Parallel Multi-Objective Branch and Bound

Wei Zhang

Kongens Lyngby 2008
IMM-M.Sc-2008-39

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-PHD: ISSN 0909-3192

Abstract

This thesis presents sequential and parallel implementation of MOBB(Multi-Objective-Branch-and-Bound) algorithm, a novel algorithm that solves MOMIP(Multi-Objective-Mixed-Integer-Programming) problem. While SOMIP(Single-Objective-Mixed-Integer-Programming) is notoriously hard to solve, MOMIP is even harder. No efficient algorithms to solve MOMIP has been known before.

MOBB is based on tradtional SOBB(Single-Objective-Branch-and-Bound) algorithm, with several major modifications of bounding feasible region and of how to prune Branch-and-Bound tree. Parallelization is implemented with the hope of speeding up calculation as much as possible. The combination of a novel algorithm and its parallelization is the highlight of this thesis project.

All classes, methods, data structures and algorithms in this thesis are implemented in C++, while certain data analysis and visualization are implemented in Matlab and Python. The result of this project shows that our MOBB algorithm is much better than the traditional brute and force algorithm. Further, our parallelization of MOBB algorithm has achieved super linear speedup, which is the best possible result of any parallelization program.

The investigations of MOBB algorithm and its parallelization has susccessfully proven that MOBB can very efficiently solve certain MOMIP problems, which could possibly lead to a major breaththrough in the Operations Research area.

Acknowledgements

First, I would like to thank Prof. Thomas Stidsen, my co-supervisor, for bringing me into the wonderful OR world so that I can easily write some programms that run out of memory. More importantly, I would like to thank him for proposing such a novel and powerful algorithm, all the test benches, and all his patience and guidance throughout this project.

Second, I would like to thank Prof. Bernd Dammann, my main supervisor, for bringing me into the wonderful HPC world so that I can enjoy the thrill of orchestrating many CPUs. Also, Bernd's expertise in HPC is second to none, while I work very hard to make the program twice faster, a compiler option added by him can easily speedup the program 5 times. Bernd's exhaustive assistance has made this thesis project a particularly enjoyable and eye-opening experience for me.

Third, I would like to thank Morten Madsen for several insightful conversations. Being one of the very few people who understand this algorithm, Morten has been a tremendous help to me.

Last but not the least, I have to thank my parents for making me possible, every single bit of me.

Contents

Abstract **i**

Acknowledgements **iii**

1 Introduction **1**

1.1 Thesis outline 2

2 Single-Objective Branch and Bound **5**

2.1 Formal description of Linear Programming and Integer Programming 5

2.2 What CLP solver has to offer 7

2.3 Description of B & B 8

3 Multi-Objective Branch and Bound **15**

3.1 Formal description of MOBB 16

3.2 Some term definitions in MOBB 16

3.3 One-Sliced MOBB 20

3.4 Multi-Sliced MOBB 24

4 Implementation of Sequential MOBB 27

4.1 Data structure needed 27

4.2 Algorithm flow chart 32

4.3 Test and analysis of sequential MOBB 38

5 Test and analysis of sequential MOBB 43

6 Parallel Computing 45

6.1 Amdahl's law 45

6.2 Category of parallel computing 46

6.3 Tools for parallel computing 46

7 Coarse Parallel MOBB 53

7.1 Design and Implementation 53

7.2 Test and Analysis 59

8 Fine parallel MOBB 63

8.1 Design and Implementation 63

8.2 Test and Analysis 72

9 Software architecture of this project 81

9.1 Collaboration between each class 81

9.2 Object-Oriented Design [15] 83

9.3 Test and Parallel simulation 84

10 Conclusion and future work **87**

10.1 Conclusion 87

10.2 Future work 88

Chapter 1

Introduction

Operations Research(OR) [1] is a branch of applied mathematics used to support decision making. It has gained its popularity ever since world war II. The paramount aim of OR is to find the maxima or minima of one objective function given a certaint complicated constraints. It uses methods of mathematical modelling, statistics, and algorithms to achieve such a goal. It is used everywhere in modern engineering, such like economy, production plan, logistics, medical engineering, computer chip designing, chemical engineering. In other words, OR is the core of all management sciences.

Linear programming(LP) [1] is a very useful tool in OR. It mainly deals with the linear objective function and linear constraints. Serveral powerful algorithms have been proposed to solve LP. These algorithms are very much efficient, usually they can solve LP problems very quickly.

While there are certain very effective ways of solving linear programming, one simple extra requirement could make linear programming extremely hard – that requirement is to make all variables be integers. This special case is called IP (Integer Programming) [2]. IP problems are notoriously hard to solve, due to the fact that the solution space is exponentially growing with the number of variables, which is categorized as an NP-hard problem. MIP(Mixed Integer Programming) and BIP(Binary Integer Programming) are two forms of IP: MIP requires the integer variables to be arbitrary integers, while BIP requires the

integer variables to be solely binary integers.

Through the last 50 years, many approaches have been proposed to better solve IP. Among them, B & B (Branch and Bound) [2] is a generic algorithm that could greatly reduce the solution space. Traditional B & B deals with single objective. MOMIP(Multi-Objective-Mixed-Integer-Programming) [4] deals with multi-objective. There are currently no efficient algorithms to deal with MOMIP. Therefore, it is a very hard problem. In this thesis, we propose a novel algorithm to efficiently solve MOMIP based on a modified version of branch and bound algorithm , i.e. MOBB (Multi-Objective Branch and Bound) .

Parallel computing [3] is a form of computing in which many instructions are carried out simultaneously. It attempts to use as many CPUs available as possible at one time, therefore calculation can be achieved much faster. Parallel computing has been the primary programming paradigm in the HPC (High Performance Computing) area, where intensive computation is required. Once to be only primarily used on super-computers, with the advent of multi-core processers' popularlity nowadays, parallel computing has been more and more important in the programming area. While there are no methods to convert an NP [5] problem to a P [5]problem, parallel computing is by now the most effective method to solve NP-hard problems. In this thesis, we propose two algorithms to parallelize our novel MOBB algorithm.

1.1 Thesis outline

Chapter2 reviews the B & B algorithm. It presents the fundamental data structures and algorithm flowcharts of B & B algorithm, also it briefly introduces CLP solver, which is the simplex solver we used in this thesis project.

Chapter3 proposes our MOBB algorithm. It compares MOBB with SOBB and illustrates several key points of MOBB. It explains how to modify a SOBB algorithm to construct a MOBB algorithm in great details.

Chapter4 describes the implementation and analysis of sequential MOBB algorithm. It also shows the result of MOBB algorithm is much better than the traditional brute and force method.

Chapter5 reviews the parallel programming. It explains classifications of paral-

lel pogramming and gives introduction to the parallelization tools that we used in this thesis project.

Chapter6 presents the coarse-grained parallelization algorithm for MOBB. Coarse-grained parallelization is a straightforward implementation of parallelization of MOBB algorithm and shows it gives a medicore speedup.

Chapter7 presents the fine-grained parallelization algorithm for MOBB. It gives a thorough explaination of how a fine-grained algorithm of parallel MOBB is constructed. And it shows a very good speedup.

Chapter8 presents the software architecture of the whole project. It explains how Object Oriented technology is adopted in this project and why they are useful.

Chpater9 presents the conclusion and future work of this thesis project.

Chapter 2

Single-Objective Branch and Bound

In this chapter, formal descriptions of linear programming and single-objective branch and bound is presented as they are the foundations of later MOBB work and further parallelization; also, the B & B algorithm is presented in greater details in the form of algorithm flow charts.

2.1 Formal description of Linear Programming and Integer Programming

Linear programming consists of two major components: (1) an objective function [1], which is a linear combination of variables and (2) a set of constraints [1] imposed on variables.
The goal for linear programming is to minimize or maximize the objective function while maintaining all the variables inside the feasible region [1], which is defined by the constraints. A formal description of one linear programming is given below:
Maximize

$$Z = cx \tag{2.1}$$

Subject to

$$Ax \leqslant b \tag{2.2}$$

And

$$x \geqslant 0 \tag{2.3}$$

Where, c is a row vector of n real numbers, x is a column vector of n continuous number(a.k.a n variables), b is a column vector of m real numbers(a.k.a m constraints), A is an $m * n$ matrix of constraints' parameters. The maximization value of the objective function is called the optimal objective. When an optimal objective is achieved, the values of all variables are called the optimal solutions. A feasible region[1] refers to the region in which all variables are sastisfied with the constraints. The variables confined in the feasible region are called feasible solutions. All the optimal solutions must be feasible solutions.

A number of algorithms have been developed to solve such problems, among those algorithms, the simplex method [1] is the most widely used.

In the formal description of linear programming, the objective function is a maximizatioin function. Duality theory [1] ensures for each maximization problem, one can always find a dual minimization problem which once solved can generate the same solutions for the counterpart maximization problem. The equivalent form of minimization problem is given below:

Minimize

$$W = yb \tag{2.4}$$

Subject to

$$yA \geqslant c \tag{2.5}$$

And

$$y \geqslant 0 \tag{2.6}$$

Where, c is the same row vector as in maximization problem, y is a row vector of m continuous numbers(a.k.a m variables), b is the same column vector as in maximization problem, A is the same $m * n$ matrix as in maximization problem.Since these two forms are equivalent to each other, one can always convert one LP problem from maximization form to minimization form, or vice versa.

Based on LP, the IP problem is to add one more constraint, which is to require some of the continous variables to be integers. The reason behind why IP is much harder than LP, is that the simplex method traverses corner points defined by the constraints set by pivoting basic variables in and out. Such a process can be done by matrix factorization [16], which is very fast on modern computer systems. However, when the integer constraint is added, such methods may still apply but the result might not be in the feasible region anymore.

To ensure a solution is both optimal and feasible requires enumeration of all the possible solutions, thus the exploration of solution space gets enormously big. Duality theory does not hold in IP problem.The focus of this thesis is minimization problem. For maximization problem, we could negate the objective coefficients and adopt several transformation and scaling techniques to convert it to a minimization problem.

2.2 What CLP solver has to offer

After giving the formal description of LP and IP, we should turn to the tool at our disposal to examine what it can offer. This thesis project is based on COIN-OR project. COIN-OR (COmputational INfrastructure for Operations Research) [6] is an initiative to spur the development of open-source software for the operations research community. COIN-OR provides a handful of open source projects for OR algorithms. More specifically, this thesis uses CLP project in COIN-OR as the backbone solver. Clp (Coin-or linear programming) is an open-source linear programming solver written in C++. It is primarily meant to be used as a callable library. Certain methods provided in CLP are of great interests for this thesis, as they provide the gateway to solving SOBB/MOBB. A brief review of these methods are given below. This review serves as a know-how primal, so that the rest of thesis can solely focus on the algorithm aspect without excessive considerations to the technicalities.

The input to the CLP solver is usually in the mps file format. MPS (Mathematical Programming System) is a file format for presenting and archiving linear programming (LP) and mixed integer programming problems. MPS is column-oriented (as opposed to entering the model as equations), and all model components (variables, rows, etc.) receive names. As in the LP format, columns refer to variables and rows refer to constraints. The below is a list of methods that are used throughout this project.

- void readMps(): It reads the .mps [17] file(mps, Mathematical Programming System format, a special file format that is used to faciliate several mathematical programming, which includes linear programming) and initiates the solver.

- int getNumCols(): It returns the number of columns of constraint matrix, i.e., the number of variables.

- int getNumRows(): It returns the number of row of constraint matrix, i.e., the number of constraints.
- string getColName(int index): It returns the column name of given index.
- string getRowName(int index) : It returns the row name of given index.
- addRow(int rowNumber, int* columnIndices, double* element, double lowerBound, double upperBound): It adds one row(i.e a constraint), rowNumber is how many variables are affected in this constraint, columnIndices is the column indices of the affected variables, element is the constraint coefficients of the affected variables, lowerBound is the lower bound of this constraint, upperBound is the upper bound of this constraint.
- removeRow(int rowIndex): It removes the row of the given row index, i.e., remove one constraint.
- isBinary(int columnIndex): It returns true if the variable of the specified columnIndex is a binary variable, false otherwise.
- setContinuous(int columnIndex): It sets the variable of the specified columnIndex to be continuous.
- setColBounds(int columnIndex, double lowerBound, double upperBound): It sets the upper bound and lower bound of the variable of the specified columnIndex.
- initialSolve(), resolve(): They both serve as the simplex solving methods, initialSolve is used when pivoting the LP system for the first time, resolve() is used afterwards as it keeps track some information from the previous initialSolve() or resolve() to speed up calcuation.
- double* getColSolution(): It returns the values of each variable.
- void setObjSense(double direction): It sets the direction of optimization, i.e. whether it is a minimization problem(when direction is 1.0) or a maximization problem(when direction is 0.0). Since the focus of this thesis is minimization, we always set direction as 1.0.
- bool isProvenPrimalInfeasible(): It returns whether the current solution is feasible or not, returns true if infeasible, returns false otherwise.

2.3 Description of B & B

After giving the formal description of IP and the tool to solve LP, now we can see how to solve an IP problem using the available tool.

To solve IP problems, Branch and Bound is by far the most widely used framework. The core idea is that instead of brute-force searching for all the possible solutions, the algorithm starts from one node(see subsection 2.3.1.1) where no binary variable is fixed, then the algorithm does a linear relaxation [2](linear relaxation means to solve this problem as an LP problem regardless of the integer constraints), the result of this linear relaxation provides the lower bound of this IP problem, lower bound means no subsequent solutions can obtain a better objective than it– a relaxation normally increases the feasible solution of original problem, thus might contain a better objective, which was not feasible in the original problem. Then the algorithm branches on the most fractional variable in the solution of last linear relaxation (i.e. the variable with a solution closest to 0.5). For each branch, the lower bound of its IP problem is the result from a linear relaxation. Bounding (or fathoming) refers to the condition under which the subtree of a given node could be discarded, there are three such condtions: (1) the relaxation value of any branch is bigger than the incumbent best feasible objective value so that no better objective values can be obtained from this subtree; (2) the solution to this node's relaxation is infeasible so that when fixing more binary variables, the solution cannot be feasible as the feasible region only gets smaller when more constraints are imposed; (3) If a linear relaxation happens to generate an intger solution(i.e. all the integer variables are fixed with an integer in the solutions), so that the optimal objective is the lower bound of the subtree then there is no need for further branching. Algorithm 1 is a general description of B & B. To better illustrate B & B algorithm, the next two subsections are presented to elaborate on the data structures and algorithm flows involved.

2.3.1 Data structure needed

To faciliate B & B algorithm, several data structures are needed. As coded in C++, each data structure is represented by an according class.

2.3.1.1 BBNode

The main purpose of the class BBNode is to keep track of which binary variables are fixed to faciliate linear relaxation in B & B algorithm. Therefore, in BBNode class, two vectors are needed, one vector records the binary variable in-

Algorithm 1 Branch and Bound

```
1: while tree(see subsection 2.3.1.2) is not empty do
2:   choose a node (see subsection 2.3.1.1) in tree
3:   reset CLP solver's bounds and fix binary variable according to this node
4:   do a linear relaxation
5:   if solution is feasible then
6:     if relaxed objective is better than the incumbent then
7:       if solution is an integer solution then
8:         update incumbent
9:       else
10:        branch on this node
11:      end if
12:    end if
13:  end if
14: end while
```

dices, to which have been assigned a fixed binary value; the other vector records the values of these fixed binary variables. These two vectors always have the same size. The minimum size of each vector is 0, which is the starting point of the whole program. The maximum size of each vector is the number of binary variables, which means that all binary variables are fixed, and such a node is called leaf node.

2.3.1.2 BBTree

The main purpose of BBTree is to contain the BBNodes, so that the algorithm can better access the BBNodes. Currently, a vector of type BBNode is used as the backbone of this class. Six methods are offered: push()/pop(), which adds/returns one node; isEmpty(), which returns if the tree still has any node; size(), which returns the size of the tree; split(), which splits a tree into two trees of equal size; and chooseNextNode(), which chooses the next node to solve, there are several strategies to choose next node to solve: choose the one with smallest relaxed objective value, or with the largest relaxed objective value, or in the order they are inserted – based on previous experiments, the one with choosing the smallest relaxed objective value has the best performance.
A typical B & B tree [7] looks like below:

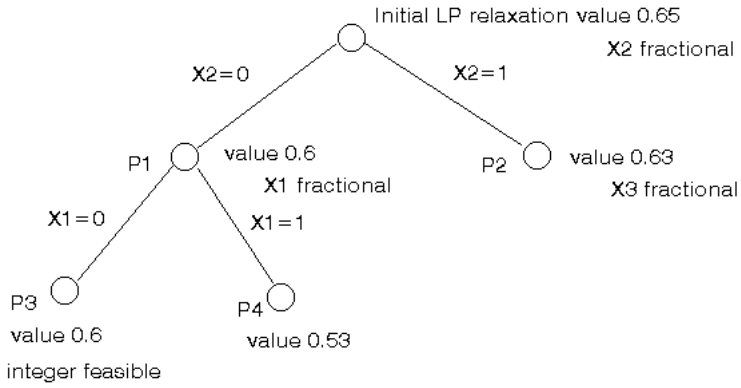

Figure 2.1: A typcial B & B tree

2.3.2 Algorithm flow chart

After designing all the required data structures, one needs to assemble them together to solve an IP problem. In this section, a big picture of how the whole program works is presented first; then, a detailed explanation of how to solve each node in the tree is presented.

2.3.2.1 Big picture of the sequential B & B algorithm

First, a flow chart of single objective B & B is presented in Figure 2.2

Explainations of the flow chart:

- Step(A) Initiate CLP solver, four tasks need be done: (1) read the input .mps file;(2) look up the binary variable indices, store them, set each binary variable to continuous(the reason to set the binary variable to continuous is that we need to treat all variables as continuous first to implement our own B & B algorithm, otherwise, CLP solver will use its own method to solve this IP problem), set lower bound to 0 and upper bound to 1 for each binary variable; (3) Construct one empty Node, which has no binary fixing, and push it into a tree; (4) set the current best objective value to be a very large number.

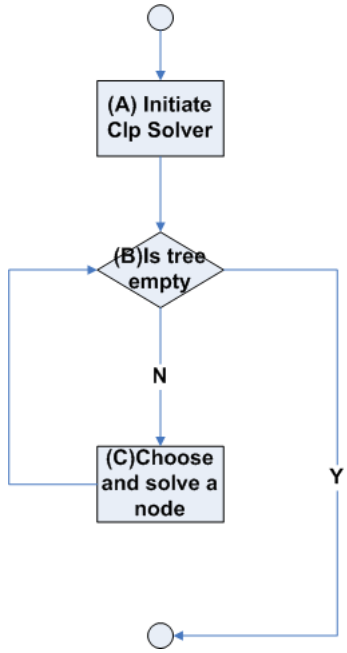

Figure 2.2: Big picture of sequential B & B algo.

