MPBO
A Distributed Pseudo-Boolean Optimization Solver

Nuno Miguel Coelho Santos

Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering

Examination Committee
Chairperson: Prof. José Manuel da Costa Alves Marques
Supervisor: Prof. Paulo Ferreira Godinho Flores
Co-supervisor: Prof. José Carlos Alves Pereira Monteiro
Member of the Committee: Prof. Vasco Miguel Gomes Nunes Manquinho

May 2013
Abstract

Parallel computing has been the subject of intensive research in the last decade and is increasingly being chosen as a solution for developing applications that require high computational power, such as the Boolean Satisfiability (SAT) and Pseudo-Boolean Optimization (PBO) problems. Research in SAT solvers has obtained relevant results in the last years, achieving significant reductions in execution times. Unfortunately, hard instances of boolean satisfiability require large computational power and even efficient SAT solvers take huge execution times to obtain their solution. Therefore, SAT solvers adaptation to parallel computing systems began to be the subject of considerable research and there already exist several distributed versions of popular SAT solvers. However, the absence of distributed solvers for the Pseudo-Boolean Optimization problem is notorious. This work intends to contribute and encourage the research into distributed solutions to solve the PBO problem. The goal of this work is to propose a distributed Pseudo-Boolean Optimization Solver, named MPBO solver, based on MPI (Message Passing Interface) and focused on an efficient search space partition, more specifically the partition of the problem optimization search space. The proposed solver achieved significant reductions in the time to solve hard PBO instances, when compared to the MiniSat+ and pwbo solvers.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>5</td>
</tr>
<tr>
<td>List of Figures</td>
<td>7</td>
</tr>
<tr>
<td>List of Tables</td>
<td>9</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>9</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>11</td>
</tr>
<tr>
<td>2 Problem Definitions</td>
<td>13</td>
</tr>
<tr>
<td>2.1 Boolean Satisfiability Problem</td>
<td>13</td>
</tr>
<tr>
<td>2.2 Pseudo-Boolean Problem</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 PB-constraint Properties</td>
<td>14</td>
</tr>
<tr>
<td>2.2.2 Normalization of Constraints</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Pseudo-Boolean Optimization Problem</td>
<td>15</td>
</tr>
<tr>
<td>3 Review of Solvers</td>
<td>17</td>
</tr>
<tr>
<td>3.1 SAT Solvers</td>
<td>17</td>
</tr>
<tr>
<td>3.1.1 Solver Operations</td>
<td>17</td>
</tr>
<tr>
<td>3.1.2 Davis-Putman Backtrack Search</td>
<td>18</td>
</tr>
<tr>
<td>3.1.3 Conflict Analysis</td>
<td>19</td>
</tr>
<tr>
<td>3.1.4 Adaptive Branching</td>
<td>20</td>
</tr>
<tr>
<td>3.1.5 Two Watched Literals Unit Propagation</td>
<td>20</td>
</tr>
<tr>
<td>3.1.6 Restarts</td>
<td>21</td>
</tr>
<tr>
<td>3.2 Pseudo-Boolean Optimization Solvers</td>
<td>21</td>
</tr>
<tr>
<td>3.2.1 Generic ILP Solvers</td>
<td>21</td>
</tr>
<tr>
<td>3.2.2 Specialized 0-1 ILP Solvers</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Parallel and Distributed Solvers</td>
<td>24</td>
</tr>
<tr>
<td>3.3.1 Distributed SAT Solvers</td>
<td>24</td>
</tr>
<tr>
<td>3.3.2 Parallel Pseudo-Boolean Optimization Solvers</td>
<td>26</td>
</tr>
<tr>
<td>4 MPBO Solver Planning</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Solver Approach</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Solver Architecture</td>
<td>29</td>
</tr>
<tr>
<td>4.2.1 MPI</td>
<td>30</td>
</tr>
<tr>
<td>4.2.2 MiniSat API</td>
<td>30</td>
</tr>
<tr>
<td>4.2.3 MiniSat+ API</td>
<td>32</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Boolean formula in CNF. ................................................................. 13
2.2 Example of a PB-constraint. ............................................................. 14
2.3 Example of a PB-constraint translation into CNF clauses. .................... 15
2.4 Example of a PB-constraint normalization. ........................................ 15
2.5 Formula of a PBO problem. ............................................................. 16
3.1 Example of a DIMACS file content. .................................................. 17
3.2 Pseudo code of a variation of the Davis-Putnam backtrack search algorithm. 18
3.3 Master/Slave topology diagram. ....................................................... 24
3.4 Decision Tree example of a problem with variables \{y, p\}. .................... 25
4.1 Example of possible initial task assignments when solving a problem with an objective function \(f(x) = x_1 + 2x_2 + 20x_3\). .................................................. 27
4.2 Optimization process example for the scenario of Figure 4.1. ................. 28
4.3 Architecture diagram of the MPBO solver. .......................................... 29
4.4 List of the MiniSat main API functions. ............................................ 30
4.5 Pseudo code of the MiniSat Search function. ....................................... 31
4.6 List of the MiniSat+ main API functions. .......................................... 32
4.7 MiniSat+ converter heuristic code. .................................................. 33
4.8 MiniSat+ optimization process code. ................................................ 34
5.1 Example of a routine to pack a dynamic structure. ................................ 38
5.2 Pre-normalization SolverInput structure. ........................................... 40
5.3 Message packet in the pre-normalization approach. ............................. 41
5.4 Pos-normalization SolverInput structure .......................................... 42
5.5 Message packet in the pos-normalization approach ............................... 43
5.6 List of the message type tags. ....................................................... 44
5.7 Task response message packet. ...................................................... 45
5.8 Pseudo code of the Master process. ................................................ 46
5.9 Code of the closerMinimum function. .............................................. 48
5.10 Code of the closerGoalValue function. .......................................... 49
5.11 Code of the NoLeftCoeffApproach function. .................................... 50
5.12 Code of the LeftCoeffApproach function. ........................................ 51
5.13 Example of the task assignments performed by the two approaches, using 5 workers and an objective function \(f(x) = 2x_1 + 5x_2 + 10x_3 + 100x_4\). .................. 52
5.14 Worker Pseudo code. ................................................................. 53
5.15 Function to check stop messages reception. ..................................... 56
5.16 Thread code to check stop messages reception. ................................ 56
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.17</td>
<td>Passive target RMA management.</td>
<td>57</td>
</tr>
<tr>
<td>5.18</td>
<td>Stop check routine code.</td>
<td>58</td>
</tr>
<tr>
<td>6.1</td>
<td>Execution times obtained in the BIGINT instances, using MiniSat+ and the MPBO solvers.</td>
<td>62</td>
</tr>
<tr>
<td>6.2</td>
<td>Speedups obtained by the MPBO solver, when solving the BIGINT instances.</td>
<td>63</td>
</tr>
<tr>
<td>6.3</td>
<td>Execution times obtained in the SMALLINT instances, with MiniSat+, pwbo and MPBO solvers.</td>
<td>64</td>
</tr>
<tr>
<td>6.4</td>
<td>Average number of task assignments when solving the BIGINT instances.</td>
<td>65</td>
</tr>
<tr>
<td>6.5</td>
<td>Average number of task assignments, excluding lower bound tasks, when solving the BIGINT instances.</td>
<td>65</td>
</tr>
<tr>
<td>6.6</td>
<td>Average idle time spent by workers, when solving the BIGINT instances.</td>
<td>66</td>
</tr>
</tbody>
</table>
List of Tables

6.1 List of problem instances that compose the benchmark suite. ............... 60
A.1 Solver execution times and speedups in the unsatisfiable instances. ............ 75
A.2 Solver execution times and speedups in the satisfiable instances. ............... 76
A.3 Measured tasks during the satisfiable BIGINT instances execution, using the simple assignment approach (Part 1). .................................................. 77
A.4 Measured tasks during the satisfiable BIGINT instances execution, using the simple assignment approach (Part 2). .................................................. 78
A.5 Measured tasks during the satisfiable BIGINT instances execution, using the optimized assignment approach (Part 1). ................................................. 79
A.6 Measured tasks during the satisfiable BIGINT instances execution, using the optimized assignment approach (Part 2). ................................................. 80
A.7 Wall times spent, in seconds, in the task assignment mechanism, during the satisfiable benchmark tests. ................................................................. 81
A.8 Average wall time (in seconds) spent by the workers in idle state, during the satisfiable instances execution. ................................................................. 82
A.9 Peak Memory obtained in the satisfiable instances, using the optimized task assignment approach. ................................................................. 83
A.10 Peak Memory obtained in the satisfiable instances, using the simple task assignment approach. ................................................................. 84
A.11 Execution times obtained in the pwbo, MPBO and MiniSat+ solvers, in all SMALLINT instances. ................................................................. 85
B.1 Full name of the benchmark instances (Part1). ........................................... 87
B.2 Full name of the benchmark instances (Part2). ........................................... 88
Chapter 1

Introduction

The well known *Boolean Satisfiability* (SAT) and *Pseudo-Boolean Optimization* (PBO) problems gained significant attention in the last years, due to their possible application in many domains, such as software and hardware verification. Since then, several algorithmic solutions have been proposed to solve both problems and many of them proved to be very efficient when solving several instances of the problems. Many SAT algorithmic solutions (known as SAT solvers), for instance GRASP [33] and Chaff [31], contributed with several techniques to improve the SAT resolution. Due to the relevant results on SAT research, PBO researchers embraced the SAT solvers efficiency and proposed efficient ways to solve PBO instances by extending SAT solvers to handle them.

Although SAT solvers are becoming more sophisticated to reduce their execution time, adopting improvement techniques such as *clause learning*, *adaptive branching*, *restarts* and *non-chronologically backtracks*, the demand for more computational power led SAT researchers to explore solutions taking advantage of parallel computing systems. Distributed systems, such as *clusters* and *grids*, are a popular type of parallel computing environments and are target of great adoption in the parallel developers community, since they allow the use of several remote resources, connected through a network, and enable the execution of distributed algorithms that can divide a problem for a concurrent resolution in such resources. Many popular SAT solvers have been migrated to such environments, adopting the well known *Task Farm* approach, which proved to be a successful solution to partition SAT problem instances for concurrent searches. However, unlike SAT solvers, the migration of PBO solvers to distributed systems has not been well explored yet. Taking such gap in consideration, this work is intended to give one more step for such migration by proposing the MPBO (*Message Passing Pseudo-Boolean Optimization*) solver, an efficient distributed PBO solver that obtains significantly reduced execution times when solving hard problem instances. The MPBO solver was implemented resorting to the industry’s defacto MPI (*Message Passing Interface*) API to allow its portability through the different distributed environments and is based on the core of the well known MiniSat+ [14] solver. The solver presented achieved speedup values of 500 in the time to solve hard PBO instances, when compared to MiniSat+, and speedups of 6 when compared to the parallel pwbo solver [30].

From the analysis of the objective function of a PBO problem, it is possible to calculate an interval where all solution candidates for the problem are included, known as the *optimization interval*, and such interval represents the optimization search space that must be explored. The approach behind the MPBO solver engine is the partition of the optimization search space by the available resources in a distributed environment, to be explored concurrently. Based on
the well known Task Farm approach, the solver is composed by two entities. The workers, which are responsible to computed the assigned tasks, and the master which manages the workers through the computation of the problem. The workers are composed by a mechanism to communicate with the master, to receive tasks and send the calculated results, and the modified core of the MiniSat+ solver to compute the tasks received. The master is responsible to assign the necessary tasks and contains a mechanism to identify the tasks to compute to reach the optimal solution for the problem. Each task corresponds to a specific restriction to the objective function of the problem that must be computed, representing a specific value of the optimization interval, and each task results in a sat or unsat response, according to the satisfiability of the given restricted problem. Through the problem optimization search, the master manages the optimization interval according to the results obtained by the computed tasks and restricts such interval until only one value remains, which is considered the optimal solution for the problem. The solver features two approaches to identify the tasks to assign: a simple approach which assigns the exact values obtained by performing a binary division of the optimization interval, and an optimized approach that assigns objective function solutions as tasks, closer to the values obtained during the binary division. The optimized approach reduces significantly the number of task assignments performed during the execution of the solver and achieves an average speedup gain of 80% than the simple assignment approach.

The following document presents the implementation details and the evaluation performed to the MPBO solver, and it is organized in 7 chapters. The definitions of the problems addressed in this project are presented in Chapter 2. Chapter 3 reviews the popular SAT and PBO solvers, detailing their methodology and improvement techniques. The Chapter 4 describes the architecture and the approach behind the MPBO solver, including some details of the resources used for its implementation. The MPBO implementation process is detailed in Chapter 5, where it is described the implementation details of all the important mechanisms that compose the distributed solver and their improvements through the implementation process. Chapter 6 presents the analysis and the evaluation performed to the proposed solver and in Chapter 7 are specified the conclusions and future work for this work.
Chapter 2

Problem Definitions

This chapter presents the definitions of two well-known problems that are the core of the context of this project, the Boolean Satisfiability and Pseudo-Boolean Optimization problems.

2.1 Boolean Satisfiability Problem

The Boolean satisfiability problem (also known as SAT) consists of determining if a Boolean formula contains at least one variable assignment that evaluates such formula to true, where it is considered satisfiable if such assignment exists. If such variable assignment does not exists, the formula is considered unsatisfiable.

Definition 2.1.1 (Boolean formula) A Boolean formula represents a set of Boolean variables related by Boolean operators (and, or, not) and can be either true or false depending on the variable values.

Definition 2.1.2 (Boolean variable) A Boolean variable is a symbol that might assume one of two values: true or false.

In SAT, such Boolean formulas are typically expressed as a conjunction of clauses (example in Figure 2.1), known as conjunctive normal form (CNF), and for a formula be considered satisfiable in such form, all clauses included in the conjunction must be satisfiable. A clause is satisfiable if at least one of its literals is true. A variable assignment that makes a formula satisfiable is known as model. \((x_1 = \text{true}, x_2 = \text{true}, x_3 = \text{true}, x_4 = \text{true})\) is a possible model for the formula presented in the Figure 2.1.

Definition 2.1.3 (Literal) A literal represents a Boolean variable or its complement. \(x_1\) is a positive literal and \(\neg x_1\) is a negative literal.

Definition 2.1.4 (Clause) A clause is a disjunction (\(\lor\)) of literals.

Definition 2.1.5 (CNF clauses) CNF clauses are clauses related with each other by a conjunction (\(\land\)).

\[
E = (x_1 \lor \neg x_2 \lor \neg x_3) \land (x_1 \lor x_2 \lor x_4)
\]

Figure 2.1: Boolean formula in CNF.
\[ C_0x_0 + C_1x_1 + \ldots + C_nx_n \geq C_{n+1} \]

Figure 2.2: Example of a PB-constraint.

**Definition 2.1.6 (Model)** A model is set of variable assignments that satisfies a given formula.

SAT was the first known example of a *NP-Complete* problem and by definition there is no known algorithm that solves all SAT instances efficiently, due to its exponential worst case complexity [16]. SAT problem is particularly popular because of it has many applications in plenty of other domains, such as electronic design automation (EDA) [21, 28] and Artificial Intelligence (AI) [7]. So there is a high demand and research with the aim of finding fast and efficient SAT Solvers.

### 2.2 Pseudo-Boolean Problem

Due to the complexity of some application domains, such as Digital Filters design [3, 22], there was the need to go beyond purely Boolean representation constraints and use a more complex representation form known as *Pseudo-Boolean constraints* (PB-constraints), a generalization of clauses where each literal has a weight associated. The *Pseudo-Boolean* (PB) problem, also known as *0-1 integer linear programming* (0-1 ILP), consists of determining if a set of PB-constraints are satisfiable or not. Like SAT, if there exists a model which evaluates all PB-constraints to true, the set is considered satisfiable.

#### 2.2.1 PB-constraint Properties

A PB-constraint is an inequality with typically five possible relations (\(\leq, <, =, >, \geq\)), where the left hand side (LHS) of the relation contains the sum of weighted literals (literals associated with integer coefficients) and the right hand side (RHS) contains an integer constant. Figure 2.2 illustrates the PB-constraint structure, where \(x_i\) is a literal and \(C_i\) is an integer coefficient. A coefficient \(C_i\) is activated, under a partial assignment if its corresponding literal \(x_i\) is assigned to true. A PB-constraint is said to be satisfied, under an assignment, if the sum of its activated coefficients on the LHS does not violate the relation with the RHS. In a PB-constraint as Figure 2.2, it will be satisfied if the sum of the activated literals exceeds or is equal to the RHS constant.

As an alternative for checking directly the PB problem satisfiability, PB-constraints can be translated into an equivalent set of CNF clauses. Such technique has the advantage of handling PB problems with SAT approaches, but suffers from some drawbacks. There exist techniques to translate PB-constraints into CNF clauses (addressed in Section 3.2.2.3) and properties such as size of the translation and mapping of implications are aspects that must be taken into account when using the translation techniques. Figure 2.3 is an example of a PB-constraint translation (mentioned in the PBS [4] paper) and it is notorious the size relation between equivalent PB and SAT problems.

