning PIPS-SBB to process branch-and-bound nodes as quickly as possible. We see in Figure 5.3b that the speedup curves are very similar in shape to that of PIPS-S, with peak speedups occurring around 25-50 cores for the larger instances. This experiment illustrates that a stripped-down PIPS-SBB implementation continues to process nodes (and therefore LP relaxations) at roughly the same rate as PIPS-S.

Interestingly, PIPS-SBB shows super-linear throughput scaling for large problem instances. While this result seems surprising and counter-intuitive, it can be explained by a careful analysis of the experimental data. Consider an experiment that processes more than

one branch-and-bound node within the prescribed time limit. Among all of these nodes, the root node LP relaxation takes the longest, while the rest are typically solved within a few simplex iterations, since the LPs at all other nodes can be warm-started from the optimal solution of the LP relaxation at their parent node. As we increase the number of cores available, the LP relaxation solves faster (by a factor given by PIPS-S speedup) enabling PIPS-SBB to process many more nodes in the time limit. Since all these extra nodes are lightweight nodes (in terms of LP relaxation solution time), this results in a super-linear increase in the number of nodes processed per unit time, skewing the speedup numbers. Using the performance of a single core as the baseline accentuated this effect, and hence 5 cores were used instead. This skew suggests that throughput (as measured in this experiment) is not an accurate indicator of PIPS-SBB's ability to process nodes. Nevertheless, the scaling results presented in Figure 5.3 indicate that PIPS-SBB throughput scales in a manner consistent with that of PIPS-S performance.

We are primarily interested in how PIPS-SBB *wall clock time* scales. To this end, we consider the default PIPS-SBB algorithm (wherein primal heuristics and other features are enabled), and measure the time required by PIPS-SBB time required to close the relative gap (RelGap) to less than 2%. Presented in Figure 5.4, we see analogous performance curves, but with a decrease in speedup relative to the results shown in Figure 5.3b. This reduction is mainly due to the time spent by PIPS-SBB in computationally expensive primal heuristics in an attempt to find good feasible solutions. As before, performance peaks around 25-50 cores for the larger instances.

5.4.2 Overall performance

We illustrate the overall performance of PIPS-SBB on many instances from the SIPLIB library. For these experiments, we present results for a representative parallel processor configuration, where the number of cores is chosen as a function of the number of scenarios, based on our scaling experiments presented in Section 5.4.1. We report the number of

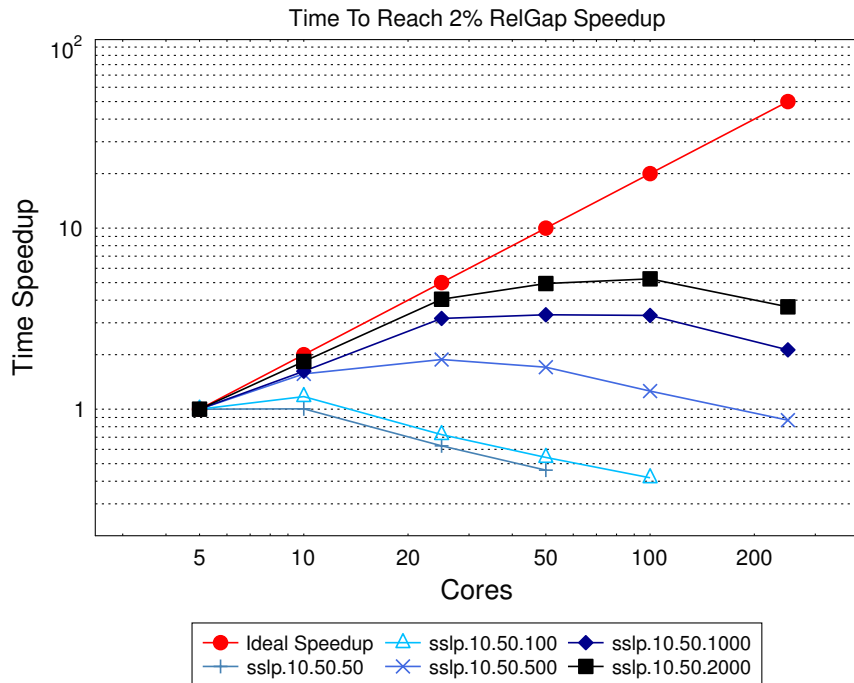


Figure 5.4: Strong scaling performance results of PIPS-SBB

scenarios and processor configuration as “Scenarios (Cores)” for all our experiments.

The instances are solved to a relative gap of 10^{-4} (CPLEX default). Each experiment is given a time limit of 1 hour (3600 seconds), and the performance results are reported as “(Time)” in seconds. If an optimal solution is not provably obtained within the time limit, then the performance results are reported in terms of relative gap, denoted by “RelGap”, and computed as in (RelGap). We also report the time (in seconds) at which PIPS-SBB found the best solution, as “Best Solution Time”. To measure the quality of U , the best solution found by PIPS-SBB, we present the percentage gap between the best upper bound found by PIPS-SBB and the best upper bound found by CPLEX, denoted as “Best Solution Quality”. This number could be negative if PIPS-SBB got a better quality solution than CPLEX at termination; such instances are marked in **bold**. For the instances solved to optimality by CPLEX but not by PIPS-SBB, this number indicates the quality of the solution obtained by

PIPS-SBB - it could still be 0%. For such instances (solved to optimality by CPLEX, but not by PIPS-SBB), the difference between Best Solution Quality and PIPS-SBB RelGap indicates how far the PIPS-SBB lower bound L is from the optimal solution. We also present the performance results of CPLEX in the “CPLEX RelGap (Time)” column. The instances where CPLEX ran out of memory are denoted with (M) next to the GAP at termination.

SSLP

The SSLP instance set is formed by 12 model server location problems [145]. As mentioned earlier, this set contains the largest variation in the number of scenarios (ranging from 5 to 2000), a pure binary first-stage and mixed-binary second-stage.

From Table 5.1, we see that PIPS-SBB outperforms CPLEX in 3 out of 10 instances. The first set of rows correspond to the *sslp.15.** instances, which have a small number of scenarios. We see that CPLEX shows better performance - it is able to solve the instances while PIPS-SBB is not. The second set of rows correspond to the easy *sslp.5.** instances, which both CPLEX and PIPS-SBB solve to optimality, though CPLEX is significantly faster than PIPS-SBB. The next two rows are instances that CPLEX solves, but PIPS-SBB does not. However, comparing the RelGap and Best Solution Quality entries, we see that the lower bounds obtained by PIPS-SBB at termination are close to the optimal solution, but its upper bound is poor. Furthermore, as the problems get more difficult as the number of scenarios increases (last three rows), PIPS-SBB is able to obtain better quality solutions than the primal heuristics implemented by CPLEX. Note that CPLEX has no knowledge that it is solving an extensive formulation, which results in its poor performance when the number of scenarios is large. In Section 5.4.3, we show that leveraging stochastic MIP problem structure significantly improves the performance of PIPS-SBB. CPLEX runs out of memory in the branch-and-bound tree search for *sslp.10.50.500*.

Table 5.1: SSLP instance set results

Problem Instance	Scenarios (Cores)	RelGap (Time)	Best Solution		CPLEX RelGap (Time)
			Time	Quality	
sslp.15.45.5	5 (2)	1.36%	1488s	1.07%	(4s)
sslp.15.45.10	10 (2)	7.93%	2129s	7.26%	(1s)
sslp.15.45.15	15 (2)	5.25%	2392s	4.84%	(12s)
sslp.5.25.50	50 (1)	(12.34s)	12s	0%	(1s)
sslp.5.25.100	100 (1)	(41.63s)	41s	0%	(1s)
sslp.10.50.50	50 (5)	1.48%	923s	1.31%	(81s)
sslp.10.50.100	100 (10)	1.74%	194s	1.56%	(442s)
sslp.10.50.500	500 (50)	1.57%	2792s	-7.32%	(M) 10.13%
sslp.10.50.1000	1000 (100)	1.60%	2397s	-11.19%	14.47%
sslp.10.50.2000	2000 (100)	24.00%	2384s	-0.73%	20.33%

SSLPRep

SSLPRep instances are slight variations of the SSLP set, available at [146]. The results displayed in Table 5.2 show a performance analogous to the SSLP instances. As before, while PIPS-SBB is not able to close the RelGap for the smaller instances, performance is comparable to CPLEX for the larger instances. As in the SSLP case, from column Best Solution Quality, we see that PIPS-SBB obtains better upper bounds than CPLEX for some large instances. Overall, PIPS-SBB outperforms CPLEX on 6 out of 50 instances.

Table 5.2: SSLPRep instance set results

Problem Instance	Scenarios (Cores)	RelGap (Time)	Best Solution		CPLEX RelGap (Time)
			Time	Quality	
sslp.15.45.5a	5 (2)	0.77%	2956s	0.6%	(2s)
sslp.15.45.5b	5 (2)	(38.05s)	38s	0%	(1s)
sslp.15.45.5c	5 (2)	6.94%	3288s	6%	(5s)
sslp.15.45.5d	5 (2)	4.14%	1378s	3.92%	(2s)
sslp.15.45.5e	5 (2)	5.18%	92s	4.4%	(4s)
sslp.15.45.10a	10 (2)	6.24%	606s	5.78%	(23s)
sslp.15.45.10b	10 (2)	6.46%	2336s	5.58%	(10s)

Continued on next page

sslp.15.45.10c	10 (2)	0.51%	3200s	0.51%	(5s)
sslp.15.45.10d	10 (2)	7.30%	3600s	6.36%	(13s)
sslp.15.45.10e	10 (2)	0.43%	1311s	0.39%	(1s)
sslp.15.45.15a	15 (2)	8.93%	2290s	7.91%	0.04%
sslp.15.45.15b	15 (2)	7.58%	11s	6.23%	(34s)
sslp.15.45.15c	15 (2)	9.06%	1326s	7.37%	(41s)
sslp.15.45.15d	15 (2)	12.32%	1362s	10.33%	(279s)
sslp.15.45.15e	15 (2)	4.07%	3215s	3.69%	(6s)
sslp.5.25.50a	50 (1)	(19.27s)	15s	0%	(1s)
sslp.5.25.50b	50 (1)	(14.88s)	15s	0%	(1s)
sslp.5.25.50c	50 (1)	(13.9s)	11s	0%	(1s)
sslp.5.25.50d	50 (1)	(13.59s)	14s	0%	(1s)
sslp.5.25.50e	50 (1)	(13.63s)	13s	0%	(1s)
sslp.5.25.100a	100 (1)	(2255.2s)	2068s	0%	(20s)
sslp.5.25.100b	100 (1)	(198.52s)	195s	0%	(1s)
sslp.5.25.100c	100 (1)	(44.14s)	44s	0%	(1s)
sslp.5.25.100d	100 (1)	(45.07s)	45s	0%	(1s)
sslp.5.25.100e	100 (1)	(43.33s)	41s	0%	(1s)
sslp.10.50.50a	50 (5)	2.36%	1451s	2.11%	(102s)
sslp.10.50.50b	50 (5)	1.67%	1708s	1.53%	(99s)
sslp.10.50.50c	50 (5)	2.31%	891s	2.02%	(765s)
sslp.10.50.50d	50 (5)	2.34%	3369s	2.16%	(15s)
sslp.10.50.50e	50 (5)	2.59%	2493s	2.32%	(251s)
sslp.10.50.100a	100 (10)	2.03%	1234s	1.70%	(233s)
sslp.10.50.100b	100 (10)	1.83%	3266s	1.64%	(161s)
sslp.10.50.100c	100 (10)	2.19%	1206s	1.96%	(248s)
sslp.10.50.100d	100 (10)	2.71%	2531s	2.45%	(52s)
sslp.10.50.100e	100 (10)	2.96%	3490s	2.60%	(267s)
sslp.10.50.500a	500 (50)	2.25%	2330s	0.53%	1.88%
sslp.10.50.500b	500 (50)	2.35%	2550s	2.03%	0.2%
sslp.10.50.500c	500 (50)	2.40%	2801s	0.63%	(M) 1.69%

Continued on next page

sslp.10.50.500d	500 (50)	2.75%	2395s	-3.69%	(M) 6.33%
sslp.10.50.500e	500 (50)	3.26%	2456s	2.57%	0.5%
sslp.10.50.1000a	1000 (100)	2.41%	3565s	-2.21%	6.1%
sslp.10.50.1000b	1000 (100)	2.52%	3451s	-2.05%	5.07%
sslp.10.50.1000c	1000 (100)	2.60%	3611s	2.24%	0.28%
sslp.10.50.1000d	1000 (100)	3.00%	3233s	2.28%	0.43%
sslp.10.50.1000e	1000 (100)	3.34%	3547s	-1.98%	5.32%
sslp.10.50.2000a	2000 (100)	24.12%	2438s	1.31%	18.62%
sslp.10.50.2000b	2000 (100)	24.72%	3610s	6.69%	12.44%
sslp.10.50.2000c	2000 (100)	21.66%	2183s	-0.12%	18.46%
sslp.10.50.2000d	2000 (100)	9.72%	2094s	-5.85%	14.46%
sslp.10.50.2000e	2000 (100)	20.83%	947s	1.64%	14.57%

It is interesting to note that the problem structure has more of an impact on solution time (for both PIPS-SBB and CPLEX) for instances with a small number of scenarios as shown by the variability in solution time among the sslp.15.45.* instances as opposed to the solution times for the sslp.10.50.* instances.

DCAP

The DCAP instances consist of a set of 12 two-stage stochastic integer programs with mixed-integer first-stage variables and pure binary second-stage variables. They model dynamic capacity acquisitions and allocations under uncertainty [147]. As seen in Table 5.3, PIPS-SBB shows a substantially inferior performance in comparison to CPLEX on finding improvements in both the lower bound and the upper bound. Advanced preprocessing and cutting-plane methods enable CPLEX to solve all instances. As we suggest in Section 5.5, the addition of cutting-plane methods in future releases of PIPS-SBB will narrow the current performance gap between PIPS-SBB and CPLEX.

Table 5.3: DCAP instance set results

Problem Instance	Scenarios (Cores)	RelGap (Time)	Best Solution		CPLEX RelGap (Time)
			Time	Quality	
dcap233_200	200 (20)	58.89%	2s	21.50%	(1s)
dcap233_300	300 (20)	68.75%	3s	47.59%	0.01%
dcap233_500	500 (50)	65.15%	8s	32.26%	(2s)
dcap243_200	200 (20)	50.15%	2s	26.46%	(1s)
dcap243_300	300 (20)	49.43%	4s	23.43%	(10s)
dcap243_500	500 (50)	53.96%	9s	31.57%	(12s)
dcap332_200	200 (20)	84.48%	1482s	63.66%	0.01%
dcap332_300	300 (20)	81.79%	248s	35.00%	(26s)
dcap332_500	500 (50)	87.36%	345s	38.85%	(78s)
dcap342_200	200 (20)	68.93%	104s	36.21%	(33s)
dcap342_300	300 (20)	69.31%	585s	30.86%	(88s)
dcap342_500	500 (50)	68.30%	536s	25.59%	(405s)

SMKP

The SMKP instance set is formed by 30 instances of a stochastic multiple knapsack problem. Each problem contains binary variables in both stages and knapsack constraints [148]. As seen in Table 5.4, Compared to DCAP, we see in Table 5.4 that PIPS-SBB performs much better in terms of the RelGap at termination. CPLEX is unable to solve two instances due to memory limitations in the branch-and-bound tree search. For the remaining ones, it is able to obtain smaller RelGap than PIPS-SBB, mainly due to finding better feasible solutions.

