Parallel cutting plane algorithms for inverse mixed integer linear programming

Zhaoyang Duan
Iowa State University
Parallel cutting plane algorithms for inverse mixed integer linear programming

by

Zhaoyang Duan

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Industrial and Manufacturing Systems Engineering

Program of Study Committee:
Lizhi Wang, Major Professor
   Sigurdur Olafsson
   Mervyn G. Marasinghe

Iowa State University
Ames, Iowa
2009

Copyright © Zhaoyang Duan, 2009. All rights reserved.
DEDICATION

I would like to dedicate this thesis to my parents Xiqi Duan and Yingzhen Su, whose love and support made me confident to complete this work. It is also dedicated to my grandparents Zengyu Duan and Qiaofeng Feng, who gave me unconditional love.
TABLE OF CONTENTS

LIST OF TABLES ................................................................. vi
LIST OF FIGURES .............................................................. vii
ACKNOWLEDGEMENTS ......................................................... viii

CHAPTER 1. OVERVIEW ......................................................... 1
  1.1 Introduction to Inverse Optimization .................................... 1
  1.2 Introduction to Parallel Computing ...................................... 2
  1.3 Applications of Inverse Optimization ................................... 3
  1.4 Research Objective ...................................................... 4
  1.5 Thesis Organization ..................................................... 6

CHAPTER 2. REVIEW OF LITERATURE ......................................... 7
  2.1 Inverse Optimization .................................................... 7
  2.2 Parallel Computing and Parallel Algorithms ........................... 8

CHAPTER 3. CUTTING PLANE ALGORITHM FOR InvMILP WITH ONE PROCESSOR ......................................................... 9
  3.1 Problem Definition ..................................................... 9
  3.2 Cutting Plane Algorithm for InvMILP .................................. 10
    3.2.1 Illustration of the cutting plane algorithm ..................... 11
    3.2.2 Limitations of the cutting plane algorithm ................... 13

CHAPTER 4. PARALLEL CUTTING PLANE ALGORITHM FOR Inv- vMILP WITH TWO PROCESSORS .................................................. 14
  4.1 Parallel Algorithm for the Cutting Plane Algorithm ................. 14
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Computational performances on different values of ( \theta )</td>
<td>19</td>
</tr>
<tr>
<td>6.1</td>
<td>Computational performances of algorithms on MIPLIB instances</td>
<td>31</td>
</tr>
<tr>
<td>6.2</td>
<td>Computational performances of algorithms on random instances</td>
<td>32</td>
</tr>
<tr>
<td>6.3</td>
<td>Computational performances of instance20 using three processors</td>
<td>33</td>
</tr>
<tr>
<td>6.4</td>
<td>Potential improvements and speedup for PCAlg(^{\text{InvMILP}}) with two processors</td>
<td>34</td>
</tr>
<tr>
<td>6.5</td>
<td>Potential improvements and speedup for PCAlg(^{\text{InvMILP}}) with three processors</td>
<td>34</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Parallel computing - shared memory</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>Research objectives</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>A two dimensional example of InvMILP</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Cutting plane algorithm for InvMILP</td>
<td>11</td>
</tr>
<tr>
<td>3.3</td>
<td>Two dimensional example for CA\textsuperscript{InvMILP}</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>Parallel algorithm for CA\textsuperscript{InvMILP}</td>
<td>16</td>
</tr>
<tr>
<td>4.2</td>
<td>Two dimensional example showing gap between lower bound and upper bound</td>
<td>17</td>
</tr>
<tr>
<td>4.3</td>
<td>Parallel cutting plane algorithm for InvMILP</td>
<td>19</td>
</tr>
<tr>
<td>4.4</td>
<td>Computational performances on different values of $\theta$</td>
<td>20</td>
</tr>
<tr>
<td>4.5</td>
<td>Gap reduction for different values of $\theta$</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Executing PC\textsuperscript{Alg\textsuperscript{InvMILP}} with three computers</td>
<td>22</td>
</tr>
<tr>
<td>5.2</td>
<td>Searching extreme points from different directions</td>
<td>23</td>
</tr>
<tr>
<td>5.3</td>
<td>Two computers search different gaps</td>
<td>25</td>
</tr>
<tr>
<td>6.1</td>
<td>Sequential algorithm for PC\textsuperscript{Alg\textsuperscript{InvMILP}}</td>
<td>27</td>
</tr>
<tr>
<td>6.2</td>
<td>Potential and actual improvements for PC\textsuperscript{Alg\textsuperscript{InvMILP}}</td>
<td>35</td>
</tr>
<tr>
<td>6.3</td>
<td>Potential and actual speedup for PC\textsuperscript{Alg\textsuperscript{InvMILP}}</td>
<td>36</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, my major professor Dr. Lizhi Wang for his guidance, patience and support throughout this research and the writing of this thesis. His insights and words have encouraged me and his enthusiasm has inspired me to pursue a high level of research work. I would also appreciate my committee members for their efforts and contributions to this work: Dr. Sigurdur Olafsson and Dr. Mervyn G. Marasinghe. I would also like to thank my friends for their help during the writing of this work.
CHAPTER 1. OVERVIEW

1.1 Introduction to Inverse Optimization

A typical optimization problem is a forward problem since it intends to find an optimal solution given the values of parameters, which include cost coefficients and constraints. In an inverse optimization problem, the objective is to find a set of cost coefficients that minimally perturbs the objective function in order to make a given feasible solution optimal under constraints. In the past few years, there have been a lot of studies in inverse optimization problems. A general definition and formulation of the inverse optimization problem that have been studied so far is given by (1), which is defined as, “to perturb the cost vector \( c \) to \( d \) so that \( x^0 \) is an optimal solution with respect to the perturbed cost vector and the cost of perturbation is minimum”.

Wang (2) presents cutting plane algorithms for solving the inverse mixed integer linear programming problem (InvMILP). In each iteration, the algorithm provides one improved lower bound, and the corresponding constraint is added into the main problem in order to get closer and closer to the final optimal solution. There are two limitations of the cutting planes algorithm, 1) the algorithm takes many iterations to get to the optimal solution; 2) no feasible solution is generated until the algorithm terminates.

The heuristic algorithm presented in my thesis is an extended research based on the cutting plane algorithms in (2). By executing this algorithm in parallel with the cutting plane algorithms, it helps increase the efficiency of improving lower bounds, while at the same time provide upper bounds and feasible solutions before the algorithm terminates. The new algorithm is called parallel cutting plane algorithm for solving InvMILP, which is noted as PCAlgInvMILP. It contains two parts. The main part is the cutting plane algorithm, which
is noted as CAlg\textsuperscript{InvMILP}, and the parallel part is the heuristic algorithm introduced in this thesis, which is noted as PAlg\textsuperscript{InvMILP}. In a conclusion, PCAlg\textsuperscript{InvMILP} is an algorithm in which CAlg\textsuperscript{InvMILP} and PAlg\textsuperscript{InvMILP} are carried out simultaneously.

### 1.2 Introduction to Parallel Computing

Parallel computing is a form of computation in which many calculations are carried out simultaneously \cite{3}. Traditionally, software has been written for serial computation. Programs are running on a single computer with a single central processing unit (CPU). An algorithm is constructed and implemented as a serial stream of instructions. A series of instructions are executed one after another and they can only be executed one at a time. Parallel computing is the simultaneous use of multiple computers to solve a problem. A problem is broken into discrete parts that can be solved concurrently with multiple CPUs \cite{4}.

![Parallel computing - shared memory](image)

**Figure 1.1** Parallel computing - shared memory

Shared memory is one of the parallel computer memory architectures that has the ability for all the processors to access all memory. As shown in Figure 1.1, different processors can operate independently but share the same memory. The need for communications between tasks depends upon the types of problems \cite{4}. Some types of problems do not need tasks to
share data while most parallel applications are not so simple, which require tasks to share data with each other. For these types of problems, frequent communications can result in spending time “waiting” instead of doing work.

Parallel computing has been used in lots of real world problems. These applications require the processing of large amounts of data. Some of applications are in the areas of atmosphere, environment, nuclear, bioscience, mechanical engineering, electrical engineering, computer science, oil exploration, web search engines, medical imaging and diagnosis, financial and economic modeling, and advanced graphics and virtual reality, etc.