#### 2.2.2 Normalization of Constraints

Due to the PB-constraints nature, a set of PB-constraints may include constraints with different relations. Since each constraint relation has its own method to check the satisfiability, processing different relations in a single set may be troublesome and implies a more complex process. So
2. PROBLEM DEFINITIONS

\[ x_1 + x_2 + x_3 + x_4 + x_5 \leq 2 \]

\[ \downarrow \]

\[ \neg x_1 \lor \neg x_2 \lor \neg x_3 \lor \neg x_1 \lor \neg x_2 \lor \neg x_3 \]

\[ \neg x_1 \lor \neg x_3 \lor \neg x_4 \lor \neg x_1 \lor \neg x_3 \lor \neg x_5 \]

\[ \neg x_2 \lor \neg x_3 \lor \neg x_4 \lor \neg x_2 \lor \neg x_3 \lor \neg x_4 \lor \neg x_2 \lor \neg x_3 \lor \neg x_5 \]

\[ \neg x_3 \lor \neg x_4 \lor \neg x_5 \]

Figure 2.3: Example of a PB-constraint translation into CNF clauses.

before processing a PB problem, it is advantageous to define a normal form to all PB-constraints of the problem, to simplify the constraints and reduce the number of relation types to handle. There is a great variety of techniques to reach a normal form and the typical technique sequence (implemented in several solvers) is the following:

1. "=" constraints are replaced by two constraints "\( \leq \)" and "\( \geq \)" , with the same LHS and RHS;
2. "<" and ">" constraints are changed to "\( \leq \)" and "\( \geq \)" , respectively, by decrementing and incrementing the RHS by 1 unit;
3. All constraints are changed to a common type (\( \leq \) or \( \geq \)) by negating all constants, if necessary;
4. Negative coefficients are eliminated by changing the associated literal \( x \) into \( \neg x \) and updating the RHS;
5. Multiple occurrences of the same variable are merged into a single term \( C_ix \) or \( C_i\neg x \);
6. Coefficients greater than the RHS are replaced with the RHS constant;
7. The coefficients of the LHS are divided by their greatest common divisor ("gcd") and the RHS is replaced by \( \lceil \text{RHS/gcd} \rceil \).

In Figure 2.4 it is present an example, from [14], of the normalization sequence described above applied to a PB-constraint.

2.3 Pseudo-Boolean Optimization Problem

Since the objective of the PB problem is to verify the satisfiability of PB-constraints, searching an arbitrary model for a given set of constraints is enough to prove its satisfiability. For some domain fields, the resulting model for a PB problem must satisfy some characteristics besides the satisfiability of the problem and obtaining an arbitrary model may not correspond to their needs. To address such demand, many PB problems often include an objective function, which is

\[ -4x_1 - 3x_2 + 3x_3 \leq 1 \]

\[ 4x_1 + 3x_2 - 3x_3 \geq -1 \quad \text{(common type)} \]

\[ 4x_1 + 3x_2 + 3x_3 \geq 2 \quad \text{(positive coefficients)} \]

\[ 2x_1 + 2x_2 + 2x_3 \geq 2 \quad \text{(trimming)} \]

\[ x_1 + x_2 + x_3 \geq 1 \quad \text{(gcd)} \]

Figure 2.4: Example of a PB-constraint normalization.
\[
\text{minimize} \quad \sum_{i=1}^{N} W_i \cdot x_i \\
\text{subject to} \quad \sum_{i=1}^{N} C_{im} \cdot x_i \geq b_m,
\]

where \( x_i \in \{0,1\} \),
\( W_i, C_{im}, b_m \in \mathbb{N} \)

Figure 2.5: Formula of a PBO problem.

intended to lead the search to obtain the desired model for a PB problem. An objective function is a linear term containing weighted literals, on the same form as a LHS of a PB-constraint, and each model will define a value of the objective function (the sum of the activated literals).

The Pseudo-Boolean optimization problem (PBO) is an extension of the PB problem and is the task of finding a specific model that not only satisfies the problem but also minimizes or maximizes a given objective function. The model which results in the lowest value (minimization) or the highest value (maximization) for the given objective function of a set of PB-constraints is considered the optimal solution. Figure 2.5 presents the mathematical formula of a PBO problem.
Chapter 3

Review of Solvers

3.1 SAT Solvers

In early years, many effective algorithmic solutions have been proposed for solving SAT instances. The most popular type of complete and efficient solvers of today are known as conflict-driven solvers. Conflict-driven solvers are variations of the well known Davis-Putnam (DP) backtrack search algorithm [11] combined with efficient improvement techniques, proposed in GRASP [33] and Chaff [31] algorithms. Techniques such as clause learning, non-chronological backtracking, “two-watched-literals” unit propagation, adaptive branching and restarts are examples of the success in the research on SAT and became a basis on SAT solvers since then. Popular conflict-driven solvers are Minisat [13], Sato [34], PicoSAT [8], Berkmin [18], Glucose [6] and zChaff (implementation of Chaff algorithm).

3.1.1 Solver Operations

The objective of SAT solvers is to determine the satisfiability of a given SAT problem and as described in Section 2.1, a given problem is satisfiable if there exists a model which satisfies it, or unsatisfiable in the absence of such model. For most SAT solvers, the problems must be represented in CNF form (due to the simplicity of checking the clauses satisfiability) and normally include an input format known as “DIMACS CNF”. A DIMACS file consists of a text file representing a CNF expression, where each line represents a clause of the expression. The text file initiates with a line “p cnf x y”, where x indicates the number of variables in the expression and y the number of clauses. The following lines represent the given clauses, where variables are expressed as numbers. The number “0” indicates the termination of a clause. Figure 3.1 presents the DIMACS file for the CNF expression in Figure 2.1.

The output of a SAT solver is the result sat/unsat and the model which was found to prove the problem satisfiable (if it is the case). Some SAT solvers have the ability to find all solutions (possible models) for a given problem, as an extension to their prove search. To achieve such functionality, the solver includes a mechanism to create new SAT problems resulting from the

```
p cnf 4 2
1 -2 -3 0
1 2 4 0
```

Figure 3.1: Example of a DIMACS file content.
while(true){
    if(!decide())
        return satisfiable;
    while(!bcp())
        if(!backtrack())
            return unsatisfiable;
}

Figure 3.2: Pseudo code of a variation of the Davis-Putman backtrack search algorithm.

original and the models which are being found. When a model for the problem is found, the complement of its literals is included as a new clause in the original problem and a new search is performed on the resulting problem. The inclusion of the new clause will restrict the problem, so that the model found is no longer a solution for the new problem, which will force the search to obtain a different (new) model. Complement clauses are added as solutions are found and the search ends when the problem reaches the unsatisfiable state, which means that there are no more possible solutions to satisfy the problem.

3.1.2 Davis-Putman Backtrack Search

Davis-Putman (DP) backtrack search algorithm [11] (commonly named DPLL), a refinement of the earlier Davis-Putman algorithm [10], is a complete algorithm for determining the satisfiability of Boolean formulas and is the core of most conflict-driven solvers. The DPLL performs a depth-first search on a binary decision tree, where each node consists of a decision, i.e., an election of a variable of the problem to assign a value. The algorithm has three main processes and the pseudo code of a common variation of the algorithm is presented in Figure 3.2.

Definition 3.1.1 (Complete Algorithm) An algorithm is complete if it finds a solution when there is one or correctly reports that no such solution exists.

The decide process decides a literal from the problem, among all unassigned ones, and assigns a value to it (true or false). Each decision represents a node on the decision tree and has a decision level associated. The decision level represents the depth of the node in the tree. The root (no assignments decided) has decision level 0, the first decision has decision level 1 and so on. Assumptions and clauses with only one literal are propagated has decision level 0. Assumptions are used to force variable values. In certain searches, it is desirable to force some variables to assume certain values beforehand.

Definition 3.1.2 (Assumptions) Assumptions are variable assignments performed beforehand, to force certain variable values at a given search.

Definition 3.1.3 (Unit Clause) A clause is an unit clause if and only if all but one of its literals is assigned to false.

Definition 3.1.4 (Implied Clause) An implied clause is an unit clause produced due to decisions performed.

After performing a decision, all clauses must be checked, since with the new assignment, some clauses may become unsatisfiable or unit clauses. This process is performed by the bcp, which carries out for Boolean Constraint Propagation (BCP), that will cover all clauses to
detect possible conflicts (unsatisfiable clauses) and implications, due to the variable state of the problem. An implication occur when, due to a variable assignment, a clause becomes a unit clause, which means that it must become implied (remaining literal must be assigned to true to satisfy the clause) and the forced assignment (implication) must be propagated. Implications have the same decision level of the decision that triggered them. Finding implications result in a repetition of the process, since they may result in more new implications or conflicts. The bcp process ends when there is no more implications to propagate or a conflict is identified.

In the case of a conflict is detected during the bcp process, means that the preformed decisions do not satisfy the problem. If such case occurs, the backtrack process backtracks the search to the most recent decision that was not tried both values (true and false) and undoes all decisions performed until then (including their implications). The remaining value is then decided to proceed the search. If during the backtrack, all previous decisions were already tried both values, the problem is considered unsatisfiable. If during the search no conflicts are detected and all variables are assigned, the problem is considered satisfiable.

### 3.1.3 Conflict Analysis

As a solution to reduce significantly the search space of the DPLL algorithm, the authors of GRASP [33] proposed an extension to the backtrack process to handle conflict analysis techniques, such as clause learning and non-chronological backtracks. In the occurrence of a conflict during the BCP process, the conflict passes through a diagnose process, where the assignments responsible for such conflict are discovered. For such assignments, a new clause denying such assignments is created and added to the given problem. These clauses are known as conflict-induced clauses (also known as learnt clauses). The addition of conflict-induced clauses to the problem restricts conflicting and unnecessary paths in the search space. Another advantage of the diagnose process is the possibility of non-chronological backtracks. By analysis of the conflict it is possible to know when there is no way of finding satisfying assignments until the search backtracks to a certain decision level. This level corresponds to the last level where assignments, which caused the conflict, were made and discovering such level allows an immediate jump to it, saving search time and effort.

To better explain the generation of conflict-induced clauses, the following definitions are presented: $R(x)$ corresponds to the “reason” of the assignment $x$ and contains a set of antecedent assignments to the unit clause $\omega$, which led to the implication of $x$. For a clause $\omega = \{x \lor y \lor \neg z\}$ and an assignment $(y = 0, z = 1)$, the “reason” of $x = 1$ is $R(x) = \{y = 0, z = 1\}$. Decision assignments have $R(x) = \emptyset$. $\delta(x)$ corresponds to the decision level of the assignment $x$. A conflicting assignment associated with a conflict $c$ is denoted as $R_C(c)$ and the respective conflict-induced clause as $\omega_C(c)$. To facilitate the computation of $R_C(c)$, the “reason” of an assignment $x$ is partitioned by decision levels and is given by:

\[
\begin{align*}
\Lambda(x) &= \{y \in R(x) | \delta(y) < \delta(x)\} \\
\Sigma(x) &= \{y \in R(x) | \delta(y) = \delta(x)\}
\end{align*}
\]

(3.1)

If a conflict $c$ is detected, $R_C(c)$ is computed by the recursive definition:

\[
R_C(x) = \begin{cases} \\
\Lambda(x) \cup \left[\bigcup_{y \in \Sigma(x)} R_C(y)\right] & \text{if } R(x) = \emptyset \\
x & \text{otherwise}
\end{cases}
\]

(3.2)

The computation of a conflict-induced clause is simply the complement of the conflicted assignment. For example, an identified conflict assignment $R_C(k) = \{x = 1, y = 0, z = 0, w = 1\}$,
the conflict-induced clause produced is $\omega_C(k) = \{\neg x \lor y \lor z \lor \neg w\}$. The backtrack level of a conflict is computed according to:

$$B(c) = \max \{\delta(x) | x \in R_C(c)\}$$

(3.3)

When $B(c) = d - 1$, where $d$ is the current decision level, the search process backtracks chronologically. When $B(c) < d - 1$ the process backtracks non-chronologically.

Conflict analysis introduces significant overhead and for small instances of SAT can lead to a larger run time, but for large instances may contribute to a significant reduction of the run times. Another drawback is the growth of the clause database, resulted by the addiction of conflict-induced clauses during an execution, which will grow with the number of conflicts found. Examples of techniques to avoid such growth are the clause size restriction techniques, where produced learnt clauses with size greater than a given threshold are discarded, and periodic cleaning processes which use some heuristics to periodically reduce the number of the learnt clauses present in the problem.

### 3.1.4 Adaptive Branching

One key aspect of SAT solvers is how variables are selected at each decision, during the search. Decisions affect the performance of an algorithm and using a good heuristic is crucial to the efficiency of a solver. Unfortunately, there is no perfect heuristic and each heuristic performs better in different ranges of SAT instances than others. Over the years, many effective branching heuristics were proposed and detailed descriptions are made in [29] by Silva.

The simplest heuristic, commonly known as RAND, is to simply select the next decision randomly, among the unassigned variables. At the other extreme are the BOHM and MOM heuristics, that involve the maximization of some moderately complex functions of the current variable state and the clauses of the problem.

Popular heuristics, somewhere in the middle of the spectrum, are the DLIS and DLCS heuristics. This two heuristics count the number of unresolved clauses in which a given variable $x$ appears as a positive literal, $C_P$, and as a negative literal, $C_N$. The DLCS (Dynamic Largest Combined Sum) heuristic selects the variable with the largest sum $C_P + C_N$ and assigns the value true if $C_P \geq C_N$ or false otherwise. The DLIS (Dynamic Largest Individual Sum) heuristic selects the variable with the largest individual sum ($C_P$ or $C_N$) and assign true if a $C_P$ value is the largest, or false if a $C_N$ value is the largest. Variations of DLIS and DLCS, referred to as RDLIS and RDLCS, consist in randomly selecting the value to be assigned to a given selected variable, instead of comparing $C_P$ and $C_N$ values.

Another efficient heuristic, proposed by the authors of Chaff [31], is VSIDS (Variable State Independent Decaying Sum). This heuristic associates a counter to each literal and every time that a clause is added, the counters of its literals are increased. The heuristic selects the variable associated with the literal that has the highest counter. Periodically, all counters are divided by a constant to give some priority to variables that appear more often in learnt clauses. Globally, the VSIDS heuristic proved to be the most efficient among the others, obtaining accurate decisions regardless the difficulty of the problems. With such results, the VSIDS became a “must have” heuristic to increase the performance of a SAT solver and it is used in most of the popular SAT solvers.

### 3.1.5 Two Watched Literals Unit Propagation

In practice, for most SAT problems, a major portion of the solvers is spent in the BCP process. So, a fast and efficient BCP engine is one of the keys to any SAT solver. Each time an assignment
is deduced during a search, there is no need to check all clauses of the problem, since some may already be satisfied and others may still inconclusive (more than 1 literal unassigned). Therefore, it is only productive to visit newly implied clauses, which are clauses that can generate new implications or result in conflicts.

Chaff [31] authors proposed an efficient implementation technique to find newly implied clauses, avoiding constant visits to every clause affected to the variable assignments and reducing significantly the time spent in the BCP process. Instead of visiting a clause each time an assignment affects one of its literals, two of its literals are chosen, not assigned to false, to be observed. Such literals are known as watches. A clause is only observed if one of its watches is assigned to false. It is guaranteed that until one of the two watches is assigned to false, there cannot be more than N-2 literals, in the clause, assigned to false and so the clause cannot be implied. When a clause is visited but is not implied, that means that at least one non-watched literal is not assigned to false. In that case, such literal replaces the watched literal currently assigned to false and the property above is maintained.

3.1.6 Restarts

A restart consists of backtracking the solver to the root level (decision level 0), during an execution of a solver, and starting the search from the beginning. Any still-relevant conflict-induced clauses added to the problem and all variable activities are preserved after the restart, so that the solver does not repeat previous searches. The new decisions are made taking into account the new activities and may result in a different search path than the previous search. The intention of restarts is to provide a chance to change early decisions in view of the previous problem state. If a search tends to a path where the probability of finding a satisfiable assignment is reduced and a lot of effort is need to leave it, a restart gives the possibility to follow a different path, that may have a highest probability of success. Restarts are performed taking into account the number of conflict-induced clauses produced during a search. If such number reaches a given threshold, a restart is performed. The threshold is increased at each restart, so that the completeness of the algorithm is guaranteed.

3.2 Pseudo-Boolean Optimization Solvers

The demand and research for efficient solvers for the PBO problem are recent and falls into two categories of algorithms: Generic ILP solvers and specialized 0-1 ILP solvers. A comparison between these two types of solvers can be found in [5].

3.2.1 Generic ILP Solvers

Generic ILP solvers are used to solve linear programming problems, in general, and since PB-constraints are a variant of linear programming problems, they were traditionally handled by this kind of solvers. Although generic ILP solvers fit naturally to PB problems, they tend to ignore the Boolean nature of 0-1 variables.

Definition 3.2.1 (Linear Programming) Linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints.