- Step(B) While the tree is not empty, choose a node and solve the node. As discussed above, the node selection strategy is minimum-objetive-first strategy.
- Step(C) Choose a node and solve this node. For details, please refer to subsection 2.3.2.2.

2.3.2.2 Elaboration on Solving a Node

This part deals with solving one node in the B & B tree, it is the core part of the whole algorithm. Figure 2.3 presents such a process.

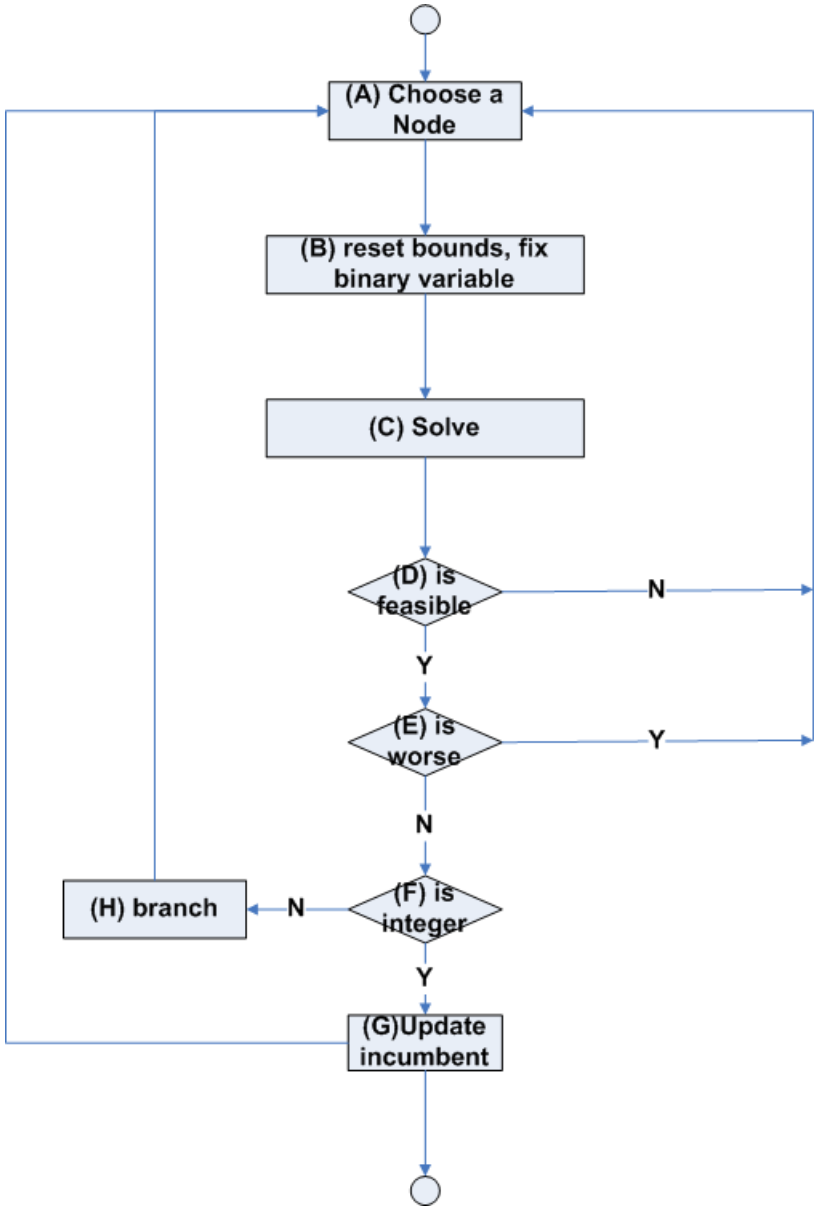

Figure 2.3: Solve a node in B & B algo.

Explainations of the flow chart:

- Step(A) Choose a Node, the selection strategy is smallest-objective-value first.
- Step(B) Reset binary variable bounds, and fix binary value according to the two vectors in Node.
- Step(C) Call CLP's simplex solver to solve the current configuration. First time call initialSolve(), later call resolve().
- Step(D) Check if the current configuration is feasible, call isProvenPrimalInfeasible().
- Step(E) Compare the linear-relaxed objective value with the current best objective value.
- Step(F) Check if all the binary variables are close to 1 or 0 enough. The gap precision here is set to 10^{-5}.
- Step(G) Update the current best objective value, i.e. if newly obtained objective is better than the incumbent, then set incumbent to the new one, otherwise keep the incumbent.
- Step(H) Choose a binary variable, whose current solution is closest to 0.5, branch on this node, i.e. set this variable's solution to 0 and 1 respectively while keeping all other solutions of the last solving process, put the two branched nodes back to the tree.

CLP solver itself provides a method that does branch and bound, however, we need to have much more flexibility of the program to construct later MOBB algorithm, so it is neccessary to have our own branch and bound implementation.

Chapter 3

Multi-Objective Branch and Bound

From this chapter, we start to discuss the core topic of this thesis – MOBB(Multi-Objective Branch and Bound). Compared to SOBB(Single-Objective Branch and Bound) , the distinctive feature of MOBB is that the objective is not only evaluated by one objective, but by many such values. More specifically, the focus of this thesis is two-objective branch and bound. In this chapter, we first give a formal description of MOBB, and then give definitions of several terms, finally we give a thorough explaination of our algorithm to solve MOBB problems.

3.1 Formal description of MOBB

This section gives the formal description of MOBB.

$$\begin{aligned} &\textit{Minimize} && \{z1, z2\} && (3.1)\\ &\textit{Subject to} && && (3.2)\\ &z1 && = && \sum_i c_i^1 x_i, \forall i, \textit{where } x_i \in B && (3.3)\\ &z2 && = && \sum_i c_i^2 x_i, \forall i && (3.4)\\ &AX && \geqslant && b && (3.5) \end{aligned}$$

Equation 10.6 defined the objective of MOBB, that is to minimize both z1 and z2. Easily seen, the objective will not only contain one point anymore (in most cases), but rather a curve of points, i.e. a Pareto front(please refer to subsection 3.2.2).

Equation 10.8 defines one dimension of the Pareto front, this constraint ensures that given a total binary fixing(i.e. all the binary variables are fixed), objective $z1$ is a fixed number, because $z1$ is only decided by binary variables .

Equation 10.9 defines the other dimension of the Pareto front, this dimension is decided by a mixture of both continuous and binary variables, so given a total binary fixing, z2 will not have a deterministic value.

Inequation 10.10 is the same constraint as in the ordiany LP.

3.2 Some term definitions in MOBB

3.2.1 Integer Point

An "integer point" is a point obtained when all the binary variables are fixed. $z1$ and $z2$ are each dimension of the point, which are defined in Equation 10.8 and Equation 10.9 respectively. For each pair of integer points (assume $ip1$, $ip2$), there are two kinds of relationship: $ip1$ dominates $ip2$, provided $ip1.z1 < ip2.z1$ and $ip1.z2 < ip2.z2$; $ip1$ draws $ip2$, provided $ip1$ cannot dominate $ip2$ and $ip2$ cannot dominate $ip1$, i.e. one of the dimensions of $ip1$ is better than that of $ip2$

but worse for the other dimension.

3.2.2 Pareto Front

A Pareto front is a curve that contains a series of Integer points, which are drawing each other. Figure 3.1 shows a Pareto front.

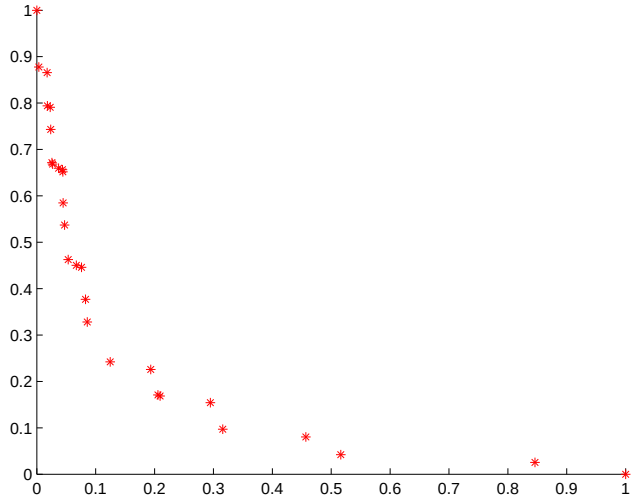

Figure 3.1: A Pareto front.

3.2.3 Semi-Nadir Point

A Semi-Nadir point is a point between two integer points which bears such property: assume two drawing integer point $ip1$, integer point $ip2$, and $ip1.z1 \geqslant ip2.z1$(thus $ip2.z2 \geqslant ip1.z2$); then, a Semi-Nadir point SP is such a point that $SP.z1 = ip1.z1$ and $SP.z2 = ip2.z2$. Figure 3.2 shows such a point.

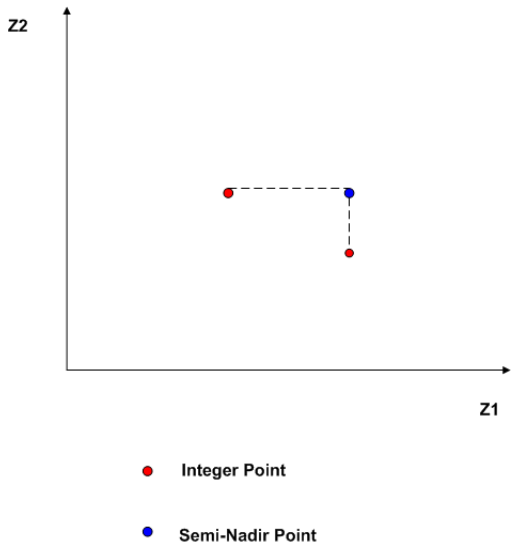

Figure 3.2: A Semi-Nadir Point

In other words, the property of a Semi-Nadir point is that any point in the region confined by two Semi-Nadir Points($snp1$, $snp2$) and the integer point(ip) between these two Semi-Nadir points will be dominated by ip, because all the points in such region will hold larger $z1$ and $z2$ than ip's. In Figure 3.3, all points in Region R1 will be dominated by the upleft integer point while all the points in Region R2 will be dominated by downright integer point.

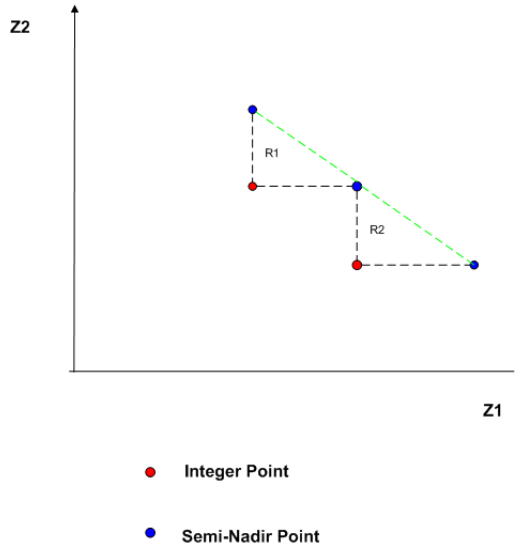

Figure 3.3: Two Integer points and Three Semi-Nadir Points with the regions dominated by the integer points

Obviously, any two integer points can decide one semi-Nadir point, therefore $N + 1$ adjacent integer points on one Pareto front can decide N semi-Nadir points. Figure 3.4 shows one Pareto front and its semi-Nadir point list(red points are the integer points and blue points are the Semi-Nadir points).

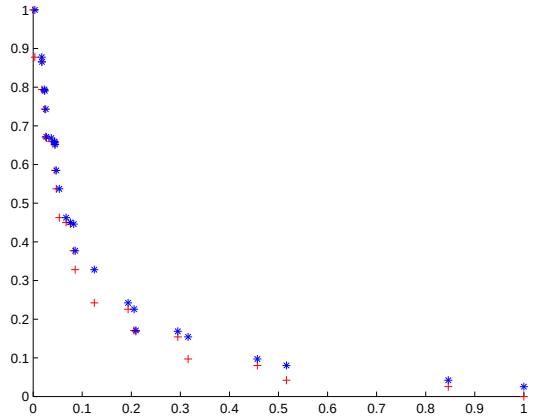

Figure 3.4: Pareto front and its Semi-Nadir point list, red points are integer points and blue points are Semi-Nadir points

3.3 One-Sliced MOBB

After introducing the basic definitions and description of MOBB, we can start to ponder how to solve such a problem with the given tool. This section deals with One-Sliced MOBB (traditional SOBB is also one-sliced, i.e, the feasible region is examined as a whole rather than being divided-and-conqured). The main topics of this section are two-fold: (1) how to modify the objective function to comply with CLP solver (2) how to branch and bound.

3.3.1 Objective function

As discussed before, the CLP solver can only solve the linear programming with one objective function. We can certainly brute force all the binary fixings and switch the objective function from $z1$ to $z2$ back and forth, but it will not be the best method. A better means would be to introduce an additional variable z and z is a linear combination of $z1$ and $z2$, and then try to find a Pareto front along with minimizing z. So the formal description of MOBB can be modified as below:

$$
\begin{aligned}
&\textit{Minimize} && z && (3.6)\\
&\textit{Subject to} && && (3.7)\\
&z && = && k_1 z_1 + k_2 z_2 && (3.8)\\
&z1 && = && \sum_i c_i^1 x_i, \forall i, \textit{where } x_i \in B && (3.9)\\
&z2 && = && \sum_i c_i^2 x_i, \forall i && (3.10)\\
&AX && \leqslant && b && (3.11)
\end{aligned}
$$

in Equation 3.8, k_1 and k_2 can be arbitrary numbers, usually we choose the perpendicular direction of the angle division line of the upper bound and lower bound of the feasible region(in one slice case, this region is the first Quadrant.) to set up k_1 and k_2. The graphic meaning of z is the distance from original point to the objective function line. Figure 3.5 shows the objective function's graphical representation in one slice. Theoretically, one should use the distance(as shown in the figure) as the objective function. However, in practice, the intersect on $z2$ axis can serve the same purpose, as it equals to $\frac{z}{sin(\alpha)}$, where α equals to $\beta - \frac{\pi}{2}$, (deducted from $\frac{\pi}{2} - (\pi - \beta)$), β is a constant degree whose tagent is the slope of objective function; therefore, $sin(\alpha)$is a constant number. In the one slice case, one can easily deduct that k_1 and k_2 are both 1.

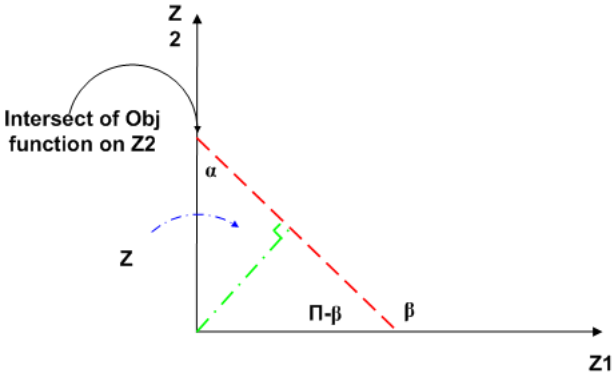

Figure 3.5: Objective function

3.3.2 Branch and bound

After the objective function is defined as in the traditional B & B algorithm, there are two major things that need to be considered: branch and bound. As in SOBB, bound refers to fathoming nodes while branch refers to branching on nodes once they cannot be fathomed. Also as in SOBB, there are three criterion to fathom: fathom by infeasibility, fathom by worse objective value, fathom by integer solution out of linear relaxation.

3.3.2.1 Fathom by infeasibility

Obviously, Equation 3.8 is the only extra constraint added compared to a normal linear programming, and this constraint cannot change the feasible region of either $z1$ or $z2$ or any other variables. So the feasiblity of MOBB is exactly the same as the SOBB, which can be tested by the CLP solver's isProvenPrimalInfeasible() method.

3.3.2.2 Fathom by worse than best incumbent

In the SOBB algorithm, the fathom scheme is done by comparing a linear relaxation objective to the current best feasible ojbective, if not better just fathom the whole branch. However, in MOBB, the fathom scheme will not apply anymore. In one slice, a linear relaxed solution has to be compared to the worst Semi-Nadir point, only if this solution is worse than the current worst Semi-Nadir Point that it can be fathomed. Figure 3.6 illustrates this. Except the worst Semi-Nadir point, any other Semi-Nadir points have the potential to jump into the feasible solution area, so do the points that are only slightly worse than them; therefore, only a point's z value worse than the worst Semi-Nadir point can it be fathomed. For instance, in Figure 3.6 point A is in the potential feasible region so it cannot be fathomed, point B can jump downwards to potential feasible region so it cannot be fathomed either, only point C is worse than the worst Semi-Nadir point so it can be fathomed.

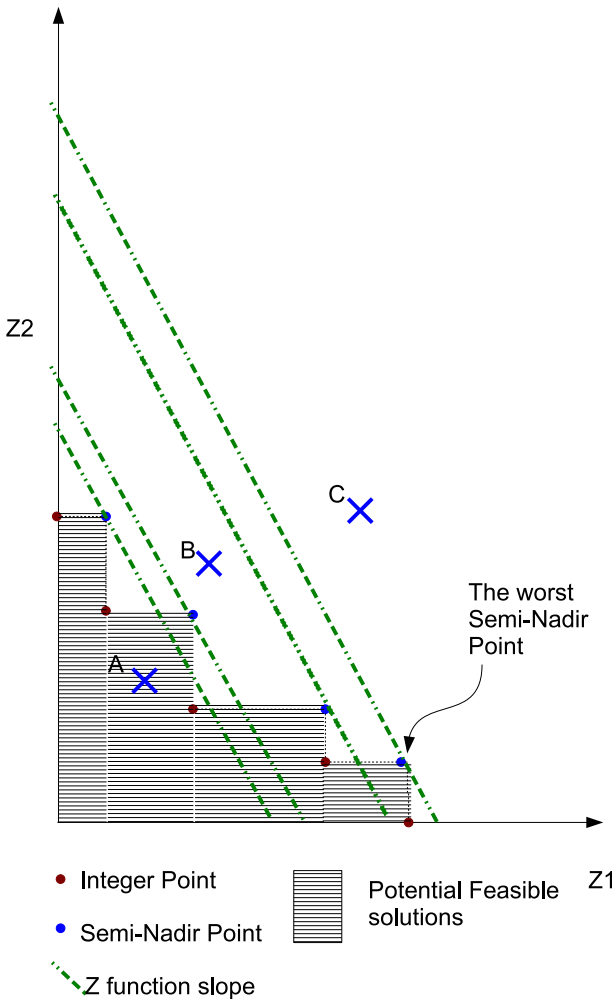

Figure 3.6: Fathom scheme

3.3.2.3 Fathom by Integer condition

In the SOBB algorithm, once a linear relaxation generates an integer solution, the branch below this node can be totally fathomed, because any other solution after this node in that branch will not possibly be better than this integer solution. But in MOBB, due to the same reason as in "Fathom by comparing to best incumbent", one cannot simply fathom all the possible solutions below this

node; instead, one has to continue to search from this node until either "fathom by infeasibility" or "fathom by worse than best incumbent" applies. However, one cannot branch on this solution, as it is an integer solution, and none of the binary fixings can be branched; further, it cannot generate a different solution if sending the previous fixing to CLP solver. Therefore, one constraint needs to be added in the CLP solver to escape this integer point. Such a constraint can be described as below:

$$\sum_i x_i + \sum_j (1 - x_j) \geqslant 1, \forall i, i \in S_0,\ \forall j, j \in S_1 \tag{3.12}$$

Where S_0 is the set in which all binary variables are fixed to 0, S_1 is the set in which all binary variables are fixed to 1.