1.3 Applications of Inverse Optimization

Inverse optimization has been applied in plenty of various areas. Geophysical scientists were among the earliest researchers that studied inverse optimization, who applied it in predicting the movements of earthquakes (1). Obayashi and Jeong (5) use inverse optimization method for blunt-trailing-edge airfoils. Torquato (6) implements inverse optimization techniques for targeted self-assembly. Multi-criteria optimization and stability analysis are also mentioned as important applications of inverse optimization in (1). Traffic equilibrium is an easily understood application of inverse problem (1). Another important application of inverse problem arises in high-speed network (7). The last two applications are explained in detail as following.

There are a number of routes between different origin-destination pairs. Drives usually select their routes which minimize their travel cost, while a transportation planner wants to minimize the total travel cost over the network. The user’s equilibrium flow does not necessarily correspond to the system optimal flow. Thus tolls may be imposed on some routes in order to make these two flows identical. How to impose the minimum total toll to achieve the identical flows becomes an inverse optimization problem.

Application in high-speed network has similar problem as in transportation networks. In asynchronous transfer mode network, when adding a new node or link to an existing network, a dynamic link-state flooding protocol will help automatically distribute the changes in link parameters to other nodes in the network. The path-selection algorithm computes the routes
for traffic sources. Since the dynamic routing contains several control and feedback loops, stability is an important concern. In order to combine stability and the advantages of a dynamic scheme, manually configured administrative weights are assigned to the links. With the use of administrative weights the route selection is simplified to finding shortest paths by well-known algorithms. The inverse shortest path problem appears here to find the optimal administrative weights in order to implement this strategy. Farago (7) introduces this inverse optimization approach in pan-European asynchronous transfer mode networks.

1.4 Research Objective

The main objective of our research is to apply parallel computing techniques to the inverse optimization algorithms in order to increase the efficiency of existing algorithm. As far as we know, this is the first algorithm using parallel computing for solving InvMILPs.

Figure 1.2 Research objectives

The first objective is to implement PCAlg$^{\text{InvMILP}}$ with two processors to increase existing algorithm efficiency. As shown in Figure 1.2, the existing algorithm is CAlg$^{\text{InvMILP}}$, which is executed in one processor. Besides efficiency improvement, we also expect that our parallel algorithm could provide more benefits, e.g., providing upper bounds and feasible solutions before algorithm terminates. After presenting our new parallel algorithm, we want to investigate more on how to improve our algorithm. Choice of parameters is an important factor that can influence the efficiency of algorithm in real applications. How to find the best values of param-
eters for a specific problem can be as important as using a good algorithm. Thus, sensitivity analysis is conducted on some important parameters of our algorithm, and we try to get more improvement by searching for the best parameter values.

The second objective is to improve more of $\text{PCAlg}^\text{invMILP}$ by using three or more processors. In general, the more processors we have, the higher efficiency we get. However, the relationship between the number of processors and the efficiency of the algorithm may not be linear. We want to get the extent to which parallel computing facilities can be explored to improve the efficiency of the algorithm. Our $\text{PCAlg}^\text{invMILP}$ with three processors is conducted in three ways. Firstly, using different values of parameters for different processors; secondly, using different processors to search different partitions; thirdly, using different processors to search from different directions.

Moreover, we want to get the potential improvement of our new algorithm and implement our algorithm using mathematical softwares. Several computational experiments are conducted to compare the efficiency of our new algorithms with existing ones.

In summary, the objectives are to:

- Present $\text{PCAlg}^\text{invMILP}$ with two processors that overcomes the limitations of existing algorithms;
- Conduct sensitivity analysis on important parameters of the $\text{PCAlg}^\text{invMILP}$ with two processors;
- Present $\text{PCAlg}^\text{invMILP}$ with three or more processors that can obtain more efficiency improvement;
- Examine the potential improvement of $\text{PCAlg}^\text{invMILP}$;
- Implement our new algorithms using mathematical softwares and examine the efficiency with some instances;
1.5 Thesis Organization

Chapter 2 reviews the relevant literatures about the inverse optimization problems and parallel computing. Chapter 3 gives the definition of InvMILP talked in my thesis and presents the CAlg\textsuperscript{InvMILP}. The detailed description and illustration of the PCAlg\textsuperscript{InvMILP} with two processors is presented in Chapter 4. Chapter 4 also provides sensitivity analysis on an important parameter $\theta$ and one algorithm improving strategy. Chapter 5 presents three strategies of conducting the PCAlg\textsuperscript{InvMILP} with three or more processors. Algorithm implementation, computational experiment results for all the algorithms, and potential improvement of PCAlg\textsuperscript{InvMILP} are reported in Chapter 6. Conclusion and future research follow in Chapter 7.
CHAPTER 2. REVIEW OF LITERATURE

2.1 Inverse Optimization

Inverse problems were firstly and extensively studied by geophysical scientists, e.g., (8), (9), and (10). As mentioned in the first chapter, an important application is to predict the movements of earthquakes. Tarantola (11) gives a comprehensive introduction of inverse problem theory in the area of geophysical sciences.

In the mathematical programming community, Burton et al. (12) are the first to study inverse optimization. They study inverse shortest path problems which have also been used in predicting earthquakes. An algorithm based on the Golfarb-Idnani method for convex quadratic programming is proposed and some preliminary numerical results are reported. Zhang et al. (13) formulate an inverse shortest path problem as a special linear programming problem. They propose a column generation method to get an optimal solution in finitely many steps. Yang and Zhang (14) study inverse maximum capacity path problem. They transform the problem into the minimum weight cut set problem and showed that the problem can be solved efficiently if an efficient algorithm for finding minimum weight cut set is available.

Ahuja and Orlin have conducted extensive research on inverse optimization and make an impressive contribution. Sokkalingam et al. (15) study the inverse spanning tree problem and develop an $O(n^3)$ algorithm under the $L_1$ norm and an $O(n^2)$ algorithm under the $L_\infty$ norm. Ahuja and Orlin (16) study the convex ordered set problem, which is a generalization of the inverse sorting problem. Ahuja and Orlin (17) present an $O(n^2 \log n)$ algorithm to solve the inverse spanning tree problem under $L_1$ norm. Ahuja and Orlin (18) study several special cases of the inverse linear programming problem under the $L_1$ and $L_\infty$ norms, e.g., the shortest path problem, the minimum cut problem, and the minimum cost flow problem. Ahuja and Orlin
(19) consider inverse network flow problems for the unit weight case and develop proofs which do not reply on the inverse linear programming theory. Ahuja and Orlin (20) prove that if the inverse versions of a problem are polynomially solvable under the weighted $L_1$ and $L_\infty$ norms for a problem with a linear cost function that is polynomially solvable.

Heuberger (21) gives a comprehensive summary of inverse combinational problems. Huang (22), Iyengar and Kang (23) conduct research on inverse problem of mixed integer and nonlinear programming. Wang (2) is among the first to report an algorithm for solving InvMILPs.

2.2 Parallel Computing and Parallel Algorithms

Parallel computing has already developed for half a century. The interest in paralleling computing dates back to late 1950s. The first article that talked about this topic is (24). After that, a lot of research work has been done in this area. For example, in 1958, IBM researchers John Cocke and Daniel Slotnick were the first to discuss the use of parallelism in numerical calculations. D825 was introduced by Burroughs corporation in 1962, which was a four-processor computer that accessed up to 16 memory modules through a crossbar switch. In the mid 1980s, a new kind of parallel computing was launched when the Caltech Concurrent Computation project built a supercomputer for scientific applications. Clusters came to complete and eventually displace massively parallel processors for many applications starting in the late 1980s. Today, clusters are the workhorse of scientific computing and are the dominant architecture in the data centers that power the modern information age.

Parallel algorithm has been applied to many areas. Some researchers have examined parallel algorithms for hierarchical clustering. Rasmussen and Willet (25) discuss parallel implementations of clustering using the single link metric and the minimum variance metric on a SIMD array processor. Bruynooghe (26) describes a parallel implementation of the nearest neighbors clustering algorithm suitable for a parallel supercomputer. Dehne et al. (27) use parallel computing in external memory algorithms. They provide a simulation technique which produces efficient parallel external memory algorithms from efficient BSP-like parallel algorithms.
CHAPTER 3. CUTTING PLANE ALGORITHM FOR InvMILP WITH ONE PROCESSOR

In this chapter we introduce the cutting plane algorithm for InvMILP with one processor presented in [2], which is noted as CAlgInvMILP in my thesis.