One popular type of generic ILP solvers are the branch-and-bound algorithm based solvers [26], that have proven to be very effective for easy PB instances. Some well known ILP solvers are CPLEX [20] and SCIP [2]. We will not detail such type of solvers since they are out of the scope of this work.
3.2.2 Specialized 0-1 ILP Solvers

Specialized 0-1 ILP solvers consist in an adaptation of conflict-driven SAT solvers to handle PB-constraints. There are two strategies to support PB-constraints: or the SAT solver is modified to handle PB-constraints directly (known as native support solvers) or each PB-constraint is translated into a set of equivalent CNF clauses, in a pre-processing step, to be processed by an unmodified SAT solver. The translation strategy has the advantage of using efficient SAT solver procedures as a black box, but suffers the possible exponential growth of CNF clauses, due to the translation nature. Native support solvers are more complex and imply a modification of the BCP process to handle PB-constraints in the way that it handles CNF clauses.

Popular solvers with native support are PBS [4], bsolo [24], vbo [27] and Pueblo [32] (an extension of Minisat to handle PB-constraints). A popular non-native support solver is the Minisat+ [14], which uses MiniSat to handle the translated constraints.

3.2.2.1 Processing of PB-constraints

Before proceeding with any searches and variable assignments, it is important to define a normal form for all constraints of the problem to be solved. As explained in Section 2.2.2, the normalization of constraints may simplify the constraints, but more importantly, reduces the number of relations into one type and consequently, reducing the cases to be handled by the solver. So, taking advantage of the normalization process, the solvers only needs to implement methods to handle one type of constraints, avoiding the implementation of complex methods to handle all types of constraints during the search.

Assign values to variables in a PB-constraint, during a search, is a more complex process than in pure CNF clauses. Each PB-constraint must contain a field representing the current LHS value, the sum of all activated $C_i$, and another field representing the maximum value possible for the LHS, the sum of all activated $C_i$ and all $C_i$ associated with unassigned variables. For simplicity, the fields are designated by $currLHS$ and $maxLHS$, respectively. Assigning true to a variable $v$ implies traversing all PB-constraints containing a literal of $v$. The $currLHS$ of every PB-constraint, containing the positive literal of $v$, is incremented by the coefficient associated with $v$ in that constraint. On the other hand, the $maxLHS$ of every PB-constraint, containing the negative literal of $v$, is decremented by the coefficient value associated with that literal. Undoing an assignment of true to a variable $v$, entails the inverse process, the $currLHS$ and $maxLHS$ of every affected PB-constraint are decremented and incremented, respectively. When the assigned value of a variable $v$ is false, the process is similar, only changing the $currLHS$ and $maxLHS$ fields in the operations.

During the process of updating PB-constraint fields, new implications and conflicts can be found. Each PB-constraint type has associated a set of rules for detecting implications and conflicts. Given a PB-constraint of type “$\leq$”, any literal $x_i$, whose coefficient is $C_i > (RHS - currLHS)$, is implied to false. On the other hand, given a PB-constraint of type “$\geq$”, a literal $x_i$ is implied to true if its coefficient is $C_i > (maxLHS - RHS)$. Conflicts are detected when the $currLHS > RHS$ (for a PB-constraint of type “$\leq$”) or the $maxLHS < RHS$ (for a PB-constraint of type “$\geq$”).

3.2.2.2 Problem Optimization Process

As described in Section 2.3, each PBO problem has an objective function associated and the purpose of the PBO solvers is to find a variable assignment for the problem that not only satisfies all problem constraints but also that obtains the optimal solution for the objective function. One popular approach to achieve such optimal solution is to perform a linear search
3. REVIEW OF SOLVERS

into the upper bound values of the problem optimization search space. Taking advantage of an iterative search procedure, it is possible to use the objective function to create new problem constraints, to restrict the sum of its literals, and perform successive searches to the problem to achieve new upper bound values for the optimization search space, until it reaches an unsatisfiable state. Assuming a problem with objective function $f(x)$, the first iteration of the procedure finds a possible model for the problem, $x_0$ (without considering any objective function restriction), which will obtain an initial solution $f(x_0) = k$. In the second iteration is included a new constraint $f(x) < k$ and performed a new search for the problem satisfiability. If the problem becomes unsatisfiable, $k$ is the optimal solution for the problem, otherwise the procedure continues to the next search iteration, where it includes a new objective function restriction to the problem with the new solution found (adding $f(x) < k_{\text{new}}$ as a new constraint). The procedure finishes when the problem becomes unsatisfiable, meaning that the optimal $k$ value was found.

Another popular approach to achieve the optimal solution for PBO problems is using unsatisfiability-based algorithms, which are detailed in [27], to search on the lower bound value of the problem optimization search space. Unsatisfiability-based algorithms are used to solve Maximum Satisifiability problems, also known as MaxSat, and it is possible to encode PBO problems into MaxSat problems. As a brief explanation, such kind of algorithms takes advantage of an iterative procedure to identify unsatisfiable formulas in the problem. In each step is identified a formula from the problem that is responsible for its unsatisfiability and a relaxation of such formula is performed (if possible) to avoid the conflict. The procedure continues until the problem becomes satisfiable, meaning that the optimal solution for the problem was found, or an unsatisfiable formula identified during the procedure cannot be relaxed, meaning that the initial problem is unsatisfiable.

There are some solvers that combine the two approaches above to achieve a better performance for their problem optimization process, which is the case of the wbo solver [27]. The wbo solver starts the optimization process using the first approach, performing a linear search into the upper bound values of optimization search space, and if the search reaches a time limit threshold without finding the optimal solution for the problem, the search is aborted and an unsatisfiability-based algorithm is used to perform a new search in the lower bound value of the optimization search space, until the optimal solution for the problem is found.

### 3.2.2.3 Translation of PB-constraints into CNF

Translation of PB-constraints into CNF clauses is a technique used in non-native support specialized 0-1 ILP solvers. It consists in a process between the normalization of PB-constraints and the calls to the SAT solver engine, and has two main steps: each constraint is converted into a single-output circuit and than each circuit is converted into CNF clauses by Tseitin transformations. A more detailed description of this conversions can be found in [1, 14].

The goal in a translation is not only to get a compact representation, but also to preserve as many implications as possible, between the PB problem and the produced CNF clauses. Such concept is known as arc-consistency. A translation maintains arc-consistency when all assignments that could be propagated on the original constraint can also be found on the SAT solver’s unit propagation, operating in the translated constraint. Although arc-consistency is always desirable, maintaining such propriety without an exponential growth in the translation is a very difficult task, since there exists a trade-off between arc-consistency and the translation size. There are several approaches to convert a PB-constraint into a single-output circuit[14], where each approach defines distinct properties, having advantages and drawbacks. Popular approaches are the conversion using BDDs, networks of adders and networks of sorters. BDDs
guarantee arc-consistency in the translation, but suffer from exponential size translation in the worst case. On the other hand, networks of adders do not guarantee arc-consistency, but result in a linear size translation. In the middle of the scope are the networks of sorters, not completely arc-consistent but with an almost linear size translation.

3.3 Parallel and Distributed Solvers

Due to the evolution and popularity of parallel and distributed computing systems, it became evident that the future of the research in SAT solvers would fall into the world of parallel computing, like many other domains did. Since hard instances result in huge search spaces, having several resources to share the effort of the search became a desirable strategy. The demand of algorithms for SAT, taking advantage of cluster and grid architectures, is growing and the first strategies appear as a stimulation to the research in this area.

3.3.1 Distributed SAT Solvers

A popular and very used approach adopted in many distributed SAT solvers is the Task Farm approach, that uses the master/slave topology presented in the Figure 3.3. The objective of such approach is to divide the SAT search space among all slaves of the system, so that different partitions of the search space can be computed simultaneously.

A slave (also known as worker) consists in an independent SAT solver that contains the entire initial description of the problem to solve. The partition of the search space is managed by the master and the assignment of such partitions to the workers is performed by a process of task delivery. The master contains and manages a task repository, which includes all tasks that must be computed by the workers. A task consists of a set of initial assumptions (variable assignments) that correspond to a given partition of the search space. Each task contains a different set of assumptions, so that each worker computes a distinct area of the search space. Workers are responsible for computing the tasks received and to respond to the master the obtained results (sat or unsat), according to the assumptions defined. A sat response may also include the model found, that satisfies the given problem, and an unsat response includes a set of conflicting assignments that originated the unsat result.

The approach explained above is the core of several distributed SAT solvers such as PSato (parallel version of Sato) [35], GrADSAT (parallel version of zChaff) [9] and PMSat (parallel version of Minisat) [23]. Although this solvers implement the same approach (differing in some upper level techniques), they differ in the parallel technologies used on their implementations.
3.3.1.1 Search Process

Using the variables of the problem, the master initially creates a task repository by defining all necessary tasks (sets of assumptions) to compute the problem. Such technique is described in the Section 3.3.1.2. After all tasks have been defined, the master initiates the search by sending one task, pulled from the task repository, to each worker and waits for their responses.

Receiving an unsat response from a worker, the master uses the set of conflicting assignments received to remove unnecessary tasks from the task repository (tasks proven to be useless due to the conflicting assignments) and to stop possible workers that are computing tasks proven to be unsat. When workers finish their tasks (or are ordered to stop one), a new task is pulled from the task repository and assigned to them. The search process ends when the master receives a sat response from a worker or when the task repository becomes empty and all workers computed their defined tasks, resulting in unsat responses (the problem is unsatisfiable).

3.3.1.2 Assumptions Generation

Assigning values to a set of variables in a problem, redirects the search into a specific subspace of the search space. Therefore, by defining a specific group of assumptions, it is possible to partition the search space in such a way that all workers could compute cooperatively the entire search space. Defining such group of assumptions must guarantee two properties: the group must cover the entire search space and each set of assumptions in the group must represent a different subspace of the search space.

As an example, given a problem with two variables \{y, p\} and the decision tree presented in the Figure 3.4, defining two assumptions with \(y = \text{true}\) and \(y = \text{false}\) covers the entire search space and each workers could compute half the search space concurrently.

The example above is a very simple example, but generating assumptions is not a simple process. Assumptions must be carefully created and aiming the “right” variables to assume may improve the performance of a solver. The authors of PMSat [23] proposed some generation methods to improve the generation of assumptions. The methods proposed take into account the popularity of variables and literals (the number of occurrences on clauses) and use the most popular variables to start partitioning the search space.

3.3.1.3 Sharing Conflict-induced Clauses

As described in Section 3.1.3, conflict analysis techniques have several advantages on SAT solvers. Since a worker consists of an independent SAT solver, it can take advantage of conflict analysis techniques and generate conflict-induced clauses, during the searches. In a parallel SAT solver, each worker may compute several tasks assigned by the master during an execution and each task may result in an addition of new clauses to the worker solver state. Such gathering
of conflict-induced clauses can be very advantageous, since each task will have access to the “knowledge” of the previous ones.

One technique adopted by several parallel SAT solvers is the ability of sharing conflict-induced clauses among workers. Instead of a task having access only to the “knowledge” of its correspondent worker, it will have access to the global “knowledge” of the system. To achieve such goal, the workers include, in their responses to the master, a set of conflict-induced clauses generated during the task computation. The master spreads the clauses received by including them in the following task assignments. The workers, after receiving a task, add the included conflict-induced clauses to their solvers. The use of such technique may help reducing the search space, avoiding unnecessary branches, but including such clauses in the communication flow implies an extra overhead in the execution. Some implementations restrict the size and the number of clauses that are shared, as a solution to amortize the overhead imposed by this technique.

3.3.2 Parallel Pseudo-Boolean Optimization Solvers

Although the PBO research lacks from popular solvers to operate in distributed environments, the first parallel versions of the PBO solvers, developed for shared-memory environments, starts to appear. The pwbo solver [30], a parallel version of the well known wbo solver [27], is one good example of such migration and it contributes with plenty of techniques to improve the efficiency of solvers with such characteristics.

Since the wbo solver uses two approaches to find the optimal solution for the problem (explained in Section 3.2.2.2), the strategy of the pwbo solver is to spawn two threads to perform the two approaches simultaneously and search the two boundaries of the problem optimization search space concurrently. One thread runs a PB solver to perform a linear search to restrict the upper bound value of the optimization search space and another thread runs an unsatisfiability-based algorithm to restrict the lower bound value of the search space.

The parallel search with these two orthogonal approaches results in a performance as good as the best approach for each problem instance. Additionally, the solver features a mechanism to share learnt clauses between the two threads to speedup the search on each thread during the problem optimization process. However, not all learnt clauses produced can be share among the threads since they have different representations of the problem to solve. To distinguish the clauses that can be safely shared, all clauses are divided by soft and hard clauses [30], where the unsatisfiability-based algorithm contains soft and hard clauses and the PB solver, performing the upper bound value search, contains only hard clauses. Considering that, only learnt clauses produced by hard clauses, that do not contain relaxation variables, can be considered for sharing.

Additionally, the pwbo solver allows the spawning of more than 2 threads to perform concurrent searches (up to 8 threads). When additional threads are spawned, the remaining threads perform linear searches on different values of the upper bound area of the optimization search space. Learnt clauses are also shared among all linear search threads, but the learnt clauses produced are only shared to threads containing upper bound values smaller or equal to the exporting thread.
Chapter 4

MPBO Solver Planning

The adaptation of popular SAT solvers into the parallel computing systems, as described in Section 3.3, proved to be very promising and their results are proof that such strategy is an effective solution, when it comes to solve hard instances. Such achievement demonstrates the great effort spent in the SAT research, to find new efficient solutions to improve the SAT resolution, and its investment in the distributed computing world. However, the same can not be said regarding to the PBO research, since there are no evidences of popular solvers taking advantage of distributed systems. To overcome such flaw, the aim of this work is to propose and implement an efficient distributed PBO solver, called MPBO (Message Passing Pseudo-Boolean Optimization) solver, to contribute to the PBO research in the distributed computing domain.

4.1 Solver Approach

The good news for developing a parallel version of a PBO solver are that the characteristics of the optimization process in PBO favor its parallel implementation. Since the optimization process (described in Section 3.2.2.2) consists of iterative calls to the search procedure, using different objective function restrictions, performing a concurrent search on the optimization search space can be a good strategy. Taking advantage of the well known Task Farm approach (described in Section 3.3), it is possible to implement a promising distributed implementation for PBO. The goal is to use independent Pseudo-Boolean solvers as workers and a master to manage and assign tasks, where each task corresponds to a specific objective function restriction that must be explored. For simplicity, the solver will only handle the minimization process.

Figure 4.1: Example of possible initial task assignments when solving a problem with an objective function \( f(x) = x_1 + 2x_2 + 20x_3 \).
since maximization problems can be converted into minimization problems.

Given an objective function $f(x)$ with $N$ variables, the size of the optimization search space to be explored (in the worst case) is $2^N$. The optimization search space can be defined by an interval $\sigma = [a, b]$, where initially $b$ is calculated using the following formula:

$$\sigma(b) = \sum_{i=1}^{n} C_i \cdot p_i, C_i \in \mathbb{N}, p_i = \begin{cases} 0 & C_i < 0 \\ 1 & \text{otherwise} \end{cases}$$

which corresponds to the highest solution possible for the objective function, and $a$ is calculated with the given formula:

$$\sigma(a) = \sum_{i=1}^{n} C_i \cdot p_i, C_i \in \mathbb{N}, p_i = \begin{cases} 0 & C_i > 0 \\ 1 & \text{otherwise} \end{cases}$$

which corresponds to the lowest solution possible for the objective function. Partitioning an interval $\sigma$ in such a way that each worker computes significant sub-spaces, with different objective function restrictions, is not an easy task. The intuitive approach is to perform a binary search in the considered values of $\sigma$, so that each worker computes different and evenly spaced values of the interval, and restricting $\sigma$ according to the results obtained from the computed tasks. To each worker will be assigned a different value $k \in \sigma$, which will be used to produce the desired restriction constraint $f(x) < k$ to be computed. However, identifying suitable values to assign is a complex task, since not all values in $\sigma$ correspond to valid variable assignments in the objective function.

In Figure 4.1 is presented an example of a possible assignment through 6 workers, when solving a problem with a simple objective function $f(x) = x_1 + 2x_2 + 20x_3$. The initial interval of the problem is $\sigma = [0, 23]$ but only eight values in such interval are valid solutions of the objective function $\{0, 1, 2, 3, 20, 21, 22, 23\}$. So, assigning arbitrary values of the interval may lead to redundant computations. Although the 6 workers are computing 6 different tasks, in the example above, the workers 3, 4 and 5 are performing redundant computations. Since there are no possible solutions between the values 3 and 20 in the objective function, computing tasks with restriction values $\{6, 12, 18\}$ result in the same subspace search (equivalent to $f(x) < 20$).

Unfortunately, identifying all possible solutions of the objective function is impractical, due to its exponential growth complexity ($2^N$), but solutions to reduce such redundancy must be taken into account.

The master performs the identification of the optimal solution by managing and restricting the interval $\sigma$ according to the responses received from the workers. If a task, associated with a $k$ value, results in an unsat response, this means that there is no possible assignment that...
satisfies the given $f(x) < k$ restriction. If $f(x) < k$ is unsatisfiable, $\forall y \in \sigma, y < k \mid f(x) < y$ are also unsatisfiable. Using such property, the master updates the lower bound $\sigma(a)$ with value $k$ and aborts all workers computing restriction values lower than $k$. On the other hand, tasks resulting in sat responses lead the master to update the upper bound $\sigma(b)$ with the solution value obtained, aborting all workers computing restrictions higher than such value. Such restrictions are performed until only one value is covered by $\sigma$, which is considered the optimal solution for the problem. A demonstration of the optimization process explained above, using the example of Figure 4.1, is presented in Figure 4.2.