It is only when all the binary fixings are exactly the same with the Integer solution that Inequation 3.12 will be violated, any other case will satisfy such a constraint. Therefore, this constraint skillfully escapes the last integer solution while still makes all the other solutions feasible.

3.3.2.4 Branching

If a node cannot be fathomed, then it has to be branched and inserted back into the tree. MOBB branching is exactly the same as in SOBB, that is to branch on the most fractional solution, fixing it to both 0 and 1 and put these two subsequent nodes to the tree.

3.4 Multi-Sliced MOBB

While Non-Sliced MOBB can perfectly handle MOBB problems, the Sliced MOBB algorithm is come up with the hope that better bounding could lead to getting solutions faster.

Generally speaking, Sliced MOBB is to divide Quadrant to a number of slices; and in each slice, a CLP solver runs as in the Non-Sliced MOBB algorithm; while a single Pareto front is kept to generate the eventual result. Since the feasible region is divided by slices yet the total region is not affected, the eventual result will not miss any potential solution. Equation 3.9 guarantees that given the same binary fixing, only one optimal solution will dominate all the

other solutions in different slice; therefore, the eventual solution will maintain unchanged. In some unlikely cases, there might be the same number of feasible solutions as in the One-Sliced MOBB, therefore, the total number of solutions could get much larger than the One-Sliced version. But in more usual cases, sliced-MOBB algorithm gives much better bounding for each slice, and it could help the algorithm to reach the points on the final solution much faster.

To migrate from One-Sliced MOBB to Multi-Sliced MOBB algorithm, two modifications need be done:

(1) For each slice, an upper bound and a lower bound need to be added so that all the solutions are confined to this slice. Therefore, the objective function's slope becomes perpendicular with the angle division line of the angle squeezed by the lower bound and upper bound. Figure 3.7 illustrates a 5-slice-MOBB setup, in which blue lines define lower bound and upper bound while red lines define objective function.

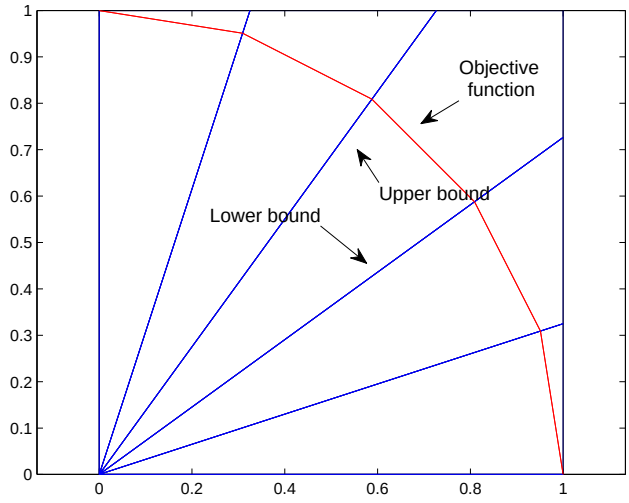

Figure 3.7: Lower bound, upper bound and objective function for a slice

(2) Definition of Semi-Nadir point is also extended, as the intersection points of two adjacent integer points and the lower/upper bound(s) between them are also categorized as Semi-Nadir points; so that within in one slice, the property that any point worse than a Semi-Nadir point will always be dominated by the Semi-Nadir point's adjacent integer points holds still. Figure 3.8 shows two integer points and all the Semi-Nadir points generated by them in a 3-slice setup. For convenience, SN(Semi-Nadir) points on lower/upper bound are called border SN points while SN points not on lower/upper bound are called conventional SN points.

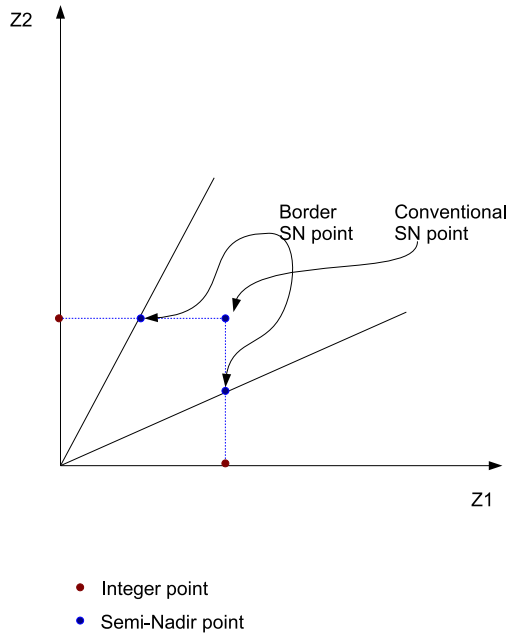

Figure 3.8: Conventional and border Semi-Nadir points

Up to this point, the formal description of sliced-MOBB and elements of MOBB algorithm has been provided. In the next chapter, a detailed implementation of MOBB algorithm will be presented.

Chapter 4

Implementation of Sequential MOBB

This chapter presents the implementation of the MOBB algorithm. As in Chapter 2, first, it presents the major data structures; second it presents the major algorithm flow charts.

4.1 Data structure needed

4.1.1 Point

As the problem domain is two dimensional, a point is the best data structure to represent z1 and z2. When working with numerical algorithms, a certain numerical precision is always needed in order to maintain numerical stability [16]. Throughout the algorithm, a great number of numerical comparisions need be done, and moreover, the final result, which is a list of points, needs to stay stable while slice numerical increases or decreases. Therefore, a precision definition is important in MOBB algorithm, since the stability of Pareto Front is totally dependent on the precision of Point. A rounding algorithm is added to cut off the extra digits to give a desired resolution for the point. As this rounding

algorithm can be solely encapsulated in this class, nothing needs to be changed except a macro define in header file when a new precision is needed. Since the precision in CLP solver is 10^{-5}, a precision of 10^{-4} is used here. Algorithm 2 illustrates the rounding scheme.

Algorithm 2 Rounding(orig)

Require: *orig* is a double number
Ensure: rounding of *orig*
1: $tmp \leftarrow orig * Prec + 0.5$ ▷ *tmp* is an integer, Prec is the reporical of precision
2: xx ← (double) tmp / Prec ▷ xx is a double number
3: **return** xx

In fact, Point is more of an abstract class, as in the real algorithm, two concrete classses are needed – SemiNadir Point and Integer Point. These two classes are child classes of Point with no further methods overloaded.

4.1.2 MOBB Node

MobbNode is one of the key data structures of the MOBB algorithm: it contains the information for each round of calculation in the whole algorithm. As in the SOBB Node, it requires two vectors, one for the binary indices of fixed values, and one for the fixed values. Also, as described in subsection 3.3.2.3, constraints to escape specific integer solutions are needed, such constraint class named EscapeConstraint is further explained in subsection 4.1.3. Therefore, three vectors are needed in MOBB Node class: two for binary variable indices and fixed values and the other for EscapeConstraint.

4.1.3 EscapeConstraint

As explained in Equation 3.12, one of the distinct differences from the SOBB algorithm is that MOBB needs to avoid checking the same Integer solutions again within one slice. An EscapeConstraint class is designed to serve this purpose. As discussed before, this class will generate a constraint to the CLP solver to ecapse an integer solution. In general, this algorithm finds all the solutions of fixed value 0, and then sets their coefficients to 1; finds all the solutions of fixed value 1 then sets their coefficients to 0. Finally, the lower bound of this

constraint is $1 - x$, where x is the number of variables that are fixed value 1. Algorithm 3 illustrates - given an integer point - how to generate the needed information for CLP solver to constrcut a constraint in order to avoid such solution in later calculation.

Algorithm 3 escape(fixing)

Require: $fixing$ is a vector of fixed values
Ensure: $elements$ an array of coefficients for a constraint in CLP solver and LB the lower bound of this constraint

```
1: elements ← arr[BN] ▷ BN is the number of binary variables
2: nonZero ← 0
3: for i = 1 to BN do
4:    if fixing[i]=1 then
5:       elements[i] ← −1
6:       nonZero ← nonZero + 1
7:    else
8:       elements[i] ← 1
9:    end if
10: end for
11: LB ← 1 − nonZero
12: return elements, LB
```

4.1.4 MOBB Tree

MobbTree is the data structure that contains a vector of MobbNodes. As in SOBB Tree, six methods are offered: push()/pop(), which adds/returns one node; isEmpty(), which returns if the tree still has any node; size(), which returns the size of the tree; split(), which splits a tree into two trees of equal size; and chooseNextNode(), which chooses next node to solve. There are several strategies to choose the next node to solve: choose the one with smallest relaxed objective value, or with the largest relaxed objectve value, or in the order they are inserted. The strategy of choosing the smallest relaxed objective value is adopted.

4.1.5 ConstraintsHelper

ConstraintsHelper is a class that faciliates the calcuation of lower bound, upper bound, and objective function. For each slice, there are 3 parameters that confine the solution region of MOBB. Theoretically speaking, given a slice number

m, and total number of slices N, the lower bound slope k_L^m, upper bound slope k_U^m and objective function slope k_O^m can be found using the equations below.

$$\Delta = \frac{\pi}{2 * N} \tag{4.1}$$
$$k_L^m = \tan(\Delta * (m-1)) \tag{4.2}$$
$$k_U^m = \tan(\Delta * (m)) \tag{4.3}$$
$$k_O^m = tan(\frac{atan(k_L) + atan(k_U)}{2}) \tag{4.4}$$

In practice, there might be some gaps between the upper bound of slice k and lower bound slice $k+1$ due to roundoff errors; so one needs to use the average value of theoretical k_U^m and k_L^{m+1} as the real ones; based on those values, a real slope of objective function can be calculated. It might be even safer to make some overlappings of each slice; however, since the Point's precision is 10^{-4} and the CLP's precision is 10^{-5}, i.e. a point can always overlap the gap caused by CLP solver precision, the current setup is sufficient.

4.1.6 ParetoFront

The ParetoFront class is a new class, which does not exist in the SOBB algorithm. Overall, ParetoFront has 2 tasks: (1)decide if an integer point in a certain slice is worse than the worst semi-nadir points; (2) add one integer point to an existing pareto front, and update the semi-nadir point list. Obviously, the ParetoFront class shall have two lists to faciliate calculation, which are an integer point lists and a semi-nadir point lists respectively. In practice, all these points are sorted based on their x(i.e. $z1$) value, so that if point p1 is ahead of point p2, then the x value of p1 is smaller than p2's x value.

The ParetoFront class needs to implement two main algorithms as described above: isWorse() and addIntegerPoint(). isWorse() is an easier algorithm, as it only needs to traverse all the SN points in a given slice and compare their z value to the calling point's z value. The algorithm 4 defines such a process.

Compared to isWorse(), addIntegerPoint() is a more complicated algorithm, it needs 4 subtasks to complete:

(1) locate(ip)(see Algorithm 5): this algorithm has, given an integer point ip, to locate a point loc in the integer point list so that the x value of ip is between

Algorithm 4 isWorse(ip, sliceID)

Require: ip a point, sliceID the slice in which ip located
Ensure: if this point is worse than the worst SN point in sliceID

```
1: for all snp in semiNadirPointList of sliceID do
2:   if ip better than snp then
3:     return false
4:   end if
5: end for
6: return true
```

loc and *loc*'s next point (or $loc + 1$).

Algorithm 5 locate(ip)

Require: ip an integer point, x value of ip is strictly between 0 and 1
Ensure: loc an integer point so that ip's x value is between that of loc and loc+1

```
1: for all loc in integer point list do
2:   if ip is between loc and loc+1 then
3:     return loc
4:   end if
5: end for
6: return loc ▷ this statement shall never be reached
```

(2) reshuffleIntegerPoints() (see Algorithm 6): this algorithm is to reshuffle the integer point list after one integer point is added in the place between *loc* and *loc*+1(as described in locate() method). Adding an integer point at such a place could lead to several effects on the integer point list:(A) added integer point is dominated by other integer points so the integer point list is not affected; (B) added integer point draws with all the other integer points, so it is added in its according position in the list and no other integer points are deleted; (C) added integer point dominates some other integer points, so that they have to be delete from list. reshuffleIntegerPoints() is such a method to decide which one of the three senarios is when an integer point is added and take the according action.

(3) reshuffleSemiNadirPoint() (see Algorithm 7): this algorithm has to generate all the SN points(be it conventional or border) once the integer point list is changed. It first calculates the conventional SN points and then finds all the border SN points between them.

(4) addIntegerPoint(*ip*)(see Algorithm 8): this algorithm has to call all the previous 3 algorithms to finish the whole task. It first locates a given *ip* and

Algorithm 6 reshuffleIntegerPoints()

```
1: for all loc in integer point list do
2:    while (loc+1)→y ⩾ loc→y do
3:       remove(loc+1)
4:    end while
5: end for
```

Algorithm 7 reshuffleSemiNadirPoints()

```
1: for all loc in integer point list do
2:    calculate the conventional semi-Nadir Point csp between loc and loc + 1
3:    based on csp,loc and loc + 1, find all the semi-Nadir Points intersected on
      the border
4:    put all the conventional and border semi-Nadir Points into the semiNadir-
      Point list
5: end for
```

adds *ip* into the integer point list without validating this integer point, then it reshuffles integer point list and the SN point list , so that both of them are valid.

4.1.7 SliceSolver

The class SliceSolver is the core part of the whole algorithm. It coordinates all the classes described above to carry on MOBB algorithm. A SliceSolver instance is assigned to each slice. It holds one *MOBB tree* to keep track of all the *MOBB nodes*. It holds one CLP solver to carry out desired LP calculation, once a time this solver needs to be modified according to *EscapeConstraint*. Whenever it obtains an integer point, it needs to talk to *ParetoFront* to handle this integer point. This class is a very complicated class, and further details are presented in subsection 4.2.2.

4.2 Algorithm flow chart

After describing the data structures and the algorithms used by them, the MOBB algorithm as a whole is presented below.

Algorithm 8 addIntegerPoint(ip)

Require: ip an integer point blaba
Ensure: add ip into the appropriate place of the integer point list

```
 1: updated ← false
 2: if ip out of 1*1 bound then
 3:    updated ← false;
 4: else
 5:    loc ← locate(ip) ▷ Algorithm (5)
 6:    if ip→x == loc→x then
 7:       ▷ handle when two points have the same x value
 8:       if ip→y ⩾ loc→y then
 9:          upated ← false
10:       else
11:          remove loc
12:          insert p
13:          upated ← true
14:       end if
15:    else if ip→ x == (loc+1)→x then
16:       if ip→y ⩾ (loc+1)→y then
17:          upated ← false
18:       else
19:          remove loc
20:          insert p
21:          upated ← true
22:       end if
23:    else
24:       ▷ when ip is strictly between loc and loc+1
25:       if ip→y ⩾ loc→y then
26:          upated ← false
27:       else
28:          insert p
29:          upated ← true
30:       end if
31:    end if
32: end if
33: if updated then
34:    reshuffleIntegerPoints() ▷ Algorithm (6)
35:    reshuffleSemiNadirPoints() ▷ Algorithm (7)
36: end if
```

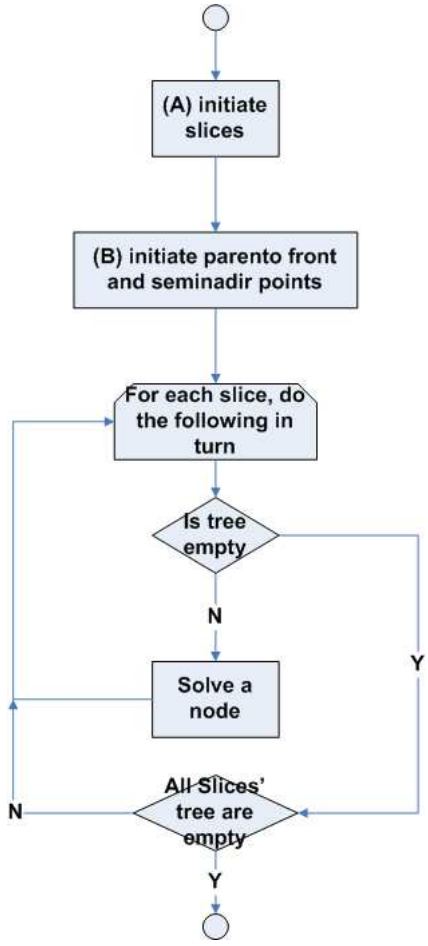

Figure 4.1: Big picture of Multi-objective B & B algo.

4.2.1 Big picture

Figure 4.1 is a big picture of MOBB algorithm. Explaination for each process is given below:

- (A) For each SliceSolver, initiate CLP solver, 6 tasks need be done: (1) read the input .mps file; (2) look up the binary variable indices, store them, set each binary variable to continuous, set lower bound to 0 and upper bound to 1 for each binary variable; (3) get the column indices of variable $z1$ and $z2$ for the convenience of accessing them in solutions later.

$z1$, $z2$are given names of "Z1.." ,"Z2.." in this project respectively; (4) get the row indice of the objective function, which was previously defined as $z = z1 + z2$ for convenience of modifying it later. This row is given the name of "NEW." in this project; (5) calculate the upper bound, lower bound and objective function of each slice. Add these constraints (rows) and change the objective function for each slice's CLP solver. To facilitate the calculation, the ConstraintsHelper instance is used to come up with the neccessary information for the CLP solver in each SliceSolver. To modify one objective function constraint, one has to remove the old one first and then add the new objective function row; (6) construct one empty MOBBNode, which has no binary fixing, and pushs it into a tree.