3.1 Problem Definition

Let \( \mathcal{IP} (A, b, c, I) \) denote an instance of mixed integer linear program (MILP)

\[
\max_x \{ c^\top x : Ax \leq b, x \geq 0, x_I \in \mathbb{Z} \},
\]

(3.1)

where \( A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n, \) and \( I \subseteq \{1, \ldots, n\} \). An InvMILP is to minimally perturb \( c \) in order to make the given feasible solution \( x^d \) optimal. In other words, it’s to find a direction \( d \) that satisfies two conditions. Firstly, the direction \( d \) makes given feasible solution \( x^d \) optimal to MILP (3.1); secondly, the difference between \( c \) and \( d \), measured by \( ||c - d|| \), is minimal among all the directions that can make \( x^d \) optimal. In this thesis, we focus our discussion on the weighted \( L_1 \) norm, so \( ||c - d|| \) becomes \( w^\top |c - d| \), where \( w \in \mathbb{R}_+^n \) is a constant vector. The straightforward formulation of the InvMILP under the weighted \( L_1 \) norm is

\[
\min_d \{ w^\top |c - d| : x^d \in \arg\max_x \{ d^\top x : Ax \leq b, x \geq 0, x_I \in \mathbb{Z} \} \}. \tag{3.2}
\]

Figure 3.1 shows a two dimensional example of InvMILP. \( c = [1/3, 1] \) is the direction that makes \( x^c = (5, 2) \) optimal to MILP (3.1). \( d^* = [-1/2, 1] \) is the direction that minimally perturbs \( c \) and makes given feasible solution \( x^d = (-1, 0) \) optimal to MILP (3.2). In another word, \( d^* \) is the optimal solution to InvMILP (3.2).
3.2 Cutting Plane Algorithm for InvMILP

The three-step CAlg\textsuperscript{InvMILP} presented by (2) is shown as following:

**Step 0** Initiate $S^0 = \emptyset$.

**Step 1** Solve the following linear programming problem (LP) and let $(y^*, e^*, f^*)$ be an optimal solution, a new direction $(c - e^* + f^*)$ is found.

$$\min_{y,e,f} \quad w^\top e + w^\top f$$ (3.3)

subject to

$$A^\top y \geq c - e + f$$ (3.4)

$$(c - e + f)^\top x^d \geq (c - e + f)^\top x^0, \quad \forall x^0 \in S^0$$ (3.5)

$$y, e, f \geq 0.$$ (3.6)

**Step 2** Let $x^0$ be an optimal solution to $\mathcal{IP}(A, b, (c - e^* + f^*), I)$. If $(c - e^* + f^*)^\top x^d \geq (c - e^* + f^*)^\top x^0$, then stop, and $d^* = c - e^* + f^*$ is optimal to InvMILP (3.2). Otherwise update $S^0 = S^0 \cup \{x^0\}$ and go back to Step 1.
The process of the cutting plane algorithm for InvMILP is shown in Figure 3.2

We give an explanation on each step of the CAlg\textsuperscript{InvMILP}.

In step 0, the extreme point set \(S^0\) is initially set to be empty.

In step 1, we define \(e, f \in \mathbb{R}_+^n\) such that \(c - d = e - f\), then the objective function (3.3) of the LP problem restricts that the direction is minimally perturbed. Constraint (3.4) is dual feasibility constraint which defines the feasible region. Constraint (3.5) requires that \(x^d\) is better than all the extreme points in \(S^0\) under the new direction found in this step.

In step 2, firstly we get the optimal solution \(x^0\) to MILP (3.1) under the new direction \((c - e^* - f^*)\) found in step 1. Then an optimality check is processed to check whether \(x^0\) is the given feasible solution \(x^d\). If yes, the new direction is the optimal solution to InvMILP (3.2); if no, we have found a lower bound direction and the corresponding extreme point \(x^0\) which is added to extreme point set \(S^0\) and the loop goes back to step 1.

3.2.1 Illustration of the cutting plane algorithm

We give an illustration of the CAlg\textsuperscript{InvMILP} based on an example with two dimensions, which is shown in Figure 3.3.

In the first iteration of our example, after solving the LP problem in step 1, a direction \(c = [1/3, 1]\) is found. This direction minimally perturbs direction \(c\) itself and makes \(x^d = (0, -1)\) better than all the extreme points in set \(S^0\), which is empty in the first iteration. In step 2,
Figure 3.3  Two dimensional example for CAlgInvMILP

$x^e = (5, 2)$ is the optimal solution to $\mathcal{IP}(A, b, c[1/3, 1], I)$ under the new direction $c$ found in step 1 of this iteration.

In the second iteration, the solution of LP in step 1 gives us a new direction $dc = [-1/3, 1]$ and an extreme point $x^e = (4, 2)$. After the optimality check, $dc$ is also a lower bound.

In the third iteration, direction $de = [-2/5, 1]$ is found in step 1 and $x^f = (1, 1)$ is a new extreme point. In step 2 of this iteration, the optimality check shows that direction $de$ is also a lower bound.

In the fourth iteration, step 1 provides us the direction $d^* = [-1/2, 1]$ and the optimality check shows it is the optimal solution to InvMILP (3.2).

Thus, for this two dimensional instance, we have found the optimal direction $d^*$ in four iterations.
3.2.2 Limitations of the cutting plane algorithm

Two limitations of the cutting plane algorithm is mentioned in (2). Firstly, the solution of each iteration is infeasible until the terminating iteration. In each iteration of his algorithm, the lower bound direction is improved until it reaches the optimal solution. Since the inverse problem is a minimization problem, lower bound is not a feasible solution until the terminating iteration. This means the solution found by each iteration is infeasible and the first feasible solution is the optimal solution which was found by the terminating iteration.

Secondly, it may take many iterations to terminate, and in each iteration an MILP instance needs to be solved, which may be time consuming. In each iteration of CAlg$^{InvMILP}$, a new direction, which is corresponding to an extreme point, is generated. Most of the new directions are not optimal to InvMILP (3.2), but lead to another direction and extreme point. The optimal direction is the one that makes $x^d$ optimal, which is found in terminating iteration. The algorithm won’t get to the optimal direction before all the necessary extreme points are found and added into extreme point set $S^0$. Thus, the number of the iterations needed to reach the optimal solution depends on the number of the necessary extreme points that are necessary before terminating iteration. The larger the problem is, the more the iterations.
CHAPTER 4. PARALLEL CUTTING PLANE ALGORITHM FOR InvMILP WITH TWO PROCESSORS

In this chapter we present the parallel cutting plane algorithm for InvMILP with two processors, which is noted as \textsc{PCAlg}_{InvMILP} with two processors. As mentioned in chapter 1, the new algorithm contains two parts. The main part is cutting plane algorithm (\textsc{CAlg}_{InvMILP}) introduced by \cite{2}, and the parallel part (\textsc{PAlg}_{InvMILP}) is a heuristic algorithm that runs simultaneously and in parallel with \textsc{CAlg}_{InvMILP} to improve the efficiency of the main algorithm.

4.1 Parallel Algorithm for the Cutting Plane Algorithm

\textsc{PAlg}_{InvMILP} is presented in this part in order to overcome the shortages of the \textsc{CAlg}_{InvMILP}. By running this algorithm in parallel with the main algorithm, we expect the number of iterations and time needed to reach the optimal solution will be reduced and feasible solutions will be generated along with the loops of algorithm.