5.4.3 Specialized structure-aware algorithms

As explained in Section 5.3.2, PIPS-SBB leverages the dual block-angular problem structure during the branch-and-bound tree search by prioritizing decisions on first-stage variables over second-stage variables. To show the effectiveness of specialized branching rules and heuristics, we consider a structure-oblivious version of PIPS-SBB where decisions in primal heuristics and branching rules are performed regardless of the variable structure, so that all variables (first- and second-stage) have an equal priority of being chosen within

Table 5.4: SMKP instance set results

Problem Instance	Scenarios (Cores)	RelGap (Time)	Best Solution		CPLEX RelGap (Time)
			Time	Quality	
smkp_1	20 (2)	0.50%	2921s	0.35%	0.14%
smkp_2	20 (2)	0.43%	1672s	0.28%	0.13%
smkp_3	20 (2)	0.57%	2080s	0.41%	0.14%
smkp_4	20 (2)	0.51%	3299s	0.34%	0.15%
smkp_5	20 (2)	0.59%	1650s	0.39%	0.12%
smkp_6	20 (2)	0.72%	3318s	0.50%	0.14%
smkp_7	20 (2)	0.70%	952s	0.46%	(M) 0.11%
smkp_8	20 (2)	0.57%	523s	0.36%	0.15%
smkp_9	20 (2)	0.62%	3584s	0.42%	0.17%
smkp_10	20 (2)	0.66%	3538s	0.33%	0.22%
smkp_11	20 (2)	0.55%	694s	0.26%	0.25%
smkp_12	20 (2)	0.68%	122s	0.38%	0.17%
smkp_13	20 (2)	0.64%	254s	0.30%	0.26%
smkp_14	20 (2)	0.72%	1029s	0.09%	(M) 0.17%
smkp_15	20 (2)	0.59%	3080s	0.34%	0.11%
smkp_16	20 (2)	0.62%	259s	0.33%	0.20%
smkp_17	20 (2)	0.62%	857s	0.36%	0.14%
smkp_18	20 (2)	0.61%	340s	0.34%	0.17%
smkp_19	20 (2)	0.77%	111s	0.45%	0.16%
smkp_20	20 (2)	0.66%	126s	0.38%	0.17%
smkp_21	20 (2)	0.84%	3148s	0.50%	0.17%
smkp_22	20 (2)	0.66%	2209s	0.29%	0.26%
smkp_23	20 (2)	0.71%	3177s	0.42%	0.18%
smkp_24	20 (2)	0.71%	1992s	0.44%	0.14%
smkp_25	20 (2)	0.60%	1552s	0.27%	0.16%
smkp_26	20 (2)	0.73%	1441s	0.48%	0.14%
smkp_27	20 (2)	0.69%	439s	0.35%	0.19%
smkp_28	20 (2)	0.67%	507s	0.35%	0.18%
smkp_29	20 (2)	0.89%	1941s	0.55%	0.20%
smkp_30	20 (2)	0.82%	1802s	0.40%	0.31%

the algorithm. We refer to this version of PIPS-SBB as General PIPS-SBB, and compare its performance against the structure-aware version of PIPS-SBB (referred to as Stochastic PIPS-SBB) in Table 5.5. We see that Stochastic PIPS-SBB is able to deliver better performance in every test instance, which shows that these specializations are critical to the success of the primal heuristics and branching rules.

Table 5.5: Comparison specialized stochastic and general structure heuristics

Problem Instance	Scenarios (Cores)	RelGap (Time)	
		Stochastic PIPS-SBB	General PIPS-SBB
sslp_15_45_5	5 (2)	1.36 %	4.23%
sslp_15_45_10	10 (2)	7.93 %	8.39%
sslp_15_45_15	15 (2)	5.25 %	8.26%
sslp_5_25_50	50 (1)	(12.34s)	289.71%
sslp_5_25_100	100 (1)	(41.63s)	65.42%
sslp_10_50_50	50 (5)	1.48%	27.13%
sslp_10_50_100	100 (10)	1.74 %	28.60%
sslp_10_50_500	500 (50)	1.57 %	29.13%
sslp_10_50_1000	1000 (100)	1.60 %	∞
sslp_10_50_2000	2000 (100)	24.00 %	∞

5.5 Conclusions and future directions

In this section, we have presented PIPS-SBB, a new exact distributed-memory parallel branch-and-bound based solver specialized for dual block-angular MIPs, which include all two-stage stochastic mixed-integer programs (SMIPs). We have shown that leveraging the problem structure of SMIPs leads to three natural advantages. The first is data distribution, allowing us to potentially solve much larger instances than before, as demonstrated by PIPS-S and by the PIPS-SBB infrastructure. Second, operating on the rows of each scenario block independently is a natural source of task parallelism for PIPS-SBB. Last but not least, we see in Section 5.4.3 that a branch-and-bound code that distinguishes between first- and second-stage data in its algorithms can result in vastly improved performance.

It is clear from Section 5.4.2 that PIPS-SBB has a long way to go before it is competitive with commercial MIP solvers. Nevertheless, as we continue to work on the algorithms and add more functionality to the PIPS-SBB codebase, we expect the performance to improve significantly. We propose four natural directions of future work.

- **Adding branch-and-bound methods:** It is well known that the success of a branch-and-bound scheme is dependent on the optimized implementation of a variety of schemes for converging MIP bounds, including cuts, presolve, and heuristics. By

specializing heuristics, branching strategies, and a very simple presolve for stochastic MIPs, our experiments show promise, but at the same time indicate how far we still have to go. Next, we will implement a variety of cutting-plane methods and a stronger presolve, followed by other methods to accelerate the branch-and-bound algorithm.

- **Developing specialized stochastic MIP methods:** The effectiveness of our algorithm can be further improved by developing specialized methods for converging the bounds. These potentially include new Benders-like cuts and Lagrangian-like heuristics.
- **Exposing additional parallelism:** Currently, PIPS-SBB can utilize as many cores in parallel as the number of scenarios in the stochastic MIP. However, our experiments in Section 5.4 show that the performance of PIPS-S is best when using fewer cores than the number of scenarios. To expose additional parallelism, especially when the number of available cores is far larger than the number of scenarios, we will extend the PIPS-SBB code to search the branch-and-bound tree in parallel. This extended framework will have two inherent levels of parallelism: parallelizing the MIP tree and parallelizing the LP relaxation for each node of the branch-and-bound tree (already done by PIPS-S). Since many relaxations are being solved simultaneously in this two-level framework (one for each node of the branch-and-bound tree), the available task parallelism will be limited by the amount of computational resources and not by the number of scenarios. Such a framework can potentially utilize a large number of cores, due to the multiplicative effect of the levels of parallelism. For instance, a 100-scenario stochastic MIP solved by using 10 parallel branch-and-bound tree searches and 100 cores for solving each LP relaxation scales to 1000 cores overall.

The current implementation of PIPS-SBB, as described in Section 5.3, can be easily

extended to instantiate multiple distributed `BBSMPSTree` objects that search separate parts of the branch-and-bound tree and are managed by a centralized coordinator. While exposing more parallelism by parallelizing the branch-and-bound search will use more cores and increase performance, it must be noted that parallel efficiency will probably decrease due to the well-known loss of parallel efficiency in branch-and-bound search [25].

- **Improving usability:** Even though releasing PIPS-SBB as open-source code allows users to modify the algorithm as needed, future versions of PIPS-SBB will also provide a callback mechanism to allow users to influence or even override its methods in a more convenient fashion. These would include callbacks for various components of a branch-and-bound tree search, such as node selection, adding cutting planes, and heuristics to find feasible solutions. A command-line interface will enable easy modification and parameterization of the PIPS-SBB solver. We also plan to interface with StructJuMP [149], a parallel extension of the algebraic modeling language JuMP for structured optimization problems, which includes two-stage SMIPs.

CHAPTER 6

PARALLEL PIPS-SBB: MULTI-LEVEL PARALLELISM FOR STOCHASTIC MIXED-INTEGER PROGRAMS

6.1 Introduction

The breakdown of Dennard scaling in CPUs has forced manufacturers to design for increasing parallelism to improve theoretical peak CPU performance, as processor clock speeds cannot be increased using current chip manufacturing processes. Realizing a nontrivial fraction of these theoretical performance improvements requires algorithms that leverage parallelism effectively. For high-performance computing (HPC) systems used in large-scale scientific and engineering applications, this trend has been exacerbated, with recent HPC systems such as Argonne National Laboratory’s Mira enabling million-process parallelism, and future systems enabling even larger process counts. Using such massive-scale parallelism – four to five orders of magnitude greater than current workstations – to its full potential in large-scale Mixed Integer Programming (MIP) applications demands MIP algorithms that scale efficiently (e.g., in a weak or strong sense) as process counts increase [25].

Current parallel algorithms for MIPs do not meet this criterion. The scalability of existing approaches, discussed later in this introduction, can vary dramatically depending on the instance being solved, exposing a significant gap between current HPC hardware and MIP algorithmic capabilities. To begin to address this algorithmic capability gap, this work investigates the parallel scaling benefits of exploiting the problem structure found in dual block-angular MIPs and exposing multiple levels of parallelism. Leveraging the parallelism in each level yields multiplicative effects on speedup: if each of two levels scales efficiently to hundreds of cores, the overall algorithm has the potential to scale efficiently to tens of thousands of cores. Our intention is to exploit this principle for dual block-angular MIPs

using PIPS-SBB [29].

6.1.1 Parallel Branch-and-Bound

We focus on the parallelization of the branch-and-bound algorithm as introduced in Section 1.2. Multiple parallel branch-and-bound tree search algorithms have been devised in order to speed up the optimization algorithm, though most of them differ significantly on how they might go about it. For a thorough taxonomy of most branch-and-bound parallelizations, we urge the reader to refer to [26]. In theory, parallel branch-and-bound tree search is not difficult because the processing of subproblems is independent. However, there are many challenges associated with designing a parallel branch-and-bound algorithm, such as maintaining a healthy work balance among all the processors and avoiding significant communication and synchronization costs. In other words, distributed-memory parallelization is not straightforward in practice, in contrast to the relatively straightforward shared-memory parallel algorithms present in most state-of-the-art MIP solvers. In this paper, we classify parallel branch-and-bound algorithms based on the smallest transferable unit of work between the various cores of the parallel search. On one hand, solvers using *fine-grained* parallelizations use a single MIP tree node as the basic unit of work. In contrast, each solver in *coarse-grained* parallelizations focuses on solving an entire MIP subtree at a time. Generally speaking, coarse-grained parallelizations require less communication. On the other hand, it may be challenging for such implementations to supply useful work to all processors.

To keep this discussion self-contained, we mention three notable parallel branch-and-bound tree search frameworks developed recently: PEBBL [150, 151], CHiPPS [152, 153, 154, 155, 156, 157], and UG [78, 158, 159, 160].

PEBBL parallelizes branch-and-bound at a fine-grained, node level within an MPI-based, shared-nothing parallel programming model using a decentralized hub-worker approach. In this approach, each hub process owns a collection of worker processes that solve

subproblems (nodes). Workers and hubs communicate problem metadata to coordinate subproblem transfers between worker processes. “Load factors” are used to estimate the work required to process the branch-and-bound subtrees rooted at the subproblems owned by workers in each hub. “Rendezvous” load balancing is then used to decide which hubs donate or receive subproblems to or from other hubs. A hub may also transfer subproblems between worker processes it owns through dynamic load-balancing algorithms specific to PEBBL.

CHiPPS parallelizes branch-and-bound using a master-hub-worker approach built upon the ALPS [153] parallel tree search framework. This approach uses coarse-grained, subtree-level parallelism in which workers explore branch-and-bound subtrees, and these subtrees are transferred among workers during load-balancing. As with PEBBL, CHiPPS uses hub processes to coordinate worker processes, but instead of employing peer-to-peer coordination among hubs, CHiPPS load-balances hierarchically, using a master process to coordinate and balance work among hub processes, each of which themselves coordinates and balances work among the worker processes it owns.

The Ubiquity Generator (UG) framework [160] was designed to parallelize existing state-of-the-art sequential branch-and-bound algorithms for MIPs, referred to as *base solvers*, using a supervisor-worker approach. In this approach, UG wraps around both the API of the base solver (e.g., CPLEX) and the parallel programming model (e.g., MPI, POSIX threads) to enable passing branch-and-bound subtrees among worker processes, each of which processes a branch-and-bound subtree using a base solver instance. A supervisor process called a “LoadCoordinator” is used to serialize branch-and-bound subtrees (e.g., using variable bound changes in the root subproblem of a subtree) and to redistribute them among worker processes during load balancing.

6.1.2 Contributions and outline

The premise of this section is to expose multiple nested levels of parallelism in MIP algorithms. We present two new frameworks for parallelizing the branch-and-bound tree, and demonstrate them on SMIPs using PIPS-SBB [29]. By parallelizing the branch-and-bound tree search in PIPS-SBB, we incorporate two levels of parallelism: parallelism (1) in the LP solver, and (2) in the branch-and-bound tree search. This additional level of parallelism to PIPS-SBB with parallel node exploration enables us to increase its scalability.

We briefly describe both frameworks, which differ in the approach used in the management and distribution of work among the parallel workers. The first framework, called PIPS-PSBB, is a new, fine-grained framework that attempts to search aggressively the promising parts of the branch-and-bound tree, with the intention of decreasing the amount of *redundant work* performed by the solver. For minimization problems, a *redundant node* can be characterized as a subproblem with an LP relaxation value greater than the optimal value. To limit communication overhead, this framework transfers node metadata (e.g., upper/lower bounds, tree sizes) asynchronously to hide latency until a load imbalance is detected, at which point MIP subproblems are redistributed among processes using synchronous communication. MPI collective operations are used instead of point-to-point operations to further limit communications overhead. The proposed architecture is also decentralized: node exchanges do not pass through a single coordinating process, which avoids a potential communication bottleneck. The second framework is based on UG. We compare the strengths and weaknesses of both approaches, and show that when implemented as extensions of PIPS-SBB, both approaches leverage multi-level parallelism to scale efficiently beyond the natural limitations of each framework in isolation.

The main contributions of this paper are therefore:

- A novel framework for fine-grained parallel branch-and-bound tree search for solving mixed-integer programs.

- Two new multi-level distributed-memory parallelisms for solving SMIPs, implemented as extensions of the distributed-memory SMIP solver PIPS-SBB.
 - PIPS-PSBB: branch-and-bound parallelism implemented using the new fine-grained parallelism presented in this paper.
 - branch-and-bound parallelism implemented using UG, the coarse-grained parallel branch-and-bound framework available as part of the SCIP optimization suite [161].
- Detailed computational experiments illustrating the scaling performance of both parallel branch-and-bound frameworks.

In general, leveraging the parallelism in each level simultaneously yields multiplicative effects on speedup and parallel scaling efficiency (in a weak or strong sense). In the specific case of branch-and-bound parallelizations for practical MIP instances, it is non-trivial to achieve even 20% strong scaling efficiency beyond 100-1000 processors (modest by HPC standards). By providing an extra level of parallelism, available processing power can be partitioned across multiple levels to improve overall efficiency and speedups over allocating the same number of processes to either single level.

The section is organized as follows. In subsection 6.2, we present PIPS-PSBB: our decentralized, lightweight parallel framework for fine-grained branch-and-bound tree search. Additionally, we delve into the details of our implementation, discussing some of the challenges, and our approach to minimizing the communication overhead associated with the distributed-memory algorithm. In subsection 6.3, we discuss the same topics for ug[PIPS-SBB,MPI]. In subsection 6.4, we begin by first comparing the performance of both of our implementations in detail on a few select instances, and then present performance results (comparing with distributed-memory CPLEX) over the complete test set. Finally, we conclude in subsection 6.5 with some directions for future research.

6.2 PIPS-PSBB: A decentralized lightweight parallel framework for Branch-and-Bound

In this section, we present PIPS-PSBB, a decentralized fine-grained, yet lightweight parallel framework built as an extension to PIPS-SBB. Our approach and ideas are general, and can be applied to any base MIP solver. We first present the main ideas and the implementation details, which are independent of PIPS-SBB. Then, in Section 6.2.3, we address the challenges specific to extending and adhering to the design principles of PIPS-SBB.

A primary aspect to consider in the design of parallel branch-and-bound algorithms is the policy used in the distribution of work, and the degree and frequency of communication used to coordinate optimization workers. On one hand, coarse-grained parallelizations typically require less communication, and allow each worker to focus on processing an entire subtree at a time. However, less frequent coordination between workers may have a detrimental effect on parallel efficiency. Workers may be likely to solve subtrees that would be otherwise fathomed in a sequential execution, especially in the context of large-scale computations with thousands of available solvers. Performing redundant work may seem unavoidable because the information required for fathoming the nodes is only discovered during the optimization and is therefore impossible to know beforehand. Figure 6.1 highlights this issue. The discovery of a feasible solution (black-colored node) by processor 2 could have resulted in the fathoming of many nodes belonging to other workers, if only this information was known beforehand. In the context of a coarse-grained distribution of work, nodes from the same part of the branch-and-bound tree typically belong to the same processor, unlike in fine-grained distributions.