- (B) Special scaling procedure is taken to make sure that (1,0) and (0,1) are always two feasible integer points, and all the solutions shall be non-negative, therefore, (1,0) and (0,1) are always in any Pareto front. Thus,given the (1,0) and (0,1) points and each upper bound, lower bound, the program initiates the pareto front and Semi-Nadir points.
- (C) Pick up each slice solver for later calcuation. The pick-up scheme can be varied: one can pick in an anti-clockwise way(slice 0 →1→2→3...) or a clockwise, or randomly. To simulate a parallel implementation of MOBB, random-pick-up scheme is adopted here.
- (D) The function call isEmpty() in MOBBTree class is invoked to decide if the tree in the selected SliceSolver has more nodes to exploit.
- (E) Solving a selected node, as described in the subsection (4.2.2)

4.2.2 Solve one Node

Figure 4.2 is the flow chart of solving one node.

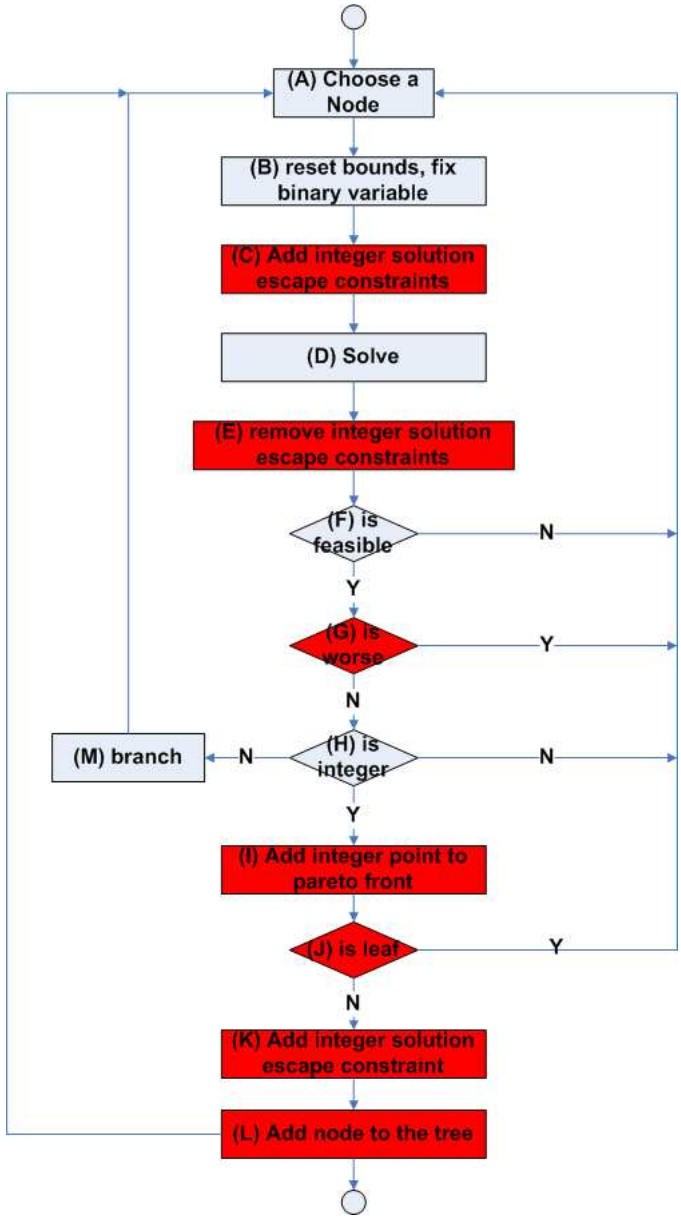

Figure 4.2: Solve one MOBB node. The different parts from SOBB is hilightened in red color

Explaination for each process:

- (A) Choose an MOBBNode, with the smallest z value.

- (B) Reset the binary variable bounds and fix the binary variables according to the selected node.

- (C) Add rows according to all the predecessors' integer-solution-escape constraints added in their step(K) operations. This process is to ensure that the CLP solver will not examine the previously examined Integer points as discussed in subsection 3.3.2.3.

- (D) Call the CLP solver's solve method to do one round of linear relaxation solve. Same as in SOBB, call method initialSolve() when the first time solving, call method resolve() afterwards.

- (E) Remove all the integer-solution-escape constraints added in (C) to clear up the CLP solver, because in the next round of solving a node, it is most likely that another node is selected, and it might have completely different integer-solution-escape constraints. Therefore a clear up of the CLP solver is demanded here.

- (F) Call ProvenPrimalInfeasible() in the CLP solver to decide the feasibility. In practice, one needs to call this method right after step (D) to record the feasibility. That is because in the CLP solver's design, once certain procedures such like in step(E) are taken, the solver's feasibility will be affected so previously infeasible solutions might be switched to feasible, which leads to inserting false nodes into tree. In such a case, indefinitely more false nodes, with the same integer fixing, will be added since they cannot be pruned by infeasibility and cannot escape the previous integer solution.

- (G) Call the ParetoFront class to compare current z value with the worst z value of all the seminadir points in that slice.

- (H) Test if all the solutions are integer. This procedure is exactly the same as in SOBB.

- (I) Call ParetoFront class to add one integer solution to the pareto front, this point could be, as discussed previously, either (1) dominated (2) inserted without affecting other integer solutions on the pareto front, or (3) dominate other integer solutions.

- (J) Calculate if the current node is a leaf node, i.e if all the binary variables are fixed before calling the solve routine as in Step(D). This procedure was not needed in SOBB because in SOBB if a node generates an integer point, this node can always be discarded so it will not be further branched. But as discussed in subsection 3.3.2.3, such a node cannot be discarded and

it need to be branched if it is not a leaf node and can only be discarded if it is a leaf node.

- (K) Add all the predecessors' integer-solution-escape constraints so that those solutions will not appear again.
- (L) Add this node(after adding the integer solution escape constraint in (K)) back to the tree because this node needs to be further branched. Adding the integer-solution-escape constraint could help avoid generating the same integer point so that a further branching on this node is made possible.
- (M) Choose the binary variable closest to 0.5 and branch on this variable. It is exactly the same as in SOBB.

4.3 Test and analysis of sequential MOBB

4.3.1 Test the validity of the program

The traditional way to calculate a Pareto front is to use a brute-force method. To ensure the validity of the program, one can use a brute-force method to obtain a Pareto front and compare it with our sliced MOBB algorithm. Specifically, in this thesis, we first write the result of brute-force and the result of a sliced MOBB algorithm to two different files, and then use the Unix command $diff$ to compare two files. The validity of the program is verified only if the two files are exactly the same. Figure 4.3 displays four Pareto fronts, which are calculated by using brute-force, slice number equals to 1, 5 and 100 respectively.

4.3.2 Profiling of the program

After verifying the validity of the program, one needs to profile the program to examine what is the bottle-neck of this program. We use Sun Studio Performance Analyzer here to profile the program. Table 4.1 is the profiling of a brute-force program working on a 1024 node problem. Table 4.2 is the profiling of a sliced-MOBB program working on a 3082 node problem(N.B. the nodes in two problem sets have different constraints, therefore each node requires different time to solve). Both profilings show that the *resolve*() routine in CLP solver is the one that takes most of time and ParetoFront related calculation take very few time, in other words, the LP related calcuation is the bottle-neck.

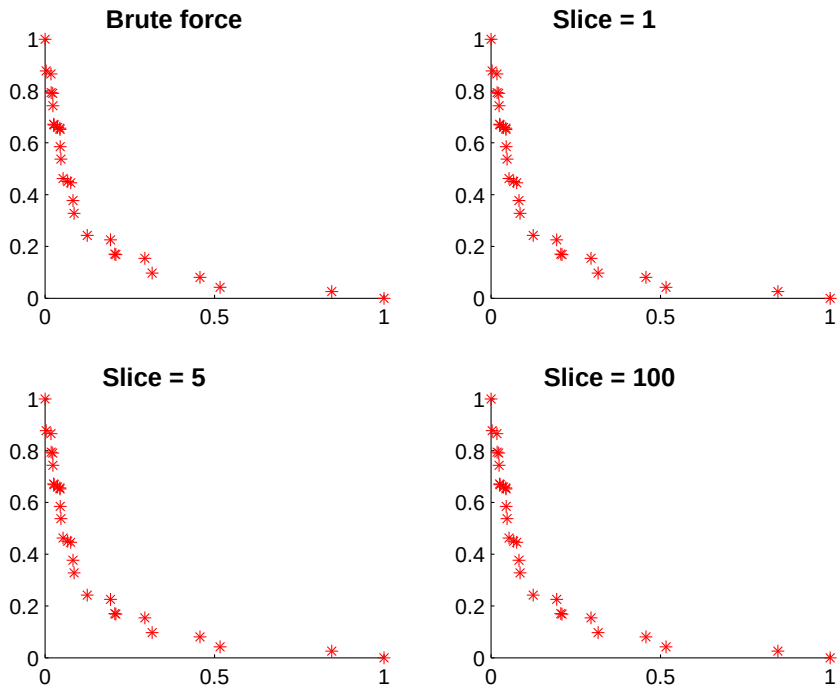

Figure 4.3: Four Pareto Fronts, calculated by using brute force, slice number equals to 1,5 and 100.

resolve() is called each time a node needs to be solved, therefore, reducing the number of nodes to solve while maintaining the validity of the program is the key part of MOBB algorithm.

4.3.3 Workload comparison between brute-force and sliced-MOBB

To compare the performance between the traditional brute-force and sliced-MOBB, two tests are done: Table 4.3 is the node number comparison between brute-force and sliced-MOBB; Table 4.4 is the time comparison between brute-force and sliced-MOBB.

Function	CPU Time(secs)	CPU Precentage
Total	9.907	100%
Clp:resolve()	9.487	95.8%
ParetoFront:addIntPoint()	0.1	1%
ParetoFront:isWorse()	0	0%
the rest	0.42	4.1%

Table 4.1: Profiling of brute force

Function	CPU Time(secs)	CPU Precentage
Total	107.805	100%
Clp:resolve()	104.323	96.8%
ParetoFront:addIntPoint()	0.07	0.06%
ParetoFront:isWorse()	0	0%
the rest	3.412	3.14%

Table 4.2: Profiling of sliced-MOBB

Binary variable number	Brute-Force	Sliced-MOBB
22	4 194 304	6365
24	16 777 216	6976
26	67 108 864	17467
28	268 435 456	34105
30	1 073 741 824	72258

Table 4.3: Node number comparison between sliced-MOBB and brute force

Binary Variable Number	Brute-Force Time(secs)	Sliced-MOBB Time(sec)
22	2662	15
24	10506	16
26	41964	42
28	166789	85
30	648151	191

Table 4.4: Time comparison between sliced-MOBB and brute force

4.3.4 Comparison between different slice number

The reason that multi-sliced MOBB algorithm is proposed rather than one-sliced MOBB is because we hope the division into slice could help better bounding and get to the final solution more quickly. Figure 4.4 shows on the same test bench(with 0% continuous variables), the node number iterated by different number of slices. Clearly seen from it, when slice gets bigger, the bounding gets better, therefore the node number gets smaller.

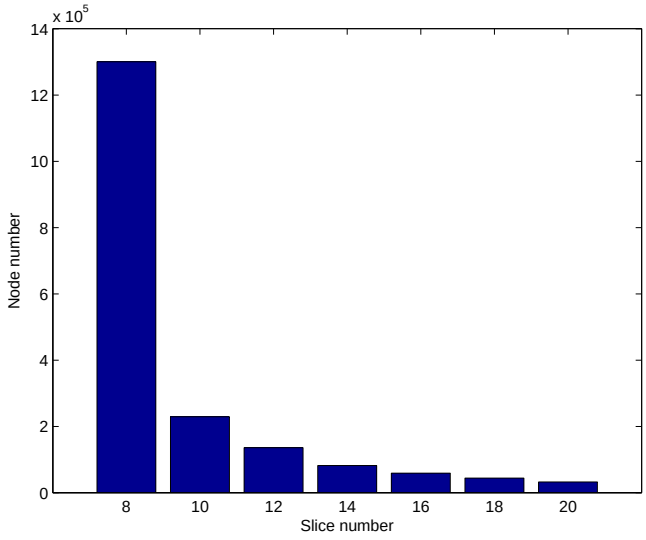

Figure 4.4: Node number comparison for different slice numbers

4.3.5 Workload growth

From the above discussion, it is easy to see that solving each node is the most time consuming task in this algorithm. It is important to generalize how the workload increases and decreases. After intensive test, it is shown that sliced-MOBB algorithm always grows and dies out gradually, the shape of workload growth resembles a bell shape. Figure 4.5 shows a typical workload growth in a sliced-MOBB algorithm. Such a workload growth implies that the workload in MOBB always grow and die out gradually, without an oscillating fashion; such

a property is of great importance for the later parallelization work.

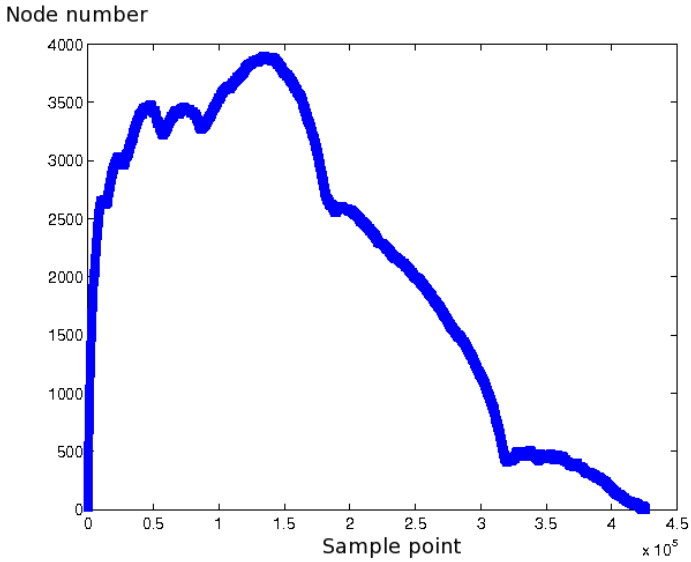

Figure 4.5: Workload growth of sliced-MOBB algorithm, 5 slices in total

Chapter 5

Test and analysis of sequential MOBB

Test and analysis of sequential MOBB. N.B. simulation of parallel program in a sequential way.

Chapter 6

Parallel Computing

This chapter illustrates serveral fundamental aspects of parallel programming. In this chapter, we first introduce how to evaluate a parallel program's performance; secondly, we categorize parallel programs based on the workload distribution; finally, we give a brief but thorough introduction to the state-of-art parallel programming tools.

6.1 Amdahl's law

In a parallel program, to evaluate the performance, Amdahl's law [8] is the criteria to apply. Amdahl's law states that if P is the proportion of a program that can be made parallel (benefit from parallelization), and $(1 - P)$ is the proportion that cannot be parallelized (remains serial), then the maximum speedup that can be achieved by using N processors is $\frac{1}{1-P+\frac{P}{N}}$. Therefore, if one can use as many processors as possible, the maximum speedup is obtained as $\lim_{N\to\infty} \frac{1}{1-P+\frac{P}{N}} = \frac{1}{1-P}$. In practice, it is very difficult to get a precise evaluation of P, so one needs to estimate P based on speedup samplings. To estimate P, one needs to test all the speedups given different Ns, and then find the maximum speedup of them all. Using that maximum speedup and the according N,

one can calculate P.

One thing needs to be pointed out here is that according to Amdahl's law, the theoretical maximum speedup of using N processors would be N, namely linear speedup. It is not uncommon to observe more than N times speedup on a machine with N processors in practice, namely super linear speedup [8]. One possible reason is the effect of cache aggregation. In parallel computers, not only does the number of processors change, but so does the size of accumulated caches from different processors. With the larger accumulated cache size, more or even the entire data set can fit into caches, dramatically reducing memory access time and producing an additional speedup beyond that arising from pure computation.

6.2 Category of parallel computing

Parallel programs[9] are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits **fine-grained parallelism** if its subtasks must communicate many times per second; it exhibits **coarse-grained parallelism** if they do not communicate many times per second, and it exhibits **embarrassingly parallelism** if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize for programmers, coarse-grained less easier, whereas fine-grained the hardest.

6.3 Tools for parallel computing

A number of concurrent programming languages, libraries, APIs, and parallel programming models have been created for programming parallel computers. Among them, Message Passing Interface (MPI) is the most widely used message passing system API, whereas , OpenMP is the most widely used shared memory APIs.

6.3.1 MPI

MPI [10] is an implementation that allows many computers to communicate with one another. It is the de facto standard for HPC computing. It can be

used on both shared memory machines and distributed machines. This project is primarily done using MPI APIs.

6.3.1.1 Data and communicators in MPI

(1) Communicator: A communicator is a MPI handle that defines a group of processes that are permitted to communicate with one another. MPI automatically provides a basic communicator called MPI_COMM_WORLD, which is the communicator consisting of all processors. Using MPI_ COMM_WORLD, every processor can communicate with every other processor. Throughout this project, MPI_ COMM_ WORLD is used as the only communicator.

(2) Primitive Datatype: All the primitive data types in $C++$ are defined in MPI, among them, MPI_INT and MPI_DOUBLE are the most used primitive data types in this project.

(3) Derived-datatypes: Creating a derived datatype and to use it just once before freeing it can be a bit wasteful. It is more effective to create derived datatypes that describe recurring patterns of access and then reuse them for each occurrence of that pattern of access. But it can be only used when the size of the derived data type is fixed.

(4) Messages: In MPI, unlike OpenMP where all the data are visible to every CPU, the local data is only visible to its CPU host. Therefore, to share the data, message passing is the only way to do it. Message, in MPI, consists of two parts: envelope and message body. The envelope has 4 parts: 1. Source – the sending process, 2. Destination – the receiving process, 3. Communicator – specifies a group of processes to which both source and destination belong, 4. Tag – used to classify messages. The message body has three parts: 1. Buffer – the message data, 2. Datatype – the type of the message data, 3. Count – the number of elements in buffer.

6.3.1.2 Methods in MPI

There are several methods of particular interest for this project.The review below serves as a primal to MPI [11], so later chapters can solely focus on the algorithm part.

The first 4 methods are about how to initiate, obtain communicator information and terminate MPI programs. They are the methods need be called by every MPI program. They serve to initiate data structures, allocate CPU and memory resources, and finally release all the resources.

(1) int MPI_Init(int argc, char** argv): This method establishes the MPI environment and will return an error code if there is a problem. Once this method is called, the operating system will dispatch the number of CPUs based on the parameter passed by argv, and also the limit of the CPUs available upon that time. It is the first method to call in every MPI program.

(2)int MPI_Comm_ rank(MPI_Comm comm, int *rank) Within each communicator, processors are numbered consecutively (starting at 0). This identifying number is known as the rank of the processor in that communicator. The rank is also used to specify the source and destination in send and receive calls.