As mentioned in Section 3.2.2, the number of iterations needed to reach the optimal solution depends on the number of necessary extreme points. In the \textsc{CAlg}_{InvMILP}, one extreme point is generated in each iteration. If we have a parallel algorithm which could continuously provide useful extreme points, the lower bound direction would be improved faster and the number of iterations and time of the algorithm would be reduced. In another aspect, if we could generate upper bound of the optimal solution, which means the directions that make the feasible solution \(x^d\) optimal to MILP \((3.1)\) but may not necessarily minimally perturb \(c\), we would have feasible solutions to InvMILP \((3.2)\) generated along with the loops of algorithm. These two ideas lead to the \textsc{PAlg}_{InvMILP} described as following:

**Step A** Solve the following inverse problem of LP relaxation of \(IP(A, b, c, I)\) and let \((y^0, e^0, f^0)\)
be an optimal solution. $d^0 = c - e^0 + f^0$ is an upper bound to InvMILP (3.2).

\[
\begin{align*}
\min_{y,e,f} & \quad \omega^\top e + \omega^\top f \quad \tag{4.1} \\
\text{s.t.} & \quad A^\top y \geq c - e + f \quad \tag{4.2} \\
& \quad b^\top y = (c - e + f)^\top x^d \quad \tag{4.3} \\
& \quad y, e, f \geq 0 \quad \tag{4.4}
\end{align*}
\]

**Step B** $g$ is the gap between current lower bound $(c - e^l + f^l)$ and upper bound $d^0$. Reduce the gap between upper and lower bounds by $\theta \in (0, 1)$ by solving the following LP and let $(y^1, e^1, f^1, s^1)$ be an optimal solution. A new direction $d^1 = c - e^1 + f^1$ is found.

\[
\begin{align*}
\max_{y,e,f,s} & \quad s \quad \tag{4.5} \\
\text{s.t.} & \quad \omega^\top e + \omega^\top f \leq \omega^\top e^l + \omega^\top f^l + (1 - \theta)g \quad \tag{4.6} \\
& \quad A^\top y \geq c - e + f \quad \tag{4.7} \\
& \quad (c - e + f)^\top x^d \geq (c - e + f)^\top x^0 + s, \quad \forall x^0 \in S^0 \quad \tag{4.8} \\
& \quad y, e, f \geq 0 \quad \tag{4.9} \\
& \quad s \geq 0 \quad \tag{4.10}
\end{align*}
\]

**Step C** Let $x^1$ be an optimal solution to $\mathcal{IP}(A, b, d^1, I)$. If $(d^1)^\top x^d \geq (d^1)^\top x^1$, then update the upper bound solution $d^0 = d^1$; otherwise update $S^0 = S^0 \cup \{x^1\}$ and go back to Step B.

The process of PAlg$^{\text{InvMILP}}$ is shown in Figure 4.1.

We give an explanation on each step of the PAlg$^{\text{InvMILP}}$.

In step A, we firstly solve the inverse problem of LP relaxation of $\mathcal{IP}(A, b, c, I)$ to get an upper bound direction $d^0$ to InvMILP (3.2).

In step B, we try to improve the upper bound by reducing the gap between upper and lower bounds by $\theta$. Constraint (4.6) restricts the obtained direction is between lower bound and improved upper bound. Constraint (4.7) requires dual feasibility which defines the feasible region. Constraint (4.8) requires that $x^d$ is better than all the extreme points in $S^0$ by at least
In step C, firstly we get the optimal solution $x^1$ to MILP (3.1) under the new direction $d^1$ found in step B. Then optimality check is processed to determine whether $d^1$ is an improved upper bound. If yes, upper bound is updated; if no, it should be a lower bound direction, and the corresponding extreme point $x^1$ is added to extreme point set $S^0$. Loop goes back to step B with updated upper bound or extreme point set $S^0$.

Each iteration of the parallel algorithm either improves the upper bound or improves the lower bound and generates a new extreme point. If the upper bound is improved, a feasible solution is generated in this iteration; otherwise the parallel algorithm can help improve the efficiency of main algorithm since it provides extreme points to $S^0$.

4.1.1 Illustration of the parallel algorithm

We give an illustration of PAIgInvMILP based on our two dimensional example, which is shown in Figure 4.2.

In the first iteration of our example, after solving the LP relaxation problem in step A, an
upper bound direction $d^0 = [-2/3, 1]$ is found. This direction makes $x^d = (0, -1)$ optimal to $\mathcal{IP}(A, b, d^0[-2/3, 1], I)$ but does not minimally perturbs direction $c = [1/3, 1]$.

In step B, the updated lower bound getting from $\text{CAAlgInvMILP}$ is $dc = [-1/3, 1]$. After reducing the gap between upper bound $d^0$ and the lower bound $dc$ by $\theta$, the LP problem is solved, and a new direction $d^1$ is found.

In step C, $x^f = (1, 1)$ is the optimal solution to $\mathcal{IP}(A, b, d^1, I)$. Then the checking process shows $x^f$ is an extreme point and it is added into $S^0$.

This iteration of $\text{PAlgInvMILP}$ helps provide a necessary extreme point $x^f$ for $\text{CAAlgInvMILP}$. With $x^f$ provided, $\text{CAAlgInvMILP}$ will directly find the optimal solution $d^* = [-1/2, 1]$ without extreme point $x^e = (4, 2)$, and the number of iterations for $\text{CAAlgInvMILP}$ in our example will be reduced by one.
4.1.2 Advantages and limitations of the parallel algorithm

There are several advantages of PAlg\textsuperscript{InvMILP}. Firstly, it overcomes the two limitations of CAlg\textsuperscript{InvMILP}. It not only reduces the time and number of iterations of the algorithm, but also generates feasible solutions before the terminating iteration. Secondly, it continuously improves the efficiency of the algorithm. In chapter 5, three strategies of improvement of the algorithm are introduced. Last but not the least, it uses parallel computing techniques for solving InvMILPs. With the development of parallel techniques, this algorithm will be more and more efficient.

There are also some limitations of PAlg\textsuperscript{InvMILP}. Firstly, it is an attaching algorithm that only runs in parallel with CAlg\textsuperscript{InvMILP}. It can generate new directions, but it can not check whether the new direction is optimal or not. Secondly, parameter $\theta$ plays an important role in generating new directions. The choice of $\theta$ directly determines the efficiency of the algorithm, but it is not easy to get the optimal $\theta$ for a certain inverse problem. Sensitivity analysis on $\theta$ will be discussed in section 4.3. Thirdly, due to the limitation of communications between different processors in parallel computing, the algorithm in practical implementation is not as efficient as theoretic.

4.2 Parallel Cutting Plane Algorithm for InvMILP with Two Processors

PCA\textsuperscript{Alg}\textsuperscript{InvMILP} with two processors is a algorithm in which CAlg\textsuperscript{InvMILP} and PAlg\textsuperscript{InvMILP} are carried out simultaneously. In a network with two computers, CAlg\textsuperscript{InvMILP} is executed on computer I and PAlg\textsuperscript{InvMILP} is executed simultaneously on computer II. The extreme point set $S^0$ is the shared memory, which is the only communication between these two algorithms. The whole process of PCA\textsuperscript{Alg}\textsuperscript{InvMILP} with two processors is shown in Figure 4.3

4.3 Sensitivity Analysis on $\theta$

Parameter $\theta$ is the percentage of gap reduction between upper bound and lower bound in step B of PAlg\textsuperscript{InvMILP}, which has been mentioned to have a significant influence on algorithm performance. The magnitude of $\theta$ directly determines whether the new direction $d_1^1$ found in
Figure 4.3 Parallel cutting plane algorithm for InvMILP

step B is a lower bound or upper bound. The sensitivity analysis in this part is to investigate how sensible the algorithm is to the value of \( \theta \).

We did experiments on two randomly generated instances, “instance15” and “instance20”. We solve each instance using randomly generated objective functions and obtain up to 50 different optimal solutions. We then use PCA\textsuperscript{Alg}_{InvMILP} to solve the inverse problem of the instance for all the optimal solutions for different values of \( \theta \). Average number of iterations and time for the algorithm are recorded, which is shown in Table 4.1.