When point-to-point communications are used, a coarse-grained distribution of work may seem the only viable option for avoiding large communication overheads without resorting to complex communication schemes, as in PEBBL. Nevertheless, it is possible to develop a practical, effective, and scalable coarse-grained parallelization framework; as has

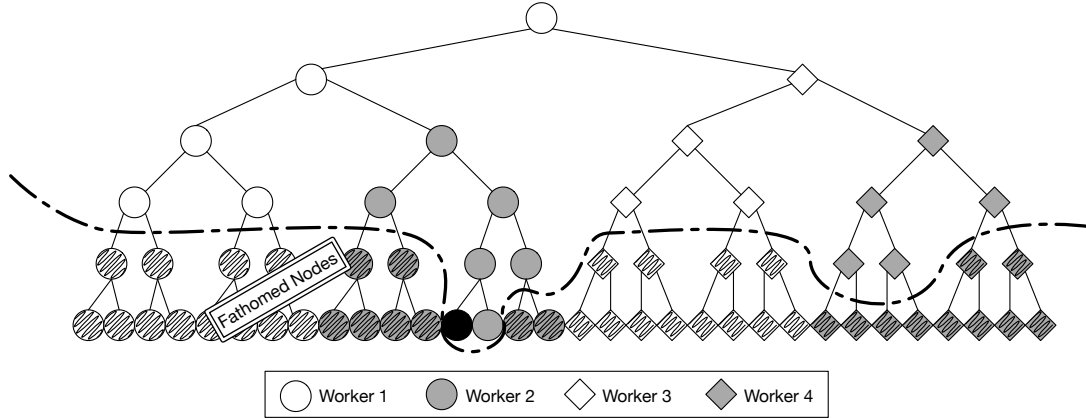


Figure 6.1: Example depicting the coarse-grained distribution of subtrees among the parallel processors at a given point in the parallel exploration. Knowledge of feasible solution at black-colored node could have fathomed many other nodes on many processors.

been demonstrated by UG. Some strategies for reducing overheads include restricting the number of workers simultaneously engaging in communication. We explore the integration of PIPS-SBB with UG in Section 6.3.

On the other side of the spectrum, fine-grained parallelizations have a MIP tree node as their smallest unit of work, which can be transferred among workers. This allows the most promising parts of the tree to be partitioned and processed in parallel, thus ensuring an effective use of the computing resources. Fine-grained control of the work performed is an adaptive approach where the amount of redundant work can be minimized effectively. At the same time, a large degree of communication is also required, which may cause an excessive overhead. By establishing complex protocols so that communication between processors is structured hierarchically, it is possible to develop scalable fine-grained parallelizations, as demonstrated by PEBBL. In PIPS-PSBB, we take a slightly different approach, overcoming many of these communication challenges using a simpler, light-weight framework using MPI collectives. We note that MPI collectives can also be implemented efficiently using hierarchical tree-based algorithms [85], enabling us to reap some of the benefits of these schemes with less implementation complexity than PEBBL. As in PEBBL, PIPS-PSBB is a decentralized framework based on asynchronous MPI communications.

6.2.1 The philosophy behind fine-grained node rebalancing

branch-and-bound can be regarded as a graph algorithm, the objective of which is to traverse the different subproblems until the optimal solution is found and proven. Borrowing from graph algorithms, we define the frontier as the collection of open subproblems at a given time. We also define the active nodes as the set of subproblems currently being explored in parallel. To be able to measure the effectiveness of any parallel branch-and-bound framework, we use the notion of redundant work (as described in [25, 26]) to refer to the collection of subproblems that would be fathomed if the optimal solution was known.

One way to reduce the amount of redundant work is to have extremely powerful heuristics. By finding good feasible solutions very early in the search, branch-and-bound fathoms as many nodes as possible, thus increasing parallel efficiency. In addition, one can try to ensure that all optimization workers focus on the most promising nodes. In the context of parallel branch-and-bound tree search, load balancing is done to ensure that all processors have access to a fraction of the *most promising* nodes, rather than an *equal number* of nodes. The quality of a node is determined by the criterion used to order it within the priority work queue. Typically the lower bound of its parent is used, or some form of node estimation as the ones described in [19, 18].

With the objective of minimizing load imbalance in PIPS-PSBB, we define the smallest transferrable unit of work to be one branch-and-bound node. As the optimization progresses and the work load is continually rebalanced, every processor gradually collects a set of nodes from different subtrees in its work queue. This feature enables great flexibility in the parallel search strategy. An example depicting the fine-grained distribution of work is shown in Figure 6.2. For instance, nodes 8-13 belong to the same subtree, but are distributed among all the available optimization workers.

The objective of our new parallel branch-and-bound implementation is to increase parallel efficiency by minimizing the number of redundant nodes explored. We actively target this goal by ensuring the set of active nodes being explored are always the most promising

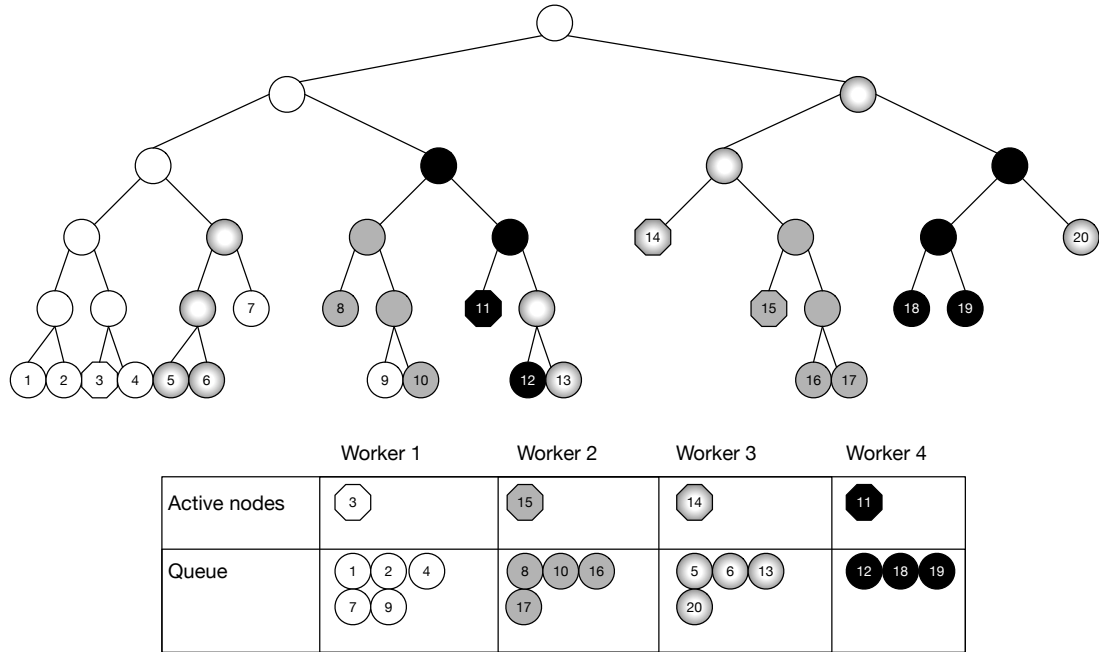


Figure 6.2: PIPS-PSBB: Example depicting the fine-grained distribution of subtrees among the parallel processes at a given point in the parallel exploration. The frontier and the active nodes in the frontier are also depicted.

ones. We present an overview of our fine-grained implementation in Algorithm 17. One of its key components is the work redistribution mechanism, described between lines 9 to 15. The approach is described in greater detail in Section 6.2.2.

PIPS-PSBB uses a lightweight mechanism for redistributing the most promising nodes among all the optimization workers without the need for a centralized load coordinator. Rather, this alternative scheme seeks to reduce the communication bottlenecks that would be caused by the existence of one. Instead of point-to-point communications, parallel processors exchange subproblems via all-to-all collective MPI asynchronous communications, enabling the framework to rebalance the computational load using a single communication step. Parallel processes proceed to solve subproblems until the problem has been solved to optimality.

Algorithm 17 Fine-grained parallel Branch-and-Bound algorithm of PIPS-PSBB

```
1: for All processors  $t \in [N]$  in Parallel do
2:    $UB^t = \infty$ 
3:    $LB^t = -\infty$ 
4:   priority queue  $Q^t = \emptyset$ 
5:   if  $t = 1$  then
6:     Add root  $r$  of original MIP problem to  $Q^1$ 
7:   end if
8:   while termination conditions are not met do
9:     mustCommunicate = TESTCONDITIONSFORCOMMUNICATION( $t, Q^t, UB^t$ )
10:    if mustCommunicate then
11:      Determine the top  $K \cdot N$  candidate subproblems from  $Q^t, t \in [N]$  and redistribute them among all processors in a
round robin fashion.
12:      if termination conditions are met then
13:        return
14:      end if
15:    end if
16:    remove subproblem  $p$  from top of  $Q^t$ 
17:    Process  $p$ 
18:     $UB_p =$  best solution found within  $p$ 
19:     $LB_p =$  lower bound of  $p$ ,  $\infty$  if infeasible,  $-\infty$  if unbounded
20:     $UB^t = \min\{UB^t, UB_p\}$ 
21:    if  $LB_p < UB^t$  then
22:      Partition  $p$  into a set  $\{s_0, \dots, s_k\}$  of subproblems, each with a lower bound defined to be  $LB_p$ 
23:      Add  $\{s_0, \dots, s_k\}$  to  $Q^t$ 
24:    end if
25:     $\triangleright$  If  $LB_p > UB^t$ , fathom problem  $p$  by bound dominance
26:     $LB^t =$  minimum lower bound among all open subproblems in  $Q^t$ 
27:  end while
28: end for
29: if  $LB < UB$  then
30:   return infeasible
31: else
32:   return optimal solution
33: end if
```

6.2.2 Decentralized node exchange

Load rebalancing among all workers is maintained via a sequence of MPI collective communications. It enables workers to rank the most promising subproblems and to redistribute them in a round robin fashion to improve load balance. A flow chart of the rebalancing process is provided in Figure 6.3.

The first step in the process consists of identifying the location of the most promising nodes. Assuming the objective is to redistribute K of the most promising nodes for every one of the N available optimization workers, the total nodes to exchange will be $K \cdot N$. In the example presented in Figure 6.3, we have $K = N = 3$, yielding 9 nodes to exchange. To achieve this exchange, every worker first collects the lower bounds and estimates of its best $K \cdot N$ ($= 9$ in this example) open subproblems, since it is possible that a single

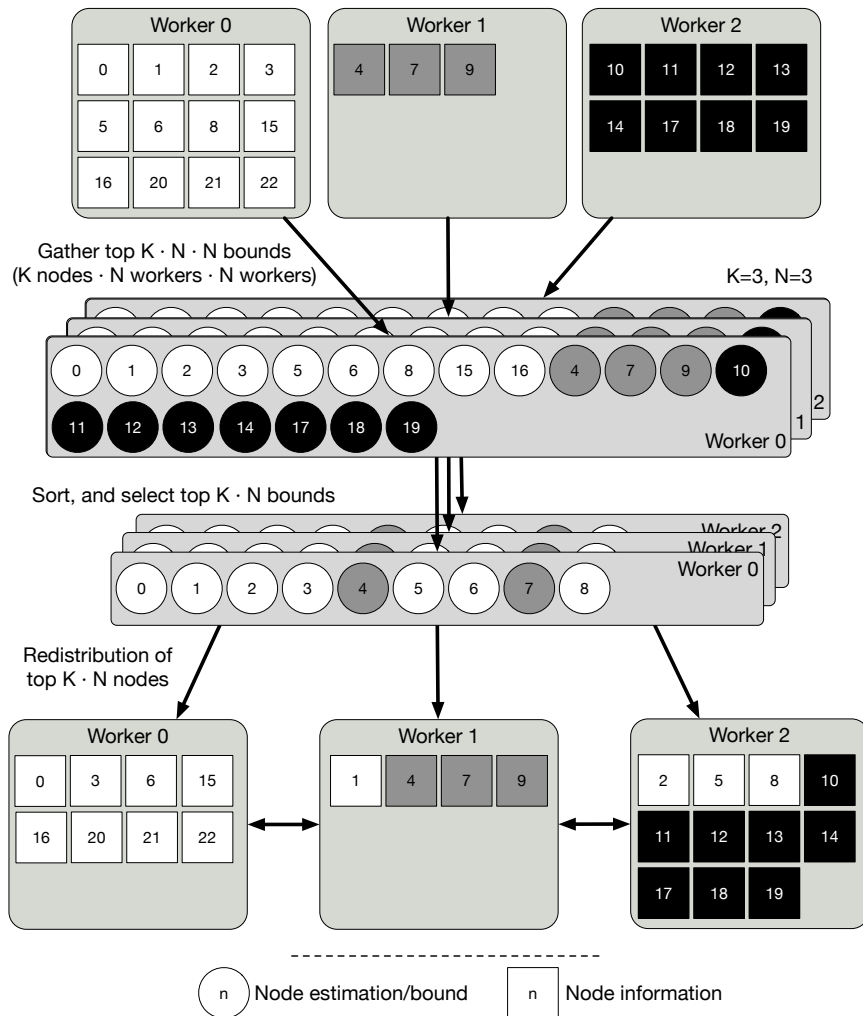


Figure 6.3: Steps of the node exchange: Node estimates (in circles) are gathered using all-to-all communication and sorted at each worker. When each worker has determined (in parallel) which nodes need to be exchanged, actual node information (in squares) is redistributed using point-to-point communication.

worker owns all the nodes to exchange. It is also possible that some worker does not have $K \cdot N$ nodes (as is the case for orkers 1 and 2), the algorithm proceeds with the maximum available. The node bounds/estimates are exchanged through a decentralized all-to-all MPI Allgather communication, collecting a total of $K \cdot N \cdot N$ lower bounds and estimates. Once the $K \cdot N$ most promising nodes have been identified in the next step by each worker (by sorting), and their origin has been determined, the actual node information is prepared to be redistributed in round-robin fashion. In this example, worker 1 is identified to receive nodes 1,4,7. Observe that it may already own some of these nodes, and therefore these nodes do

not need to be exchanged. Once it is determined which nodes need to be redistributed, every worker proceeds to serialize the actual node information prior to their exchange. In this example, none of the nodes from worker 2 were deemed promising nodes, but it received three new promising subproblems.

Communication must be used strategically in order to avoid overheads. While node transfers are carried out synchronously, exchanges of worker statuses such as upper/lower bounds, tree sizes, times, and solutions are performed asynchronously. Nodes are transferred synchronously only after all workers signal that communication is needed. When communicating over a large number of processes, MPI collective communication primitives have been shown to significantly outperform their point-to-point equivalents[85], provided a tuned MPI implementation is used.

We provide details regarding the asynchronous detection of load imbalance in Algorithm 18. The algorithm establishes a threshold λ that determines the number of branch-and-bound iterations between asynchronous communication calls. This parameter is adjusted throughout the parallel branch-and-bound tree search to adapt load rebalancing as needed. The parameter is modified based on the difference between the minimum and maximum optimality gap (gap between lower and upper bounds) among the workers. A difference in gap greater than a provided threshold δ indicates load imbalance, and therefore the parameter λ is decreased in order to rebalance more aggressively in the future. During ramp-up and ramp-down, frequent rebalancing is expected. Work queues are also rebalanced whenever any worker has no active nodes in its priority queue. Otherwise, if the number of iterations since the last communication is greater than λ , asynchronous all-to-all communication is performed.