(3)int MPI_Comm_size(MPI_Comm comm, int *size): The argument comm is of type MPI_COMM, a communicator. The second argument is the address of the integer variable size. Size is the number of CPUs available for this program.

(4)MPI_Finalize(): It is the last MPI routine called in a program. It terminates the program by cleaning up all MPI data structures, canceling operations that never completed, and so on. MPI_FIinalize() must be called by all processes. Once MPI_Finalize() has been called, no other MPI routines may be called.

The next 6 methods are about how to pass messages in MPI. There are two major communication fashions: blocking and non-blocking. Blocking is to block the calling process until the communication operation is completed. That is to say the sending/receiving will hang if they are not finished and all the statements after them cannot be reached until communication is finished. Non-blocking is the send and receive operations that do not block the calling process. It is possible to separate the initiation of a send or receive operation from its completion by making two separate calls to MPI. The first call initiates the operation, and the second call completes it. In practice, blocking and nonblocking communication can be mixed together. The source processor might use a blocking send and the destination process could use a nonblocking receive process, or vice-versa.

(5) int MPI_Send(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm): MPI_Send is the blocking sending method. As discussed before, the first 3 parameters are message body, and the latter 3 parameters are envelope(the sender rank is defined implicitly as it is the rank of this process).

(6)int MPI_Recv(void *buf, int count, MPI_ Datatype dtype, int source, int

tag, MPI_Comm comm, MPI_Status *status): MPI_Recv is the blocking recieving method. As discussed before, the first 3 parameters are message body, and the latter 3 parameters are envelope(the receiver rank is defined implicitly as it is the rank of this process). The output argument status is to keep a track of receiving status.

(7) int MPI_Isend(void *buf, int count, MPI_Datatype dtype, int dest, int tag, MPI_Comm comm, MPI_Request *request): MPI_Isend posts a non-blocking send message. The calling sequence is similar to the calling sequence for the blocking routine MPI_Send but includes an additional output argument, a request handle. The request handle can be used to check the status of the posted send or to wait for its completion.

(8) MPI_Irecv(void *buf, int count, MPI_Datatype dtype, int source, int tag, MPI_Comm comm, MPI_Request *request): A process calls the routine MPI_Irecv to post a non-blocking receive . The calling sequence is similar to the calling sequence for the blocking routine MPI_Recv, but the status argument is replaced by a request handle; both are output arguments. The request handle identifies the receive operation that was posted and can be used to check the status of the posted receive or to wait for its completion.

(9)int MPI_Wait(MPI_Request *request, MPI_ Status *status): A process that has posted a send or receive by calling a nonblocking routine (for instance, MPI_Isend or MPI_Irecv) can subsequently wait for the posted operation to complete by calling MPI_Wait . The posted send or receive is identified by passing a request handle. The arguments for the MPI_Wait routine are: request – a request handle (returned when the send or receive was posted; status – for receive, information on the message received).

(10) int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status): A process that has posted a send or receive by calling a nonblocking routine can subsequently test for the posted operation's completion by calling MPI_Test. The posted send or receive is identified by passing a request handle. The arguments for the MPI_Test routine are: request – a request handle (returned when the send or receive was posted), flag – "true" if the send or receive has completed; status – undefined if flag equals "false". Otherwise, it is just like MPI_Wait. In addition, an error code is returned.

The above 10 methods can construct almost all the MPI programs. Besides the above 10 methods, there are many other methods available in MPI, such like those dealing with collective communications, virtual topologies; however, they are not particularly useful to serve the purpose of this thesis, therefore, they are not discussed further.

6.3.1.3 Runtime Behavior

The runtime behavior in this subsection is specifically refered to the blocking communication runtime behavior. when a message is sent using MPI_Send one of two things may happen:(1) The message may be copied into an MPI internal buffer and transferred to its destination later, in the background, or (2) The message may be left where it is, in the program's variables, until the destination process is ready to receive it. At that time, the message is transferred to its destination. In the first case, the sending process is allowed to move on to other things after the copy is completed. That is to say, it is asynchronized sending/receiving. Such a property can lead to certain "communication delay" – for instance, Process A first sends a message to Process B and then sends a message to Process C, due to the runtime behavior discussed above, Process C might receive its message before Process B. Such a behavior can cause several problems, thus some approaches are needed to solve these problems. Those approaches will be further discussed in Chapter8.

The most common pitfall in MPI program is the deadlock[13] problem. Suppose such a senario–process A waits for a message from process B and then sends a message to process B; in the mean time, process B waits for a message from process A and then sends a message to process A. In this senario, either process can move on only when it receives message from the other, therefore, a deadlock is generated.

6.3.2 OpenMP

The OpenMP (Open Multi-Processing) [12] is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It is comprised of three complementary components: (1) a set of directives used by the programmer to instruct the compiler on parallelism; (2) a runtime library which enables the setting and querying of parallel parameters such as number of participating threads and the thread number. (3) a limited number of environment variables that can be used to define runtime system parallel parameters such as the number of threads. It is by far the best API for parallel programming on shared-memory machines. The most common pitfall in OpenMP is data racing [13], in which shared variables are falsely updated or accessed by different threads. Due to the time limit, an OpenMP implementation of this thesis project was not finished. Therefore, no further introduction on OpenMP is presented in this thesis.

Chapter 7

Coarse Parallel MOBB

MPI provides a simpler interface to construct parallel programs that applies to MOBB problem. So it has been the primary choice during this thesis work. Two version of parallel programs have been developed– coarse-grained version and fine-grained version.

In coarse-grained version, it is a straightforward mapping from the sequential version to parallel version. Suppose N CPUs are available, 1 of them can serve as the ParetoFront handler, which handles the isWorse inquiry and addIntegerPoint inquiry; while the other $N-1$ CPUs serve as the Slice Solvers.

7.1 Design and Implementation

7.1.1 How to migrate from sequential version to parallel version

In the sequential version of MOBB, each SliceSolver instance shares the same ParetoFront instance, so when they need to call isWorse() or addIntegerPoint(), they can just use that ParetoFront instance because it is shared by each Slice-

Solver. However, when we move to MPI programs, we need to separate the ParetoFront instance from each SliceSolver instance. Therefore, the most important migration in this algorithm is to use MPI to pass messages between SliceSolver and ParetoFront. Since there are only two methods that need to be communicated, the messages are therefore very straightforward to code: (1) For isWorse() , only the relaxed z value(as the sliceID is implicitly passed by the sender rank) need to be transferred to ParetoFront, and ParetoFront then passes a boolean value back. (2) For addIntegerPoint() , the integer point as a whole(an double array of size 2) needs to be transferred to ParetoFront and no response need be transferred back.

It would be sufficient to just send the above two messages in this algorithm, however, there is a problem in such a design – while SliceSolver terminates when it enumerates all the nodes in its tree, ParetoFront does not know when to terminate itself. To solve this, before SliceSolver terminates itself, it needs to send a finish message to ParetoFront. When ParetoFront has received all the finish messages, it then can terminate.

7.1.2 State Transition graph

To better illustrate the design, state transition graphs are made to represent how a state is changed within one process. Dashed lines are used to represent non-blocking communication, while solid lines are used to represent blocking communication. Red is used to mark slave process, while green is used to mark master process; for example, a red edge in the state transition graph of master process means at this point the communication peer is the slave process. In one communication, the talking peers are labeled with the same number, while sending peer with the number and receiving peer with the number and its prime , for example 1 and 1' mark two talking peers in the same round.

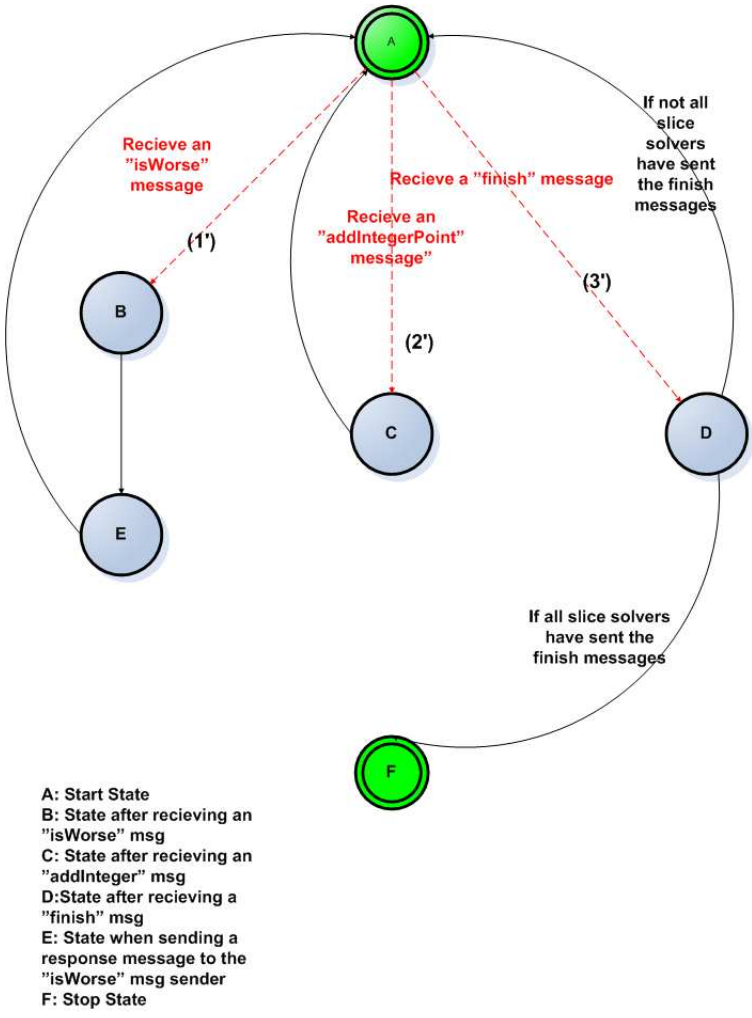

Figure 7.1: State transition graph for Master CPU .

Figure 7.1 is the state transition graph of the master process. It is divided to 3 parts:(1) Path $StateA \hookrightarrow StateB \hookrightarrow StateE$ is to handle an isWorse message. The master will call the isWorse() routine of its ParetoFront instance to generate a response for this message. (2) Path $StateA \hookrightarrow StateC$ is to handle an addIntegerPoint message. The master will call the addIntegerPoint() routine of its ParetoFront to add the received integer point. (3) Path $StateA \hookrightarrow StateD \hookrightarrow StateF$ is to handle a finish message from one slave process. If the master collects the finish messages from all the slave processes then it terminates.

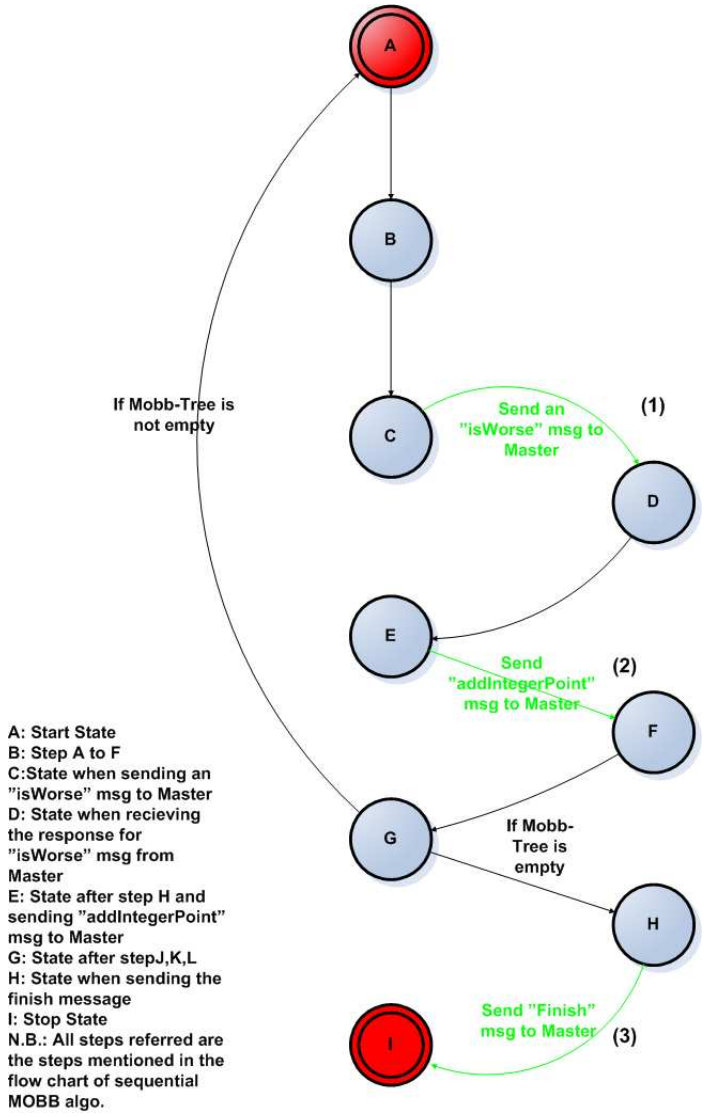

Figure 7.2: State transition graph for Slave CPU .

Figure 7.2 is the state transition graph of the slave process. It is divided to 3 parts: (1) Path $StateA \hookrightarrow StateB \hookrightarrow StateC$ is all the calculations before calling isWorse(), as in the sequential MOBB, and sends the isWorse message to the master; (2) Path $StateD \hookrightarrow StateE$ is all the calculations between calling isWorse() and addIntegerPoint(), as in the sequential MOBB, and sends

the addIntegerPoint message to the master; (3) Path $StateF \hookrightarrow StateG$ is all the remaining calculation in solving a node, as in the sequential MOBB. Path $StateG \hookrightarrow StateA$ is taken if there are more nodes in the tree; Path $StateG \hookrightarrow StateH$ is taken if all the nodes are solved, $StateH$ is to send the finish message to the master. $StateI$ is the termination state.

7.1.3 Implementation

7.1.3.1 Communication

All these three receiving communications are implemented as non-blocking messages. Each communication follows the pattern in algorithm 9. That is to say the message receiver first posts a non-blocking receiving and then periodically checks if the desired message is received, if so it will handle the message and then post another non-blocking receiving, otherwise, it will keep going on. Such a pattern will be repeatedly used throughout this project.

Theoretically, the coarse-grained version's master can be modeled as block-

Algorithm 9 non blocking communication

```
1: msg ← MPI_Irecv() ▷ post non-blocking receive
2: while true do
3:    MPI_Test(..., &flag) ▷ test if the message has been received yet
4:    if flag==RECEIVED then
5:       handle the received msg ▷ handle the message
6:       msg ← MPI_Irecv() ▷ continue to post this non-blocking receive
7:    end if
8: end while
```

ing communication. To do so, the master is always receiving one format of message. To ensure isWorse(), addIntegerPoint(), and finish() message can be all formatted in the same pattern, one has to first find the largest message size, which is 2 by addIntegerPoint(), as it needs to send the $z1$ and $z2$ value; second, one needs to pad one more number for isWorse() and finish() message so that all three formats are consistent; finallly, one needs to add one more bit of ID after each message so that the master could distinguish what each message is for. Algorithm 10 describes such an algorithm. Obviously it requires more communication space and is less flexible. Using blocking communication can serve the parallelization purpose in this simple case, it will not hold true in a later more complex case.

Algorithm 10 blocking communication master

```
1: finish ← 0
2: slaveNum ← (size − 1)
3: while finish < slaveNum do
4:    msg ← MPI_Recv() ▷ blocking receive
5:    if msg.ID == isWorse then
6:       handle isWorse message
7:    else if msg.ID == addIntegerPoint then
8:       handle add-integer-point message
9:    else
10:      finish ← (finish + 1) ▷ finish message
11:   end if
12: end while
```

7.2 Test and Analysis

7.2.1 Execution time and Speed up

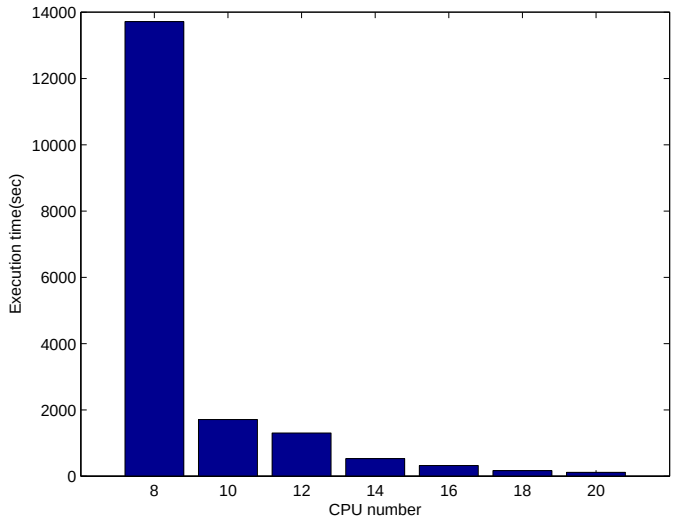

Figure 7.3: Coarse-grained Execution Time

Figure 7.3 is the execution time when CPU number grows from 8 to 20. It has shown the execution time drops very quickly; however, as discussed before, it is also because when the slice number is different, the sequential algorithm has very different execution time. To better evaluate the performance, we shall use the Amdahl's law.

Figure 7.4 is the speed up when CPU number grows from 8 to 20. Clearly, we can see that the speed up in this case is bound to only a little over 2, the parallelization factor is 68%, given Amdahl's law, the speed up can achieve no more than 3.12.

7.2.2 Workload Distribution

The reason that the speed up is not very high in coarse-grained implementation is because the workload is not evenly distributed. Figure 7.5 is the work load distribution of a testbench when $CPU = 8$. It is clearly seen that the workload

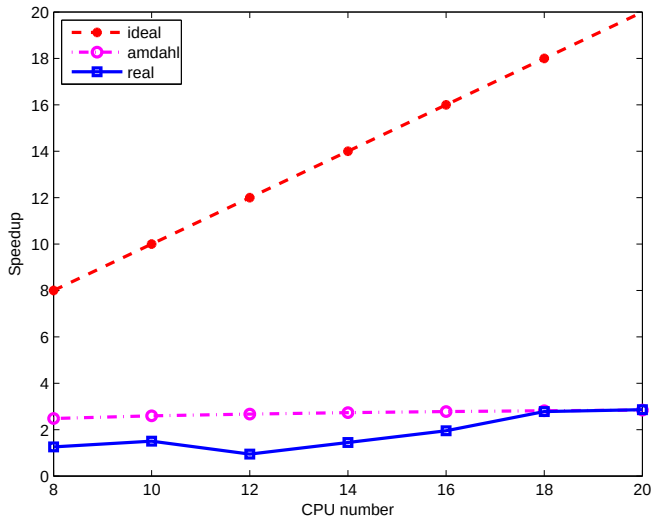

Figure 7.4: Coarse-grained Speed-up

distribution is very unevenly distributed. It might be a very rare case, however, it clearly indicates the weakness of this implementation.