Table 4.1 Computational performances on different values of \( \theta \)

<table>
<thead>
<tr>
<th>( \theta )</th>
<th>instance15</th>
<th>instance20</th>
<th>instance20</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># iter.</td>
<td># sec.</td>
<td># iter.</td>
</tr>
<tr>
<td>0.1</td>
<td>23.7</td>
<td>8.9</td>
<td>0.1</td>
</tr>
<tr>
<td>0.5</td>
<td>22.3</td>
<td>8.4</td>
<td>0.5</td>
</tr>
<tr>
<td>0.6</td>
<td>22.8</td>
<td>8.7</td>
<td>0.8</td>
</tr>
<tr>
<td>0.7</td>
<td>22.6</td>
<td>8.5</td>
<td>0.95</td>
</tr>
<tr>
<td>0.8</td>
<td>23.0</td>
<td>8.7</td>
<td>0.96</td>
</tr>
<tr>
<td>0.95</td>
<td>24.8</td>
<td>9.3</td>
<td>0.97</td>
</tr>
<tr>
<td>0.99</td>
<td>—</td>
<td>—</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>—</td>
<td>0.99</td>
</tr>
</tbody>
</table>
From the computational experiments, we can see the number of iterations and time that the algorithm needs to solve the problems are quite different for different value of $\theta$. This means the algorithm is sensible to the value of $\theta$. For different instances, the optimal value of $\theta$ is different. For instance15, the optimal $\theta$ lies between 0.5 and 0.7, while for instance20, the optimal $\theta$ lies between 0.8 and 1. We can also see the results graphically in Figure 4.4.

![Figure 4.4](image)

**Figure 4.4** Computational performances on different values of $\theta$

### 4.4 Improving Strategy: Dynamic Adjustable $\theta$

From Section 4.3 we find that the algorithm is sensitive to the value of $\theta$. We can improve our parallel algorithm by continuously adjusting $\theta$.

We know the optimal direction is somewhere between upper bound and lower bound, as shown in Figure 4.5. In step B of PA$\text{lg}^{\text{InvMILP}}$, if $\theta$ is set to be a small value, e.g., $\theta$ is equal to 0.2, and we still find an upper bound in step C. This means the optimal direction may possibly be far from upper bound and close to lower bound. In the next iteration, we increase the value
of $\theta$, e.g., $\theta$ is equal to 0.8. If we find a lower bound in step C, which means that we cut too much to the lower bound, in the next iteration, we set $\theta$ to a small value in order to get an upper bound. In a conclusion, the value of $\theta$ for the next iteration depends on the results of step C in this iteration. The reason for adjusting $\theta$ in this way is to get to the optimal direction as quick as possible.

One limitation of this improved method is that a certain way of adjusting $\theta$ does not work well for all instances. In other words, it is not easy to find the best way of adjusting $\theta$ for a specific instance. Actually, the best $\theta$ for the next iteration does not absolutely depends on the results of previous iteration. The way of adjusting $\theta$ presented above is just one of many methods that we think should be proper for most general cases. Our experiment results also proved our concerns. We have randomly generated 50 instances, and compared their performances between PCAgl$^{\text{InvMILP}}$ with dynamic $\theta$ and with constant $\theta$. 21 out of 50 instances need less number of iterations to terminate while the other 29 instances have even worse performances with adjustable $\theta$.

One way to overcome this limitation is using three or more computers and different computers have different values of $\theta$, which will be discussed in next chapter.
CHAPTER 5. PARALLEL CUTTING PLANE ALGORITHM FOR InvMILP WITH THREE OR MORE PROCESSORS

In this chapter, we execute PCAlg$^{\text{InvMILP}}$ with three or more processors to get more improvement and overcome the limitations of PCAlg$^{\text{InvMILP}}$ with two processors. In a network with three or more computers shown in Figure 5.1, CAAlg$^{\text{InvMILP}}$ is still executed in computer I, and PAAlg$^{\text{InvMILP}}$s with different values of parameters are executed in the other computers.

Figure 5.1 Executing PCAlg$^{\text{InvMILP}}$ with three computers
5.1 Basic Strategy of Using Three or More Processors

We know that the key of getting to the optimal solution is to find all the necessary extreme points. The parallel algorithm is actually helping provide the extreme points for the main algorithm. That is why the number of iterations and time have been reduced. If we use three or more computers to provide extreme points simultaneously, more extreme points will be gathered and the main algorithm should reach the optimal solution even faster. However, several issues should be considered when more computers are used. Firstly, the extreme points found by different computers should be different, which means different computers should search for extreme points from different directions, as shown in Figure 5.2. Secondly, all of the computers need to read from or write to a same file in shared memory. Priority should be set as well. Since plenty of time will be spent on waiting for the release of the file, the greater of number of computers in use, the greater of waiting time. Thus we think the practical efficiency of the algorithm may not be improved as fast as the increase of the number of computers.

![Diagram](image.png)

Figure 5.2 Searching extreme points from different directions

In a conclusion, in order to gather different necessary extreme points, we need different
processors to search extreme points from different directions. The less of the reduplicative points searched by processors, the more efficiency of the algorithm. We have three strategies to realize this purpose which are shown in the following sections. All of them are embodied in step B of PAIg^InvMILP.

5.2 Strategy I: Different Values of $\theta$ for Different Processors

In the first strategy, we use different values of $\theta$ for different processors. For example, in a network with three computers, we use computer I to execute CAlg^InvMILP, and use computer II to execute PAIg^InvMILP with $\theta$ equals 0.2, and use computer III to execute PAIg^InvMILP with $\theta$ equals 0.8. The shared extreme point set $S^0$ is updated by all three computers simultaneously.

Large value of $\theta$ generates lower bound most of times, while small value of $\theta$ generates upper bound most of times. Both of them have limitation if we only make use of one. This strategy is to use both values of $\theta$ simultaneously. In this way, we can overcome the limitation of only searching small value or large value of $\theta$, and generate both lower bound and upper bound. Experiments have been conducted to prove the advantage of this strategy. The results and explanation of computational experiments for this method are shown in next chapter.

5.3 Strategy II: Different Searching Partitions for Different Processors

In the second strategy, we divide the gap between upper bound and lower bound into several exclusive and exhaustive partitions and let different processors search different partitions by changing the constraint (4.6), which defines the searching area between the lower bound and upper bound.

As shown in Figure 5.3, the gap between upper bound and lower bound is divided into two exclusive and exhaustive parts. In a network with three computers, computer I executes CAlg^InvMILP. For the other two computers, each computer searches directions in each part and both provide useful information to the main algorithm. For computer II, constraint (4.6) is changing into

$$\omega^T e^l + \omega^T f^l + (1 - \theta)g \leq \omega^T e + \omega^T f \leq \omega^T e^l + \omega^T f^l + g. \quad (5.1)$$
Figure 5.3 Two computers search different gaps

For computer III, constraint (4.6) is changing into

\[ \sum_{j=1}^{n} \omega_j e_j + \sum_{j=1}^{n} \omega_j f_j \leq \sum_{j=1}^{n} \omega_j e'_j + \sum_{j=1}^{n} \omega_j f'_j + (1 - \theta)g. \]

(5.2)

Here in our example, \( \theta = 0.5 \). The results and explanations of computational experiments for this method are shown in next chapter.

5.4 Strategy III: Different Searching Directions for Different Processors

In the third strategy, we use different values of \( \omega \) in constraint (4.6) for different computers. In constraint (4.6) \( \sum_{j=1}^{n} \omega_j e_j + \sum_{j=1}^{n} \omega_j f_j \leq \sum_{j=1}^{n} \omega_j e'_j + \sum_{j=1}^{n} \omega_j f'_j + (1 - \theta)g \), different values of \( \omega \) actually stand for different searching directions.

In a network with three computers, computer I still executes \( \text{CA} \! \! \text{l}g^{\text{InvMILP}} \). The other two computers execute \( \text{PA} \! \! \text{l}g^{\text{InvMILP}} \) with different values of \( \omega \). Normally, \( \omega \) equals \([1, 1, ..., 1]^T\) in \( L_1 \) norm. In this strategy, we let computer II use \( \omega = [100, 1, 1, ..., 1] \) and let computer III use \( \omega = [1, 1, ..., 1, 100] \). In this way, we make the two computers to search the extreme points from different directions. We expect that this strategy could reduce the number of reduplicative extreme points getting from different computers. The results and explanations of computational experiments for this method are shown in next chapter.
CHAPTER 6. IMPLEMENTATION AND EXAMPLES

In this chapter we firstly discuss the potential improvement of the PCAlg\textsuperscript{InvMILP}. Ideal speedup and general speedup are two levels of efficiency objective for our algorithm in implementation. Then we implement PCAlg\textsuperscript{InvMILP} with both two processors and more processors in a computer network that shares the same hard disk. The extreme point set $\mathcal{S}^0$ is stored in hard disk that can be accessed (e.g., read from and written to) by each computers. The algorithm is programmed in Matlab, Tomlab and Cplex. We demonstrate the PCAlg\textsuperscript{InvMILP} using two groups of instances. In order to compare the efficiency of the PCAlg\textsuperscript{InvMILP} and CAlg\textsuperscript{InvMILP}, instances are solved in both algorithms, and time and number of iterations for both algorithms are recorded. The last section shows the computational results of potential improvement, which leads two ways to improve our algorithm.