6.2.3 Processor distribution

Every software component and algorithm in PIPS-PSBB is designed to comply with the distributed data representation imposed by PIPS-SBB. Thus, every MPI processor is respon-

Algorithm 18 Asynchronous communication mechanism in PIPS-PSBB

```
1: procedure TESTCONDITIONSFORCOMMUNICATION( $t, Q^t, UB^t$ )
2:   if asynchronous communication pending then
3:     Test communication flags
4:     if communication complete then
5:       Update best solutions,  $UB^t = \min_{1 \leq i \leq N} \{UB^i\}$ 
6:       Update global lower bound, and check termination conditions
7:        $Gap_{min}$  = smallest optimality gap among solvers
8:        $Gap_{max}$  = largest optimality gap among solvers
9:       if  $|Gap_{max} - Gap_{min}| \geq \delta$  then
10:        Decrease number of iterations threshold  $\lambda$  before next communication
11:        Return true
12:       else
13:        increase number of iterations threshold  $\lambda$  before next communication
14:       end if
15:     end if
16:     Return false
17:   end if
18:   if number of iterations since last communication is greater than  $\lambda$  or  $Q^t = \emptyset$  then
19:     initiate asynchronous exchange of current best lower bound, upper bound, tree size, solution time, and best solution
20:   end if
21:   Return false
22: end procedure
```

sible for performing all operations on the data it owns. With the addition of branch-and-bound parallelism to the framework, processes are arranged in a matrix of MPI communicators, as depicted in Figure 6.4. MPI processes belonging to a given PIPS-S solver communicator operate as a single worker, conform to the data distribution, and exchange information in order to be able to solve LP relaxations in parallel. In turn, all MPI processes belonging to a branch-and-bound communicator own the same data of the problem, and communicate to exchange nodes during the rebalancing process. The process/communicator arrangement is such that with a total of $M \cdot N$ processes, each collective communication only involves either M or N processes.

6.3 Parallelizing PIPS-SBB with UG

The main concept of UG is to exploit the performance of a powerful state-of-the-art “*base solver*” by coordinating multiple instances in parallel. The UG framework is depicted in Figure 6.5. Using the established notation, $ug[\text{PIPS-SBB}, \text{MPI}]$ is the product of parallelizing the base solver PIPS-SBB using MPI under the framework of UG. UG carefully abstracts all functions related to managing parallel branch-and-bound search from the ac-

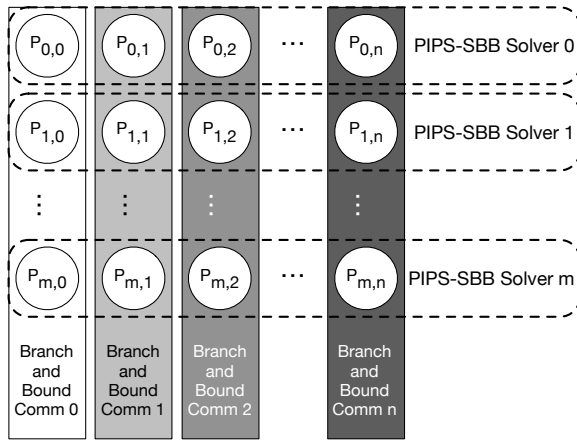


Figure 6.4: Processor distribution used in PIPS-PSBB: Each row corresponds to the communicator of a single PIPS-SBB solver. Each column corresponds to the communicator used by processes from different PIPS-SBB solvers that own the same data.

tual processing of the branch-and-bound tree itself. It supports many common features needed in parallel branch-and-bound, such as multiple communication protocols, ramp-up, dynamic load balancing, check-pointing, and restarting mechanisms.

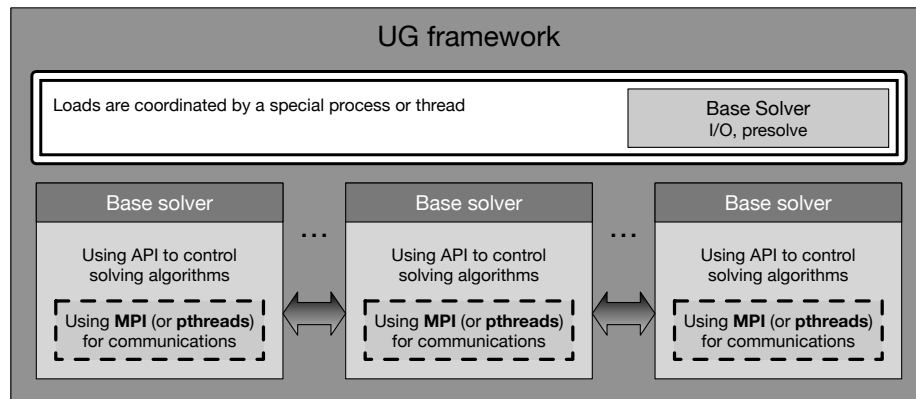


Figure 6.5: Design of UG and ug[PIPS-SBB, MPI]

This framework has been instrumental in solving to optimality several open instances of the MIPLIB2003 and MIPLIB2010 libraries [158]. Its coordination paradigm has also been used in the distributed-memory implementation of the CPLEX solver. The design of UG is such that its integration with PIPS-SBB is straightforward, only requiring the implementation of the interface between both software components.

Algorithm 19 UG LoadCoordinator (UG Solvers 1 to N with the PIPS-SBB are spawned)

```
1: collectMode  $\leftarrow$  False
2:  $x^* \leftarrow$  NULL
3:  $I \leftarrow N \setminus \{1\}$ 
4:  $A \leftarrow \{1\}$ 
5:  $Q \leftarrow \emptyset$ 
6:  $R \leftarrow \{(1, 0)\}$   $\triangleright$  Subproblems currently being processed, 0 is the index of the root problem
7: Send the root problem to UG workers 1 (see Algorithm 20)
8: while  $Q \neq \emptyset$  and  $R \neq \emptyset$  do
9:    $(i, \text{tag}) \leftarrow$  Wait for message  $\triangleright$  Returns UG workers identifier and message tag
10:  if tag = solutionFound then
11:    Receive solution  $\hat{x}$  from UG worker  $i$ 
12:    if  $x^* = \text{NULL}$  or  $c^\top \hat{x} < c^\top x^*$  then
13:       $x^* \leftarrow \hat{x}$ 
14:    end if
15:  else if tag = subproblem then
16:    Receive a subproblem indexed by  $k$  from UG worker  $i$ 
17:     $Q \leftarrow Q \cup \{k\}$ 
18:  else if tag = terminated then
19:     $R \leftarrow R \setminus \{(i, j)\}$   $\triangleright j$  is the index of the terminated subproblem
20:     $A \leftarrow A \setminus \{i\}, I \leftarrow I \cup \{i\}$ 
21:  else if tag = status then
22:    if collectMode = True then
23:      if there are enough heavy subproblems in  $Q$  then
24:         $\triangleright$  heavy subproblem is a subproblem which is expected to generate a large subtree
25:        Send message with tag = stopCollecting to UG workers in collecting mode. (see Algorithm 20)
26:        collectMode  $\leftarrow$  False
27:      end if
28:    else
29:       $\triangleright$  collectMode = False
30:      if there are not enough heavy subproblems in  $Q$  then
31:        Select UG workers which have heavy subproblems
32:        Send message with tag = startCollecting to the selected UG workers (see Algorithm 20)
33:        collectMode  $\leftarrow$  True
34:      end if
35:    end if
36:  end if
37:  while  $I \neq \emptyset$  do
38:     $i \in I, I \leftarrow I \setminus \{i\}, A \leftarrow A \cup \{i\}$ 
39:    subproblem  $j \in Q, Q \leftarrow Q \setminus \{j\}, R \leftarrow R \cup \{(i, j)\}$ 
40:    Send subproblem  $j$  and  $x^*$  to UG worker  $i$  (see Algorithm 20)
41:  end while
42: end while
43:  $\forall i \in S$ : Send message with tag = termination to UG worker  $i$  (see Algorithm 20)
44: Output  $x^*$ 
```

Algorithm 20 UG solver i with the PIPS-SBB ($i = 1, \dots, N$)

```
collectMode  $\leftarrow$  False
terminate  $\leftarrow$  False
while terminate = False do
   $(i, \text{tag}) \leftarrow$  Wait for message from LoadCoordinator (from Algorithm 19)  $\triangleright$  Returns LoadCoordinator identifier 0 and message tag
  if tag = subproblem then
    Receive subproblem and solution from LoadCoordinator (from Algorithm 19)
    Solve the subproblem using PIPS-SBB, periodically communicating with LoadCoordinator (from Algorithm 19) as follows
    - Send message with tag solutionFound any time a new solution is discovered.
    - Periodically send message with tag status to report current lower bound for this subproblem.
    - When messages with tag startCollecting or stopCollecting are received, toggle collectMode.
    - When collectMode = True,
      periodically send message with tag subproblem containing best candidate subproblem.
    Send a message with tag = terminated
  else if tag = termination then
    terminate  $\leftarrow$  True
  end if
end while
```

6.3.1 Processor distribution within UG

The integration requires an additional MPI processor to act as coordinator in addition to some design compromises. The supervisor-worker paradigm of UG designates a single processor within The ug[PIPS-SBB,MPI] design requires an additional MPI processor to act as coordinator and some additional design compromises. The supervisor-worker paradigm of UG designates a single processor within each PIPS-S communicator to exchange solver information and subproblems. Therefore, only a single communicator is needed in order to enable parallelism at the branch-and-bound level, as depicted in Figure 6.6. The main downside to this approach is the need to break the data distribution policy imposed by PIPS-SBB when communication takes place. This design limitation limits the scalability of the parallel implementation to large data sizes.

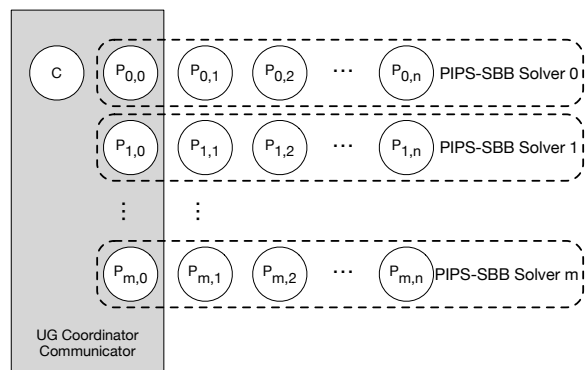


Figure 6.6: Processor distribution used in ug[PIPS-SBB,MPI]

6.3.2 Supervised workload coordination mechanism

UG follows a supervisor-worker paradigm, in which a supervisor, the LoadCoordinator, monitors and coordinates the workload among multiple optimization workers. The LoadCoordinator coordinates the workload and does not store data associated with the search tree. Load balancing is accomplished mainly by toggling the collection mode flag in the UG solvers. Turning on collecting mode sends additional “high quality” subproblems to other UG solvers via the LoadCoordinator. Algorithms 19 and 20 show a simplified co-

ordination mechanism used in UG. The LoadCoordinator can tune the frequency of the status updates produced by the optimization workers to avoid communication bottlenecks in large-scale parallel optimizations. In addition, the number of workers simultaneously participating in the collection mode can be restricted and selected dynamically.

Naturally, there are tradeoffs among the frequency of communication, the number of UG workers participating in collection mode, and the degree to which the parallel search order replicates the sequential one. As the number of processes increases, these tradeoffs must be navigated carefully.

An exchanged subproblem that contains additional bound changes of variables and warmstart information different from the original problem must be solved from scratch on the receiver side. This is a price to pay for externalization. However, there is a potential benefit from this practice, as better performance can be obtained by applying additional pre-solving, cutting planes and heuristics in the subproblem. In the case of `ug[PIPS-SBB,MPI]`, the impact of externalization is minimal because the information exchange level is the same as the one produced in `PIPS-PSBB`.

6.4 Experimental results

We test the performance of our parallel implementations on the Stochastic Server Location Problem (SSLP) [145] instances from the SIPLIB library [121]. The SSLP instances model server location problems with pure binary variables in the first-stage and mixed-binary variables in the second-stage. The problems are encoded as `sslp_m_n_s`, where m is the number of potential server locations, n is the number of potential clients, and s is the number of scenarios. All computations were performed on the `cab` Cluster at Lawrence Livermore National Laboratory, which consists of 1,296 computing nodes. Each computing node features two Intel Xeon E5-2670 8-core processors and 32 GB of RAM. In all experiments, bindings of MPI processes were configured to prevent over-subscription. We evaluate the performance of our frameworks from the perspective of parallel scaling, as

well as the overall performance when compared against CPLEX 12.6.2 on a distributed-memory parallel environment.

We evaluate the scaling performance of our methods using four metrics: time to optimality, tree size, communication overhead, and node inefficiency. We define communication overhead as the fraction of the total time spent by a process performing communication, being idle due to parallel synchronization, or waiting to receive work from the central coordinator. In other words, it is the fraction of time a process spends not performing computation. Recall that for minimization problems, a redundant node can be characterized as a subproblem with an LP relaxation value greater than the optimal solution. If the optimal solution is known at the beginning of the branch-and-bound tree search, all redundant nodes would be fathomed immediately, and therefore never processed. Hence, the number of redundant nodes processed is a measure of the extra work performed by the solver in order to prove optimality. We define node inefficiency as the fraction of the total number of nodes processed that are redundant nodes. In a serial branch-and-bound implementation, node inefficiency is purely a measure of how quickly the optimal solution is found. In parallel branch-and-bound implementations, there will be additional redundant work because some processes may not be working on the most promising nodes. Therefore, node inefficiency is a good surrogate for how well the parallel branch-and-bound framework ensures that the processors are doing useful work.

6.4.1 Scaling performance: Branch-and-Bound parallelization

We first present results that demonstrate the scaling performance of both parallel implementations. For these results, we use *sslp_15_45_5*, a problem instance with 5 scenarios, 3390 binary variables, and 301 constraints. Because of the relatively small number of scenarios, the LP solver PIPS-S is unable to scale beyond a handful of cores. Therefore, The focus of these experiments is to understand the benefits from branch-and-bound parallelization, the trade-offs involved, and the impact of certain algorithmic parameters.

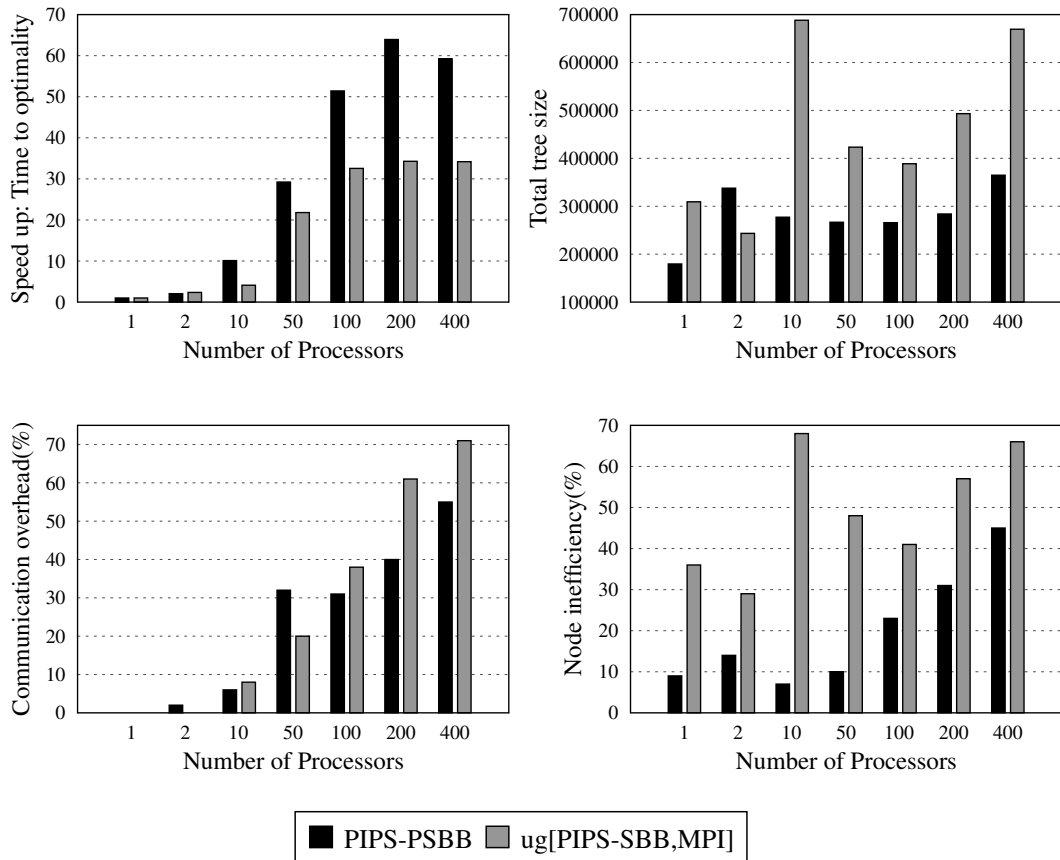


Figure 6.7: Scaling performance of PIPS-PSBB and ug[PIPS-SBB,MPI] when solving *sslp_15_45_5*: Speedup (time to optimality), branch and bound nodes processed, parallel communication overhead, and proportion of redundant nodes

Figure 6.7 shows the scaling performance of PIPS-PSBB (black) and ug[PIPS-SBB,MPI] (gray). For both frameworks, we use as many solvers as the number of available MPI processes. We see that PIPS-PSBB is able to scale up to 200 processes with a speedup of 66x with respect to the baseline serial execution (2920s). This represents a parallel efficiency of 33%. Further progress (increasing the number of processes to 400) is hampered by the overhead caused by the communication required to synchronize the solvers. The total number of nodes explored remains fairly constant, under 400000 nodes. The proportion of redundant nodes is below 15% for configurations under 50 processes, but increases as the number of solvers is increased further. Later, in Section 6.4.2, we will see that decreasing the number of solvers (by giving more processes to each PIPS-S solver) can reduce signif-

icantly the communication overhead as well as the node inefficiency. We next look at the scaling performance of `ug[PIPS-SBB,MPI]`, presented in gray in Figure 6.7. This parallel solver is able to scale up to 200 processes, with a speedup of 33x with respect to the baseline serial execution (4491s). This represents a parallel efficiency of 16.5%. Compared to PIPS-PSBB, `ug[PIPS-SBB,MPI]` shows slightly worse performance. This performance decrease is caused by a larger communication overhead and a larger node inefficiency.