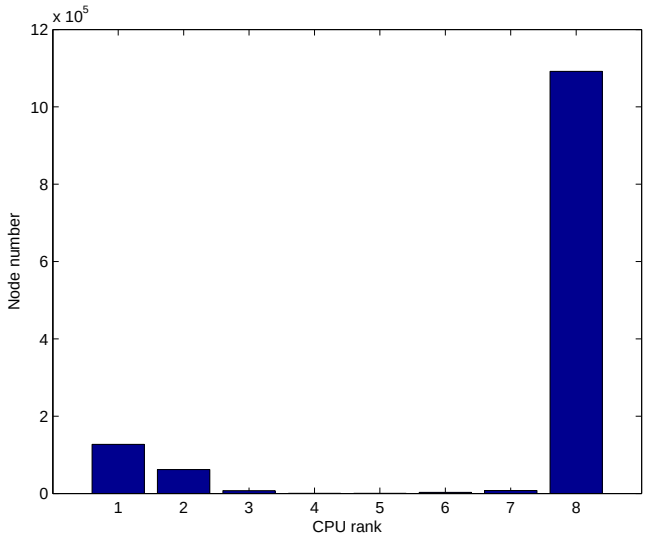

Figure 7.5: Coarse-grained work load

Chapter 8

Fine parallel MOBB

This chapter further investigates how to better utilize CPU recourses. To accomplish this, a better load-balancing scheme is designed and implemented. Then, a thorough test and analysis of this design is presented.

8.1 Design and Implementation

8.1.1 How to improve load balance

The essential problem that occured in the coarse load balancing algorithm, is that when a lightly-loaded slave process finishes working, it will just exit and its CPU becomes idle. So the total execution time is dependent on the last finished slave process. To mitigate this effect, a mechanism need to be designed, that, once a slave process finishes its workload, it can check with the master process to see if it can get some extra work from other slave processees, and if so, it will talk to the targeted slave process and get the work from it and continue until all the slave processes are finished. Such a strategy could lead to a better performance and is also based on the fact that transmitting a tree in MPI is much faster than solving a tree.

Therefore, the master process must keep a record of how many nodes each slave process's tree contains, so that it can allocate trees when a requirement is made. In theory, only one master process is needed to do both work allocation and pareto front calculation. However, it is of great importance to update pareto front quickly; therefore it is better to assign 2 masters, one of which is to do the workload allocation, while the other is to do the pareto front calculation.

Overall, 3 major classes are needed – Pareto Front Master process, Scheduling process, and Worker Process.

8.1.2 Design

The scheduling master process is a brand new process. It essentially has three tasks: (1) keep a record of the tree size of each working process. (2) re-allocate workload. (3) signaling Pareto front process and each Slave process to terminate.

The Pareto front master process is the process that calculates the pareto front. It does almost the same work as the master process in Coarse version. The only difference is that it stops only when signaled by the scheduling master process.

The slave process is still the working process that does most of the linear relaxation, branch and bound job. It will, however, not stop when it finishes solving all the nodes in its tree; but rather solicit the scheduling master to require more jobs from another slave process. It will only stop when there are no more trees in other slave processes available.

As in the previous chapter, state transition graphs are made to represent how state is changed within one process. Dashed lines are used to represent non-blocking communication, while solid lines are used to represent blocking communication. Red is used to mark slave process, green is used to mark pareto front master process, blue is used to mark scheduling master process. In one communication, the talking peers are labeled with the same number, while sending peer with the number and receiving peer with the number and its prime , for example 1 and 1' mark two talking peers in the same round.

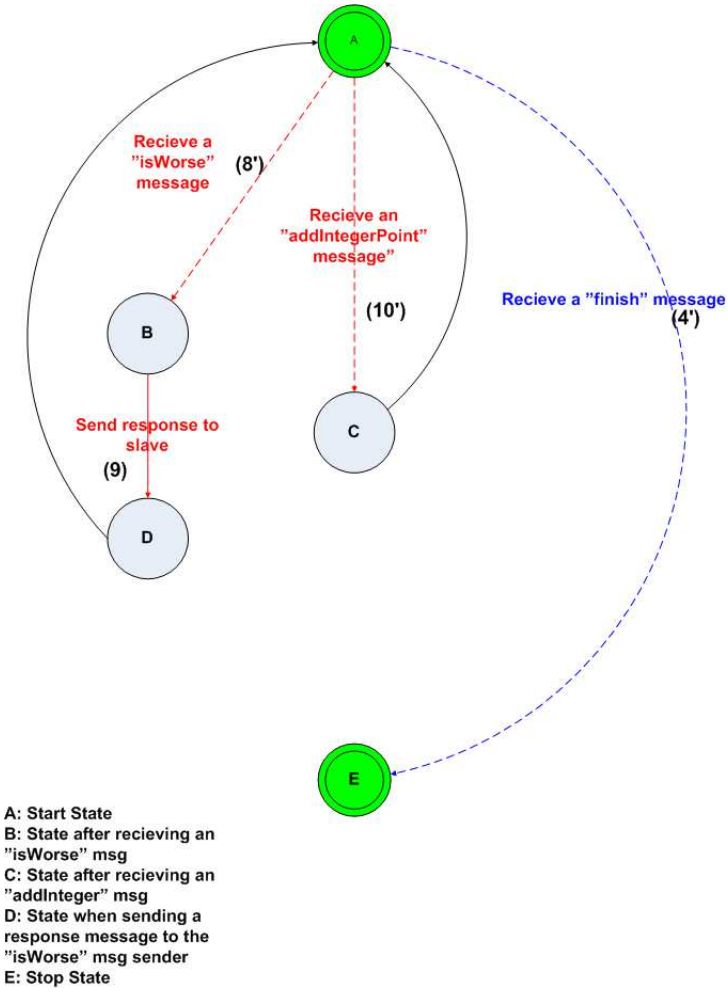

Figure 8.1: State transition graph for ParetoFront Master

8.1.2.1 Pareto Front Master

Figure 8.1 shows the state transition of the Pareto front master. It is divided to 3 major parts:(1) StateA $\hookrightarrow$ StateB $\hookrightarrow$ StateD is that master receives an isWorseMessage from an arbitrary slave process, handles this message by calling ParetoFront's isWorse routine and returns the response. (2) StateA $\hookrightarrow$ StateC is that master receives an addIntegerPoint message from an arbitrary slave process and then calls the addIntegerPoint routine in ParetoFront to handle it. (3)

StateA $\hookrightarrow$ StateE is that when master receives message from Scheduling master and then terminates.

8.1.2.2 Scheduler Master

Figure8.2 shows the Scheduling master. It is responsible for recording and scheduling the workload. It is divided into 3 parts:

(1) StateA $\hookrightarrow$ StateB is that Scheduling master receives an updated tree size message from an arbitrary slave process and updates this message in its tree size table.

(2) $StateA \hookrightarrow StateC \hookrightarrow StateD \hookrightarrow StateE \hookrightarrow StateK$ is when the scheduler gets all the termination acknowledge message from slave processes and then send out the final termination message to all communication entities. $StateC$ is when the scheduler cannot find any other slave process that has extra nodes to solve, thus it realizes that all the work has been finished. The states after this state is that scheduler starts to send termination signals to the Pareto front master and all the slave processes. $StateD$ is that the Scheduling master sends termination messages to each slave process. $StateE$ is that the Scheduling master sends termination message to the Pareto front master. $StateK$ is that the Scheduling master terminates itself.

(3) $StateF \hookrightarrow \{StateG, StateH\} \hookrightarrow StateI$ is that how the Scheduling master handles a message of requiring a tree contributor from an arbitrary slave process when this slave process' tree is empty. $StateF$ is that the Scheduling master receives a tree size request and looks for the biggest tree size in its table. $StateG$ is that when all the tree sizes in its table are 0 and then it sends a -1 as a termination message to the slave process. $StateH$ is that when not all the tree sizes in its table are 0 and then it sends the rank of the slave process that possesses the biggest tree to the message-sending slave process. $StateI$ is that when the scheduling master finishes sending the tree-size-response message to the slave process.

The reason that the scheduler master, in part(2) , needs to collect all the termination acknowledge messages before it sends out the final termination message is further explained in the subsection 8.1.2.3.

8.1.2.3 Slave Slice Solver

The Slave Slice Solver is the solver process to solve the nodes. Generally speaking, it solves all the nodes in its tree and solicit more nodes when its own tree dies out. Figure 8.3 shows the state transition graph of a slave process.

To better elaborate the whole strategy, this state transition graph is divided into 3 parts– (a) solving a node, (b) tree transmitting and (c) solver re-Initiation, and termination.

(A)Solving a node: Brown-colored part in Figure8.6 is the solve-node-state transition graph.

$\{StateG, StateH, StateI, StateJ, StateK, StateL\}$ together serve the purpose of LP-related programming, as in the coarse-grained slave process. Among these states, $StateH, StateI$ are the states in which isWorse message is sent and the response for isWorse is received; StateK is the state in which an addIntegerPoint message is sent; all the other states are the auxilary states for LP calculation. $StateM$ is when a node is solved and then updates the slave's tree size to the scheduling master.

$StateN$ is when the slave process decides whether it will continue to solve nodes in its tree (provided that there are more nodes in tree), in this case it will go back to $StateA$; or requesting a new tree (provided that there are no more nodes in its tree), in this case, it will move to $StateO$. In $StateO$, it will send the tree size request to the scheduling master. After acquiring the scheduling master, if a positive message is received (i.e. a valid rank is received), the slave process will follow the path $StateR \hookrightarrow StateS \hookrightarrow StateT$(which will be discussed later) to request a tree from the slave process of the responded rank; otherwise, the slave process will follow the path $StateP \hookrightarrow StateQ \hookrightarrow StateA$. $StateP$ sets an internal signal $justWait$. $justWait$ means the slave process from now on will not solve any node or request any tree, but rather waits for the finish signal from the scheduling master or handle a tree request from other slave processes. The reason that a slave process might still receive a tree request from other slave processes even if the scheduling master has issued a negative response is that due to certain communication delay, it is possible that slave process A initially got a positive response to talk to process B while B has finished solve all its nodes and also got a negative response message from the scheduing master. Figure 8.5 shows such a senario. Suppose a slave process skips $StateP$ and $StateQ$ and terminates itself, then if another slave process is requiring a tree from this slave process, such a request will never be answered, thus a starvation occurs(in

Figure8.5 process$B$ quits at time$b2$ before the requesting message arrives at time$b3$). Therefore, in $StateQ$ the slave process needs to send the termination acknowledge message, so that all the slave processes will have to wait at the same barrier to finish to prevent a starvation.

(B) Send a tree: the brown-colored part in Figure 8.6 is the tree-transmission-state graph. An ACK is needed so that it won't generate a deadlock when two slave processes are trying to request trees from each other. There is only one case that these state transition graphs won't handle well, that is when all slave processes are finished, and one slave process obtains one node from others, and this node grows to a very large tree late. This case is highly impossible though, given the node number always follows a growing and dying out fashion.

The path $StateA \hookrightarrow StateB \hookrightarrow StateC \hookrightarrow StateD \hookrightarrow StateE \hookrightarrow StateF$ is when a slave process receives a message from another slave process to send a tree. $StateB$ is to split its own tree. The current strategy is to split a tree to two trees of equal size. If the current tree has only one node, then it keeps this node to itself and later sends an empty tree. $StateC$ is to update the split tree size to the scheduling master. $StateD$ is to send an acknowledgement to the tree-requiring slave process. $StateE$ is to actually transfer the tree to the tree-requiring slave process. This transferring is a rather complicated procedure, please refer to subsection 8.1.3.1 for details.

The path $StateR \hookrightarrow StateS \hookrightarrow StateT$ is when a slave process runs out of nodes and receives a positive response from the scheduling master and then requires a tree from the designated slave process (i.e. the tree contributor). $StateR$ is to send the tree requesting message to the designated slave process. $StateS$ is to receive the acknowledgement message from the tree contributor. $StateT$ is to receive a tree from the tree contributor, this procedure is the counter part of tree transferring, which is also explained more in details in subsection 8.1.3.1. The reason an acknowledgement message is needed is that due to the communication delay (something similar to the situation in the Figure 8.5), there is a potential senario in which slave process A and B are requiring a tree from each other, then the algorithm will generate a deadlock. That is the reason why there is a dashed line from $StateS$ to $StateB$, because even when a slave process is waiting to receive a tree, it might still get some tree requests from other slave processes and it needs to handle such messages promptly.

(C) Termination: In Figure 8.3, the path $StateA \hookrightarrow StateU$ is when a slave process receives the termination message from the scheduling master and then it finally quits. The whole program is finished from this point.

8.1.3 Implementation

8.1.3.1 Transmitting a tree

Obviously, a MobbTree instance can not be represented by a simple array, therefore it cannot be transmitted in the primitive data type provided by MPI. Ideally, it would be efficient to pack up a new $MPI\_Datatype$ and transmit the instance of such a type. However, as discussed in subsection 6.3.1.1, the premise of packing up this new datatype is the size of such a type is fixed. This is evidently not the case in the senario of MobbTree, as the size of the transmitted tree is not determined each time. The tree size grows or shrinks during each slave process solving; and even for two trees of the same size, their memory footprint may most likely be different because the EscapeConstraints are different in each node. In conclusion, it is impossible to predifine a size to pack up the tree. So the only feasible way to pack up a tree is to "serialize" a tree using arrays of MPI primitive datatypes. Considering a class that needs to be serialized as a graph(in Graph theory), each vertex on this graph needs to be serialized. There are two issues involved with serialization: serialization of each vertex and in what order to traverse each vertex.

Serialization of each vertex: To serialize a MOBB Tree, all the member field must be serialized. In the design of class MobbTree, there are no self-referenced data members, that is to say there is no cycle in the graph; therefore, if all member fields are serializable, the whole tree can be serialized. MobbNode is the fundamental "vertex" to serialize. MobbNode has 3 member fieds to serialize: a vector of binary variable indices of size m(i.e. m fixed values), a vector of fixed values of size m, and a vector of EscapeConstraints of size N(i.e. N fixed values or the number of all the binary variables in a MOBB problem, which is a constant number unlike m, and $N \geqslant m$). Apparently, the first two elements can be both serialized as integer arrays of size m. To serialize an EscapeConstraints, the most straightforward way is to use one integer array of size N to record it. However, assume for each node that it has m fixed-value binary variables and n EscapeConstraints of size N, the total communciation memory cost would be $m \times 2 + n \times N$ integers. Considering the upper bound of n is 2^N and $N \geqslant m$, it would be more efficient if such a cost could be reduced. A better strategy is to represent a fixed-value-series $x_1, x_2, x_3, ..., x_N$ $(x_i \in B)$using a single number X. Such a strategy is similar to convert a binary variable to a decimal variable. Algorithm 11 and Algorithm 12 describes how to code such an EscapeConstraint to X and how to convert it back. Given the value of number X grows exponentially with N, it is not feasible to represent X using data type of integer, instead, we should use double. Using such a strategy, the memory cost of communication is significantly reduced to $m \times 2 + N$, which is nearly n

times better than the original proposal. Given the fact that the upper bound of n is 2^N, such a reduction is very desired.

How to traverse each vertex: To traverse each vertex, there are two ways:

Algorithm 11 code(bin)

Require: *bin* is a binary number
Ensure: decimal representation of *bin*
1: $bitNum \leftarrow$ *bit number of bin*
2: $dec \leftarrow 0$
3: **for all** i such that $0 \leq i \leq (bitNum - 1)$ **do**
4: $dec \leftarrow bin.at(i+1) \times 2^i$ ▷ bin.at(x) refers to the xth bit of bin
5: **end for**
6: **return** *dec*

Algorithm 12 decode(dec)

Require: *dec* is a decimal number
Ensure: binary representation of *dec*
1: $bin \leftarrow 0$ ▷ *bin* is implemented as a vector
2: $bitNum \leftarrow$ *bit number of bin*
3: $bigNum \leftarrow 2^b itNum$
4: $residue \leftarrow dec$
5: **for all** i such that $0 \leq i \leq (bitNum - 1)$ **do**
6: $tmp \leftarrow residue \times 2$
7: **if** $tmp \geqslant bigNum$ **then**
8: $bin.set(i+1, 1)$ ▷ $bin.set(x, y)$ is to assign the value of y to the xth bit of *bin*
9: $tmp \leftarrow (tmp - bigNum)$
10: **else**
11: $bin.set(i+1, 0)$
12: **end if**
13: $residue \leftarrow tmp$
14: **end for**
15: **return** *bin*

depth-first and breadth-first. Generally speaking, serialization adopts depth-first.In practice, a different strategy(i.e. breath-first) is used, there are two reasons to do so: (1) MobbTree is composed of MobbNode instances, MobbNode class is coded in both integer and double types. It is only economical to transmit integers as MPI_Int and doubles as MPI_Double. The maneuver to only use MPI_Double to code a MOBBNode instance is not desirable. Therefore, if one needs to serialize MobbNode using different data types and in each sending/receiving round only one data type can be specified in MPI specifi-

cation, depth-first is not the way to go. (2)In the current implementation, a tree-transmission is always finished within one round, i.e. there is no need to chop a tree into chunks to transmit. Yet, in the future research work, a tree might get so large that it exceeds the MPI limit for finishing one round of tranmsission. Then, a tree needs to be divided into piece and later assembled. In such a case, depth-first will require the algorithm to look through each MobbNode to figure out how to best divide the tree. Yet, using breath-first strategy only requires to divide serveral MPI primitive data type arrays, which is considerably faster.

After discussing how to serialize a MobbTree, now it comes to how to transmit the serialized data using MPI. It is clear that three key elements(indices and values of FVB(Fixed Value Binary), EscapeConstraints) need be transmitted. So the program first needs to concatenate these three elements in each MobbNode. Secondly, the size of each element is required to be known. Therefore, the transmitting process is clearly divided into 5 phases: (1) size of FVB(in fact, the sliceID of this tree is piggy-packed too, for the reason, please refer to subsectionsec:reinit), (2) indices of FVB, (3) values of FVB, (4) size of EscapeConstraint, (5) values of EscapeConstraints. The above discussion can ensure a valid transmittion of a tree. However, since the selection rule of each node is based on their z value. So to ensure the consistency with the sequential program, each node's z need be transmitted. If in the future, a better selection strategy is adopted, the data structure related to that strategy need be transmitted too.

Figure 8.7 shows the handshake of the sending/receving processes. When the receiving part receives these elements, it can re-construct the MobbTree, the procedure of which is a reverse of serialization, i.e. the de-serialization.