6.1 Potential Improvement of the Parallel Cutting Plane Algorithm

In parallel computing, speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. It is defined by the following formula:

$$S_p = \frac{T_1}{T_p}$$ \hfill (6.1)

where $p$ is the number of processors, $T_1$ is the execution time of sequential algorithm, and $T_p$ is the execution time of the parallel algorithm with $p$ processors.

6.1.1 Ideal speedup

In general, the ideal speedup is obtained when $S_p = p$. When running an algorithm with ideal speedup, doubling the number of processors doubles the speed. Normally it’s an upper
bound of the speedup. Thus, the ideal speedup is 2 for the PCAlg\textsuperscript{InvMILP} with two processors and 3 for the PCAlg\textsuperscript{InvMILP} with three processors.

### 6.1.2 General speedup

General speedup is defined as the possible maximum efficiency improvement of PCAlg\textsuperscript{InvMILP} compared with CAlg\textsuperscript{InvMILP} when there is no wasting time spent on communication between processors. Here we use the number of iterations as the measurement of efficiency.

In order to get the general speedup of the PCAlg\textsuperscript{InvMILP} with two processors, we make a new algorithm by combining the CAlg\textsuperscript{InvMILP} and the PAlg\textsuperscript{InvMILP} as a sequential algorithm. The idea is to execute CAlg\textsuperscript{InvMILP} and PAlg\textsuperscript{InvMILP} in sequence with one processor. The new algorithm is noted as sequential algorithm for PCAlg\textsuperscript{InvMILP} with two processors. The process of the new algorithm is shown in Figure 6.1. Steps in this algorithm are the same with the ones in PCAlg\textsuperscript{InvMILP}.

![Sequential algorithm for PCAlg\textsuperscript{InvMILP}](image)

We also have sequential algorithm for PCAlg\textsuperscript{InvMILP} with three processors. It is made by combining the CAlg\textsuperscript{InvMILP} and the two PAlg\textsuperscript{InvMILP} as a sequential algorithm. For example, for strategy III of PCAlg\textsuperscript{InvMILP} with three processors. The corresponding sequential algorithm is using one computer to firstly execute CAlg\textsuperscript{InvMILP}, secondly execute PAlg\textsuperscript{InvMILP} with $\omega_1 =$
[100, 1, 1, ..., 1], and at last execute PAIg\textsuperscript{InvMILP} with $\omega_2 = [1, 1, ..., 1, 100]$. By executing the sequential algorithm, we can get the general speedup of the PCAIg\textsuperscript{InvMILP}, which is also an upper bound of the actual speedup.

6.2 Implementation

Parallel computing uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of algorithm simultaneously with the others. The processing elements can be diverse and be implemented in several networked computers. Since different processing elements may access memory location at the same time, how to deal with read and write conflict is one of the main problems in parallel computing.

6.2.1 Read and write conflict

Two incompatible operations (e.g., read and write) conflict if they both access the same data item. In a shared memory parallel architecture such as the parallel algorithm introduced in this thesis, more than one processing elements will read from or write to the same memory location simultaneously, which means that the possibility of conflict arises. In the field of database of computer science, there are four types of file operations conflict, (1) read-write conflict, also known as unrepeatable reads, is a computational anomaly associated with interleaved execution of transactions; (2) write-read conflict, also known as reading uncommitted data, is a computational anomaly associated with interleaved execution of transactions; (3) write-write conflict, also known as overwriting uncommitted data is a computational anomaly associated with interleaved execution of transactions. There is no read-read conflict, which means more than one processor can read from the same file simultaneously. Parallel algorithm are assumed to be in error if a read or write conflict ever rises and this is one of the most realistic problems in practice (28).
6.2.2 Read and write strategy

In the parallel cutting plane algorithm, CAlg\textsuperscript{InvMILP} and PAlg\textsuperscript{InvMILP} need to work simultaneously. The communication between two algorithms is the extreme point set $S^0$, which needs to be stored in memory and updated by both algorithms continuously. In order to implement this parallel algorithm, we need two computers to work on one hard disk. Moreover, a file that is being updated can not be read or updated by another algorithm at the same time. One algorithm needs to wait until the other finishes updating the file. To realize the communication between two algorithms, our strategy is to create and delete protective files. Creating a protective file sends a signal to others which says “It is being used”. Deleting the protective file sends a signal to others which says “It is available”. One algorithm creates a protective file in memory before it accesses the shared file, and delete the protective file after it finishes using the shared file. The other algorithm checks and detects the protective file created by others exist, then it waits until receiving the second signal and goes on using the shared file.

For example, if the main algorithm needs to update the file $S^0$, a protective file is created before updating, which sends the signal showing that the CAlg\textsuperscript{InvMILP} is going to update $S^0$. During the process of updating, if PAlg\textsuperscript{InvMILP} needs to read from or update $S^0$, it firstly checks and notices that the protective file created by CAlg\textsuperscript{InvMILP} exists. Then the PAlg\textsuperscript{InvMILP} needs to keep checking until it gets the signal which said that the main algorithm finishes updating the file $S^0$. In the CAlg\textsuperscript{InvMILP} part, as soon as the updating process ends, another signal is sent by deleting the protective file. At this time, the parallel part will get the signal and go on reading or updating $S^0$. In the other side, PAlg\textsuperscript{InvMILP} will also create a protective file while it is updating the file, and CAlg\textsuperscript{InvMILP} also needs to check whether a protective file created by PAlg\textsuperscript{InvMILP} exists.

Since both algorithms need to check and create protective files, there is a priority problem that should be considered. This problems happens in the following circumstance: when both algorithms need to access the $S^0$ file simultaneously, they will both check whether a protective file exists nearly at the same time. The answer will be no for both of them since they are both in the checking process, no protective file is created. Then wrongly believing that no one is
using the file, they will both create a protective file and go on accessing $S^0$ simultaneously. In order to present above circumstance happening, we need to set priority to both algorithms. In our examples, CAlg$^{\text{InvMILP}}$ is set to have the first priority to using the file $S^0$. The way to set priority is easy and smart. We know that when the algorithm runs into reading or updating the file, two steps should be done, checking whether a protective file exists and creating a new protective file. We let CAlg$^{\text{InvMILP}}$ create protective files before checking, and let PAlg$^{\text{InvMILP}}$ check protective files before creating. In this way, CAlg$^{\text{InvMILP}}$ is set to have first priority to access the file when the above conflict exists.

6.2.3 Matlab, Tomlab and Cplex

The PCAlg$^{\text{InvMILP}}$ is programmed in Matlab, Tomlab and Cplex. Matlab is the interface software we use. Tomlab and Cplex are called by Matlab as tools to take charge of the calculations of optimization problems.

Matlab stands for MATrix LABoratory, which is developed by The Mathworks, inc. It is one of the fastest and most enjoyable ways to solve problems numerically. It is particularly easy to generate results, draw graphs to find interesting features, and then explore the problem further. The version we used in our implementation is Martlab 7.6.0 (R2008a). Tomlab optimization environment is a powerful optimization platform and modeling language for solving applied optimization problems in Matlab. It is flexible, easy-to-use, robust and reliable for the solution of all types of applied optimization problems. The version that we use is Tomlab v6.1. Cplex has a strong mathematical optimization technology, which enables better decision-making for efficient resource utilization. It especially helps solve planning and scheduling problems in virtually every industry. The version that we use is Cplex 11.0.