Figure 6.8 provides further information regarding the origin of the overhead for both frameworks. For `ug[PIPS-SBB,MPI]`, the overhead coming from the ramp down gains importance as the number of processes increases. This overhead is due to the centralized nature of the load coordinator, which fails to provide all available solvers with open subproblems to process. As a result, most processes remain idle during ramp down. On the other hand, PIPS-PSBB has much lower overhead in general. However, its overhead in the primal phase increases as the number of processors increases, whereas `ug[PIPS-SBB,MPI]` has relatively stable overhead in this phase. These experiments suggest that the two parallelization frameworks have different strengths, and may perform differently depending on problem instance.

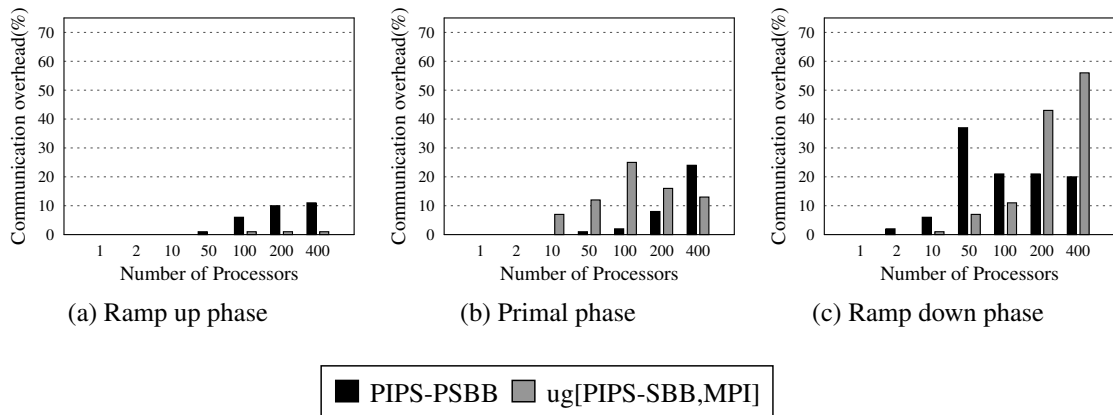


Figure 6.8: Communication overhead for PIPS-PSBB and `ug[PIPS-SBB,MPI]` when solving `sslp_15_45_5` broken down by stage in the optimization process

Finally, we analyze the behavior of PIPS-PSBB in more detail. As described in Section 6.2.2, the control of communication between solvers in PIPS-PSBB is dictated pri-

marily by the communication frequency parameter λ . In the experiments presented in Figure 6.7, the parameter λ is set to fluctuate within a minimum of 50 iterations and a maximum of 1000. In Figure 6.9, we study the effects on performance when the communication frequency is altered. When solvers are forced to communicate more frequently, PIPS-PSBB suffers from an increased communication overhead and a corresponding decrease in performance, especially when the number of processes is increased to 400. The positive side-effect from more frequent communication is a comparatively smaller tree size. When synchronization is less frequent, we see the opposite effect: a decrease in overhead but a significantly large number of nodes processed.

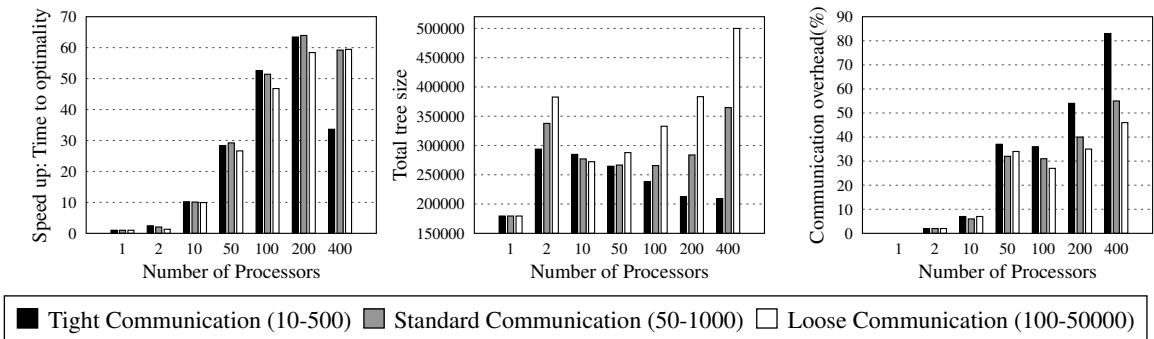


Figure 6.9: Performance of PIPS-PSBB for different communication frequencies λ . $(x - y)$ denotes the minimum (x) and the maximum (y) number of solver iterations before communication is attempted.

6.4.2 Scaling performance: effects of multi-level parallelism

Parallel scaling results on small problems such as *sslp_15_45_5* provide an interesting picture for understanding the behavior and the interactions between the different solvers in the branch-and-bound parallelization frameworks. From a practical standpoint, it is more valuable to test the effects of parallelism when optimizing larger problems, especially those with a larger number of scenarios, because the UG framework was designed with large-scale parallel optimization in mind [158]. The following set of results describe the performance of PIPS-PSBB and ug[PIPS-SBB,MPI] when optimizing *sslp_10_50_500*, a problem with 500 scenarios, 250010 variables, and 504510 constraints. We particularly take a look at the

combined effect of the two levels of parallelism by splitting a budget of 500 parallel cores and testing the solvers' behavior under different configurations.

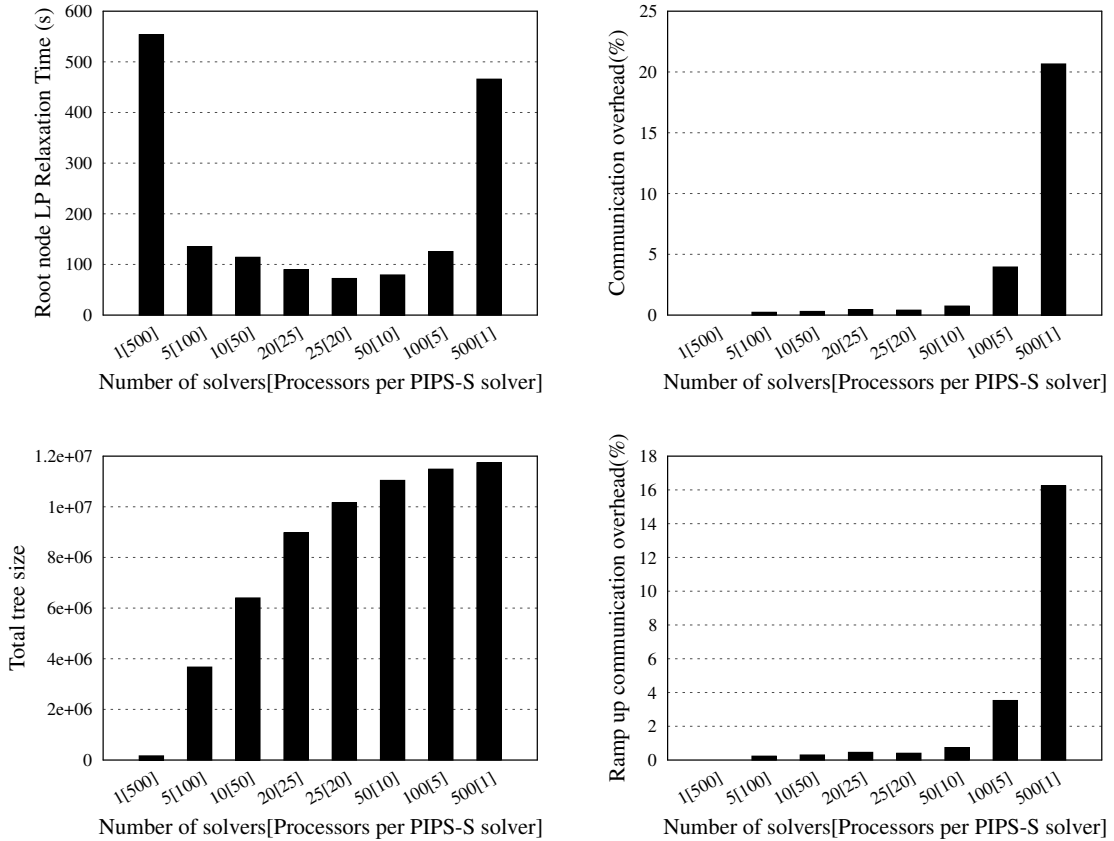


Figure 6.10: Scaling performance of PIPS-PSBB when solving *sslp_10_50_500*: Time to solve the LP relaxation, communication overhead, number of nodes processed, and communication overhead at ramp-up.

In Figure 6.10, we first analyze the performance of the LP solver in solving the root node of the branch-and-bound tree. When a single core per solver is used (last column), the performance of the LP solver is significantly inferior to other configurations. However, its speed improves when more processors are used (moving to the left), achieving a speedup of 6x when using 10 cores per solver. As the number of processors increase, the returns for using more cores per solver diminishes. The trend is inverted after the mark of 20 cores per PIPS-S solver because the time needed to solve the LP relaxation grows and the efficiency of PIPS-S drops off quickly. This dropoff in PIPS-S efficiency suggests that the total

available parallelism should be divided between the two levels of parallelism. In this case, the best configuration is to distribute the 500 available processors among 20-25 PIPS-SBB solvers, with each solver getting 20-25 processors. This timing result regarding multilevel parallelism is an important point to make, and further confirmed by the remaining charts in Figure 6.10: When large process counts are available, performance can be significantly improved by distributing them among different levels of parallelism.

The remaining charts display the performance of PIPS-PSBB in terms of total tree size and communication overhead. When solving larger problems, PIPS-PSBB is able to keep a low communication overhead until 100 solvers are used. The overhead spikes significantly when using 500 solvers. This spike in overhead is due to the slow performance of PIPS-S when solving the initial LP relaxations, and the problems PIPS-PSBB faces when generating work for all solvers quickly at the beginning of the optimization. As a result, most solvers remain idle at ramp-up, causing the significant overhead. This finding can be confirmed in the last plot, where the communication overhead at ramp up is plotted for the different processor arrangements, and is highly correlated with the overall communication overhead. In general, the node throughput increases as more solvers are used, though its progress is hampered by the overhead created by slow LP relaxations.

ug[PIPS-SBB,MPI] displays a similar performance behavior, as seen in Figure 6.11.

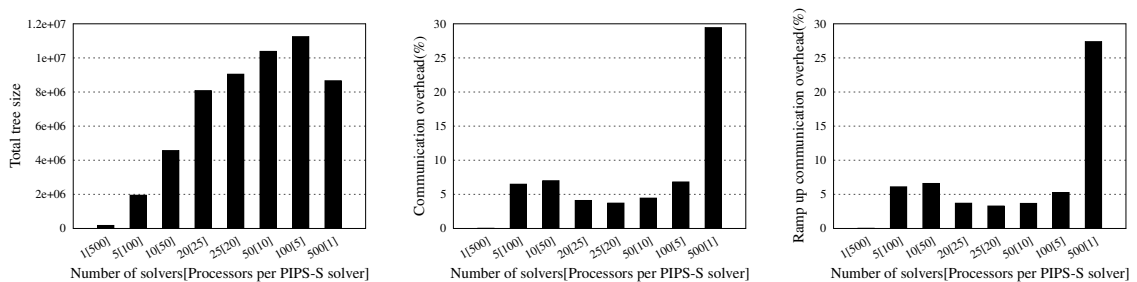


Figure 6.11: Scaling performance of ug[PIPS-SBB,MPI] when solving *slp_10_50_500*: Time to solve the LP relaxation, communication overhead, number of nodes processed, and communication overhead at ramp-up.

6.4.3 Overall performance and comparison to CPLEX

We illustrate the overall performance of PIPS-PSBB and ug[PIPS-SBB, MPI] by testing them on all instances from the SSLP set. For these experiments, we present results for a representative parallel processor configuration, where the number of cores used for each PIPS-SLP solver is chosen as a function of the number of scenarios. In turn, a configuration of 200 solvers is used for all but the trivial problems. We also add CPLEX 12.6.2 to the comparison, both in its shared-memory and distributed-memory implementations using a comparable number of computing cores.

Instances are solved to a relative gap of 10^{-4} (CPLEX default). Each experiment is given a time limit of 1 hour (3600 seconds), and the performance results are reported as “(Time)” in seconds needed to prove optimality. If an optimal solution is not provably obtained within the time limit, then the performance results are reported in terms of the relative gap, which is calculated based on the best found upper and lower bounds, z_{UB} and z_{LB} :

$$Gap = \frac{z_{UB} - z_{LB}}{z_{UB}} \quad (6.1)$$

The instances where CPLEX ran out of memory are denoted with (M) next to the GAP at termination.

From Table 6.1, we see that PIPS-PSBB and ug[PIPS-SBB, MPI] perform comparably through the entire problem set. The first set of rows correspond to the trivial *sslp_5_** instances, which all solvers are able solve to optimality, though CPLEX is significantly faster than all PIPS-SBB variants. The next set consists of three easy *sslp_15_** instances, which have a small number of scenarios. CPLEX is able to solve all instances in this case, while PIPS-SBB implementations are able to solve only one of the three instances, and at a significantly slower pace. In the case of *sslp_10_** instances, the problem difficulty increases as the number of scenarios grows. Distributed-memory CPLEX runs out of memory before solving the instances with 50, 100 and 500 scenarios to optimality. We also suspect the

Table 6.1: Performance comparison for all SSLP instances

Problem Instance	Configuration		PIPS-PSBB Gap(%) (time)(s)	ug[PIPS-SBB,MPI] Gap(%) (time)(s)	CPLEX SM		CPLEX DM	
	Solvers	PIPS-S procs			Procs	Gap(%) (time)(s)	Procs	Gap(%) (time)(s)
sslp_5_25_50	2	2	(7.45s)	(8.03s)	4	(0.27s)	4	(0.27s)
sslp_5_25_100	2	2	(22.37s)	(17.7951s)	4	(0.64s)	4	(0.64s)
sslp_15_45_5	200	2	(107.11s)	(163.53s)	16	(1.97s)	400	(6.26s)
sslp_15_45_10	200	2	0.09	0.16	16	(1.81s)	400	(15.94s)
sslp_15_45_15	200	2	0.25	0.30	16	(7.8s)	400	(15.75s)
sslp_10_50_50	200	10	0.13	0.21	16	(43.88s)	2000	0.15(M)
sslp_10_50_100	200	10	0.17	0.20	16	(221.69s)	2000	0.16(M)
sslp_10_50_500	200	10	0.24	0.24	16	4.91(M)	2000	1.25(M)
sslp_10_50_1000	200	10	0.24	0.24	16	9.21	2000	6.08
sslp_10_50_2000	200	10	0.26	0.26	16	19.93	2000	8.11

same would have happened for the larger instances if the solver was allowed more time. The performance for *sslp_10_** is similar in all distributed-memory parallel algorithms, with PIPS-SBB implementations taking a significant advantage for instances with more than 500 scenarios. Note that CPLEX has no knowledge that it is solving an extensive formulation, which results in its poor performance when the number of scenarios is large. It is also worth mentioning the poor performance of distributed-memory CPLEX compared to its shared-memory counterpart, which is only able to provide a better performance for the two largest problems in the set.