8.1.3.2 Re-initiation

After a slave process receives a tree from another slave process. It needs to reinitiate its CLP solver to operate on the tree, because the tree is only valid given its binding sliceID. Therefore, throughout the calculation, one slave process maintains the same MPI rank, but its sliceID may vary from time to time. To re-initiate the CLP solver is straight-forward: firstly, remove the old upper/lower bound and objective function; secondly, add the new upper bound, lower bound and the objective function.

8.2 Test and Analysis

8.2.1 Execution time and Speed up

Figure 8.8 is the execution time when CPU number grows from 8 to 20. As in the coarse-grained implementation, it has shown the execution time drops very quickly; however, as discussed before, it is also because when the slice number is different, the sequential algorithm has very different execution time. To better evaluate the performance, we shall use Amdahl's law.

Figure 8.9 is the speed up when CPU number grows from 8 to 20. Clearly, a very high speed up is achieved. When CPU number equals to 8, 10, a super-linear speed up is even achieved. That is because sequential programming, the program needs to load the data into cache in different rounds; while in the parallel programming, the data can be fit into cache more efficiently. We can see the speed up maintains very high even when CPU number gets larger.

8.2.2 Workload Distribution

The reason that the speed up is very high in fine-grained implementation is because the workload is evenly distributed. Figure 8.10 is the work load distribution of a testbench when $CPU = 8$. Compared to the work load distribution in the coarse-grained implementation (see Figure 7.5), It is clearly seen that the workload distribution is almost evenly distributed. Therefore, it leads to a very big improvement compared to the coarse-grained implementation.