“MipAssign” is the main function that we use to set up a mix integer linear problem: “Prob = mipAssign(...”). It’s then possible to solve the problem by using the Tomlab solver “mipSolve” with the call: Result = mipSolve(Prob).
6.3 Computational Experiments for PCAlg\textsuperscript{InvMILP} with Two Processors

We demonstrate the PCAlg\textsuperscript{InvMILP} with two processors using two groups of instances. The first group of instances is from mixed integer problem library (MIPLIB) \cite{29}. The second group of instances is randomly generated. For each instance in both groups, we use randomly generated objective functions and obtain up to 50 different optimal solutions. We then solve the inverse problem of all instances for all the optimal solutions using both parallel cutting plane algorithm and cutting plane algorithm for comparison. Number of iterations and time have been recorded as measurements of algorithm performance.

6.3.1 MIPLIB instances

Most instances from MIPLIB have special characteristics. They can be classified into set partition problems, set packing problems, knapsack problems, mixed binary problems, etc. The computational performances of PCAlg\textsuperscript{InvMILP} on these instances are shown in Table 6.1.

<table>
<thead>
<tr>
<th>instance</th>
<th>PCAlg\textsuperscript{InvMILP}</th>
<th>CAAlg\textsuperscript{InvMILP}</th>
<th>improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># iter.</td>
<td># sec.</td>
<td># iter.</td>
</tr>
<tr>
<td>egout</td>
<td>20.5</td>
<td>7.3</td>
<td>27.9</td>
</tr>
<tr>
<td>gt2</td>
<td>65.7</td>
<td>40.7</td>
<td>96.2</td>
</tr>
<tr>
<td>mod008</td>
<td>15.3</td>
<td>11.0</td>
<td>18.0</td>
</tr>
<tr>
<td>Iseu</td>
<td>49.5</td>
<td>19.7</td>
<td>62.3</td>
</tr>
<tr>
<td>p0201</td>
<td>55.9</td>
<td>34.0</td>
<td>70.0</td>
</tr>
<tr>
<td>misc03</td>
<td>1.0</td>
<td>0.6</td>
<td>1.0</td>
</tr>
</tbody>
</table>

From the experiment results we can see that generally speaking, the PCAlg\textsuperscript{InvMILP} has better performances than CAAlg\textsuperscript{InvMILP}. Smaller instances have relatively smaller number of iterations and seconds. That is why the improvement of instance “mod008” is not as significant as other larger instances. For the instance “misc03”, there is no improvement since it only takes one iteration to terminate. The time spent on communication between computers causes the negative improvement of time.
6.3.2 Random instances

Since instances from MIPLIB may have special characteristics, we randomly generated the second group of instances to get more general results. We randomly generate the parameters $A$ and $b$ of MILP (3.1) of different sizes. The number in the name of the instance stands for the size of $A$ and $b$, e.g., matrix $A$ for “instance20” is 20 by 20 and $b$ for “instance20” is 20 by 1. For both $\text{PCAlg}^{\text{InvMILP}}$ and $\text{CAlg}^{\text{InvMILP}}$, number of iterations and time have also been recorded for comparison.

Table 6.2 Computational performances of algorithms on random instances

<table>
<thead>
<tr>
<th>instance</th>
<th>$\text{PCAlg}^{\text{InvMILP}}$</th>
<th>$\text{CAlg}^{\text{InvMILP}}$</th>
<th>improvement $%$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># iter.</td>
<td># sec.</td>
<td># iter.</td>
</tr>
<tr>
<td>instance15</td>
<td>22.9</td>
<td>8.4</td>
<td>28.6</td>
</tr>
<tr>
<td>instance20</td>
<td>39.1</td>
<td>16.6</td>
<td>50.8</td>
</tr>
<tr>
<td>instance30</td>
<td>102.5</td>
<td>76.7</td>
<td>123.8</td>
</tr>
<tr>
<td>instance50</td>
<td>126.6</td>
<td>487.6</td>
<td>161.8</td>
</tr>
</tbody>
</table>

From Table 6.2, the experiment results on random instances also show that the $\text{PCAlg}^{\text{InvMILP}}$ has better performances than $\text{CAlg}^{\text{InvMILP}}$. Although improvements are more significant for first two smaller instances than last two, but the percentages have no absolutely increasing or decreasing tendency with the size of the problems. For general instances, the improvement of $\text{PCAlg}^{\text{InvMILP}}$ compared with $\text{CAlg}^{\text{InvMILP}}$ is not related to the size of the problem.

6.4 Computational Experiments for $\text{PCAlg}^{\text{InvMILP}}$ with Three Processors

The $\text{PCAlg}^{\text{InvMILP}}$ with three or more processors is introduced in Chapter 5. Three strategies are presented on how to implement the algorithm with more processors. In this section, computational experiments is conducted with “instance20” for all three strategies. The number of iterations and time are recorded to show the improvement and limitation of $\text{PCAlg}^{\text{InvMILP}}$ with three processors.

From the computational results shown in Table 6.3, by comparing with results of $\text{CAlg}^{\text{InvMILP}}$ and $\text{PCAlg}^{\text{InvMILP}}$ with two processors, we can see the numbers of iterations for all three strate-
Table 6.3  Computational performances of instance20 using three processors

<table>
<thead>
<tr>
<th># processors</th>
<th>instance20</th>
<th># iter.</th>
<th># sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>CAlg$^{\text{InvMILP}}$</td>
<td>50.8</td>
<td>22.5</td>
</tr>
<tr>
<td>Two</td>
<td>PCAlg$^{\text{InvMILP}}$</td>
<td>39.1</td>
<td>16.6</td>
</tr>
<tr>
<td>Three: strategy I $\theta=0.2$ and 0.8</td>
<td>36.9</td>
<td>26.7</td>
<td></td>
</tr>
<tr>
<td>Three: strategy I $\theta=0.8$ and 0.8</td>
<td>37.5</td>
<td>23.5</td>
<td></td>
</tr>
<tr>
<td>Three: strategy I $\theta=0.8$ and 0.9</td>
<td>35.6</td>
<td>21.7</td>
<td></td>
</tr>
<tr>
<td>Three: strategy II $\theta=0.5$ and 0.5-1</td>
<td>38.9</td>
<td>24.3</td>
<td></td>
</tr>
<tr>
<td>Three: strategy III different $\omega$s I</td>
<td>42.9</td>
<td>36.0</td>
<td></td>
</tr>
<tr>
<td>Three: strategy III different $\omega$s II</td>
<td>36.8</td>
<td>30.2</td>
<td></td>
</tr>
</tbody>
</table>

gies are reduced by using three computers, but the time needed increases. From the number of iterations view, using more computers indeed reduces the iterations of the algorithm, which means the algorithm has been improved. However, the time that the algorithm needs increases. The reason why this happens is that a large amount of time is spent on waiting for the release of the communication file $S^0$. The greater of number of computers in use, the greater of waiting time. For a small instance like instance20 we use, this waiting time covers a significant part of the total time. The relationship between the number of processors and the efficiency of the algorithm is not linear. However, With the development of high speed computers and parallel computing skills, the communication between processors should be more efficiency and the waiting time should be reduced.

6.5 Computational Experiments for Potential Improvements of PCAlg$^{\text{InvMILP}}$

We conduct experiments using both MIPLIB and random instances to get potential improvement of PCAlg$^{\text{InvMILP}}$. The numbers of iterations in sequential algorithm for PCAlg$^{\text{InvMILP}}$ with two and three processors are regarded as the highest efficiency that PCAlg$^{\text{InvMILP}}$ could get. They are used to calculate the potential improvement and general speedup.

Table 6.4 shows the computational results for potential improvement of PCAlg$^{\text{InvMILP}}$ with two processors. The potential number of iterations are shown in the third column, which is the
highest possible efficiency for the instances that our algorithm could get. The corresponding improvement percentages and speedup are shown in the forth and fifth columns.