During the problem optimization search, each worker will probably compute several values of $\sigma$ and some changes to the solver state of a worker may be required, between each task computation. The computation of following tasks with lower values does not imply any changes to the solver state, since if a solution satisfies the new restriction, it mandatorily satisfies the old objective function restriction. On the other hand, in the computation of higher values than $k$, the old objective function restriction cancels the new restriction to be explored. To avoid such property, a removal of all constraints, created due to the old objective function restriction, must be performed before adding and computing the new objective function restriction.

### 4.2 Solver Architecture

To implement the MPBO solver, the well known Minisat+ [14] solver is used, which operates through translation using the MiniSat solver engine, and the MPI API [15] to handle the communication and synchronization between processes. Minisat+ was chosen because of its clarity in the code and its good documentation. It also takes advantage of the MiniSat solver, which proved to be a leader at the SAT solvers. The use of MPI comes from its high performance and portability, characteristics that made such API the industry’s de-facto standard.

In Figure 4.3 is presented the architecture diagram of the MPBO solver to introduce the modules disposal that compose each entity of the solver. The master entity is composed by a problem optimization module to perform the optimization approach explained in Section 4.1 and by set set of routines, using MPI calls, to manage the workers. According to the optimization process state, the master assigns or aborts the necessary tasks to reach the optimal solution for the problems. The worker entity consists of an independent instance of the MiniSat+ solver, to compute the assigned tasks, and a set of communication routines, resorting to MPI library calls, to receive the tasks assigned by the master and consequently send the obtained results.

To implement the optimization process module, some possible challenges need be taken into
bool solve(vec<Lit> assumps);
lbool search(int nof_conflicts);
CRef propagate();
void analyze(CRef confl, vec<Lit>& out_learnt, int& out_btlevel);
void cancelUntil (int level);
Lit pickBranchLit();
bool addClause (vec<Lit>& ps);

Figure 4.4: List of the MiniSat main API functions.

account to improve the solver performance. As in the case of parallel SAT solvers, deciding the “right” tasks to be computed early can lead to a significant reduction in the execution time. So, defining efficient heuristics for a better partition of the optimization search space (instead of the pure binary division) is a good approach to improve the MPBO solver algorithm. Another aspect that must be taken into account is the assignment of less restrictive tasks to the workers. As explained in Section 4.1, if a worker receives a task with a higher $k$ value than the previous task assigned, it is not possible to compute the new task correctly since the worker contains more restrictive constraints due to the previous task computation. To avoid such situation, the worker must include a clause removal mechanism to remove all clauses related to the computation of previous tasks when it receives a task with a higher $k$ value.

The following sections give some details of the MPI and MiniSat+ APIs, which are the core of several modules of the MPBO solver.

4.2.1 MPI

The demand for more computational power lead the programmers to adopt parallel computing systems to run their programs. Super computers, clusters, grids and even actual personal computers (multi-core computers) are considered parallel computing systems, although with different characteristics. Unfortunately, developing programs for parallel systems is more complex than programming in a sequential system, since parallel systems involve multiple processing elements during a program execution. Parallel programming systems are used to handle the communication and synchronization between those processes and the MPI is one example of such systems.

MPI (Message Passing Interface) [15] is a popular and widely used API that provides essential virtual topology, synchronization and communication functionality between a set of processes in parallel computing systems. MPI consists of a specific set of routines that are directly callable from a variety of languages (for example, C and C++) to integrate its features into any program. MPI was defined with the contribution of many commercial and open-source vendors with the aim of creating a parallel programming standard with high performance, scalability and portability. Since then, it has become the industry’s de-facto standard to implement portable parallel programs for a wide variety of parallel computing systems.

4.2.2 MiniSat API

MiniSat [13] is a minimalistic, open-source SAT solver, which implements all improvement techniques described in Section 3.1, combined with efficient data structures that improve the memory access and management. Such combination results in a significant reduction of effort and time required to solve SAT problems, which led MiniSat to win all industrial categories
while(learnts < nof_conflicts){
    propagate() // propagate unit clauses
    if(no conflict found){
        if(all variables assigned)
            return true; // Satisfiable
        else
            pickBranchLit(); // pick a new literal and assign it
    }else{
        analyze(&bt_level); // analyze confl. and add learnt clause
        if(top-level conflict found)
            return false; // Unsatisfiable
        else
            cancelUntil(bt_level); // do the backtrack
    }
} cancelUntil(root_level);
return Undef;

Figure 4.5: Pseudo code of the MiniSat Search function.

of SAT 2005 competition. MiniSat is compact and well documented, containing a small and well-designed amount of code.

4.2.2.1 Core Engine

The MiniSat engine contains an intuitive and well defined set of functions that implement the SAT search mechanism (the list of main functions are defined in Figure 4.4). The solve function is the main function of MiniSat, which is responsible to initiate the search mechanism. It is composed of a loop, with iterative calls the the search function (to implement the restart technique), which returns true if the problem is satisfiable or false otherwise. The search function has a threshold argument to limit the learnt clauses added to the problem and backtracks the search to the root level if such threshold is reached. The solver function increases the threshold each time the search result in Undef (undefined) and proceeds with another call containing a new threshold value, higher than the previous one. The search ends when the search function returns true or false depending on the satisfiability of the problem in question.

The search function implements the DPLL algorithm (details in Section 3.1.2) and its pseudo code is presented in Figure 4.5. The three DPLL processes are implemented using the propagate function (implementing the BCP), the pickBranchLit function (implementing the decide process) and the cancelUntil (which implements the backtrack process). The analyze function implements the Conflict Analysis technique (detailed in Section 3.1.3) which is responsible to analyze the conflicts found and to add the corresponding learnt clauses.

MiniSat performs its decisions based on the VSIDS and RAND heuristics (detailed in Section 3.1.4). Each variable has an activity associated, which increases each time it occurs in a learnt clause produced during the search. MiniSat contains a heap which includes all decision variables (variables that can be decided), ordered by descending activities. The heap is ordered at the beginning of a search and reordered each time a learnt clause is inserted into the problem. Each time a decision must be made, the pickBranchLit function will produce a random value
void addGoal(vec<Lit>& ps, vec<Int>& Cs);
bool addConstr(vec<Lit>& ps, vec<Int>& Cs, Int rhs, int ineq);
bool normalizePb(vec<Lit>& ps, vec<Int>& Cs, Int& C);
void storePb(vec<Lit>& ps, vec<Int>& Cs, Int lo, Int hi);
void propagate();
bool convertPbs(bool first_call);

Figure 4.6: List of the MiniSat+ main API functions.

and if such value is smaller than a given threshold, the function picks a random variable from
the heap to assign a value. If the produced random value is equal or greater than the threshold,
pickBranchLit picks the variable with the highest activity from the heap to assign.

One important characteristic of MiniSat is the possibility for incremental clause addiction.
Besides the addiction of learnt clauses (which are created during a search), MiniSat contains an
interface to include more “core” clauses to the problem when the search is located at the root
level. Such mechanism is useful when we need to expand the problem and re-execute the search,
depending on the result obtained (example of MiniSat+), and such characteristic enriches the
MiniSat for its integration into other application fields.

4.2.2.2 SimpSolver Extension

Although MiniSat core engine is very robust and efficient, the authors made additional effort
to continue the improvements to their solver. SimpSolver [12] is an extension to the core
algorithm, which performs preprocessing of clauses to reduce their size. The preprocessing
combines variable elimination with subsumption and selfsubsuming resolution (details can be
found in [12]), which reduces the problem size and consequently decreases substantially the
runtime of the solver. However, it was observed some occurrences of segmentation faults in
calls to the SimpSolver function backwardsSubsumptionCheck, which made us to disable such
functionality in our solver implementation.

4.2.3 MiniSat+ API

MiniSat+ [14] is a specialized 0-1 ILP solver (description in Section 3.2.2) which solves PBO
problems resorting to the conversion of PB-constraints into SAT clauses, using MiniSat as the
SAT solver engine to handle the SAT clauses produced. Therefore, MiniSat+ can be seen as an
extension to MiniSat, to allow the resolution of PB problems taking advantage of the efficient
MiniSat search engine. MiniSat+ represents PB-constraints using instances of a class called
Linear. Such class contains information about the size (number of literals) of the constraint,
the high and low values which the sum of the activated literals must obey, and the corresponding
literals (and associated constants) that compose the given constraint. To go beyond the typical
implementation of integers, MiniSat+ supports big numbers representation on their problem
coefficients, resorting the well known GNU Multiple Precision Arithmetic (GMP) [17] library.
GMP allows the representation of operands without size limitations and it is carefully designed
to be as fast as possible during its operations.

To take advantage of the MiniSat engine to solve PB problems, MiniSat+ performs two
steps to efficiently achieve an equivalent SAT problem from the PB problem in question: the
pre-processing of all PB constraints and their conversion into equivalent SAT clauses. Regarding
their implementation, MiniSat+ contains a set of well defined functions (presented in Figure 4.6)
to perform the two steps mentioned.
Linear& c = *constrs[i];
int adder_cost = estimatedAdderCost(c);

// Converting using BDD
Formula result = convertToBdd(c, adder_cost * opt_bdd_thres);
if (result == _undef_)
    // Converting using Sorters
    result = buildConstraint(c, adder_cost * opt_sort_thres);
if (result == _undef_)
    // Converting using Adders
    linearAddition(c, converted_constrs);
else
    converted_constrs.push(result);

Figure 4.7: MiniSat+ converter heuristic code.

4.2.3.1 Pre-processing of Constraints

Before converting any constraint, MiniSat+ performs some techniques to simplify (if possible) the constraints of a problem. Normalization of constraints and propagation of assigned literals are techniques that may reduce the problem size, thereby reducing the necessary conversions to obtain the equivalent SAT problem. Each constraint of the problem is allocated using the function addConstr, which is composed by a normalization process (normalizePb) and a storage process (storePb). The normalization process (detailed in Section 2.2.2), besides defining a normal form for the constraints and simplify each constraint, can minimize the problem size by discarding already satisfied constraints and by detecting conflicting constraints. A constraint is stored if and only if it is not already satisfied or it is not in conflict, where the normalizePb function returns false if one of the two cases is not met. MiniSat+ allocates its constraints using the StackAlloc class. Such class improves the constraints allocation by reducing the number of memory allocations, avoiding a malloc instruction each time a constraint is stored. Unlike PB constraints, the objective function in PBO problem is not allocated using the StackAlloc and it is stored using a dedicated function addGoal.

During the normalization of a constraint, some variables may be forced to be assigned, so that the given constraint can still be satisfiable. All forced assignments are recorded in a vector called trail and after all constraints are allocated and normalized, MiniSat+ propagates all variable assignments in the trail through all constraints of the problem. Such process is performed by the propagate function and its essence is similar to the BCP process used in SAT solvers (described in Section 3.1.2). Propagating the assignments reduces the size of the constraints, by eliminating assigned literals and updating the relation limits by the corresponding constants, and may eliminate unnecessary constraints for conversion, which become satisfiable due to the variable assignments performed.

4.2.3.2 Conversion of Constraints

Conversion of constraints is the key mechanism of MiniSat+ since it is the bridge between the PB problems to solve and the equivalent SAT problems obtained. The conversion type used in MiniSat+ is described in Section 3.2.2.3, which is composed of two steps: construction of a single-output circuit for each constraint and the clausification of each circuit into SAT clauses. To implement the converter mechanism, MiniSat+ uses methods and data structures defined by the namespace known as Formula Environment (FEnv). Typically, constraints are
bool sat = false;
while (sat_solver.solve()){
    sat = true;
    // Obtain value found
    best_goalvalue = evalGoal(*goal, sat_solver.model);
    // Add new objective restriction
    if (!addConstr(goal_ps, goal_Cs, best_goalvalue))
        break;
    convertPbs(false);
}
if(sat) printf("Optimal solution: %d\n", best_goalvalue);
else printf("UNSAT\n");

Figure 4.8: MiniSat+ optimization process code.

converted into Formulas, which are a representation of the nodes in the single-output circuits obtained by the conversion of the PB constraints. After all circuits are created, MiniSat+ uses a Clausifier class that decomposed the Formulas into SAT clauses, depending on the type of the Formulas.

MiniSat+ features three techniques to create the single-output circuits: BDDs, Sorters and Adders (described in Section 3.2.2.3). Each technique obtains a different SAT problem for the same PB problem, which differs in the size and in the arc-consistency between the SAT and PB problems in question. It is possible to choose one of the available techniques to convert all constraints or leave such responsibility to MiniSat+, which integrates a mechanism to decide the suitable technique to use for each constraint, resorting to the Adders technique cost. Before converting a constraint into Formulas, MiniSat+ calculates an estimate cost value of converting the given constraint using Adders. MiniSat+ gives priority to the BDDs technique and always initiates a constraint conversion using it. If during the BDDs conversion, the BDD cost overflows the estimate Adders cost calculated, it is aborted and the Sorters conversion takes place. If the same happens with the Sorters conversion (Sorters cost overflows the estimate Adders cost calculated), MiniSat+ converts the constraint using Adders, which is the last resort technique to convert any constraint due to its characteristics. Figure 4.7 presents the code snippet used in MiniSat+ which implements the heuristic explained above.

4.2.3.3 Operation Modes

MiniSat+ features three modes of operation when solving a problem:

• finding the first solution (sc_FirstSolution);
• finding all possible solutions (sc_AllSolutions);
• finding the optimal solution, for PBO instances (sc_Minimize);

The first two options are simple, MiniSat+ converts the given PB problem into an equivalent SAT problem and resorts to MiniSat to obtain the solution. Since MiniSat already features a mechanism to obtain all solutions for a given SAT problem, MiniSat+ can obtain all solutions for a PB problem resorting to MiniSat.

To implement the last operation mode, MiniSat+ features an optimization mechanism to identify the optimal solution, when solving PBO problems. The approach used is described
in Section 3.2.2.2 and consists of successive restrictions to the problem, using the objective function and the goal values obtained, until the problem reaches an unsatisfiable state. Since \texttt{MiniSat+} converts constraints one by one and using incremental addition support by \texttt{MiniSat}, it is possible to add new constraints between searches in \texttt{MiniSat} and implement the approach explained above. Obtaining the first model for a given PB problem, \texttt{MiniSat+} calculates the value of the objective function, using such model, and includes a new constraint “<” into the problem, using the objective function as LHS and the objective value obtained as RHS. The new constraint will force the next searches to obtain models that result on a lower value in the objective function. After the new constraint is converter into SAT clauses, \texttt{MiniSat} solver is used to obtain a new solution, taking into account the new requirements. If no model is found, the search ends and the last model is considered the optimal solution, otherwise, the process is repeated by adding a new constraint, using the solution value obtained. The implementation is presented in Figure 4.8, which describe the optimization loop adopted by \texttt{MiniSat+}. 
Chapter 5

MPBO Solver Implementation

This chapter presents the implementation details of the MPBO solver and its evolution during the implementation process. The implementation of the MPBO solver went through several stages, related with the task assignment on the master, communication between master and workers, and the solver restore mechanisms on the worker. Along with the details of each mechanism implemented, this chapter describes the implementation process and the reason behind their usage. The following sections describe the interactions between the master and worker processes, such as type of messages transferred and their purpose, and the implementation details of each process type, master and worker, that composes the solver.

5.1 Process Communication

Communication is indispensable for the interaction of processes in a distributed system, for synchronization and sharing of information among them. There are plenty of available technologies which provide process communication solutions for distributed system, but for the motives already mentioned in Section 4.2, MPI was chosen for such purpose in the implementation of this work.

In the implementation of the proposed solver, the communication is performed involving the master process and each individual worker, and such communication is divided in two main stages: the broadcast of the content of the problem file descriptor and the point-to-point communication during the problem optimization process. The following sections detail how communication is performed in each stage and how the necessary data is transferred between the master and worker processes.

5.1.1 Complex Data Transfer

MPI calls that exchange information between processes, resorting to messages, support data transfer in most of data types available in C, such as integers, booleans, characters and many more. However, not limiting the amount of data in a message, they only support sending data in one data type directly, making the transference of multiple data types, in a single message, a more complex task.

Almost all communications in the MPBO solver are complex, containing data of different data types to be transferred (sending PB constraints and task responses are examples of such), and sending them in separate messages is an inefficient approach, due to the overhead imposed by the multiple MPI calls and the headers of each message (resulting in more data to be
struct TestStruct {
    int *i;
    double d[length];
    char c[length];
} data;

data.i = (int*) malloc(sizeof(int) * length);
char* buf = (char*) malloc(sizeof(char) * BUF_SIZE);
int pos = 0;

MPI_Pack(data.i, length, MPI_INT, buf, BUF_SIZE, &pos, comm);
MPI_Pack(&data.d, length, MPI_DOUBLE, buf, BUF_SIZE, &pos, comm);
MPI_Pack(&data.c, length, MPI_CHAR, buf, BUF_SIZE, &pos, comm);

MPI_Send(buf, pos, MPI_PACKED, dest, tag, comm);

Figure 5.1: Example of a routine to pack a dynamic structure.