In Table 6.2, we show the performance improvements made from the version of PIPS-SBB introduced in [29] to the current parallel implementations presented in this paper. Significant performance improvements are made possible, not only with the introduction

Table 6.2: Version-to-version performance comparison of PIPS-SBB solvers

Problem Instance	PIPS-SBB presented in [29]		Parallel versions in current paper			
	Procs	Gap(time)	Configuration Solvers	PIPS-S procs	PIPS-PSBB Gap(time)	ug[PIPS-SBB,MPI] Gap(time)
sslp_5_25_50	1	(12.34s)	2	2	(7.45s)	(8.03s)
sslp_5_25_100	1	(41.63s)	2	2	(22.37s)	(17.7951s)
sslp_15_45_5	2	1.36	200	2	(107.11s)	(163.53s)
sslp_15_45_10	2	7.93	200	2	0.09	0.16
sslp_15_45_15	2	5.25	200	2	0.25	0.30
sslp_10_50_50	5	1.48	200	10	0.13	0.21
sslp_10_50_100	10	1.74	200	10	0.17	0.20
sslp_10_50_500	50	1.57	200	10	0.24	0.24
sslp_10_50_1000	100	1.60	200	10	0.24	0.24
sslp_10_50_2000	100	24.00	200	10	0.26	0.26

of parallel branch-and-bound, but also with the addition of further specialized heuristics and branching methods.

6.5 Conclusions

In this paper, we present PIPS-PSBB and ug[PIPS-SBB,MPI]: two implementations of parallel distributed-memory Branch-and-Bound. The first of the proposed methods, PIPS-PSBB, is a new fine-grained algorithm for parallelizing the tree search. The coordination and load-balancing of the different parallel solvers is done in a decentralized fashion, and designed to ensure that all available cores are processing the most promising parts of the branch-and-bound tree. The second method is ug[PIPS-SBB,MPI]: a parallel implementation using UG, a generic framework for parallelizing branch-and-bound tree search that is relatively coarse-grained in its approach. The UG framework has been effectively used to parallelize other MIP solvers such as Xpress and SCIP.

We implement both frameworks for parallelizing branch-and-bound tree search as extensions of PIPS-SBB, a distributed memory solver for Stochastic MIPs (SMIPs). Therefore, both our implementations leverage two levels of nested parallelism in order to improve parallel scalability. We study the effects of leveraging multiple levels of parallelism in improving scaling performance. We also compare our algorithms against the distributed-memory branch-and-bound implementation of the state-of-the-art commercial solver CPLEX. The latter proves to be the best performer for small problems. However, the specialized nature of the methods present in PIPS-SBB-based solvers enable them to outperform CPLEX in large SMIP instances.

PIPS-SBB has seen a dramatic performance improvement since its inception. As new features get added, it will become a more viable option when solving generic two-stage SMIPs. A natural extension of this work is to improve the performance of the base solver. In particular, further specialized methods to improve the convergence of bounds, such as specialized cuts and preprocessing can be added. Another natural extension of this work is

to incorporate *three* levels of parallelism by hierarchically incorporating both frameworks of parallelism presented in this paper. Since UG can already handle a distributed-memory base solver, it can be integrated with the fine-grained parallel branch-and-bound implementation PIPS-PSBB, resulting in ug[*PIPS-PSBB*,MPI]. Adding a third level of parallelism would enable more opportunities for parallel speedup with large process counts.

CHAPTER 7

CONCLUSIONS

The ever-growing needs of industry demand solving larger MIPs with faster times to high quality solutions. The introduction of many algorithmic techniques have contributed to satisfy this demand by making MIP solving more computationally efficient. However, these developments have not been backed up by recent improvements in sequential processor design. With the end of Moore's law, performance improvements in CPU design have come from replicating logic units and CPU cores. The objective in this thesis is to examine techniques and engineer new parallel algorithms for MIP solving, with the intent of fully utilizing the new advancements and benefits in computer architecture. These algorithms go beyond the traditional uses of HPC, and introduce novel ways to expose parallelism.

7.1 Main contributions

We investigate the features required for building highly effective parallel primal heuristics and the multiple problem decomposition techniques required to expose parallelism. We target general MIPs and specialized problems alike. We also study the role of parallel heuristics within the context of parallel branch-and-bound. The direct product of this thesis is a set of algorithms ready to be used in massively parallel systems to quickly find high quality solutions to any MIP problem. In some cases, we show our algorithms to be several orders of magnitude more effective than current state-of-the-art approaches.

Research is carried out using a progressive bottom-up approach. In a first instance, we use focus on the Fixed Charge Multicommodity Network Flow problem. The proposed algorithm relies on the network characteristics of the problem as well as the already found solutions to improve the work balance among parallel processors and the effectiveness of the method at finding high quality solutions. The presented method proves to be more

effective at finding substantially better primal solutions when compared to state-of-the-art commercial MIP solvers, even when the latter are allowed five times as much time. The performance advantage becomes especially significant in the context of large sized problem instances.

Fixed Charge Multicommodity Network Flow problem instances are easy from a feasibility point of view. In addition, one might argue that our problem-specific heuristic holds a great advantage compared to a general-purpose tool, such as a commercial MIP solver. This is why we introduce Parallel Alternating Criteria Search, a parallel primal heuristic specifically designed to handle any general MIP instance. This new method specifically targets instances for which a single feasible solution may be hard to find. We test our tool as a standalone heuristic, but we also study how to combine it with an exact algorithm, such as branch-and-bound. Our parallel method is able to produce competitive or better and faster results for more than 90% of the tested instances against CPLEX. We find our approach to be particularly suitable for large instances. It is in this context where opportunities to exploit parallelism can be found, and a significant advantage with respect to commercial solvers can be seen.

The proposed general algorithm could benefit from further improvements. Besides the parallelization, we introduced general strategies for addressing the choice of starting solutions and variable fixings in the context of general MIPs. However, their specification is independent of the parallel framework. Thus, it could be possible to substitute them for more effective fixings when considering specific classes of problems with defined structures. In our next work, we take the Maritime Inventory Routing Problem as an example and focus on tailoring the modular components of Parallel Alternating Criteria Search to the structure of the problem. The new specialized parallel heuristic is able to significantly outperform state-of-the-art MIP solvers and domain specific heuristics. The advantage increases considerably when solving hard instances featuring long horizon periods, and a large number of ports and vessels. All in all, we show that Parallel Alternating Criteria

Search is an excellent platform that the MIP practitioner can rely on as is, or for a rapid prototyping of effective parallel heuristics for any particular MIP domain.

We also investigate alternative parallelization strategies for branch-and-bound, which try to alleviate the performance and scaling issues often found in most frameworks. PIP-SBB is a new exact distributed-memory parallel branch-and-bound based solver specifically designed for dual block-angular MIPs. It supports parallel data distribution, by partitioning the problem among multiple computing nodes, allowing to potentially solve much larger problems. PIPS-SBB also brings parallelism to the heart of branch-and-bound by performing the LP relaxations in parallel. To achieve this goal, it uses the parallel simplex solver PIPS-S as its backbone LP solver. A second level of nested parallelism is added to increase the scalability of our solver. Concretely, we present two implementations of parallel memory-distributed branch-and-bound extending the already parallel solver. The first of the proposed methods is a new fine-grained algorithm for parallelizing the tree search. The coordination and load-balancing of the different parallel workers is done in a decentralized fashion, and it is designed to ensure all available cores are processing the most promising parts of the branch-and-bound tree. Additionally, we present `ug[PIPS-SBB,MPI]`: a parallel implementation using UG, a generic framework for parallelizing branch-and-bound tree search that is relatively coarse-grained in its approach. We use these frameworks to study the effects of leveraging multiple levels of parallelism and their part in improving scaling performance beyond thousands of cores. We also compare our algorithms against the distributed-memory branch-and-bound implementation of the state-of-the-art commercial solver CPLEX. The latter proves to be the best performer at small problem scales. However, the specialized nature of the methods present in PIPS-SBB-based solvers allow them to outperform CPLEX in large SMIP instances. PIPS-SBB has seen a dramatic performance improvement since its inception. As new features get added, it will become a more viable option when solving generic two-stage SMIPs.

7.2 Future research areas

Most primal heuristics within a solver require an extensive amount of parameter tuning in order to be effective at finding solutions. In most commercial solvers, proprietary tunings are developed for each suite of heuristics. These settings are usually based on experience and remain fixed throughout the optimization. Machine Learning can potentially improve the performance of heuristics by automatically tuning these parameters and to dynamically modify them throughout the optimization. By adding parallel computing to the combination, such learning has the potential to be sped up significantly.

Cut generation, primal heuristics, preprocessing, conflict analysis and branching are all essential algorithmic components of the branch-and-bound that could potentially benefit from parallelism but remain untapped fields of study. In our view, this may be partially caused by the current parallel paradigm, where threads are focused exclusively on solving multiple sub-problems in parallel. At a first glance, adding a new parallel algorithm within such scheme may seem incompatible, as it would require a diversion of the parallel resources to that effect. If parallelizations of such algorithms prove to be effective, additional research will be needed to completely understand the arrangement and interaction between them.

Appendices

APPENDIX A

NOMENCLATURE USED IN THE MARITIME INVENTORY ROUTING

PROBLEM FORMULATIONS

A.1 Indices and sets

$t \in \mathcal{T}$	set of time periods with $T = \mathcal{T} $
$v \in \mathcal{V}$	set of vessels
$vc \in \mathcal{VC}$	set of vessel classes
$j \in \mathcal{J}^P$	set of production ports
$j \in \mathcal{J}^C$	set of consumption ports
$j \in \mathcal{J}$	set of all ports: $\mathcal{J} = \mathcal{J}^P$
$n \in \mathcal{N}$	set of regular nodes or port-time pairs: $\mathcal{N} = \{n = (j, t) : j \in \mathcal{J}, t \in \mathcal{T}\}$
$n \in \mathcal{N}_{0,T+1}$	set of all nodes, including the source node n_0 and a sink node n_{T+1}
$a \in \mathcal{A}$	set of all arcs
$a \in \mathcal{A}^v$	set of arcs associated with vessel $v \in \mathcal{V}$
$a \in \mathcal{A}^{vc}$	set of arcs associated with vessel class $vc \in \mathcal{VC}$
$a \in \mathcal{FS}_n^v$	set of all outgoing arcs associated with node $n = (j, t) \in \mathcal{N}_{0,T+1}$ and vessel $v \in \mathcal{V}$
$a \in \mathcal{FS}_n^{vc}$	set of all outgoing arcs associated with node $n = (j, t) \in \mathcal{N}_{0,T+1}$ and vessel class $vc \in \mathcal{VC}$
$a \in \mathcal{RS}_n^v$	set of all outgoing arcs associated with node $n = (j, t) \in \mathcal{N}_{0,T+1}$ and vessel $v \in \mathcal{V}$
$a \in \mathcal{RS}_n^{vc}$	set of all outgoing arcs associated with node $n = (j, t) \in \mathcal{N}_{0,T+1}$ and vessel class $vc \in \mathcal{VC}$
$\mathcal{FS}_n^{vc,inter}$	set of all outgoing interregional arcs for node $n = (j, t) \in \mathcal{N}_{s,t}$ and vessel class $vc \in \mathcal{VC}$

A.2 Problem data

$\alpha_{j,t}^{max}$	upper bound on the amount of product that can be bought/sold at the spot market at port $j \in \mathcal{J}$ and time $t \in \mathcal{T}$
α_j^{max}	upper bound on the cumulative amount of product that can be bought/sold at the spot market at port $j \in \mathcal{J}$ over the entire planning horizon
B_j	number of berths available at port $j \in \mathcal{J}$
C_a^v	cost for vessel $v \in \mathcal{V}$ to traverse arc $a \in \mathcal{A}^v$
C_a^{vc}	cost for vessel class $vc \in \mathcal{VC}$ to traverse arc $a \in \mathcal{A}^v$
$d_{j,t}$	number of units produced or consumed at port $j \in \mathcal{J}$ in time period $t \in \mathcal{T}$
Δ_j	an indicator parameter taking value $+1$ if $j \in \mathcal{J}^P$, and -1 otherwise
ϵ_z	nonnegative cost parameter associated with attempting to load or discharge at a port
$F_{j,t}^{min}(F_{j,t}^{max})$	minimum (maximum) amount of product that can be loaded or discharged at port $j \in \mathcal{J}$ from a single vessel in time period $t \in \mathcal{T}$
$P_{j,t}$	nonnegative penalty parameter associated with one unit of lost production or stockout at port $j \in \mathcal{J}$ in time period $t \in \mathcal{T}$
Q^v	capacity of vessel $v \in \mathcal{V}$
Q^{vc}	capacity of vessel class $vc \in \mathcal{VC}$
R_n	the unit sales revenue for product discharged at port-time pair $n = (j, t) \in \mathcal{N}$
$S_{j,t}^{min}(S_{j,t}^{max})$	lower bound (capacity) at port $j \in \mathcal{J}$ in time period $t \in \mathcal{T}$
$s_{j,0}$	initial inventory at port $j \in \mathcal{J}$
s_0^v	initial inventory on vessel $v \in \mathcal{V}$

A.3 Problem decision variables

$\alpha_{j,t}$	(continuous) amount of product that port $j \in \mathcal{J}$ purchases or sells to the spot market in time period $t \in \mathcal{T}$
----------------	---

Continued on next page

f_n^v	(continuous) amount loaded or discharged at port-time pair $n = (j, t) \in \mathcal{N}$ from vessel $v \in \mathcal{V}$
$s_{j,t}$	(continuous) number of units of inventory at port $j \in \mathcal{J}$ available at the end of period $t \in \mathcal{T}$
s_t^v	(continuous) number of units of inventory on vessel $v \in \mathcal{V}$ available at the end of period $t \in \mathcal{T}$
x_a^v	(binary) takes value 1 if vessel $v \in \mathcal{V}$ uses arc a incident to node $n = (j, t) \in \mathcal{N}$
x_a^{vc}	(integer) takes value 1 if vessel class $vc \in \mathcal{VC}$ uses arc a incident to node $n = (j, t) \in \mathcal{N}$
z_n^v	(binary) takes value 1 if vessel $v \in \mathcal{V}$ attempts to load or discharge product at node $n = (j, t) \in \mathcal{N}$

REFERENCES

- [1] G. L. Nemhauser and L. A. Wolsey, “Integer programming and combinatorial optimization,” Wiley, Chichester. *GL Nemhauser, MWP Savelsbergh, GS Sigismondi (1992). Constraint Classification for Mixed Integer Programming Formulations. COAL Bulletin*, vol. 20, pp. 8–12, 1988.
- [2] A. H. Land and A. G. Doig, “An automatic method of solving discrete programming problems,” *Econometrica*, vol. 28, no. 3, pp. 497–520, 1960.
- [3] T. Berthold, “Primal heuristics for mixed integer programs,” *Diploma Thesis, Technische Universitat Berlin*, 2006.
- [4] M. Fischetti and A. Lodi, “Heuristics in mixed integer programming,” *Wiley Encyclopedia of Operations Research and Management Science*, 2011.
- [5] M. Savelsbergh, “Preprocessing and probing techniques for mixed integer programming problems,” *ORSA Journal on Computing*, vol. 6, no. 4, pp. 445–454, 1994.
- [6] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger, “Presolve reductions in mixed integer programming,” Technical Report 16-44, ZIB, Berlin, Tech. Rep., 2016.
- [7] H. Marchand, A. Martin, R. Weismantel, and L. Wolsey, “Cutting planes in integer and mixed integer programming,” *Discrete Applied Mathematics*, vol. 123, no. 1–3, pp. 397–446, 2002.
- [8] H. Crowder, E. L. Johnson, and M. Padberg, “Solving large-scale zero-one linear programming problems,” *Operations Research*, vol. 31, no. 5, pp. 803–834, 1983.
- [9] Z. Gu, G. L. Nemhauser, and M. W. Savelsbergh, “Lifted cover inequalities for 0-1 integer programs: Computation,” *INFORMS Journal on Computing*, vol. 10, no. 4, pp. 427–437, 1998.
- [10] T. J. Van Roy and L. A. Wolsey, “Solving mixed integer programming problems using automatic reformulation,” *Operations Research*, vol. 35, no. 1, pp. 45–57, 1987.
- [11] R. Gomory, “An algorithm for the mixed integer problem,” *No. RAND-P-1885*, 1960.