Table 6.4 Potential improvements and speedup for PCAlg\textsuperscript{InvMILP} with two processors

<table>
<thead>
<tr>
<th>instance</th>
<th>CA_{Alg}^{InvMILP} # iter.</th>
<th>Sequential #iter.</th>
<th>Potential improvement</th>
<th>Speedup General</th>
<th>Ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>egout</td>
<td>27.9</td>
<td>15.4</td>
<td>44.8%</td>
<td>1.8</td>
<td>2</td>
</tr>
<tr>
<td>mod008</td>
<td>18.0</td>
<td>13.0</td>
<td>27.8%</td>
<td>1.4</td>
<td>2</td>
</tr>
<tr>
<td>Iseu</td>
<td>62.3</td>
<td>32.3</td>
<td>48.3%</td>
<td>1.9</td>
<td>2</td>
</tr>
<tr>
<td>p0201</td>
<td>70.0</td>
<td>34.1</td>
<td>51.3%</td>
<td>2.0</td>
<td>2</td>
</tr>
<tr>
<td>instance15</td>
<td>28.6</td>
<td>16.7</td>
<td>41.4%</td>
<td>1.7</td>
<td>2</td>
</tr>
<tr>
<td>instance20</td>
<td>50.8</td>
<td>34.0</td>
<td>32.9%</td>
<td>1.5</td>
<td>2</td>
</tr>
<tr>
<td>instance30</td>
<td>123.8</td>
<td>70.9</td>
<td>42.7%</td>
<td>1.7</td>
<td>2</td>
</tr>
<tr>
<td>average</td>
<td>54.5</td>
<td>30.9</td>
<td>41.3%</td>
<td>1.7</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.5 shows the computational results for potential improvement of PCAlg\textsuperscript{InvMILP} with three processors.

Table 6.5 Potential improvements and speedup for PCAlg\textsuperscript{InvMILP} with three processors

<table>
<thead>
<tr>
<th>instance</th>
<th>CA_{Alg}^{InvMILP} # iter.</th>
<th>Sequential #iter.</th>
<th>Potential improvement</th>
<th>Speedup General</th>
<th>Ideal</th>
</tr>
</thead>
<tbody>
<tr>
<td>egout</td>
<td>27.9</td>
<td>9.7</td>
<td>65.1%</td>
<td>2.9</td>
<td>3</td>
</tr>
<tr>
<td>mod008</td>
<td>18.0</td>
<td>9.0</td>
<td>49.7%</td>
<td>2.0</td>
<td>3</td>
</tr>
<tr>
<td>Iseu</td>
<td>62.3</td>
<td>23.8</td>
<td>61.8%</td>
<td>2.6</td>
<td>3</td>
</tr>
<tr>
<td>p0201</td>
<td>70.0</td>
<td>21.9</td>
<td>68.7%</td>
<td>3.2</td>
<td>3</td>
</tr>
<tr>
<td>instance15</td>
<td>28.6</td>
<td>12.6</td>
<td>57.1%</td>
<td>2.3</td>
<td>3</td>
</tr>
<tr>
<td>instance20</td>
<td>50.8</td>
<td>21.7</td>
<td>57.2%</td>
<td>2.3</td>
<td>3</td>
</tr>
<tr>
<td>instance30</td>
<td>123.8</td>
<td>46.0</td>
<td>62.9%</td>
<td>2.7</td>
<td>3</td>
</tr>
<tr>
<td>average</td>
<td>54.5</td>
<td>20.6</td>
<td>60.4%</td>
<td>2.6</td>
<td>3</td>
</tr>
</tbody>
</table>

In Table 6.5, a surprising result is shown in the forth instance “p0201”. The general speedup 3.2 is higher than ideal speedup 3. This means PCAlg\textsuperscript{InvMILP} with three processors may have a speedup that is greater than ideal speedup. The reason for this phenomenon is that parallel computing can search different regions in extreme point set at the same time, while algorithm
with one processor can only search one region at one time. Since the solution density in different regions of extreme point set is non-uniform, algorithm with one processor may search a low density region. On the contrary, parallel algorithm can search multiple regions at the same time, ensuring that at least some processors are searching the high density regions [30].

Figure 6.2 and Figure 6.3 give us a clear comparison of potential and actual improvements of the parallel cutting plane algorithm. In both figures, horizontal axis $p$ stands for the number of processors in use.

![Graph showing potential and actual improvements for PCA\text{lg}^{\text{InvMILP}}](image)

**Figure 6.2** Potential and actual improvements for PCA\text{lg}^{\text{InvMILP}}

In Figure 6.2, the ideal line (highest line) shows the ideal improvement and speedup that can be obtained by using parallel computing techniques. The general line (middle line) shows the average result of the sequential algorithms, which means the possible maximum efficiency improvement of PCA\text{lg}^{\text{InvMILP}} compared with CA\text{lg}^{\text{InvMILP}} without considering the time wasting in communications between processors. This gives us the potential improvements of
Figure 6.3 Potential and actual speedup for \( \text{PCAlg}^{\text{InvMILP}} \)

The actual line (lowest line) shows the actual computational improvement results of \( \text{PCAlg}^{\text{InvMILP}} \) in our experiments. Since the communication skills between processors are time consuming in our experiments, the actual improvements are much lower than the potential improvements.

We can see the actual improvement is much smaller than potential improvement that \( \text{PCAlg}^{\text{InvMILP}} \) can get, and the potential improvement of \( \text{PCAlg}^{\text{InvMILP}} \) is smaller than the ideal improvement that generally parallel computing can get. These are the two ways that we can improve our algorithm. Thus, we have two aspects of future work. In the first aspect, we need to improve the parallel communication skills in order to reduce the wasting time in our implementation. In this way, we try to make the actual improvement approaching the potential improvement, which is shown as the actual line approaching general line in the figures. In the other aspect, we need to improve the \( \text{PCAlg}^{\text{InvMILP}} \) in order to increase the
potential improvement. In this way, we try to make the general line approaching the ideal line.
CHAPTER 7. CONCLUSION AND FUTURE WORK

7.1 Conclusion

In my thesis, we extend the CAlg$^{\text{InvMILP}}$ to the PCAlg$^{\text{InvMILP}}$ with two and more processors. A three-step parallel algorithm is presented. With the help of parallel computing techniques, we can execute the parallel algorithm simultaneously with the original cutting plane algorithm and generate several benefits. Firstly, it can continuously provide extreme points to the main algorithm in order to improve the algorithm efficiency, i.e., reduce time and number of iterations that algorithm needs; secondly, it generates feasible solution to the inverse problem before the algorithm terminates; thirdly, the new algorithm can be executed with multiple processors, which makes continuous improving of the algorithm to be possible. Moreover, sensitivity analysis is conducted on an important parameter $\theta$. The result shows that this parameter has an obvious influence on the efficiency of the algorithm.

We present several improving strategies for the PCAlg$^{\text{InvMILP}}$ with both two and three processors. Potential improvement of our new algorithm is also discussed. We implement our parallel cutting plane algorithm with mathematical softwares and realize the parallel computing with a network of computers that share the same memory. The read and write conflict is solved by creating protecting files. Then We demonstrate the parallel cutting plane algorithms using two groups of instances. The experiment results indicate that there is a significant improvement of the PCAlg$^{\text{InvMILP}}$ compared with the CAlg$^{\text{InvMILP}}$.

In a conclusion, the contribution of this research can be summarized as follows:

- Incorporated parallel computing into the inverse algorithm to solve the InvMILPs;
- Presented the PCAlg$^{\text{InvMILP}}$ with two processors which overcomes the limitation of ex-
isting algorithm;

- Extended the PCAlg^InvMILP to be executed with multiple processors;
- Presented the potential improvement of the PCAlg^InvMILP.

7.2 Future Work

7.2.1 Further improvement of the algorithm

The original idea of the PCAlg^InvMILP is to improve the efficiency of existing algorithm. However, since two processors are used, we expect that the theoretic optimal improvement is 50%, which does not realize from the experiment results. This means we still get a large improvement space. We will conduct more research on further improvement of our algorithm. In inverse optimization area, we still need to find more efficient algorithms.

7.2.2 Further improvement of parallel computing techniques

Parallel computing techniques have high development recent years. Due to the shortage of related knowledge of this area, the communication strategy between processors in our implementation is inefficient. Thus, one of the future work is to incorporate more efficient parallel computing techniques into our algorithms.

7.2.3 Writing an open source software

As far as we know, there is no software available for solving inverse optimization problems. We can write our algorithm into an open source software and publish it on internet for researchers to solve related problems.
BIBLIOGRAPHY