Fortunately, MPI features mechanisms which enable the transfer of messages containing multiple data types, in a single MPI call. Two popular solutions to perform such complex transfers are the creation of derived MPI data types and data packing.

The derived MPI data types are built from basic MPI data types and can represent data structures to be transferred directly in the MPI message transfer calls. Combining multiple basic MPI data types and using displacements, to delimit the data of each type, is possible to create a new MPI data type to map a particular structure and use it to transfer the structure instances afterwards, in a single message. Upon receiving, MPI mechanisms automatically fill the structure instance, which is given as the receive buffer in the MPI call, without any further user mechanisms to handle the data. Each derived MPI data type must be created and allocated (using the function MPI_Type_commit), before any transfer of such type, and freed before the application finishes. Derived MPI data types are the fastest and elegant solution to achieve complex data transfers, since they provide portability and efficiency for communications using mixed data types in a single message. However, such solution is not indicated when the structures to be transferred are dynamic. Since derived MPI data types perform a field mapping of the structures, if the structures contain pointers to allocated memory, such memory is not transferred. So, all memory addressed by such pointers must be handled separately, using a derived data type extension or transferring it in additional messages.

Data packing is a more primitive solution and is commonly used to implement data transfers in MPI wrappers. The approach of data packing is similar to the transfers using the MPI_BYTE data type, where the data to transfer is arranged into a byte buffer, just like it is allocated in memory, and sent to the destination through a single message. However, in data packing, the data is arranged into the buffer using MPI data types to guarantee the portability of the code through distributed systems with heterogeneous architectures. The solution consists of packing the data to a buffer using successive calls to the function MPI_Pack, where each call packs data of a different type to the buffer, and sending such buffer in a single message using the MPI_PACKED data type. After the transfer is complete, the receiver must unpack the data, in the buffer, in the same order that it was packed in the sender, using the MPI_Unpack function. Figure 5.1 presents an example of the necessary routine to pack a dynamic structure.

Data packing is a simpler solution compared to the derived MPI data types but it is more
tedious to implement and suffers from the overhead of successive calls to MPI functions to pack and unpack the transferred data. However, it is more flexible in how data is organized in the message, since it is the user responsibility to implement the packing and unpacking routines. Since the MPBO solver uses dynamic structures to manage the data to be transferred between processes, it was decide to use the data packing solution for its implementation.

5.1.2 Problem Content Broadcast

The MPBO solver was developed taking into consideration distributed environments that do not feature a network file system. Without such feature, some processes (workers) may not have access to the problem description (DIMACS file), which is necessary to prepare their solvers for the problem computation. Therefore, the content of the file must be propagated to all workers, so that they have knowledge of the problem to be solved. Since the master must obligatorily contain the problem description, since it is responsible for the application ignition, it must assure that the content of the file is distributed to each worker. Such distribution is performed at the beginning of the solver execution, by packing the content of the problem description into a buffer and broadcasting such buffer to all processes.

MiniSat+ contains a parser (PbParser) which is used to extract the problem specification to be solved, from a file descriptor. Information such as number of variables, number of constraints, objective function and constraints are data obtained during the parsing process, where constraints are handled and allocated one by one along the parsing process. During the parse, information is extracted line by line and components (such as literals and coefficients) are aggregated to compose constraints (or the objective function), passing them as arguments into the MiniSat+ solver interface. Such aggregated components must be intercepted and packed, since they represent the description of the problem to be solved which need to be propagated to the workers, so that their solver obtains the similar state as if they parsed the file descriptor directly. To achieve such goal, the parser was modified so that the master not only prepares its solver state, using the extracted components from the file descriptor, but also to record the necessary information that must be propagated to the workers so that they achieve the same solver state as the master. So during the parsing process, all the extracted information is stored in a structure, which is packed into a buffer (using the data packing technique explained in Section 5.1.1) by a packing routine, after the parsing process finishes. The resulting buffer is then transferred using the MPI_Bcast function, which broadcasts the buffer to all worker processes. MPI_Bcast is used due to its complexity of $O(n \log(p))$, where $n$ is the message size and $p$ the number of processes in the communication, that is better than using the pair MPI_Send/MPI_Recv to all processes, which represents a complexity of $O(np)$. Since the buffer size differs for each problem instance and using a static size buffer is not a suitable solution, two consecutive MPI_Bcast calls must be performed, where the first broadcast the size of the buffer and the second broadcasts the buffer itself.

During the implementation process, the modified parsing process suffered several improvements, that resulted in significant reductions in the buffer size to be broadcast, reducing consequently the time spent in the problem content broadcast. The first approach implemented is referenced as pre-normalization approach and the optimized approach, currently implemented, is referenced as pos-normalization approach.

5.1.2.1 Pre-normalization approach

Pre-normalization approach was the first attempt to propagate the problem content to the workers. It is referred as pre-normalization due to the absence of normalization and simplification techniques in the information transferred. This means that constraints contained in the
typedef struct{
    int n_vars;
    int n_constrs;
    bool hasGoal;
    vec<vec<char> > goal_ps;
    vec<Int> goal_Cs;
    vec<Constraint*> constrs;
}SolverInput;

typedef struct{
    vec<vec<char> > ps;
    vec<Int> Cs;
    int ineq;
    Int rhs;
}Constraint;

Figure 5.2: Pre-normalization SolverInput structure.

file descriptor do not suffer any simplification before the broadcast and all the content of file
is allocated into a structure without modifications. In other words, the data propagated to the
workers is a direct representation of the problem file description.

To achieve the pre-normalization approach, the parser was modified so that the information
contained in the file were allocated directly into the SolverInput structure, presented in
Figure 5.2. Once the parsing process is finished, the SolverInput instance contains all the
information of the problem description. Afterwards, the structure goes through a packing rou-
tine, where the structure content is encapsulated into a buffer to be transmitted to all workers.
Figure 5.3 demonstrates how data was arranged in the buffer, representing the message packet
which was broadcast.

After the broadcast, all processes have the same instance of SolverInput. Taking advantage
of an extra routine, the processes prepare their solvers using the structure as the problem
description, executing the MiniSat+ interface calls with the correct fields of the SolverInput
as arguments and performing the normalization and unit propagation techniques locally. Unlike
the original parser approach, the pre-normalization parser does not contain any calls to the
MiniSat+ solver interface, having only the responsibility to fill the SolverInput. Therefore, the
master uses the same routine as the workers, to prepare its solver for the problem computation,
after the broadcast process is performed.

Although the approach seems simple and features small modifications to the parser, it suffers
from great amount of data that is sent in the buffer, which can be avoided by taking advantage
of some techniques to “shrink” the data needed to pack. Taking such knowledge into account,
a new approach was adopted, which reduces significantly the amount of data to be transferred
to the workers.

5.1.2.2 Pos-normalization approach

The pos-normalization approach is the current implementation for the problem content propa-
gation. It was developed as a solution to avoid redundant and unnecessary data to be packed
and broadcast. The pos-normalization approach, like the name indicates, records the problem
constraints to be broadcast to the workers after their normalization. As referred in Section 2.2.2,
the normalization technique may simplify a constraint, such as removing literals and reducing
constant values, or even discard it, if already satisfied. Taking such property in consideration, it
is more advantageous to pack the constraints after their normalization than before, since some
constraints may be simplified, by reducing their size, while others do not need to pack, since
they may be already satisfied. Consequently, this will result in a reduction in the buffer size
to broadcast, therefore reducing the time and effort to propagate the problem content to the
workers.

Although all constraints packed are simplified, other solutions are used to minimize the
data to be packed. Another important and efficient technique is to avoid the variables string representation in the objective function and constraints. Variables are represented as integers in the solver and the conversion of the string representation of a variable into the correspondent integer value is done using the function getVar. The integer values are assigned by occurrence, where the first variable to occur in the file gets the value 0 and so on. So if all processes create the variables in the same order in their core solver, it is guaranteed that they all obtain the same variables integer representation and the same literals representation. Therefore, it is possible to substitute their string representation in the buffer by their integer representation, which occupies less amount of space.

For an efficient implementation of the new approach, the parser process was modified to use and fill a new SolverInput structure, presented in the Figure 5.4. To obtain and record the normalized constraints, the function addConstr of MiniSat+, which performs the normalization of constraints, was modified to return the content of the constraints after their normalization. To ensure the order of the variables creation, the getVar function was modified to fill a string vector containing the variables ordered by their occurrence. Such vector is one of the first things to pack, since it is indispensable to ensure that all processes contain the same integer representation for each literal. The packing routine was also modified to pack the new fields of the SolverInput structure. Variables are no longer identified by strings, due to their substitution by their literals integer representation, and the relation field is no longer need, since all the normalized constraints have the same relation after the normalization.

There is another important aspect that must be taken into account, during the normalization process, is the possible variable assignments. Some literals are removed from the constraints due to their assignment during the normalization process. Such assignments can not be predicted by the workers and they must be propagated so that all workers have knowledge of their values when solving the problem. The assigned literals are removed from the constraints but their assigned values are important since some assigned literals may reside in the objective function and their values are important when creating the objective function restrictions. To record all assignments performed during the normalization process, the function addConstr not only returns the normalized constraint content but also the assigned literals during the constraint normalization. Figure 5.5 presents the buffer structure that is broadcast in the pos-normalization approach.

This approach proved to be more efficient, compared with the pre-normalization approach, regarding to the amount of data to be packed in the buffer. With just a bit more of computation to process the file descriptor data, the buffer sizes obtained and broadcast are less than a half.
typedef struct{
    int n_vars;
    int n_constrs;
    bool hasGoal;
    vec<Lit> goal_ps;
    vec<Int> goal_Cs;
    vec<Constraint*> constrs;
    vec<Lit> addedUnits;
}SolverInput;

typedef struct{
    vec<Lit> ps;
    vec<Int> Cs;
    Int rhs;
}Constraint;

Figure 5.4: Pos-normalization SolverInput structure

of the sizes obtained in the pre-normalization approach, reducing significantly the time spent in the message broadcast. Moreover, it saves effort to the workers, regarding to the constraints normalization, since they are propagated already normalized.

5.1.3 Optimization Process Communication

The optimization process communication is referred to all the communications performed during the problem optimization process, which are indispensable for the master’s management of the workers computation during such process. After the problem content broadcast, which involves the transfer of the same message to all workers, the master initiates the optimization process and starts to assign tasks to the workers and compute the problem optimization solution taking into account their responses. Such communication involves the master and each individual worker, where the master sends commands to each worker directly and waits for a response for it. Such commands can be a task assignment or simply a stop message to cancel the computation of the current task.

In this type of communication, each message contains a different content and a different sender/receiver, so the communication must be point-to-point, using the pair of functions MPI_Send/MPI_Recv and always including the master. The optimization process communication is composed of several types of messages, which are explained and detailed in the next subsections. Each message type is identified by a tag, a distinct integer identifier, and a list of the available tags are presented in Figure 5.6.

5.1.3.1 Task Assignment Messages

Task assignment messages are the mechanism for the master to assign the necessary tasks to the available workers. As explained in Section 4.1, the assignment of tasks is a key aspect of the proposed solver and each task consists of a specific objective function restriction to be computed by a worker. A task assignment message is sent when a worker becomes idle or followed after a task abortion message. Since the workers only need to know the constant value to restrict their problem, using the objective function, the data to be sent is an Int value (a GMP value).

The task assignment messages are sent to individual workers by the master, resorting to point-to-point communication, using the pair of functions MPI_Send/MPI_Recv, where the messages are sent with the tag TASK_TAG to identify their purpose. The MPI_Send function blocks a process until the given message is fully allocated in the destination process buffer. Since all messages dispatched by the master contain short buffers, it is unlikely that a worker process buffer becomes full, meaning that the probability of the master block for an exceeded period of time is very low. Considering such characteristic, it was decided to perform standard-mode send
operations for task assignment messages, instead of the *immediate* send operations (MPI\_Isend) that do not block the sending process but are more laborious to manage.

### 5.1.3.2 Task Response Messages

After assigning a task, the master waits for a response from the worker who is supposed to compute it. Responses are indispensable for the master to manage and restrict the goal interval until an optimal solution is found. After a worker compute a solution for the assigned task, it must notify the result to the master, being it satisfiable or not. Such notification is performed by sending a message to the master, identifying the result obtained and the correspondent task value that originated the result. Identifying the task associated to the response is important to avoid mistakes on the master process side. Since the master assigns immediately a new task after performing a task abortion, without waiting for a stop message response, it is possible that the response of the aborted task reaches the master after the assignment of the new task. If the responses are not marked with the correspondent task, the master can wrongly assume that the response received is related to the last task assignment, which could compromise the search for the optimal solution.

Figure 5.7 presents the task response buffer structure. The task response message is composed of three data components packed into a buffer. The first components is a boolean value that indicates if the worker computed a satisfiable solution or not. The second component is the Int restriction value of the task associated to the response. The third component is only present in satisfiable responses and corresponds to the model found that satisfied the computed task. Sending the model is indispensable for the master to compute the objective function value associated to such model and restrict the upper bound of the optimization interval according to such value. It is also important for the master output the model when the optimal solution is found.

Unlike the task assignment messages, sending the task response messages with MPI standard-mode transfers can degrade drastically the performance of the workers. Since all responses are sent to the master, the probability of the buffer of the master process become full is very high. To avoid the possibility of a worker to block during the transfer of a task response message, due to the buffer overflow of the master process, all transfers of the task response are performed using the MPI immediate send function (MPI\_Isend), which performs the message transfers in background, allowing the process to proceed its computation. The task response messages are sent using the GOAL\_RESPONSE\_TAG tag to identify their purpose.
5.1.3.3 Stop Messages

Due to the optimization interval restriction during the optimization process, some assigned tasks may become outside of the interval and useless in the search for the optimal solution. Such tasks must be discarded and their computation substituted with a new task within the boundaries of the current optimization interval. However, the workers know nothing about the optimization interval state and it is the responsibility of the master to order the workers to stop computing their current task, if necessary, before assigning a new one. Ordering a worker to stop is performed by sending a stop message, that is sent using the tag STOP_TAG. A stop message is composed of a boolean value, representing if the stop order is only to abort the current task computation or if it is an order to abort the current task and finalize the process. Upon receiving the stop message, the worker warns its task abortion by sending a message to the master, using the tag STOP_RESPONSE_TAG. Such stop response is important for the master to keep record of which tasks are currently being computed by all workers and present such information to the user.

When the master finds the optimal solution, it must order all workers to terminate for a successful and clean application exit. It is performed by sending a stop message with the boolean value as true. Such message orders the abortion of the current task computation, if the worker is computing any, and indicates that the worker must terminate afterwards.

5.1.3.4 Debug Messages

To allow a better debug of the MPBO solver, some optional communication features can be used to obtain extra information about the computation state of each worker. Using the available debug options, the workers inform the master not only with the indispensable information for the correct execution of the solver, but also additional information about their status during each task computation.

Using the solver option -statedebug, the workers notify their current state to the master. The notifications are performed resorting to signal messages, using the last four tags in Figure 5.6, which are sent before any execution of the mechanisms that compose the workers. Each time a worker executes the converter, restore or SAT engine mechanisms, at any given task, the correspondent signal is sent to the master to warn that such mechanism is about to be executed. So, the master can present to the user what is the current mechanism that is being executed by each worker. Although this option is very useful for development, its usage must be avoided due to the performance degradation from the excessive communication.

Another two options available are the -idletimedebug and -memorydebug. This two options extend the responses of the workers, such as tasks and stop responses, to give additional
information about each task computation. The -idletimedebug option includes an additional double value to the response, that represents the time spent by the worker in idle state, waiting for the assignment of the task computed. The -memorydebug option also includes another boolean value to the responses and such value represents the peak memory allocated by the worker at the time that the given task was computed.

5.2 Master Implementation

Characterized as the core of the solver, the master process is responsible to manage the workers computation until the optimal solution for the problem is found. It is composed by a MiniSat+ solver instance, to help in the problem content simplification, and by a problem optimization loop to find the optimal solution of the PBO problems. Using point-to-point communication, the master guides the computation of each individual worker during the optimization process and use their computation results to restrict the optimization interval until only one solution remains on it, which is considered the optimal solution of the problem in question.

Figure 5.8 presents the pseudo code that represents the master process execution flow. The master process was the component in which most modifications were introduced during the implementation process, to reduce the overhead of communication and for a better assignment of tasks during the optimization process. The following sections detail the important mechanisms that compose the master process, regarding the management of workers and the optimization interval management, and their evolution during the implementation process.

5.2.1 Workers Management

Workers are the entities responsible for discovering the solutions for the optimization interval values and the higher the number of workers available, the higher the number of searches to be performed simultaneously. However, workers do not communicate with each other and have no clues about the searches performed by others. They are dependent on a central entity and it is the master process the entity responsible of guiding all workers during the execution of the solver. So, an efficient guidance and management of the workers effort, during the resolution of the problems, are important aspects to maximize the performance of the MPBO solver.