- [12] E. Balas, S. Ceria, and G. Cornuéjols, “A lift-and-project cutting plane algorithm for mixed 0–1 programs,” *Mathematical programming*, vol. 58, no. 1-3, pp. 295–324, 1993.
- [13] A. N. Letchford and A. Lodi, “Strengthening chvátal–gomory cuts and gomory fractional cuts,” *Operations Research Letters*, vol. 30, no. 2, pp. 74–82, 2002.
- [14] H. Marchand and L. A. Wolsey, “Aggregation and mixed integer rounding to solve mip,” *Operations research*, vol. 49, no. 3, pp. 363–371, 2001.
- [15] M. Bénichou, J.-M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O Vincent, “Experiments in mixed-integer linear programming,” *Mathematical Programming*, vol. 1, no. 1, pp. 76–94, 1971.
- [16] J.-M. Gauthier and G. Ribière, “Experiments in mixed-integer linear programming using pseudo-costs,” *Mathematical Programming*, vol. 12, no. 1, pp. 26–47, 1977.
- [17] T. Achterberg, T. Koch, and A. Martin, “Branching rules revisited,” *Operations Research Letters*, vol. 33, no. 1, pp. 42–54, 2005.
- [18] J. T. Linderoth and M. W. Savelsbergh, “A computational study of search strategies for mixed integer programming,” *INFORMS Journal on Computing*, vol. 11, no. 2, pp. 173–187, 1999.
- [19] D. T. Wojtaszek and J. W. Chinneck, “Faster mip solutions via new node selection rules,” *Computers & Operations Research*, vol. 37, no. 9, pp. 1544–1556, 2010.
- [20] M. Fischetti, F. Glover, and A. Lodi, “The feasibility pump,” *Mathematical Programming*, vol. 104, no. 1, pp. 91–104, 2005.
- [21] T. Berthold, “Rens,” *Mathematical Programming Computation*, vol. 6, no. 1, pp. 33–54, 2014.
- [22] E. Danna, E. Rothberg, and C. Le Pape, “Exploring relaxation induced neighborhoods to improve mip solutions,” *Mathematical Programming*, vol. 102, no. 1, pp. 71–90, 2005.
- [23] E. Rothberg, “An evolutionary algorithm for polishing mixed integer programming solutions,” *INFORMS Journal on Computing*, vol. 19, no. 4, pp. 534–541, 2007.
- [24] D. A. Bader, W. E. Hart, and C. A. Phillips, “Parallel algorithm design for branch and bound,” in *Tutorials on Emerging Methodologies and Applications in Operations Research*, Springer, 2005, Chapter 5.

- [25] T. Koch, T. Ralphs, and Y. Shinano, “Could we use a million cores to solve an integer program?” *Mathematical Methods of Operations Research*, vol. 76, no. 1, pp. 67–93, 2012.
- [26] T. Ralphs, Y. Shinano, T. Berthold, and T. Koch, “Parallel solvers for mixed integer linear optimization,”
- [27] L.-M. Munguía, S. Ahmed, D. A. Bader, G. L. Nemhauser, V. Goel, and Y. Shao, “A parallel local search framework for the fixed-charge multicommodity network flow problem,” *Computers & Operations Research*, vol. 77, pp. 44–57, 2017.
- [28] L.-M. Munguía, S. Ahmed, D. A. Bader, G. L. Nemhauser, and Y. Shao, “Alternating criteria search: A parallel large neighborhood search algorithm for mixed integer programs,” *Computational Optimization and Applications*, pp. 1–24, 2016.
- [29] L.-M. Munguía, G. Oxberry, and D. Rajan, “Pips-sbb: A parallel distributed-memory branch-and-bound algorithm for stochastic mixed-integer programs,” in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, IEEE, 2016, pp. 730–739.
- [30] T. L. Magnanti and R. T. Wong, “Network design and transportation planning: Models and algorithms,” *Transportation Science*, vol. 18, no. 1, pp. 1–55, 1984.
- [31] I. Ghamlouche, T. G. Crainic, and M. Gendreau, “Path relinking, cycle-based neighbourhoods and capacitated multicommodity network design,” *Annals of Operations research*, vol. 131, no. 1-4, pp. 109–133, 2004.
- [32] I. Ghamlouche, T. G. Crainic, M. Gendreau, and I. Sbeity, “Learning mechanisms and local search heuristics for the fixed charge capacitated multicommodity network design,” *IJCSI*, p. 5, 2011.
- [33] M. Chouman and T. Crainic, *A MIP-tabu search hybrid framework for multicommodity capacitated fixed-charge network design*. CIRRELT, 2010.
- [34] M. Yaghini, M. Momeni, and M. Sarmadi, “A simplex-based simulated annealing algorithm for node-arc capacitated multicommodity network design,” *Applied Soft Computing*, vol. 12, no. 9, pp. 2997–3003, 2012.
- [35] M. P. Kleeman, B. A. Seibert, G. B. Lamont, K. M. Hopkinson, and S. R. Graham, “Solving multicommodity capacitated network design problems using multiobjective evolutionary algorithms,” *Evolutionary Computation, IEEE Transactions on*, vol. 16, no. 4, pp. 449–471, 2012.

- [36] A. M. Alvarez, J. L. González-Velarde, and K. De-Alba, “Scatter search for network design problem,” *Annals of Operations Research*, vol. 138, no. 1, pp. 159–178, 2005.
- [37] T. G. Crainic and M. Gendreau, “Metaheuristics: Progress in complex systems optimization,” in Boston, MA: Springer US, 2007, pp. 25–40, ISBN: 978-0-387-71921-4.
- [38] D. C. Paraskevopoulos, T. Bektaş, T. G. Crainic, and C. N. Potts, “A cycle-based evolutionary algorithm for the fixed-charge capacitated multi-commodity network design problem,” *European Journal of Operational Research*, pp. –, 2016.
- [39] M. Fischetti and A. Lodi, “Local branching,” *Mathematical Programming*, vol. 98, no. 1-3, pp. 23–47, 2003.
- [40] I. Rodríguez-Martín and J. J. Salazar-González, “A local branching heuristic for the capacitated fixed-charge network design problem,” *Computers & Operations Research*, vol. 37, no. 3, pp. 575–581, 2010.
- [41] N. Katayama, M. Z. Chen, and M. Kubo, “A capacity scaling heuristic for the multicommodity capacitated network design problem,” *Journal of Computational and Applied Mathematics*, vol. 232, no. 1, pp. 90–101, 2009.
- [42] N. Katayama, “A combined capacity scaling and local branching approach to capacitated multi-commodity network design problem,” *Far East Journal of Applied Mathematics*, vol. 92, no. 1, p. 1, 2015.
- [43] M. Hewitt, G. L. Nemhauser, and M. W. P. Savelsbergh, “Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem,” *INFORMS Journal on Computing*, vol. 22, no. 2, pp. 314–325, 2010.
- [44] ———, “Branch-and-price guided search for integer programs with an application to the multicommodity fixed-charge network flow problem,” *INFORMS Journal on Computing*, vol. 25, no. 2, pp. 302–316, 2013.
- [45] V. A. Badrinarayanan, K. C. Furman, V. Goel, Y. Shao, and G. Li, “Parallel large-neighborhood search techniques for lng inventory routing,” *Optimization Online*, 2014.
- [46] T. G. Crainic and M. Gendreau, “Cooperative parallel tabu search for capacitated network design,” *Journal of Heuristics*, vol. 8, no. 6, pp. 601–627, 2002.
- [47] T. G. Crainic, Y. Li, and M. Toulouse, “A first multilevel cooperative algorithm for capacitated multicommodity network design,” *Comput. Oper. Res.*, vol. 33, no. 9, pp. 2602–2622, Sep. 2006.

- [48] T. G. Crainic and M. Toulouse, “Parallel meta-heuristics,” in *Handbook of Meta-heuristics*, ser. International Series in Operations Research & Management Science, vol. 146, Springer US, 2010, pp. 497–541, ISBN: 978-1-4419-1663-1.
- [49] A. Frangioni, *Multicommodity problems*, [Online; accessed 2016-07-05], 2013.
- [50] B. W. Kernighan and S. Lin, “An efficient heuristic procedure for partitioning graphs,” *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.
- [51] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
- [52] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.
- [53] M. Hewitt, *Gt instances*, [Online; accessed 2016-07-05], 2010.
- [54] P. S. P. Ltd, *Passmark software*, [Online; accessed 2017-08-05], 2017.
- [55] M. Fischetti and A. Lodi, “Heuristics in mixed integer programming,” in *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Inc., 2010, ISBN: 9780470400531.
- [56] L. Bertacco, M. Fischetti, and A. Lodi, “A feasibility pump heuristic for general mixed-integer problems,” *Discrete Optimization*, vol. 4, no. 1, pp. 63–76, 2007.
- [57] T. Achterberg and T. Berthold, “Improving the feasibility pump,” *Discrete Optimization*, vol. 4, no. 1, pp. 77–86, 2007.
- [58] M. Fischetti and D. Salvagnin, “Feasibility pump 2.0,” *Mathematical Programming Computation*, vol. 1, no. 2-3, pp. 201–222, 2009.
- [59] N. L. Boland, A. C. Eberhard, F. G. Engineer, M. Fischetti, M. W. Savelsbergh, and A. Tsoukalas, “Boosting the feasibility pump,” *Mathematical Programming Computation*, pp. 1–25, 2011.
- [60] D. Baena and J. Castro, “Using the analytic center in the feasibility pump,” *Operations Research Letters*, vol. 39, no. 5, pp. 310–317, 2011.
- [61] J. Naoum-Sawaya, “Recursive central rounding for mixed integer programs,” *Computers & Operations Research*, vol. 43, no. 0, pp. 191–200, 2014.
- [62] G. Gamrath, T. Berthold, S. Heinz, and M. Winkler, “Structure-based primal heuristics for mixed integer programming,” in *Optimization in the Real World: Toward*

Solving Real-World Optimization Problems, K. Fujisawa, Y. Shinano, and H. Waki, Eds. Tokyo: Springer Japan, 2016, pp. 37–53.

- [63] T. Berthold and G. Hendel, “Shift-and-propagate,” *Journal of Heuristics*, vol. 21, no. 1, pp. 73–106, Feb. 2015.
- [64] C. Wallace, “Zi round, a mip rounding heuristic,” *Journal of Heuristics*, vol. 16, no. 5, pp. 715–722, 2010.
- [65] T. Achterberg, T. Berthold, and G. Hendel, “Rounding and propagation heuristics for mixed integer programming,” in *Operations Research Proceedings 2011*, Springer, 2012, pp. 71–76.
- [66] F. Glover, A. LøKketangen, and D. L. Woodruff, “Scatter search to generate diverse mip solutions,” in *Computing Tools for Modeling, Optimization and Simulation: Interfaces in Computer Science and Operations Research*, M. Laguna and J. L. G. Velarde, Eds. Boston, MA: Springer US, 2000, pp. 299–317.
- [67] F. Glover and M. Laguna, “General purpose heuristics for integer programming—part i,” *Journal of Heuristics*, vol. 2, no. 4, pp. 343–358, 1997.
- [68] ———, “General purpose heuristics for integer programming—part ii,” *Journal of Heuristics*, vol. 3, no. 2, pp. 161–179, 1997.
- [69] E. Balas, S. Schmieta, and C. Wallace, “Pivot and shift—a mixed integer programming heuristic,” *Discrete Optimization*, vol. 1, no. 1, pp. 3–12, 2004.
- [70] E. Balas, S. Ceria, M. Dawande, F. Margot, and G. Pataki, “Octane: A new heuristic for pure 0-1 programs,” *Operations Research*, vol. 49, no. 2, pp. 207–225, 2001.
- [71] T. Achterberg, “Constraint integer programming,” PhD thesis, 2007.
- [72] G. Hendel, “New rounding and propagation heuristics for mixed integer programming,” *Bachelor’s thesis, TU Berlin*, 2011.
- [73] P. Hansen, N. Mladenović, and D. Urošević, “Variable neighborhood search and local branching,” *Computers & Operations Research*, vol. 33, no. 10, pp. 3034–3045, 2006, Part Special Issue: Constraint Programming.
- [74] S. Ghosh, “Dins, a mip improvement heuristic,” in *Integer Programming and Combinatorial Optimization*, ser. Lecture Notes in Computer Science, vol. 4513, Springer Berlin Heidelberg, 2007, pp. 310–323, ISBN: 978-3-540-72791-0.
- [75] M. Fischetti and M. Monaci, “Proximity search for 0-1 mixed-integer convex programming,” *Journal of Heuristics*, vol. 20, no. 6, pp. 709–731, 2014.

- [76] *IBM CPLEX optimizer*, <http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/>, 2015.
- [77] *Gurobi optimizer*, <http://www.gurobi.com>, 2015.
- [78] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch, “Parascip: A parallel extension of SCIP,” in *Competence in High Performance Computing 2010*, Springer, 2012, pp. 135–148.
- [79] M. Fischetti, A. Lodi, M. Monaci, D. Salvagnin, and A. Tramontani, “Improving branch-and-cut performance by random sampling,” *Mathematical Programming Computation*, vol. 8, no. 1, pp. 113–132, 2016.
- [80] R. Carvajal, S. Ahmed, G. Nemhauser, K. Furman, V. Goel, and Y. Shao, “Using diversification, communication and parallelism to solve mixed-integer linear programs,” *Operations Research Letters*, vol. 42, no. 2, pp. 186–189, 2014.
- [81] U. Koc and S. Mehrotra, “Generation of feasible integer solutions on a massively parallel computer,” *Article in submission*,
- [82] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter, “MIPLIB 2010,” *Mathematical Programming Computation*, vol. 3, no. 2, pp. 103–163, 2011.
- [83] M. Fischetti and A. Lodi, “Repairing mip infeasibility through local branching,” *Computers & Operations Research*, vol. 35, no. 5, pp. 1436–1445, 2008, Part Special Issue: Algorithms and Computational Methods in Feasibility and Infeasibility.
- [84] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard,” *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.
- [85] R. Thakur, R. Rabenseifner, and W. Gropp, “Optimization of collective communication operations in mpich,” *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, 2005.
- [86] E Danna, “Performance variability in mixed integer programming,” in *Presentation at Workshop on Mixed Integer Programming*, 2008.
- [87] T. Berthold, “Measuring the impact of primal heuristics,” *Operations Research Letters*, vol. 41, no. 6, pp. 611–614, 2013.
- [88] D. J. Papageorgiou, M.-S. Cheon, S. Harwood, F. Trespalacios, and G. L. Nemhauser, “Recent progress using matheuristics for strategic maritime inventory routing,” in

Modeling, Computing and Data Handling Methodologies for Maritime Transportation, Springer, 2018, pp. 59–94.