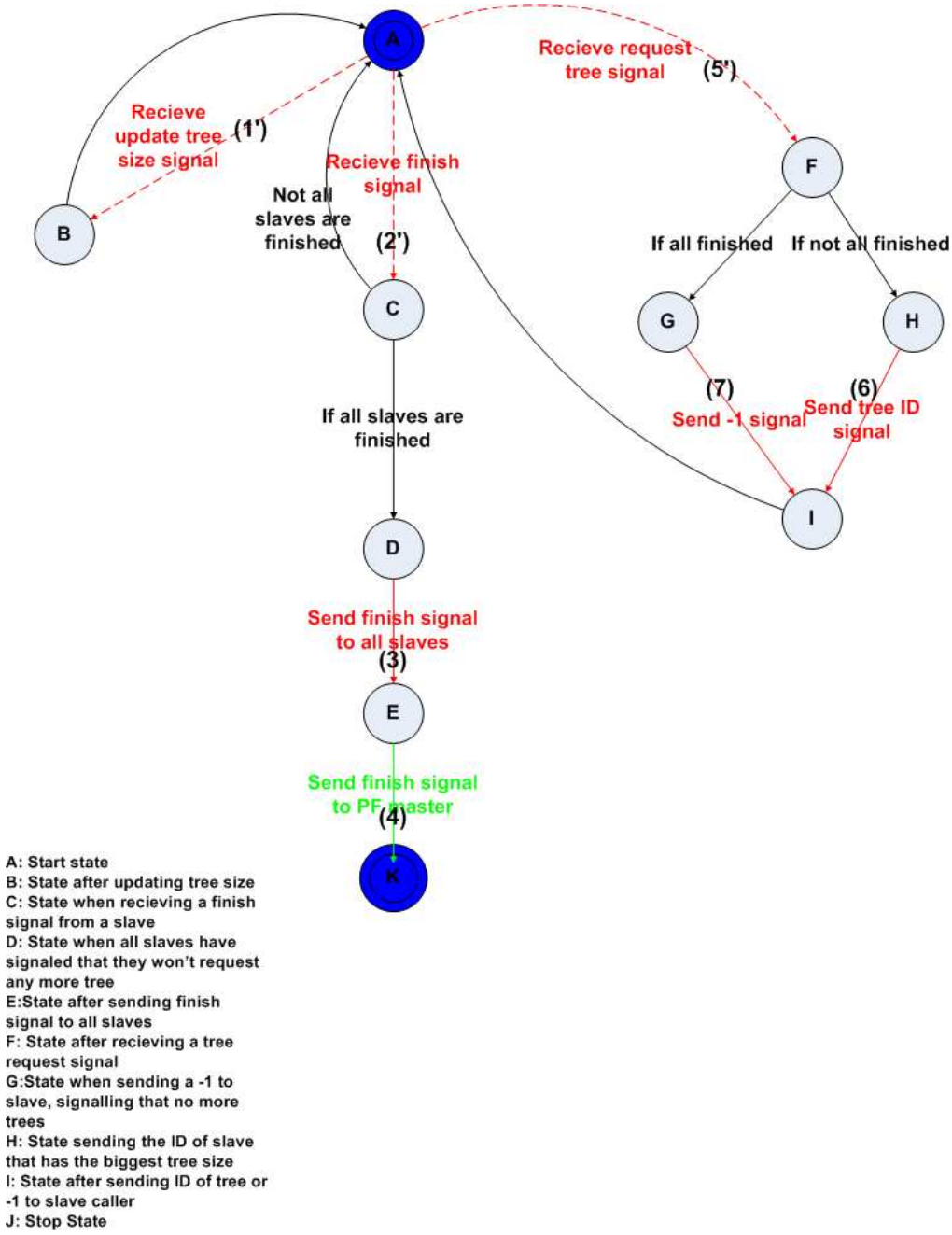

Figure 8.2: State transition graphs for Scheduling Master

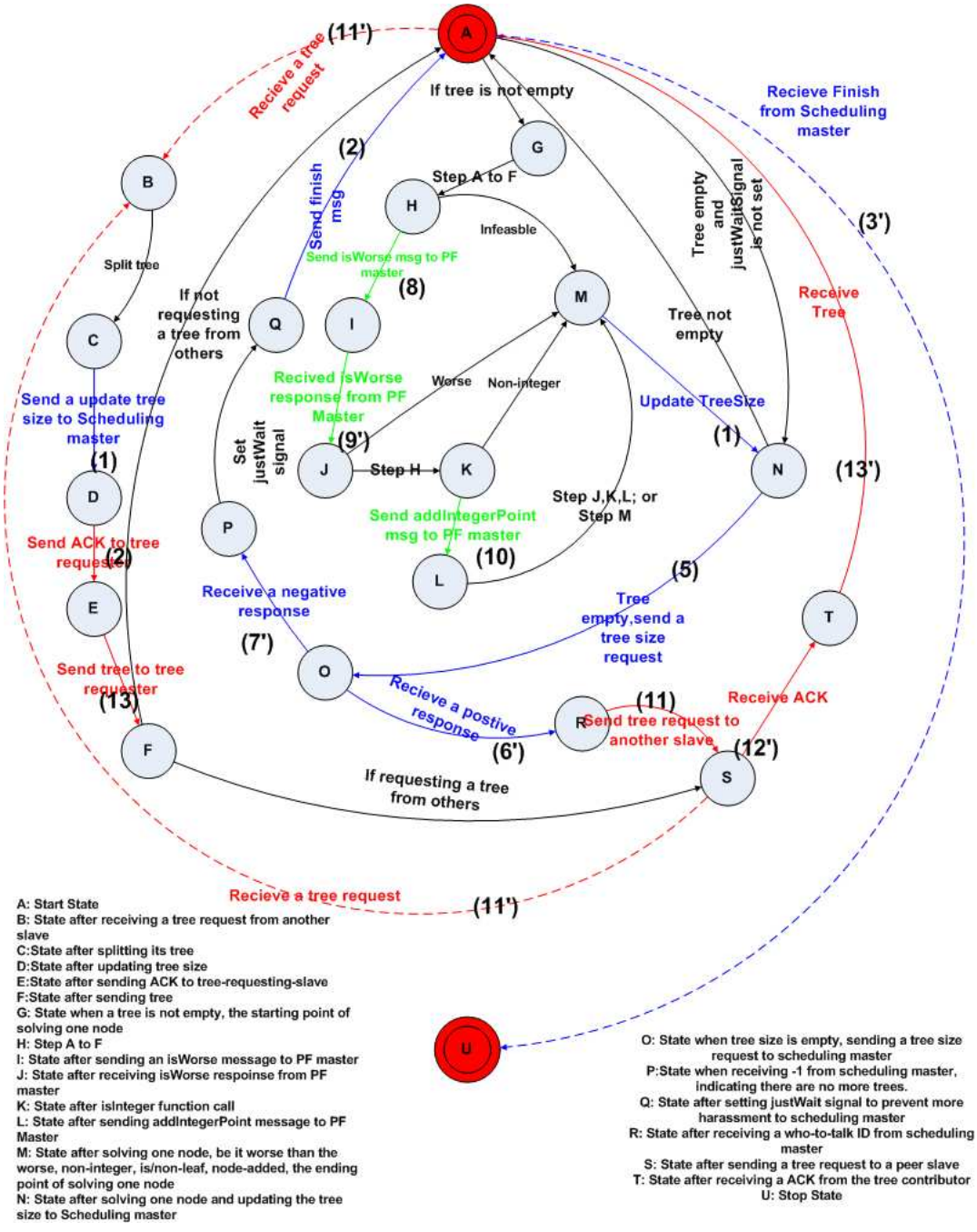

Figure 8.3: State transition graph for Slave process

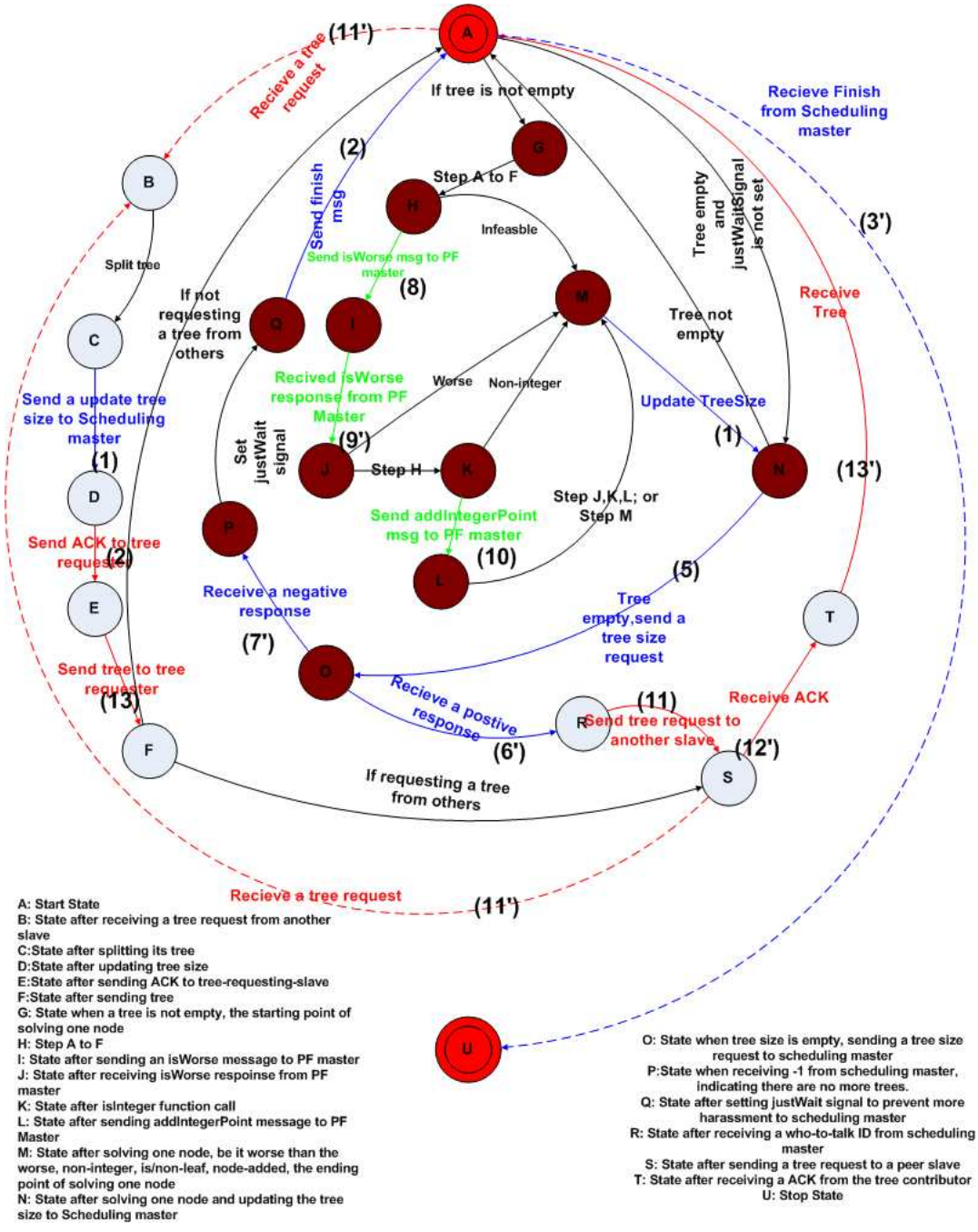

Figure 8.4: State transition graph for solving a node in Slave process

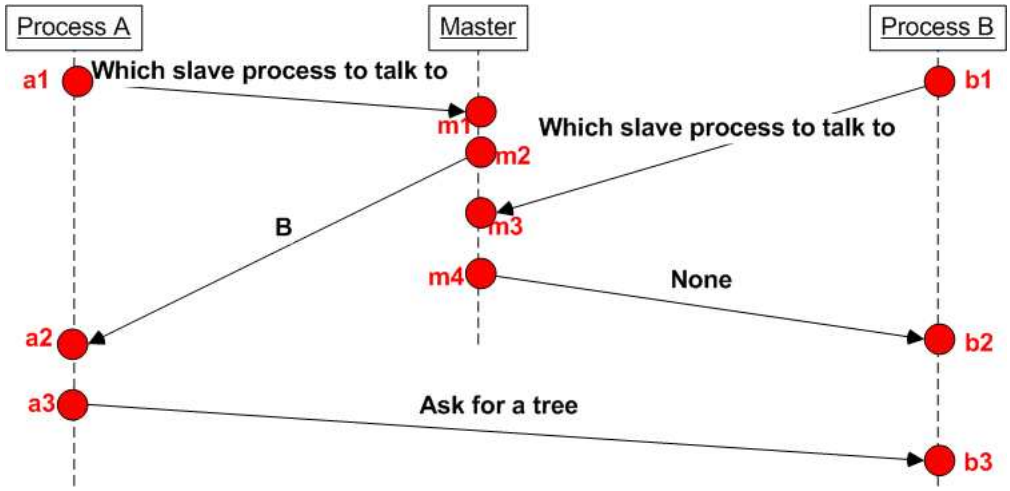

Figure 8.5: A Starvation senario

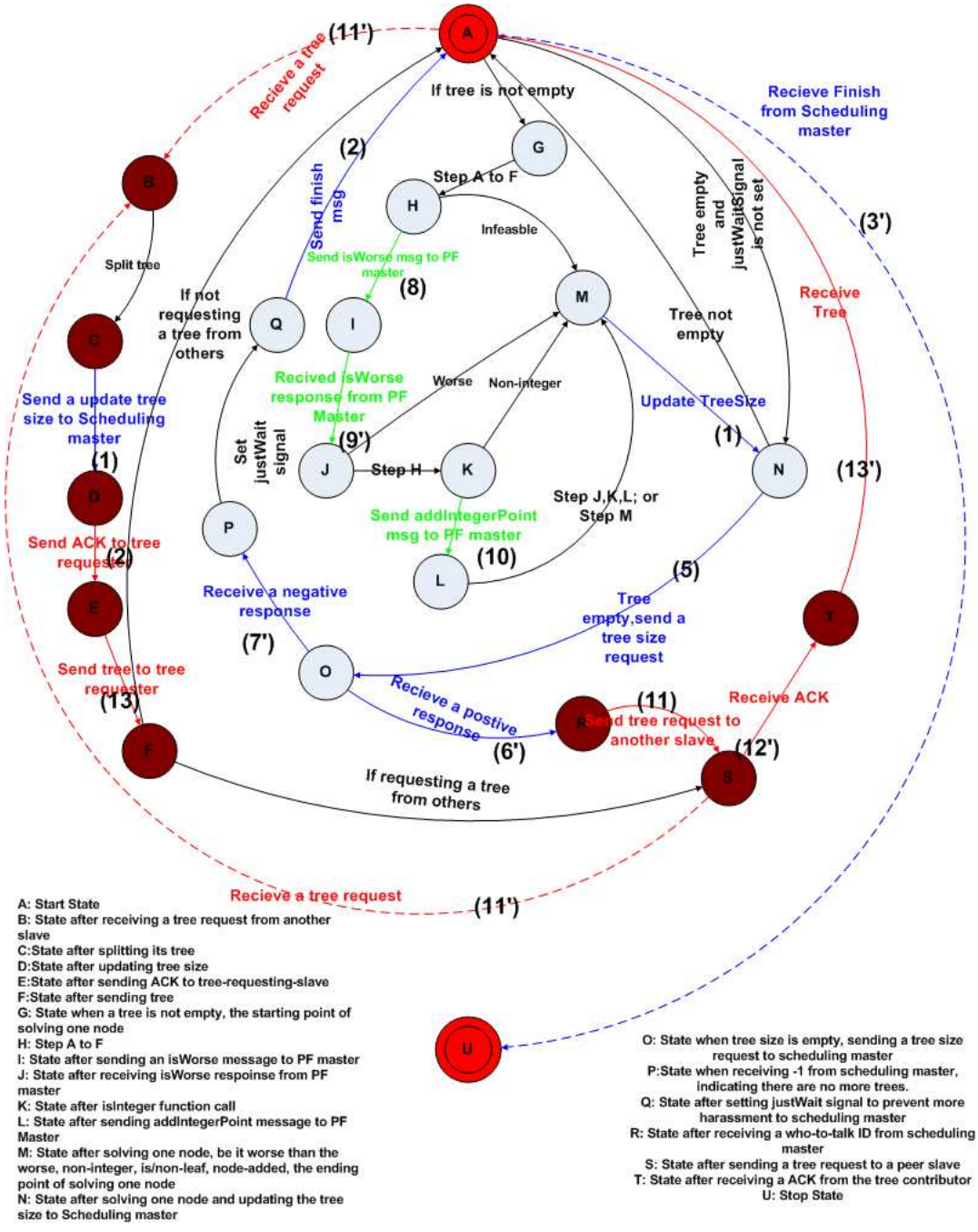

Figure 8.6: State transition for sending a tree

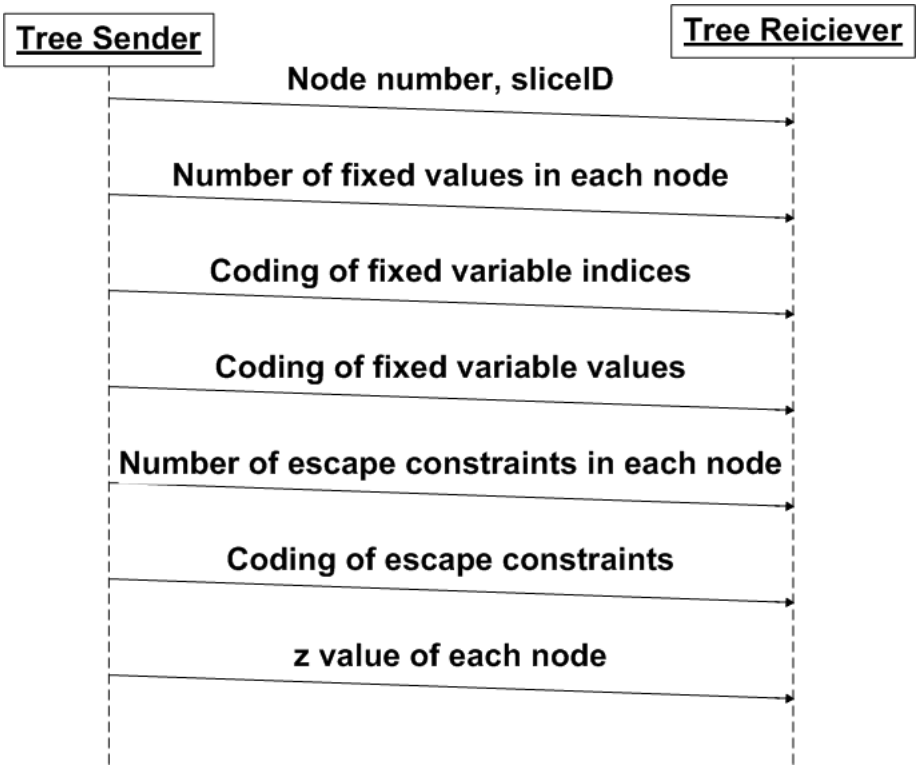

Figure 8.7: Handshakes of sending/receiving a tree

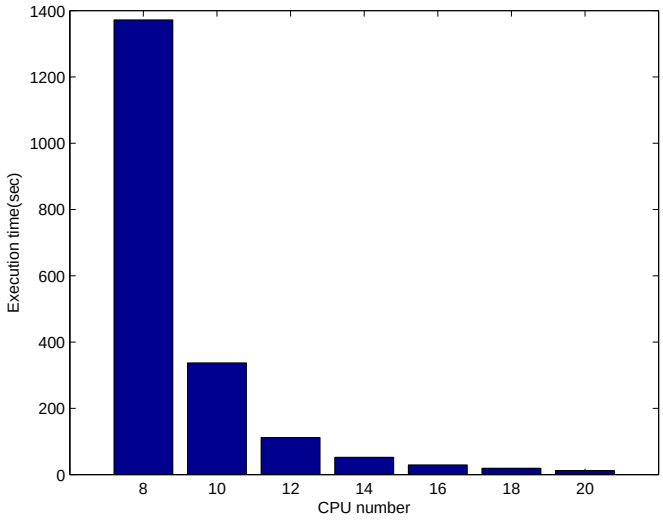

Figure 8.8: Fine-grained Execution Time

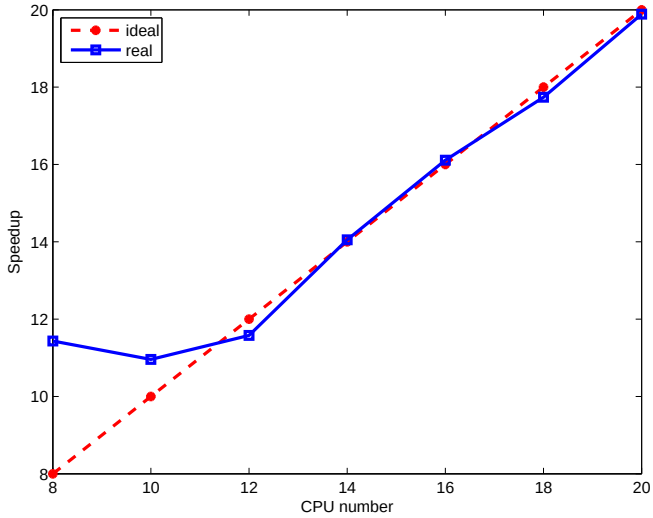

Figure 8.9: Fine-grained Speed-up

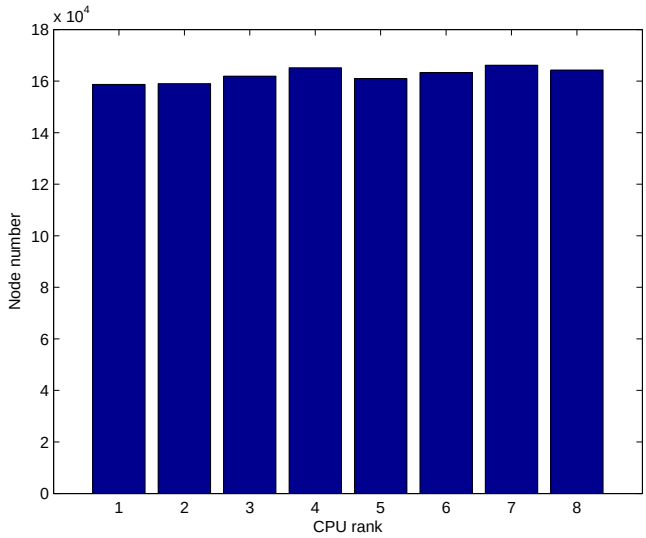

Figure 8.10: Fine-grained work load

Chapter 9

Software architecture of this project

After describing the theories, data structures, and algorithms. This chapter will present the software architecture of this project. In fact, this software architecture was designed before all the actual coding was started. However, it is only sensible to present it here after all the theories and jargons are explained in this thesis. Figure 9.1 is the UML [14] graph of the whole project. Only the core parts of the project are shown in this UML graph.

9.1 Collaboration between each class

The whole project is mainly divided to three major parts: *SliceSolver* group(*SliceSolver* related classes), *ParetoFront* group(*ParetoFront* related classes) and *XXXScheduler* clique(class $SeqMobbScheduler$, class $ParMPICoarseScheduler$, class $ParMPIFineScheduler$).

SliceSolver group is the key part of the whole algorithm. class *SliceSolver* contains an instance of class $MobbTree$, an instance of class $StatisticsTracker$(to keep record of several key estimator data) and an instance of class $ConstraintsHelper$ to calculate upper/lower bound and objective function. Class *SliceSolver* is also

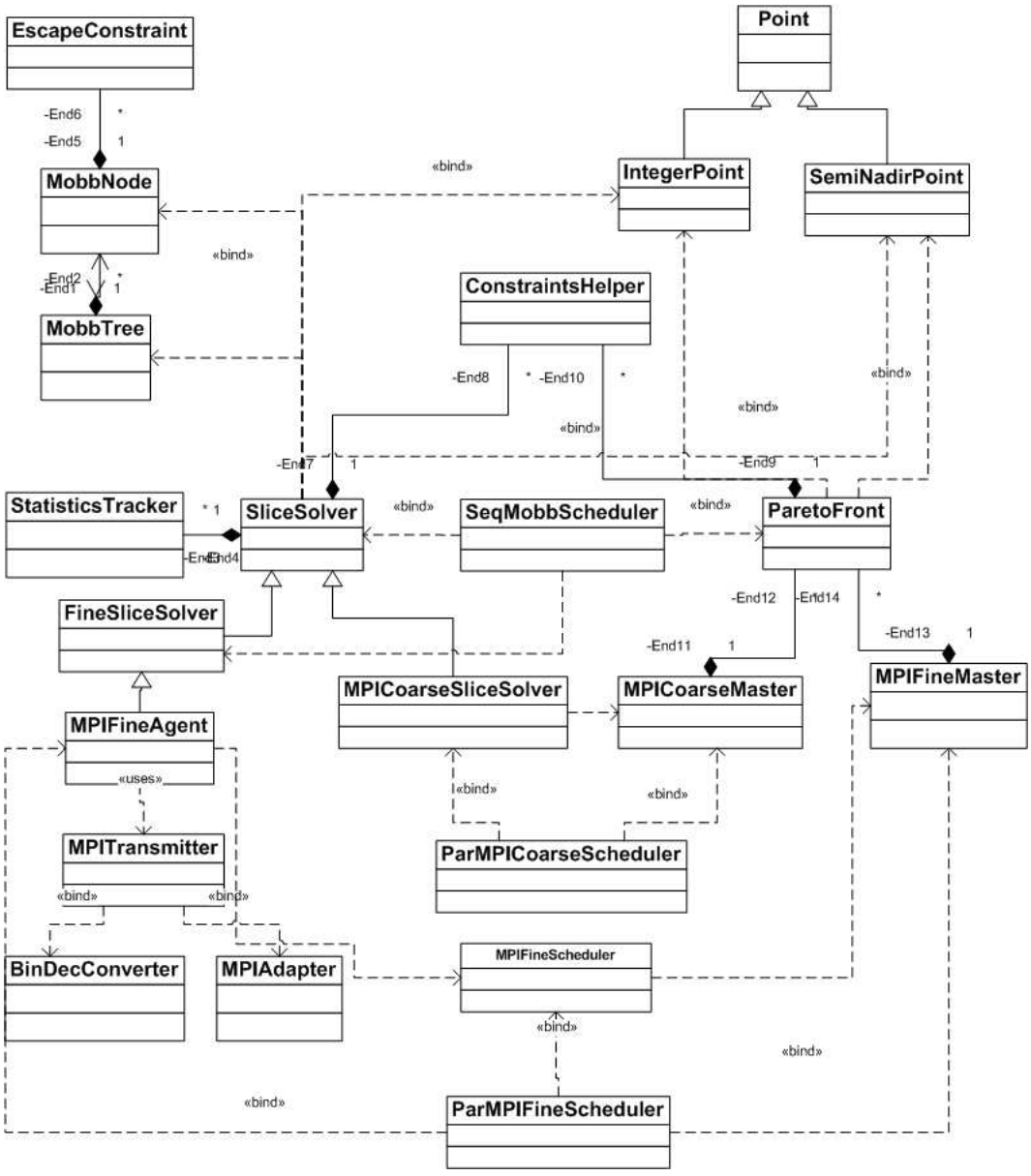

Figure 9.1: UML of this project

the parent class $MPICoarseSliceSolver$ and class $MPIFineAgent$, which are both the solver class in parallelization implementation. For class $MPIFineAgent$, it needs to resort to class $MPIAdapter$ and class $MPITransmitter$ to transmit trees.

$ParetoFront$ group is the part in which $ParetoFront$ related algorithms are implemented. class $ParetoFront$ has an instance of $ConstraintsHelper$ to calculate each slice's upper/lower bound and objective function. In parallelization implementation, class $MPICoarseMaster$ and class $MPIFinePFMaster$ both contain an instance of class $ParetoFront$ to facilitate calculation. Class $MPIFineScheduler$ is a special class to faciliate sequential simulation of fine-grained parallelization.

$XXXScheduler$ group refers to class $SeqMobbScheduler$, class $ParMPICoarseScheduler$, and class $ParMPIFineScheduler$. All these classes are the entry to the whole program, which initiates $SliceSolver$ group and $ParetoFront$ group and then orchestras them to work.

9.2 Object-Oriented Design [15]

This project is done in $C++$, an OO language, so there are some OO features used. Among OO features, Inheritance and encapuslation are the mostly used. Inheritance refers to inherit attributes and behaviors from their parent classes, and can introduce their own methods by overloading the parent methods. Encapsulation refers to conceal the functional details of a class from objects that communicate with it, therefore to guarantee the used class is not modified. Encapsulation vs. Inheritance is therefore also called Is-a vs. Has-a relationship.

9.2.1 Inheritance

Throughout this project, there are two places where inheritance is used. One place is for the class $Point$ and its derived classes $IntegerPoint$ and $SemiNadirPoint$. As discussed in chapter 4, $Point$ is to keep record of the two objectives of MOBB, namely $z1$ and $z2$, and also to maintain the numerical stability. So once class $Point$ is implemented and tested, the other two classes only need very little modification.

The other place for inheritance in this project is the class $SliceSolver$, and

its derived classes $MPICoarseSliceSolver$ and $MPIFineAgent$. $SliceSolver$ is the core class of the whole algorithm, it implements the core algorithm. So once $SliceSolver$ is coded and tested, one only needs to add the parallelization implementation in its derived classes. Inheritance turns out to be a very powerful tool in this case– even in the class $MPIFineAgent$, only a number of lines of code need to be added. In the future work, if one decides to use OpenMP (or any other parallel tools supported by C++) one can simply inherit this class and add the parallelization related code.

9.2.2 Encapsulation

Encapsulation is also a very powerful tool. Compared to inheritance, encapsulation does not inherit any class, rather, it uses another class as its member field. Encapsulation is used when a class has a very close relationship with another class, but not a direct deriviation of another class. So it will not have any effect on the encapsulated class but only use the existant interface. In this project, both $MPICoarseMaster$ and $MPIFinePFMaster$ have an instance of $ParetoFront$, but neither of them is a derived class of $ParetoFront$. It is because that to use the facility offered in $ParetoFront$, one does not need to change anything; all the existent interfaces in class $ParetoFront$ are sufficient for later parallelization.

9.3 Test and Parallel simulation

The parallel implementation is considerably more difficult than its sequential counter-part due to the fact that the total execution sequence is not deterministic. To lower the risk of generating bugs in parallel implementation, one needs to first rigorously test all the algorithm related code. For each above class XXX, a $TestXXX$ is accordingly implemented to test the class so that it will have the desired result. Once the algorithm is tested to work, we can make some parallel simulation in the sequential program before actually implementing the parallel code. In this project, $SeqMobbScheduler$ is such a class, in which several methods are implemented to simulate parallelization–(1) executeRandom(), in this method, the $SeqMobbScheduler$ will randomly choose a $SliceSolver$ with non-empty tree to solve a node. (2) executeFine(), in this method, $SeqMobbScheduler$ will randomly choose a $SliceSolver$, if it has a non-empty tree, then this solver solves a node, otherwise, it will solicit a tree from another $SliceSolver$. Evidently, executeRandom() is a parallel simula-

tion of Coarse-grained parallel implementation, while executeFine() is a parallel simulation of Fine-grained parallel implementation. To simulate parallelization could help to reduce the risk of algorithm fault. For example, in Fine-grained parallel implementation, re-initialization and tree-transmitting are both delicate issues. To simulate this, one can make sure all the OR-related parts are working fine, which would leave programmer to only deal with the parallelization issue in later parallelization work.

Chapter 10

Conclusion and future work

10.1 Conclusion

This thesis project has finished several tasks:

- Implement the sliced-MOBB algorithm.Test the algorithm and show it is generally much better than the traditional brute force algorithm and slicing the feasible region gives much better bounding.

- Implement a coarse-grained MOBB parallelization program in MPI. Test the implementation and shows that it gives reasonable performance, but it cannot achieve better performance even when CPU number gets larger than a certain threshold.

- Implement a fine-grained MOBB parallelization program in MPI. Test the implementation and show that it gives a very good performance, in some cases, it even achieves a super-linear speedup. Such a parallelization algorithm can be applied to any general branch-and-bound-like program.

- Overall, parallel MOBB algorithm displays a great improvement of solving MOMIP problems, the excellent performance is a combination of both better bounding scheme in MOBB and also a well-designed parallel algorithm. Such a result might lead to a major breakthrough in OR research.

10.2 Future work

- $z1$ can be modeled not only by binary variables but also a mix of binary and continuous variables.
- Problem domain can be improved to more than 2 objectives.
- Branch and cut technology may be applied to MOBB algorithm so that the sequential program could even achieve better performance.
- Parallelization can be implemented in OpenMP, so several problems such like transferring a tree would be significantly reduced; as in OpenMP, one only need to transfer a pointer to the tree.

$$\begin{aligned}
&\textit{Minimize} && \{z\} && && (10.1)\\
&\textit{Subject to} && && && (10.2)\\
&z && = && cx && (10.3)\\
&Ax && \geqslant && b && (10.4)\\
&x && \geqslant && 0 && (10.5)
\end{aligned}$$

$$\begin{aligned}
&\textit{Minimize} && \{z1, z2\} && && (10.6)\\
&\textit{Subject to} && && && (10.7)\\
&z1 && = && \sum_i c_i^1 x_i, \forall i, \textit{where } x_i \in B && (10.8)\\
&z2 && = && \sum_i c_i^2 x_i, \forall i && (10.9)\\
&Ax && \geqslant && b && (10.10)
\end{aligned}$$

Bibliography

[1] Frederick S. Hiller and Gerald J. Libeberman(2005) *Introduction to Operations Research,* eighth edition McGraw-Hill ISBN-10: 007-123828-X

[2] Wolsey, L.A. (1998) *Interger Programming,* Jone Wiley and Sons, Inc. ISBN-10: 0-471-28366-5

[3] Barry Wilkinson , Michael Allen (2004) *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers,* second edition, Prentice Hall, ISBN-13: 978-0131405639

[4] Matthias Ehrgott(2005) *Multicriteria Optimization,* second edition, Springer, ISBN-13: 978-3540213987

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2001) *Introduction to Algorithms,* second edition, The MIT Press, ISBN-13: 978-0262032933

[6] COmputational INfrastructure for Operations Research *http://www.coin-or.org/*

[7] Branch and Bound Tree *http://people.brunel.ac.uk/ mastjjb/jeb/or/ipt2.gif*

[8] Amdahl's Law *http://en.wikipedia.org/wiki/Amdahl's_law*

[9] Parallel Programming Category *http://en.wikipedia.org/wiki/Parallel_computing,*

[10] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra (1998) *MPI: The Complete Reference (Vol. 1),* second edition, The MIT Press, ISBN-13: 978-0262692151

[11] The National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. *http://ci-tutor.ncsa.uiuc.edu*

[12] Barbara Chapman, Gabriele Jost, Ruud van der Pas(2007) *Using OpenMP: Portable Shared Memory Parallel Programming,* The MIT Press, ISBN-13: 978-0262533027

[13] Gregory R. Andrews(1999) *Foundations of Multithreaded, Parallel, and Distributed Programming,* Addison Wesley, ISBN-13: 978-0201357523

[14] Paul R. Reed(2001) *Developing Applications with Java and UML,* Addison-Wesley Professional, ISBN-13: 978-0201702521

[15] Robert Lafore(2001) *Object-Oriented Programming in C++,* 4th Edition, Sams, ISBN-13: 978-0672323089

[16] Faires, J.D. & Burden, R.(2002) *Numerical Methods,* seconde edition, Brooks/Cole Publishing Company ISBN-13: 978-0534407612

[17] MPS(Mathematical Programming System) http://en.wikipedia.org/wiki/MPS_%28format%29