Being the central entity of the solver, the first thing the master does, after parsing the file descriptor, is to initialize the workers by broadcasting a boolean value. If the parsing process does not detect any root conflict in the problem, the value to broadcast is true, otherwise the problem is considered unsatisfiable and there is no point to initialize the workers, broadcasting the value false to terminate the worker processes immediately. After the workers are ordered to
`bool conflict_found = Parse_File_Problem(&buffer); Initialize_Workers(conflict_found); if(!conflict_found) print("Problem unsatisfiable"); else {
  Broadcast_File_Input(buffer);
  Propagate();
  interval = Obtain_Interval();
  while(true){ // Problem Optimization process
    Stop_OutOfRange_Workers(&workers_state, interval);
    Task_Assignment(&workers_state, interval);
    response = Wait_for_Response();
    if(response == Root_Unsat){ // Problem with root conflict
      print("Problem unsatisfiable");
      break;
    }
    Restrict_Interval(&interval, response);
    if(interval < 1){
      print_solution();
      break;
    }
  }
  Quit_Workers();
} exit();

Figure 5.8: Pseudo code of the Master process.

proceed, the problem content is broadcast (as described in Section 5.1.2) to prepare the workers to the problem computation.

Entering into the problem optimization loop, the master manages each worker individually. Using the available messages to assign or cancel tasks (detailed in Section 5.1.3), the master can control the computation of each worker and manage them depending on the problem optimization interval state, to obtain a better management of resources. At the beginning, one task is assigned to each worker and every time a worker terminates a given task, a new task from the current optimization interval is assigned to it. Each time a worker is computing a task that became outside the current optimization interval, such worker is ordered to abort the current task and a new one is assigned.

After an optimal solution is found, the master sends a stop message with the boolean flag to true (exit signal) to abort the current task being computed and to order the termination of all processes. However, stop messages are detected at different time frames by the workers and to ensure that no worker is stuck in a MPI block send routine, the master cannot perform the MPI_Finalize function until all workers detect the stop message. To guarantee that all processes terminate at the same time, not only to ensure the correct termination but also to measure the correct termination wall time, a MPI barrier is used before each process execute the MPI_Finalize function.
5.2.2 Optimization Interval Management

The objective of the proposed solver is to find the optimal solution for PBO problems, which are determined by calculating an initial optimization interval with all possible solution values of the objective function and restricting such interval until only one value is covered by such interval, that is considered the optimal solution value for the problem. As the central entity and having the knowledge of all tasks computed, including their results, the master manages the optimization interval using information obtained with the responses from the computed tasks (approach explained in Section 4.1).

The optimization interval is a discrete interval covering all values between the highest unsatisfiable objective function solution known and the lowest satisfiable solution computed. However, many of the values covered by the interval can not be obtained using the objective function, since they do not correspond to any valid variable assignment, and using them to create tasks may result in redundant computations. Unfortunately, identifying the values in the interval that correspond to valid variable assignments in the objective function is a $2^N$ complexity problem (where $N$ is the number of variables in the objective function) and there is no feasible solution to identify them in reasonable time. So, each value in the optimization interval is considered a possible solution for the problem and it is up to the task assignment mechanism to identify the best values to assign, to speedup the optimization interval restriction.

Added all problem constraints and propagated all units assigned during the parsing process, the master calculates the first boundaries of the interval. The lower bound is the sum of all activated coefficients of the objective function and all not assigned literals to false, if the corresponding coefficients are positive, or to true, if the corresponding coefficients are negative. The upper bound is calculated the other way around, not assigned literals to true, if the corresponding coefficients are positive, or to false, if the corresponding coefficients are negative. After the boundaries are identified, the workers responses will restrict the interval until the optimal solution is found. Each satisfiable response will obtain a solution and the corresponding model is used to calculate the given objective function solution, which will substitute the current upper bound of the interval. Unsatisfiable responses will substitute the lower bound of the interval with the objective function restriction value used to create such tasks.

In the current implementation of the solver, the master does not compute any solution for the problem. However, in an initial implementation of the solver, the master computed the problem (without any restriction) to obtain a first model for the problem and with it, obtain the first upper bound of the optimization interval before starting to assign tasks to the workers. Such problem computation is less expensive than any other computation including objective function restrictions and has the advantage of identifying unsatisfiable problems without the workers cooperation. Unfortunately, during such time, all workers were idle, waiting to be initialized, which resulted in a huge waste of resources in satisfiable hard problem instances, where the time to compute the problem without restriction may take minutes, meaning that all workers would be inactive during such time. So, to overcome such waste in the current implementation of the solver, the master does not compute any solution for the problem and starts to assign tasks to the workers as fast as possible, to avoid the waste of resources. But to continue to take advantage of the fast computation of the initial solution, the first task assigned by the master is the computation of the problem without objective function restrictions. Such task results in a fast response, identifying if the problem is unsatisfiable or resulting in a fast new upper bound to the optimization interval.
Int closerMinimum(Int min, Int max){
    Int count = worst_min;
    int i;
    if(count >= min && count < max) return count;
    if(negative_boundary > -1)
        for(i = 0; i <= negative_boundary; i++){
            if(goal_function[i] >= min && goal_function[i] < max)
                return goal_function[i];
            count -= goal_function[i];
            if(count >= min && count < max) return count;
        }
    for(i = negative_boundary+1; i < goal_function.size(); i++){
        if(goal_function[i] >= min && goal_function[i] < max)
            return goal_function[i];
        else if(goal_function[i] >= max) break;
        count += goal_function[i];
        if(count >= min && count < max) return count;
    }
    for(i = 0; i <= negative_boundary; i++){
        count += goal_function[i];
        if(count >= min && count < max) return count;
    }
    return min;
}

Figure 5.9: Code of the closerMinimum function.

5.2.3 Task Assignments

Since the master is responsible to manage the problem optimization interval, it must assign
tasks efficiently to guarantee that each worker computation has the best impact on the interval
restriction. A faster interval restriction results in a faster search to obtain the optimal solution
for a given problem. To identify the best tasks to assign, taking into account the optimization
interval state, the master contains a task assignment mechanism that identifies relevant objec-
tive function values to assign to each individual worker. The mechanism performs successive
divisions of the optimization interval to perform a binary search on its values and uses the
values selected in each division to create tasks for the workers.

The task assignment mechanism is triggered at the beginning of the solver execution, where
all workers are waiting for a task assignment, and each time the master receives a response
from a worker. Since a response shrinks the optimization interval, some tasks being computed
may become useless for the problem optimization and therefore, their computation need to be
substituted with new tasks. When triggered, the mechanism starts by identifying the number of
workers that need a new task assignment and performs successive divisions of the optimization
interval until all the necessary tasks are created and allocated to the identified workers. The
first task is created using the upper bound value of the interval and the second task with the
lower bound value. The remaining tasks are created during the interval division, where the
value that divides the interval in two equal sub-intervals, referenced as division value, is used to
create a new task. The process is repeated with the resulted sub-intervals until the number of
Int closerGoalValue(Int div, Int min, Int max){
    int i, leftCindex = findLeftCoefficient(div);
    Int closer;
    if(leftCindex < 0)
        // There is no goal coefficient equal or lower than div
        closer = NoLeftCoeffApproach(div, min, max);
    else
        // There is at least one goal coefficient equal or lower than div
        closer = LeftCoeffApproach(div, leftCindex, min, max);
    return closer;
}

Figure 5.10: Code of the closerGoalValue function.

tasks needed for the necessary assignments are created or there are no remaining values in the interval to select, meaning that all values were already used for task assignments. If during the division process, a created tasks is already being computed by a worker, such task is discarded and another value is selected to create a different task.

Since not all values of the interval correspond to valid variable assignments in the objective function, using the division values directly to create the tasks may lead to redundant computations by the workers, situation that was already explained in Section 4.1. Considering this problem, the task assignment mechanism was implemented with two approaches to select the values to create the tasks, according to the division values calculated: the simple approach, which uses the division values directly to create the tasks (can be select using the solver option -goaldivoptim-off), and a new optimized approach (selected by default), which identifies possible computable objective function solutions, closer to the division values calculated, to create the tasks. The optimized approach consists in a set of two functions that use heuristics to find computable objective function solutions closer to the values obtained during the interval division. One function is used to identify an objective function solution that better substitutes the lower bound of the interval for a task creation and the other is used to identify an objective function solution that is closer to a given division value. If the functions do not identify a possible objective function solution or the identified one is outside the boundaries of the optimization interval, the actual division value is selected to create the task. The heuristics do not cover all possible solutions of the objective function, due to the complexity of such problem, but cover a small set of solutions that may result in better tasks than the division values calculated. To help in the heuristics computation, the coefficients of the objective function are ordered by ascending order and the index of the higher negative coefficient of the objective function is recorded (negative_boundary), to delimit the negative and positive coefficient areas of the objective function.

The first function is known as closerMinimum (presented in Figure 5.9) and it is used to find a computable objective function solution to substitute the lower bound value of the interval when creating the corresponding task. To identify such solution, the function closerMinimum traverses the coefficients of the objective function by ascending order and the first coefficient identified as being higher than the lower bound value of the interval, and lower than the upper bound value, is selected to create the task. However, the actual coefficients only represent a tiny set of the possible objective function solutions and a bigger set must be considered to increase the number of alternatives. To obtain a large set to traverse, the sum of the coefficients is also compared with the lower bound value. The function initializes the sum value with the
Int NoLeftCoeffApproach(Int div, Int min, Int max){
    Int closer = div; int i;
    if(negative_boundary > 0){
        // There is at least 2 negative coefficients in the goal function
        closer = goal_function[negative_boundary];
        for(i = negative_boundary-1; i >= 0; i--){
            closer += goal_function[i];
            if(closer <= div) break;
        }
        if(i <= 0){
            // Did not reach the limit
            if(closer <= min) closer -= goal_function[i];
            else{
                // Choose the closer value to the mid
                Int upper = closer - goal_function[i];
                if(abs(div - closer) > abs(upper - div)) closer = upper;
            }
        }
    }
    if(closer >= max) closer = div;
    return closer;
}

Figure 5.11: Code of the NoLeftCoeffApproach function.

smaller solution that can be obtained in the objective function (referenced as worst_min) and each time a coefficient is traversed (and is not selected), it is added to the sum. If during the process, the sum becomes higher than the lower bound value but lower than the upper bound value, it is selected for the task creation. If all coefficients are traversed and none of the values compared are selected, the function returns the actual value of the lower bound to create the corresponding task.

The second function is known as closerGoalValue (presented in Figure 5.10) and is used to identify the objective function values closer to the division values calculated during the interval division. The function implements an algorithm that traverses a set of objective function solutions to identify a possible value in such set that is closer to the provided division value. To reach such goal, the function starts to calculate the index of the higher coefficient of the objective function that is smaller or equal than the given division value, which will be referred in the rest of this document by the left coefficient. If it obtains an index value equal or higher than 0 means that was identified a left coefficient for the division value. In the case of a negative index value means that all coefficients are greater than the division value and no left coefficient can be deduced. Taking into account the existence of a left coefficient or not, the function uses two different approaches to identify the closer objective function value.

If no left coefficient is identified for the given division value, the NoLeftCoeffApproach function (presented in Figure 5.11) takes place to identify a closer objective function solution. The NoLeftCoeffApproach function traverses the sum of the negative coefficients of the objective function (if there is at least two), starting by the highest coefficient and adding the remaining coefficients by descending order, until the sum reaches a smaller value than the division value provided. If such smaller value is identified, it is compared to the previous value of the sum.
Int LeftCoeffApproach(Int div, int index, Int min, Int max){
    Int closer = goal_function[index];
    if(closer == div) return closer;
    int i = negative_boundary+1;
    for(; i < index; i++)
        if(closer + goal_function[i] <= div)
            closer += goal_function[i];
        else break;
    if(i < index && closer + goal_function[i] <= max){
        // Did not reach the limit
        // Choose the closest value to the mid
        Int upper = closer + goal_function[i];
        if(abs(div - closer) > abs(upper - div))
            closer = upper;
    }
    if(closer <= min) closer = div;
    return closer;
}

Figure 5.12: Code of the LeftCoeffApproach function.

obtained (the last higher value than the division value) and the closest to the division value
provided is returned. If the sum does not reach a smaller value, the actual division value is
returned.

In the case of a left coefficient is identify for the given division value, another approach
takes place to find the closer objective function solution and such approach is implemented
in the function LeftCoeffApproach (presented in Figure 5.12). When used, the function
LeftCoeffApproach returns the actual left coefficient value if it is equal to the division value
provided. Otherwise, the function starts to increment, to the left coefficient value, all the
smaller positive coefficients by ascending order, to achieve a higher solution value than the
division value provided. The coefficients are incremented one by one until the sum reaches a
higher value than the division value or until the left coefficient index is reached, meaning that
there is no more smaller coefficients to increment. If the higher sum value is identify, there are
two alternative values that can be chosen to be the closer objective function solution, the value
of the sum that exceeded the division value and the value of the sum before such crossover.
The value to be selected, between the two alternatives, is the one that is closer to the division
value provided. If the sum never exceeds the division value, the value of the sum at the end of
the process is returned as the closest objective function solution.

To compare the behavior of the two task assignment approaches, Figure 5.13 presents the
assignments performed by the two approaches, for five workers, when solving a problem with
an objective function \( f(x) = 2x_1 + 5x_2 + 10x_3 + 100x_4 \). The horizontal lines in the figure
represent the initial scope of the problem optimization interval and in this particular scenario,
all possible objective function solutions are located near the boundaries of the interval, leaving
a huge gap in the middle of the interval without solutions. As observed in the example, only
one task assigned by the simple approach corresponds to an objective function solution and
three of the tasks, besides to not correspond to objective function solutions, represent the same
task computation. Since the simple approach uses the division values directly to create the
tasks, the division of the interval leads the simple approach to assign three tasks to explore the
middle area of the interval. Such tasks result in redundant computations since there are no objective function solutions between the values 17 and 100 of the optimization interval, where the tasks with values 30, 59 and 88 represent the same computation as a task with value 100. So, with such assignments, three of the workers are computing the same area of the optimization interval and such situation results in a waste of resources. Regarding the optimized approach, it is observable that all five tasks assigned correspond to objective function solutions and they are evenly distributed among the optimization interval, which results in a better management of the workers computation for the problem optimization, when compared to the assignments of the simple approach. However, such scenario is not always possible for all problem instances since the optimized approach does not guarantee the identification of a closer objective function solutions for all interval division values and eventually the actual division values are used to create the tasks to assign. So, it is possible that in some scenarios, the optimized approach behaves exactly as the simple approach, performing task assignments using the division values directly.

5.3 Worker Implementation

The worker process is the entity responsible to obtain the problem solutions, during the solver execution. Its purpose is to compute the tasks assigned by the master, which correspond to the computation of the problem containing an specific objective function restriction and check if the problem stills satisfiable with such restriction. After its initialization, the worker receives the content of the problem (a simplified version of the information contained in the problem file description) and prepare its solver state to compute the tasks that will be assigned, by adding the necessary information to the solver (objective function and PB constraints) and converting the initial PB problem into an equivalent SAT problem. If during the preparation a conflict is found, which is considered a root conflict, the problem unsatisfiability is reported to the master and the worker process begins its termination routine. If no conflict is found, the worker enters into a loop, where the worker will compute the assigned tasks, until the master orders to terminate.
Each iteration consists in a task reception, followed by its computation, and the solution report to the master. Since tasks can become useless for the search of the optimal solution, an iteration (a task computation) may be aborted immediately upon the reception of a stop message from the master. The computation of each task consists of restricting the problem using the value sent by the master, by including the proper objective function restriction constraint into the problem, converting such constraint into SAT clauses and verifying if the problem stills satisfiable or not. The worker process features a restore mechanism, which backtracks the current solver state into the initial state, when an assigned task is less restrictive (an higher restriction value) than the previous task. Figure 5.14 presents the pseudo-code that represent the execution flow of the worker process.

Developing the restore and stop mechanisms was the big challenge, during the implementation of the worker process, and many improvements were made to guarantee the efficiency and correctness of the tasks computation. The implementation of each mechanism is detailed in the following sections, describing all solutions found to implement them and the reason for the chosen solutions in the current implementations.

5.3.1 Restore Mechanism

As explained in Section 4.2.3.3, MiniSat+ features the incremental addiction of constraints, which give the possibility of adding constraints into the problem between each search. Such functionality facilitates the mechanism for the workers to compute tasks consecutively only by including the new restrictions into the problem. However, such property is only true if the tasks to be computed have a lower restriction than the last ones, since they do not conflict with each other (property described in Section 4.1). Yet, workers may need to compute a task with a higher value than the last task computed and to allow the computation of the new task, the
clauses related to the last task have to be removed. For such cases, the worker process features a restore mechanism which is triggered each time an assigned task has a higher value than the previous task computed.