- [89] D. J. Papageorgiou, G. L. Nemhauser, J. Sokol, M.-S. Cheon, and A. B. Keha, “Mirplib—a library of maritime inventory routing problem instances: Survey, core model, and benchmark results,” *European Journal of Operational Research*, vol. 235, no. 2, pp. 350–366, 2014.
- [90] M. Christiansen, K. Fagerholt, T. Flatberg, Øyvind Haugen, O. Kloster, and E. H. Lund, “Maritime inventory routing with multiple products: A case study from the cement industry,” *European Journal of Operational Research*, vol. 208, no. 1, pp. 86–94, 2011.
- [91] S. Dauzère-Pérès, A. Nordli, A. Olstad, K. Haugen, U. Koester, P. O. Myrstad, G. Teistklub, and A. Reistad, “Omya hustadmarmor optimizes its supply chain for delivering calcium carbonate slurry to european paper manufacturers,” *Interfaces*, vol. 37, no. 1, pp. 39–51, 2007.
- [92] A. Agra, M. Christiansen, A. Delgado, and L. M. Hvattum, “A maritime inventory routing problem with stochastic sailing and port times,” *Computers & Operations Research*, vol. 61, pp. 18–30, 2015.
- [93] K. C. Furman, J.-H. Song, G. R. Kocis, M. K. McDonald, and P. H. Warrick, “Feedstock routing in the exxonmobil downstream sector,” *Interfaces*, vol. 41, no. 2, pp. 149–163, 2011.
- [94] V. Goel, K. C. Furman, J.-H. Song, and A. S. El-Bakry, “Large neighborhood search for lng inventory routing,” *Journal of Heuristics*, vol. 18, no. 6, pp. 821–848, 2012.
- [95] M. Stålhane, J. G. Rakke, C. R. Moe, H. Andersson, M. Christiansen, and K. Fagerholt, “A construction and improvement heuristic for a liquefied natural gas inventory routing problem,” *Computers & Industrial Engineering*, vol. 62, no. 1, pp. 245–255, 2012.
- [96] Y. Shao, K. C. Furman, V. Goel, and S. Hoda, “A hybrid heuristic strategy for liquefied natural gas inventory routing,” *Transportation Research Part C: Emerging Technologies*, vol. 53, pp. 151–171, 2015.
- [97] F. Mutlu, M. K. Msakni, H. Yildiz, E. Sönmez, and S. Pokharel, “A comprehensive annual delivery program for upstream liquefied natural gas supply chain,” *European Journal of Operational Research*, vol. 250, no. 1, pp. 120–130, 2016.

- [98] D. J. Papageorgiou, M.-S. Cheon, G. Nemhauser, and J. Sokol, "Approximate dynamic programming for a class of long-horizon maritime inventory routing problems," *Transportation Science*, vol. 49, no. 4, pp. 870–885, 2014.
- [99] D. J. Papageorgiou, A. B. Keha, G. L. Nemhauser, and J. Sokol, "Two-stage decomposition algorithms for single product maritime inventory routing," *INFORMS Journal on Computing*, vol. 26, no. 4, pp. 825–847, 2014.
- [100] J.-H. Song and K. C. Furman, "A maritime inventory routing problem: Practical approach," *Computers & Operations Research*, vol. 40, no. 3, pp. 657–665, 2013.
- [101] F. G. Engineer, K. C. Furman, G. L. Nemhauser, M. W. P. Savelsbergh, and J.-H. Song, "A branch-price-and-cut algorithm for single-product maritime inventory routing," *Operations Research*, vol. 60, no. 1, pp. 106–122, 2012.
- [102] M. Hewitt, G. Nemhauser, M. Savelsbergh, and J.-H. Song, "A branch-and-price guided search approach to maritime inventory routing," *Computers & Operations Research*, vol. 40, no. 5, pp. 1410–1419, 2013.
- [103] F. Al-Khayyal and S.-J. Hwang, "Inventory constrained maritime routing and scheduling for multi-commodity liquid bulk, part i: Applications and model," *European Journal of Operational Research*, vol. 176, no. 1, pp. 106–130, 2007.
- [104] J. G. Rakke, M. Stålhane, C. R. Moe, M. Christiansen, H. Andersson, K. Fagerholt, and I. Norstad, "A rolling horizon heuristic for creating a liquefied natural gas annual delivery program," *Transportation Research Part C: Emerging Technologies*, vol. 19, no. 5, pp. 896–911, 2011.
- [105] A. Agra, M. Christiansen, A. Delgado, and L. Simonetti, "Hybrid heuristics for a short sea inventory routing problem," *European Journal of Operational Research*, vol. 236, no. 3, pp. 924–935, 2014.
- [106] K. T. Uggen, M. Fodstad, and V. S. Nørstebø, "Using and extending fix-and-relax to solve maritime inventory routing problems," *Top*, vol. 21, no. 2, pp. 355–377, 2013.
- [107] V. Goel, M. Slusky, W.-J. van Hoesve, K. C. Furman, and Y. Shao, "Constraint programming for lng ship scheduling and inventory management," *European Journal of Operational Research*, vol. 241, no. 3, pp. 662–673, 2015.
- [108] B. V. Asokan, K. C. Furman, V. Goel, Y. Shao, and G. Li, "Parallel large-neighborhood search techniques for lng inventory routing," *Submitted for publication*, 2014.
- [109] Y. Shao, K. C. Furman, V. Goel, and S. Hoda, "Bound improvement for lng inventory routing," *Submitted for publication*, 2014.

- [110] R. Byrd, A. Goldman, and M. Heller, “Recognizing unbounded integer programs,” *Oper. Res.*, vol. 35, no. 1, pp. 140–142, Feb. 1987.
- [111] W. Klein Haneveld and M. van der Vlerk, “Stochastic integer programming: General models and algorithms,” *Annals of Operations Research*, vol. 85, pp. 39–57, 1999.
- [112] A. Kleywegt, A. Shapiro, and T. Homem-de Mello, “The sample average approximation method for stochastic discrete optimization,” *SIAM Journal on Optimization*, vol. 12, no. 2, pp. 479–502, 2002.
- [113] *FICO Xpress Optimization Suite*, <http://www.fico.com/en/products/fico-xpress-optimization-suite>.
- [114] T. Achterberg, “Scip: Solving constraint integer programs,” *Mathematical Programming Computation*, vol. 1, no. 1, pp. 1–41, 2009.
- [115] Y. Shinano and T. Fujie, “Paralex: A parallel extension for the CPLEX mixed integer optimizer,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer, 2007, pp. 97–106.
- [116] T. Ralphs, L. Ladányi, and M. Saltzman, “Parallel branch, cut, and price for large-scale discrete optimization,” *Mathematical Programming*, vol. 98, pp. 253–280, 1 2003.
- [117] ———, “A library hierarchy for implementing scalable parallel search algorithms,” *J. Supercomput.*, vol. 28, no. 2, pp. 215–234, 2004.
- [118] C. Phillips, J. Eckstein, and W. Hart, “Massively parallel mixed-integer programming: Algorithms and applications,” in *Parallel Processing for Scientific Computing*. SIAM Books, 2006, ch. 17, pp. 323–340. eprint: <http://epubs.siam.org/doi/pdf/10.1137/1.9780898718133.ch17>.
- [119] J. Hall, “Towards a practical parallelisation of the simplex method,” *Computational Management Science*, vol. 7, no. 2, pp. 139–170, 2010.
- [120] M. Lubin, J. Hall, C. Petra, and M. Anitescu, “Parallel distributed-memory simplex for large-scale stochastic LP problems,” *Computational Optimization and Applications*, vol. 55, no. 3, pp. 571–596, 2013.
- [121] S. Ahmed, R. Garcia, N. Kong, L. Ntaimo, G. Parija, F. Qiu, and S. Sen, *Siplib: A stochastic integer programming test problem library*, 2013.
- [122] S. Sen, “Algorithms for stochastic mixed-integer programming models,” *Handbooks in operations research and management science*, vol. 12, pp. 515–558, 2005.

- [123] F. Louveaux and R. Schultz, “Stochastic integer programming,” *Handbooks in Operations Research and Management Science*, vol. 10, pp. 213–266, 2003.
- [124] T. Ralphs and A. Hassanzadeh, “A generalization of Benders’ algorithm for two-stage stochastic optimization problems with mixed integer recourse,” Lehigh University, Technical Report 14T-005, 2014.
- [125] M. Zhang and S. Küçükyavuz, “Finitely convergent decomposition algorithms for two-stage stochastic pure integer programs,” *SIAM Journal on Optimization*, vol. 24, no. 4, pp. 1933–1951, 2014.
- [126] H. Serali and B. Fraticelli, “A modification of Benders’ decomposition algorithm for discrete subproblems: An approach for stochastic programs with integer recourse,” *Journal of Global Optimization*, vol. 22, no. 1-4, pp. 319–342, 2002.
- [127] S. Ahmed, M. Tawarmalani, and N. Sahinidis, “A finite branch-and-bound algorithm for two-stage stochastic integer programs,” *Mathematical Programming*, vol. 100, no. 2, pp. 355–377, 2004.
- [128] J.-P. Watson, D. Woodruff, and W. Hart, “Pysp: Modeling and solving stochastic programs in Python,” *Mathematical Programming Computation*, vol. 4, no. 2, pp. 109–149, 2012.
- [129] D. Gade, G. Hackebeil, S. Ryan, J.-P. Watson, R. Wets, and D. Woodruff, “Obtaining lower bounds from the progressive hedging algorithm for stochastic mixed-integer programs,” *Optimization Online*, 2014.
- [130] C. Carøe and R. Schultz, “Dual decomposition in stochastic integer programming,” *Operations Research Letters*, vol. 24, no. 1, pp. 37–45, 1999.
- [131] M. Lubin, K. Martin, C. Petra, and B. Sandıkçı, “On parallelizing dual decomposition in stochastic integer programming,” *Operations Research Letters*, vol. 41, no. 3, pp. 252–258, 2013.
- [132] G. Guo, G. Hackebeil, S. Ryan, J.-P. Watson, and D. Woodruff, “Integration of progressive hedging and dual decomposition in stochastic integer programs,” *Operations Research Letters*, vol. 43, no. 3, pp. 311–316, 2015.
- [133] S. Ahmed, “A scenario decomposition algorithm for 0–1 stochastic programs,” *Operations Research Letters*, vol. 41, no. 6, pp. 565–569, 2013.
- [134] G. Gamrath and M. Lübbecke, “Experiments with a generic Dantzig-Wolfe decomposition for integer programs,” in *Experimental Algorithms*, ser. Lecture Notes in Computer Science, vol. 6049, Springer Berlin Heidelberg, 2010, pp. 239–252, ISBN: 978-3-642-13192-9.

- [135] F. Vanderbeck, *BaPCod – A generic branch-and-price code*, <https://wiki.bordeaux.inria.fr/realopt/pmwiki.php/Project/BaPCod>, 2005.
- [136] T. Ralphs and M. Galati, *DIP – Decomposition for integer programming*, <https://projects.coin-or.org/Dip>, 2009.
- [137] K. Kim and V. Zavala, “Algorithmic innovations and software for the dual decomposition method applied to stochastic mixed-integer programs,” *Optimization Online*, 2015.
- [138] A. Langer, R. Venkataraman, U. Palekar, and L. Kale, “Parallel branch-and-bound for two-stage stochastic integer optimization,” in *High Performance Computing (HiPC), 2013 20th International Conference on*, 2013, pp. 266–275.
- [139] *Stochastic programming software and test sets*, <http://stoprog.org/index.html?software.html>.
- [140] R. Bixby and E. Rothberg, “Progress in computational mixed integer programming—a look back from the other side of the tipping point,” *Annals of Operations Research*, vol. 149, pp. 37–41, 1 2007.
- [141] M. Lubin, C. Petra, M. Anitescu, and V. Zavala, “Scalable stochastic optimization of complex energy systems,” in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, IEEE, 2011, pp. 1–10.
- [142] C. Petra, O. Schenk, and M. Anitescu, “Real-time stochastic optimization of complex energy systems on high-performance computers,” *Computing in Science Engineering*, vol. 16, no. 5, pp. 32–42, 2014.
- [143] C. Petra, O. Schenk, M. Lubin, and K. Gärtner, “An augmented incomplete factorization approach for computing the schur complement in stochastic optimization,” *SIAM Journal on Scientific Computing*, vol. 36, no. 2, pp. C139–C162, 2014. eprint: <http://dx.doi.org/10.1137/130908737>.
- [144] J. Gondzio and A. Grothey, “A new unblocking technique to warmstart interior point methods based on sensitivity analysis,” *SIAM Journal of Optimization*, vol. 19, no. 3, pp. 1184–1210, 2008.
- [145] L. Ntaimo and S. Sen, “The million-variable “march” for stochastic combinatorial optimization,” *Journal of Global Optimization*, vol. 32, no. 3, pp. 385–400, 2005.
- [146] L. Ntaimo, *Stochastic mixed-integer programming test problems*, http://ise.tamu.edu/people/faculty/ntaimo/personal_web/test_instances.htm, 2015.

- [147] S. Ahmed and R. Garcia, “Dynamic capacity acquisition and assignment under uncertainty,” *Annals of Operations Research*, vol. 124, no. 1-4, pp. 267–283, 2003.
- [148] G. Angulo, S. Ahmed, and S. Dey, “Improving the integer L-shaped method,” *INFORMS Journal on Computing*, 2016, To appear.
- [149] J. Huchette, M. Lubin, and C. Petra, “Parallel algebraic modeling for stochastic optimization,” in *High Performance Technical Computing in Dynamic Languages (HPTCDL)*, 2014, pp. 29–35.
- [150] J. Eckstein, W. E. Hart, and C. A. Phillips, “Pebbl: An object-oriented framework for scalable parallel branch-and-bound,” *Mathematical Programming Computation*, vol. 7, no. 4, pp. 429–469, 2015.
- [151] J. Eckstein, C. A. Phillips, and W. E. Hart, *Pebbl 1.0 user guide*, <https://software.sandia.gov/acro/releases/votd/acro/packages/pebbl/doc/uguide/user-guide.pdf>, 2007.
- [152] Y. Xu, “Scalable algorithms for parallel tree search,” PhD thesis, Lehigh University, 2007.
- [153] Y. Xu, T. K. Ralphs, L. Ladányi, and M. Saltzmann, *ALPS version 1.5*, <https://github.com/coin-or/CHiPPS-ALPS>, 2016.
- [154] —, *BiCePs version 0.94*, <https://github.com/coin-or/CHiPPS-BiCePS>, 2017.
- [155] —, *BLIS version 0.94*, <https://github.com/coin-or/CHiPPS-BLIS>, 2017.
- [156] Y. Xu, T. K. Ralphs, L. Ladányi, and M. J. Saltzmann, “Alps: A frameowkr for implementing parallel search algorithms,” ser. The Proceedings of the Ninth INFORMS Computing Society Conference, 2005, pp. 319–334.
- [157] —, “Computational expereicne with a software framework for parallel integer programming,” *The INFORMS Journal on Computing*, vol. 21, pp. 383–397, 2009.
- [158] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler, “Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Los Alamitos, CA, USA: IEEE Computer Society, 2016, pp. 770–779.
- [159] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler, “FiberSCIP – a shared memory parallelization of SCIP,” Tech. Rep., to appear.

- [160] *Ug: ubiquity generator framework*, <http://ug.zib.de/>.
- [161] S. J. Maher, T. Fischer, T. Galley, G. Gamrath, A. Gleixner, R. L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, D. Weninger, J. T. Witt, and J. Witzig, “The SCIP optimization suite 4.0,” Zuse Institute Berlin, Tech. Rep. ZIB-Report 17-12, 2017.

VITA

Lluís-Miquel Munguía was born in Girona, Catalonia, on July 11th of 1989. On September, 2007, he enrolled at the school of Informatics at the Universitat Politècnica de Catalunya, from where he received a Bachelor and Masters in computer science (May 2012). While pursuing his studies in Barcelona, Lluís worked with Professor Ernest Teniente in the development of automatic reasoning tools for validating UML Software Engineering schemas. In the last year of his degree, he visited the Georgia Institute of Technology as an exchange student. During his stay, he collaborated with Professor Frank Dellaert on Computer Vision applications for mobile platforms, and Professor David A. Bader on graph algorithms using parallel heterogeneous architectures. In August 2012, he started his Ph.D. studies under the supervision of Professor Bader at the Computational Science and Engineering School at the Georgia Institute of technology. There, he worked on several research projects. He first focused on parallel graph algorithms with Professor Bader and Dr. Oded Green. The most significant part of his degree was dedicated to developing parallel algorithms for discrete optimization in collaboration with Professor George L. Nemhauser, Professor Shabbir Ahmed, and Dr. Deepak Rajan. This thesis focuses on the latest work.