The restore mechanism is responsible to restore the solver into the initial state, which represents the solver state without any clauses related to objective function restrictions. It is composed of two routines, the \texttt{SaveState} function which saves the initial state of the solver (composed only by problem constraints) and the \texttt{RestoreState} which backtracks the current state of the solver into the initial state, cleaning up all occurrences of the objective function restriction constraints. The \texttt{SaveState} function is triggered before adding the first objective function restriction constraint and the \texttt{RestoreState} function is triggered each time a worker needs to compute a less restrictive task. Initially it was implemented using the backup approach, but due to its drawbacks, the mechanism evolved to the mark and sweep approach. Another promising approach was also found as an alternative to the mark and sweep approach, the clause grouping solution, but due to the absence of efficient mechanisms for its successful implementation, it was left apart.

5.3.1.1 Backup Approach

The backup solution was the first implementation of the restore mechanism and consists in creating a copy of the solver state before any objective function restriction is added to the solver (known as initial state) and use such copy to restore the solver to the initial state when necessary. So, each time the \texttt{RestoreState} is triggered, the solver returns to the initial state, containing no restrictions and ready to compute any objective function restriction.

To achieve such backup, the MiniSat+ solver was modified to distinguish each type of constraints added to it (problem constraints and objective function restriction constraints) and to backup its state before the first objective function restriction is added to the solver. Since constraints are added to the solver in order, starting by the problem constraints and adding objective function restriction constraints afterwards, it is guaranteed that after the first restriction constraint is added, there will be no more problem constraints to be included, meaning that the backup state does not need to change until the problem resolution terminates. To perform the backup, the SAT solver and the Converter contain a duplication of each field, where the duplicates contain a copy of the content of the original fields, at the time that the function \texttt{SaveState} is called. When the function \texttt{RestoreState} is called, the content of the original fields are deallocated and substituted with the content of their backup replicas. The backup of the SAT solver includes the initial problem clauses (activity, literals and coefficients), the problem variables and their related information (activity, assigned value, watches and reason) and the assignment trail.

Such solution is flawless but requires too much extra memory, due to the duplication of the original problem data, which may lead the processes to run out of memory quickly. Besides, the \texttt{RestoreState} routine is too heavy, since all the content of the fields must be deallocated and a copy of the replicas must be allocated afterwards, representing a high number of \texttt{malloc} and \texttt{free} instructions. As a consequence of the drawbacks, the restore mechanism changed into the mark and sweep approach, which proved to be more efficient in terms of memory and performance.

5.3.1.2 Mark and Sweep Approach

The mark and sweep is the final implementation of the restore mechanism, which substituted the backup approach. It appeared as a solution for the high demand of memory imposed by the backup approach, by eliminating the need to duplicate the problem clauses and therefore, less field duplication of the solver state. Each state information is identified as being from the
initial problem or being from a restriction, and each time a restore is performed, all content related to problem restrictions is removed from the solver state. Although some fields need to remain duplicated, such as variable activities, such solution reduces significantly the amount of memory required to perform a correct and efficient restore.

To implement the mark and sweep approach, the Clause class was modified to contain a new flag \textit{goal}. The flag is assigned to true if the clause corresponds to a problem restriction and false if corresponds to the initial problem. The references of the clauses are organized into two vectors, \textit{clauses} and \textit{goal\_clauses}, so that the restriction clauses can be quickly accessed during the restore process. When the \texttt{RestoreState} function is called, all the clauses referenced by the \textit{goal\_clauses} vector are removed and consequently, their references in the \textit{watches} of their literals are also removed. Since variables are identified by integers and they are added in order, where first are created the problem variables and second the problem restriction variables, there is an integer value that separates each type of variables, which corresponds to the last initial problem variable. Such value is recorded during the \texttt{SaveState} routine and during each restore, all information related to a variable higher than such value is removed from the watches and order\_heap fields. The activities and reasons of such variables are also removed. The activity and the assignment values of the problem variables are restored using the content of their duplicated replicas, which contain their values at the time of the \texttt{SaveState} routine call. The assignment trails is another field that remains replicated. Due to the nature of the Converter mechanism code, it was impossible to find a mark and sweep approach for the clausifier state. So the Converter remains with the backup approach as the restore mechanism to its state.

Another big challenge of the mark and sweep approach was on the \texttt{SimplSolver} extension of the SAT solver. It removes variables to simplify the problem and some modifications to its mechanisms were needed to prevent the removal of the initial problem variables. Although its simplification mechanism is disabled by default, it is possible to turn it on and continue to restore the solver correctly and efficiently.

5.3.1.3 Clause Grouping Approach

The clause grouping approach was identified as a promising alternative approach to implement the restore mechanism. The approach consists of grouping the clauses by restriction values, where each clause is added to the solver containing an extra literal related to their restriction value (known as group literal). Different restriction clauses will contain their own group literal and each clause contains the group literal of the higher restriction that is associated to it, meaning that all restriction values equal or lower than such restriction need to consider the clause as part of the problem and higher values need to ignore such clause to obtain the correct solution. So each time a task needs to be computed, all group literals associated to restriction values equal or higher than the task to computed are assigned to false, which will not discard the clauses, and all group literals associated to restriction values lower than the assigned task are assigned to true, turning the clauses already satisfiable to not interfere with the computation of the assigned task. Group literals are only used to activate or inactivate the clauses for a given search computation and they cannot be treated as literals of the problem. So, their intended values are assigned before initializing the search.

This solution could speedup the conversion process, since clauses are never removed and could be reutilized after the solver is restored, avoiding the need for consecutive conversions of the same clauses. Many clauses are present in almost all problem restrictions and with the mark and sweep approach, such clauses are converted several times during the solver execution, which make some computation of the converter redundant. Much of the effort of the workers is spent...
inline bool stop_signal(){
    int stop;
    // Check if was received a Stop message
    MPI_Iprobe(0, STOP_TAG, comm, &stop, MPI_STATUS_IGNORE);
    return stop ? true : false;
}

void *check_stop_sentinel(void *ptr){
    while(1){
        sleep(opt_sentinelttime);
        if(!stop_flag && stop_signal())
            stop_flag = true;
    }
}

Figure 5.15: Function to check stop messages reception.

Figure 5.16: Thread code to check stop messages reception.

in the conversion of constraints and the clause grouping solution could speedup significantly such computation. However, it was impossible to find a way of grouping the clauses successfully, since it is not possible to discover, during the conversion of constraints, the highest restriction value that a clause must be associated.

5.3.2 Stop Mechanism

A stop mechanism is indispensable for a better management of tasks in an asynchronous application for distributed systems. In the MPBO solver, when some worker finishes the computation of a given task, it will result in a shrinkage of the optimization interval of the given problem and other workers may become computing tasks associated to values outside of the new interval (which become irrelevant to obtain the optimal solution), resulting in a waste of effort and resources. Such tasks need to be aborted, so that new task assignments can take place in their computation.

Although it appears to be a simple functionality, it can be one source of great performance degradation, if not implemented carefully. Unlike other types of communication, such as task assignments and responses, stop messages are sent by the master at undefined times, during the problem execution, so their reception must be constantly checked by the workers. Since checking if a message was received involves calls to the MPI library, it imposes some overhead to the worker execution and the tradeoff between stop response speed and worker performance must be taken into account. Much of the worker effort is spent in the converter and SAT solver engines and such mechanisms are composed of several loops, where each iteration is computed in a order of milliseconds. So, including MPI calls to check for stop messages inside such loops will increase the execution time of each iteration and consequently degrade the performance of the mechanisms.

The first implementation of the stop mechanism was the inclusion of calls to the `stop_signal` function (implementation in Figure 5.15) inside the loops of the converter and inside the `search` function of the SAT solver, to immediately abort the task computation, if the function returns `true`. The `MPI_Iprobe` is a non-blocking MPI function, that tests if a message with a given
tag was received and is waiting to be handled. For the purpose of the solver, the `stop_signal` function uses `MPI_Iprobe` to check for received messages with a tag `STOP_TAG`, returning `true` if there is a reception or `false` otherwise. Although the detection of stop messages is practically instantaneous after their reception, such implementation proved to be very inefficient. Due to the characteristics and fast computation of the functions in the converter and SAT solver engine, including the call to the `stop_signal` on this functions grows drastically their execution time, which may lead to a delay of an order of minutes, when solving hard problem instances. So, an alternative approach was taken into account to solve such problem, maintaining an acceptable response time to stop messages.

As a solution to reduce the overhead imposed by the successive calls to the `MPI_Iprobe`, during the converter and the SAT solver engine, each worker spawns a new thread to handle the necessary calls to such function. The thread executes a function (presented in Figure 5.16) that calls the `stop_signal` function at a given configurable interval (1 second by default) and will set a global boolean `stop_flag` to `true` if a stop message reception is detected. With such solution, the worker main thread is then alleviated, with respect to MPI calls to discover stop messages, and only needs to check the `stop_flag` value to terminate the given task computation, if `true` is verified. So every call to the `stop_signal` function, in the main thread, is replaced by a boolean check to the `stop_flag`, which reduces significantly the overhead of the main thread and the effort needed for the Converter and the SAT solver engines. The main thread sets the `stop_flag` value back to `false`, when it pulls the stop messages from the MPI message buffer.

The ideal approach for the stop mechanism would be taking advantage of MPI Remote Memory Access (RMA). This functionality is present in many recent MPI implementations and can be used to develop MPI applications that behave closely to shared-memory programming applications. The intended goal of RMAs is to give the possibility of a given node to read or
bool stop = true;
MPI_Win_lock(MPI_LOCK_EXCLUSIVE, worker_id, 0, win);
MPI_Put,&stop, 1, MPI::BOOL, worker_id, 0, 1, MPI::BOOL, win);
MPI_Win_unlock(worker_id, win);

Figure 5.18: Stop check routine code.

write a particular region of memory in another target node, with no explicit interaction required by the target node or any other node. To achieve such functionality, the MPI uses abstract objects called windows that intuitively specify regions of memory of a process that have been made available for remote operations by other MPI processes. The closest approach to such goal is the Passive Target RMA type, where the target process allocates a certain amount of memory with the MPI_Alloc_mem function and the other MPI processes can access such memory with the MPI_Get and MPI_Put to read and write on it. To handle the remote memory access concurrency, a process must acquire a lock, from the target process (using MPI_Win_lock), before performing a set of remote memory operations and release it afterward (using MPI_Win_unlock). Figure 5.17 illustrates how MPI manages passive target remote memory accesses.

Using the MPI Passive Target RMA functionality, it would be possible to implement a stop mechanism less costly for workers. Instead of using stop messages to communicate a task abortion and have an extra thread in each worker to detect their arrival (to update the flag), it would be possible to set the stop flag of the workers remotely, without the need for point-to-point communication. Workers would have their stop flag allocated with MPI_Alloc_mem and the master would be responsible to set such flag to true when a stop is intended. Using such solution, the workers only need to check their stop flag value to detect a task abortion, without the need for any message “sentinel” and extra effort, and the master would set the flags of the workers directly, instead of sending stop messages. The necessary code for the master change remotely a worker stop flag is presented in Figure 5.18.

Unfortunately, OpenMPI [19] has a poor passive target synchronization implementation so far, limiting the asynchronous behavior of MPI lock/unlock, which made this last solution be pushed aside until the OpenMPI supports the minimum requirements to use Passive RMA efficiently. During some attempts in an implementation for this approach, it was noticed that in the current OpenMPI implementation, for a Passive RMA operation succeed, the target process needs to enter into the MPI library to complete the operation. This property violates the intended asynchronous behavior of the Passive Target RMAs and makes this solution improper for the proposed solver, where the workers (target processes) need to avoid MPI calls during their computation.
Chapter 6

Solver Evaluation

Concluded the implementation process and achieved a stable version of the MPBO solver, it was performed a benchmark test to analyze and evaluate the MPBO solver performance and efficiency. This chapter defines and details the evaluation performed, presenting the methodology used and the benchmark test results obtained, including some commentaries and conclusions about the results. To easily prove its efficiency, the MPBO solver was compared against the MiniSat+ 1.0 [14] and pwbo 2.2 [30] solvers, to have an idea of the reductions in the execution times achieved. Since the PBO research lacks from popular distributed solvers, it was difficult to discover a solver with the same characteristic of the MPBO solver to compare with. However, since MPI allows the execution of its applications in single machines, spawning all processes through the local cores, it was possible to make a comparison of executions between our solver and the pwbo solver, which is a parallel solver developed for shared-memory environments.

The evaluation performed focuses the solver scalability, comparing its execution when using different numbers of workers available, and the impact of the two task assignment approaches implemented, in the solver execution. To accomplish the performance evaluation of the MPBO solver and gather a reasonable range of results for a more fine grained conclusion, it was used a benchmark suite containing 47 PBO problem instances from the problem optimization category in the 2010 PB competition [25]. The problem instances selected range, in difficulty, from easy to hard instances, with execution times ranging from milliseconds to hours, when using MiniSat+. Since the pwbo solver does not handle high decimal numbers for the problem coefficients, the benchmark suite contains two groups of instances. The first group is composed of 35 instances from the BIGINT category and the second group by 12 instances from the SMALLINT category, which were used exclusively for the evaluations performed with the pwbo solver. The benchmark list is presented in Table 6.1, which includes some details of each instance and their execution time using the MiniSat+ solver. The full names of the benchmarks are present in Appendix B. All problem instances are ordered by ascending difficulty and grouped by answer type (unsatisfiable and satisfiable).

All benchmark executions were performed in the available grid at INESC-ID. The INESC-ID grid infrastructure is composed of 13 Intel Q6600 (quad-core) machines at 2.4 GHz, with 8 GB RAM, running the OpenSuse 11.0 Linux with Kernel 2.6.25.16. All machines are connected by a gigabit private subnet and feature a network file system (OpenAFS 1.4.x). To compile and run MPI applications, the grid provides the OpenMPI 1.2.8 and Glibc 2.8 libraries.
<table>
<thead>
<tr>
<th>Benchmark ID</th>
<th>Benchmark Name</th>
<th>Optimal Solution</th>
<th># Variables</th>
<th># Constraints</th>
<th>Sequential Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>normalized-658.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>533</td>
<td>2638</td>
<td>0.02</td>
</tr>
<tr>
<td>2</td>
<td>normalized-mps-v2-20-10-bore3d.opb</td>
<td>UNSAT</td>
<td>9277</td>
<td>342</td>
<td>8.43</td>
</tr>
<tr>
<td>3</td>
<td>normalized-mps-v2-20-10-hands.opb</td>
<td>UNSAT</td>
<td>14116</td>
<td>205</td>
<td>2.77</td>
</tr>
<tr>
<td>4</td>
<td>normalized-mps-v2-20-10-momentsmpl1.opb</td>
<td>UNSAT</td>
<td>41495</td>
<td>40182</td>
<td>3.49</td>
</tr>
<tr>
<td>5</td>
<td>normalized-mps-v2-20-10-norms.opb</td>
<td>UNSAT</td>
<td>60075</td>
<td>2257</td>
<td>4.04</td>
</tr>
<tr>
<td>6</td>
<td>normalized-factor-sizeN=20-sizeP=11-sizeQ=20-900543.opb</td>
<td>UNSAT</td>
<td>1</td>
<td>251</td>
<td>0.01</td>
</tr>
<tr>
<td>7</td>
<td>normalized-factor-sizeN=20-sizeP=11-sizeQ=20-66708-max.opb</td>
<td>UNSAT</td>
<td>39663726592</td>
<td>56</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>normalized-mps-v2-20-10-testprob.opb</td>
<td>UNSAT</td>
<td>9277</td>
<td>242</td>
<td>0.43</td>
</tr>
<tr>
<td>9</td>
<td>normalized-mps-v2-20-10-bandm.opb</td>
<td>UNSAT</td>
<td>14116</td>
<td>205</td>
<td>2.77</td>
</tr>
<tr>
<td>10</td>
<td>normalized-mps-v2-20-10-momentum1.opb</td>
<td>UNSAT</td>
<td>41495</td>
<td>40182</td>
<td>3.49</td>
</tr>
<tr>
<td>11</td>
<td>normalized-mps-v2-20-10-nesm.opb</td>
<td>UNSAT</td>
<td>60075</td>
<td>2257</td>
<td>4.04</td>
</tr>
<tr>
<td>12</td>
<td>normalized-factor-sizeN=20-sizeP=11-sizeQ=20-900543.opb</td>
<td>UNSAT</td>
<td>1</td>
<td>251</td>
<td>0.01</td>
</tr>
<tr>
<td>13</td>
<td>normalized-mps-v2-20-10-norms.opb</td>
<td>UNSAT</td>
<td>60075</td>
<td>2257</td>
<td>4.04</td>
</tr>
<tr>
<td>14</td>
<td>normalized-mps-v2-20-10-momentsmpl1.opb</td>
<td>UNSAT</td>
<td>41495</td>
<td>40182</td>
<td>3.49</td>
</tr>
<tr>
<td>15</td>
<td>normalized-mps-v2-20-10-norms.opb</td>
<td>UNSAT</td>
<td>60075</td>
<td>2257</td>
<td>4.04</td>
</tr>
<tr>
<td>16</td>
<td>normalized-mps-v2-20-10-momentsmpl1.opb</td>
<td>UNSAT</td>
<td>41495</td>
<td>40182</td>
<td>3.49</td>
</tr>
<tr>
<td>17</td>
<td>normalized-mps-v2-20-10-norms.opb</td>
<td>UNSAT</td>
<td>60075</td>
<td>2257</td>
<td>4.04</td>
</tr>
<tr>
<td>18</td>
<td>normalized-rand59f7dd.cudf.paranoid.opb</td>
<td>UNSAT</td>
<td>344</td>
<td>6360</td>
<td>19076</td>
</tr>
<tr>
<td>19</td>
<td>normalized-rand69f8c.cudf.paranoid.opb</td>
<td>UNSAT</td>
<td>2669</td>
<td>6382</td>
<td>19351</td>
</tr>
<tr>
<td>20</td>
<td>normalized-rand69f8c.cudf.paranoid.opb</td>
<td>UNSAT</td>
<td>408</td>
<td>6845</td>
<td>21270</td>
</tr>
<tr>
<td>21</td>
<td>normalized-rand69f8c.cudf.paranoid.opb</td>
<td>UNSAT</td>
<td>0</td>
<td>5835</td>
<td>16856</td>
</tr>
<tr>
<td>22</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>23</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>24</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>25</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>26</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>27</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>28</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>29</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>30</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>31</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>32</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>33</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>34</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
<tr>
<td>35</td>
<td>normalized-rand58554.cudf.trendy.opb</td>
<td>UNSAT</td>
<td>3534613</td>
<td>6922</td>
<td>20527</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SMALLINT Category</th>
<th>Benchmark Name</th>
<th>Optimal Solution</th>
<th># Variables</th>
<th># Constraints</th>
<th>Sequential Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>37</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>38</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>39</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>40</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>41</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>42</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>43</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>44</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>45</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
<tr>
<td>46</td>
<td>normalized-ul31e1.opb</td>
<td>162</td>
<td>444</td>
<td>1408</td>
<td>6.94</td>
</tr>
</tbody>
</table>

Table 6.1: List of problem instances that compose the benchmark suite.
6. SOLVER EVALUATION

6.1 Evaluation Methodology

Evaluating the performance of a distributed application is a complex task and exploring characteristics such as scalability and speedups is very important to demonstrate and analyze the application efficiency over a distributed environment. To achieve such type of evaluation in the MPBO solver, it is indispensable to analyze the solver execution under different number of resources, measuring the execution times obtained and comparing the evolution of speedups during the execution environment changes. Therefore, four scenarios were selected to test the solver scalability, using 4, 12, 24 and 48 workers. So, all instances of the first group of the benchmark suite were executed using the four execution scenarios with the objective of testifying the solver scalability and check the scenario that obtains the best performance.

To have a reference point when checking the solver efficiency, the MPBO solver is compared to MiniSat+, which can be considered a sequential version of the proposed solver. The speedups obtained in each execution scenario were calculated using the following formula:

\[ S_p = \frac{T_1}{T_p} \]

where \( T_1 \) is the execution time obtained using MiniSat+ and \( T_p \) is the execution time obtained in the execution scenario using \( p \) processors (where \( p \) includes the master process and all the available worker processes). However, the execution times measured in each solver are different, turning the comparison somehow unfair. In MiniSat+ solver was measured in terms of the cpu time, since such measurement is already implemented in the solver, and in the MPBO solver was measured in terms of the wall time resorting to the MPI function MPI_Wtime. To achieve a reasonable execution time measurement in the proposed solver, the wall time measured in each execution is the wall time spent by the master process, from the beginning of the execution until the discovery of the optimal solution (or the proof that the problem is unsatisfiable).

Since the proposed solver features two approaches to assign tasks to the workers, the other point of interest of the evaluation is to analyze the difference in the solver performance obtained when adopting each assignment approach, to the same number of workers. So, for a more fine grained collection of results, the two approaches were used in each execution scenario to observe the gain of performance of the optimized approach against the simple approach.

To summarize, all instances of the BIGINT category were tested 8 times, where each execution differs in the number of workers available and in the task assignment approach used. To evaluate the two approaches, the number of tasks assigned and the number of tasks successfully computed in each execution were measured, and the overhead imposed by each approach identified.

Regarding the comparison between the MPBO and pwbo solvers, all executions were performed in a single machine from the grid, where all instances of the SMALLINT category were executed with both solvers. To take advantage of the 4 cores available, the pwbo solver was executed using 4 threads and the MPBO solver was executed using 3 workers, leaving one core dedicated for the master process.

6.2 Evaluation Results

6.2.1 Execution Times

All measurements performed during the benchmark tests are presented in Tables A.1 to A.11, in Appendix A. The evaluation focused on the satisfiable instances of the benchmark suite, since the purpose of the MPBO solver is to solve efficiently hard instances of the PBO problem.
Figure 6.1: Execution times obtained in the BIGINT instances, using MiniSat+ and the MPBO solvers.

However, five unsatisfiable instances were included in the benchmark suite to demonstrate the loss of performance when using the distributed solver for such type of instances. Table A.1 presents the wall times and speedups obtained in all execution scenarios when solving the unsatisfiable instances with the distributed solver, where it can be observed that no execution reached a speedup higher than 0.8.

Regarding the satisfiable instances, the execution times and speedups obtained in all BIGINT instances, during the benchmark tests, are presented in Table A.2 and the chart in Figure 6.1 illustrates the execution times obtained for the scenarios using the optimized task assignment approach in all executions. The results obtained by the MiniSat+ solver are also presented and is observable that the MPBO solver achieves better results while the difficulty of the instances grows, which proves the MPBO solver efficiency when solving hard problem instances of the PBO problem. However, the chart does not provide a clear comparison between the execution scenarios of the MPBO solver but is observable that there is no best MPBO execution scenario to solve all problem instances. To enforce the results in Figure 6.1, the average speedup obtained by all execution scenarios when solving all BIGINT instances is presented in Figure 6.2. From the chart, it is observable that all execution scenarios achieved better results when compared to MiniSat+. However, the execution scenario with 48 workers stood out better when solving all instances, achieving an average speedup value of 500 when compared to the MiniSat+ executions.

Considering the comparison between the MPBO and pwbo solvers, the wall times obtained by the both solvers in all SMALLINT problem instances are presented in Table A.11, including the speedups obtained by the MPBO solver when compared to the pwbo solver executions. Figure 6.3 presents the results obtained by the three solvers in all executions of the SMALLINT
6. SOLVER EVALUATION

instances and it is observable that the MPBO solver achieve better results than pwbo, achieving an average speedup of 6. However, it is also observable that MiniSat+ achieved better results in the first instances of the benchmark, beating the two parallel solvers when solving the easiest instances. But it is notorious the discrepancy of the pwbo results compared to the others.

6.2.2 Task Assignment Approaches

Note that each problem instance has a unique search scenario and considering that the optimal solution can be located at any position of the problem optimization interval, dividing the interval by 4 workers may lead a worker directly to the optimal solution, but dividing the same interval by 12 workers may lead all workers away from the optimal solution value, resulting in a more time consuming search. So, the performance obtained in each instance does not depend only on the number of resources used but also on how the optimization interval is divided and where the optimal solution is located in such interval, which is observable by the results in the chart of Figure 6.1 where there is no perfect scenario for all problem instances. However, using an efficient mechanism to assign tasks is one key aspect to better restrict the optimization interval and consequently improve the optimization search.

Comparing the two available approaches of the MPBO solver, in the chart of Figure 6.2, it is perfectly clear that the optimized approach obtained the best results, independently on the number of available workers used, achieving an average of 80% more speedup when compared to the simple approach. Since the optimized approach assigns more accurate tasks, regarding to the objective function, it can obtain significant reductions in the number of assignments along the optimization of the problem, reducing consequently the time and communication during the optimization search. Tables A.3 to A.6, in the Annex, indicate the number of tasks assigned and the number of tasks successfully computed, measured during the execution of all BIGINT instances. Figure 6.4 presents the average number of task assigned during all execution scenarios when solving all problem instances and it is notorious the reduction of task assignments of the optimized approach when compared to the simple approach, reducing up to 70% the number of tasks assigned. In terms of overhead, between the two approaches, the optimized approach has a higher overhead, since it performs linear searches over the objective function to identify the task to assignment. So, depending on the number of variables in the objective function, the time spent in the assignment mechanism may not be insignificant. However, observing Table A.7 in Annex, the optimized approach never exceeds 1% of the total time spent to compute the problem instances, which is considered a negligible overhead for the great reduction of task
During the task assignments measurement, it was noticed that an excessive number of tasks are being assigned during each execution and the relation between the tasks assigned and the ones that are successfully computed is not satisfactory, observing that only 1% of the tasks assigned are aborted. Such abortion rate indicates that much of the tasks computed do not contribute for a great restriction of the optimization interval. Such behavior was not the expected one and during the solver executions was observed that tasks restricting the lower bound of the interval are continually being assigned and such tasks are the ones that result in a smaller restriction of the optimization interval. To conclude if such type of tasks are responsible for the huge amount of assignments during the executions, it was excluded all tasks related to interval lower bound values in all task assignment measurements and observed their impact in the task assignment approaches. The results obtained are presented in Figure 6.5. Taking into account such results, the optimized approach continues to reduce 10% of the tasks that do not correspond to lower bound values, which stills a reasonable reduction rate for the task being assigned. However, comparing the results from the charts 6.4 and 6.5, it is observed that only 50% of the total task assigned, for the execution scenario with 48 workers, and 5% for the remaining execution scenarios, correspond to task assignments that are not related to the interval lower bound values. So, much of the tasks assigned during an execution correspond to restrictions to the lower bound of the optimization interval, which typically are faster to compute, hence their large amount, but obtain a lower impact in the computation of the optimization interval. Such excessive number of tasks augments the communication during the solver execution and hampers the effort of the master process to manage the remaining and more promising tasks, since the lower bound restriction tasks are successively being created and assigned.

Figure 6.3: Execution times obtained in the SMALLINT instances, with MiniSat+, pwbo and MPBO solvers.


6. SOLVER EVALUATION

Figure 6.4: Average number of task assignments when solving the BIGINT instances.

Figure 6.5: Average number of task assignments, excluding lower bound tasks, when solving the BIGINT instances.

6.2.3 Solver Scalability

To achieve a reasonable evaluation for the solver scalability, it was used a broad number of workers to test our MPBO solver. The maximum number of workers used was 48 since the grid of INESC-ID contains 13 machines, resulting in a total of 52 cores available. Whereas the proposed solver adopts the popular Task Farm approach containing one master entity to manage all resources, such entity can become the bottleneck for the solver efficiency and the more workers available, the more time is needed to handle and manage all the available workers, consequently extending the window of time to handle each individual worker. Considering such, 48 workers is considered a large number of resources for a single entity to manage.

In Section 6.2.1 was concluded that regardless the task assignment approach used, the execution scenarios using the highest number of resources achieved the best results during the solver evaluation. So, for the evaluation presented, the growth of workers did not affect the solver performance, rather the opposite, such growth increased the solver performance when solving the benchmark suite. However, to better conclude the master efficiency when managing all available workers, the average idle time of the workers was measured during each execution. Table A.8, in the Annex, presents the average wall time spent by the workers in idle state at each execution and Figure 6.6 compares the average idle time obtained in the execution...
scenarios using 4 and 48 workers (using the optimized task assignment approach). As it can be observed in the chart of Figure 6.6, the idle time measurements obtained in hard problem instances using 48 workers are not much higher than using 4 workers, where they never exceeded the 7 seconds of idle time, which is a negligible time comparing to the whole execution time to solve the problem instances. Such results show that the master manages successfully the 48 workers during the solver executions and since the idle times observed are very low, it may be possible to use a higher value of workers, in the execution of the solver, without a drastic degradation in the solver efficiency.
Chapter 7

Conclusions and Future Work

Considering that the adoption of distributed solutions to solve the SAT problem was a success in the SAT research, resulting in several efficient distributed solvers, the aim of this work was to make a contribution for the PBO research by proposing a distributed solution to efficiently solve the PBO problem. Using the knowledge acquired by SAT research in the parallel computing field, it was possible to develop and implement a PBO solver based in the approaches proposed by the popular distributed SAT solvers available. The MPBO solver developed in this thesis is a distributed solution that contribute for the research in the PBO problem, when it comes to its migration into the parallel computing world. The solver was implemented using the popular task farm approach, adopted in many distributed SAT solvers, and using the MPI API, the industry de-facto standard for the implementation of distributed applications.

The MPBO solver proved to be a very effective solution to solve hard PBO problem instances, achieving speedup values of 500 when compared to the efficient MiniSat+ solver and speedup values of 6 when compared to the pwbo solver. The MPBO solver features two task assignment approaches: the simple approach, where the optimization search space is divided using a technique similar to binary search, and the optimized approach, where the optimization search space is divided considering the objective function to minimize. The optimized approach achieves a reduction of 70% in the number of task assignments, during an execution, when compared to the simple approach, which consequently, improves the solver performance up to 80% when solving hard problem instances. Although the master/slave topology tends to degrade the performance of the distributed applications, due to the overhead imposed by the master process when managing a large amount of resources, the proposed solver accomplished significant results when facing execution environments using up to 48 workers. Such results prove the great level of scalability of the solver, which is the result of the efficient worker management performed by the master process during the solver executions. The reduced average idle time spent by the workers, during the executions, proves the effectiveness of the master process to handle all workers in a feasible time window.

Although an efficient distributed PBO solver have been developed, there are some aspects that should be considered to improve the solver efficiency and were left for future work. One aspect to consider is the development of new approaches for the task assignment mechanism. The MPBO solver implements a binary division approach to divide the optimization interval to be computed by the several resources and features an optimized approach which proved to be efficient to reduce the number of task assignments during the solver execution. Such reduction is a result of the assignment of more promising tasks than a pure binary division approach. However, as in SAT research, there are no perfect heuristics to better partition the search space...
to be explored and the same can be said to the optimization search space in the PBO problems. Besides the binary division of the optimization interval, it can be advantageous to explore new solutions to divide the optimization interval. Another approach could be to prioritize the upper boundary area of the interval to be computed first or even a completely different method to assign tasks resorting to assumptions for the objective function variables.

Another aspect that can be explored and implemented is the learnt clause sharing technique, which is used in several parallel SAT solvers. In the MPBO solver, each task corresponds to a different objective function restriction and consequently, a different sub-problem to compute. Many of the learnt clauses created during each task computation cannot be shared among the workers since they are produced due to conflicts involving the objective function restriction imposed by the task assigned. However, learnt clauses produced by conflicts encountered in the initial problem (without involving objective function restriction clauses) can be shared among the workers since they do not corrupt the computation of other tasks and in turn can speed up their computation. Since all clauses in the MPBO solver are marked as being from a objective function restriction or from the initial problem, it is easy to identify if the produced learnt clauses are related to the given objective function restriction or not. So, defining a good heuristic to restrict the learnt clauses to share, it is possible to implement an efficient approach to share the “right” learnt clauses through the workers and speed up the computation of each task. However, such mechanism must be carefully designed to avoid excessive communications among the processes.

Another aspect that can be considered for future worker, to improve the solver’s performance, is the partition of the SAT search space of each task to be computed in parallel by several resources, which can be useful to allow other workers to help in a task computation of another. In all MPBO solver executions, when the optimization interval covers less values than the number of workers available, some workers become idle until the optimal solution for the problem is found. Such situation represents a waste of resources and in the current implementation of the solver, such situation is inevitable. Since the workers of the MPBO solver are based on MiniSat+, they use an independent SAT solver engine to compute their tasks. So, another SAT solver can be used instead of the MiniSat engine adopted. Such substitution could be performed using any distributed SAT solver with the same characteristic of the MPBO Solver (implemented with C++ and the MPI API), to allow the partition of a given task computation to all idle workers, when the scenario explained above is verified. One possible candidate for such substitute could be the PMSat [23] solver, since it has the same characteristics as the MPBO solver.
References


REFERENCES


Appendices
## Appendix A

### Benchmark Results

<table>
<thead>
<tr>
<th>ID</th>
<th>MiniSat+</th>
<th>4 Workers</th>
<th>12 Workers</th>
<th>24 Workers</th>
<th>48 Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simple</td>
<td>Optimized</td>
<td>Simple</td>
<td>Optimized</td>
<td>Simple</td>
</tr>
<tr>
<td></td>
<td>Time (s)</td>
<td>Sp</td>
<td>Time (s)</td>
<td>Sp</td>
<td>Time (s)</td>
</tr>
<tr>
<td>1</td>
<td>0.02</td>
<td>0.03</td>
<td>0.63</td>
<td>0.73</td>
<td>0.03</td>
</tr>
<tr>
<td>2</td>
<td>0.43</td>
<td>0.61</td>
<td>0.61</td>
<td>0.71</td>
<td>0.61</td>
</tr>
<tr>
<td>3</td>
<td>2.77</td>
<td>4.12</td>
<td>4.13</td>
<td>0.67</td>
<td>4.04</td>
</tr>
<tr>
<td>4</td>
<td>3.49</td>
<td>4.59</td>
<td>4.53</td>
<td>0.77</td>
<td>4.59</td>
</tr>
<tr>
<td>5</td>
<td>4.04</td>
<td>5.65</td>
<td>5.62</td>
<td>0.72</td>
<td>5.67</td>
</tr>
</tbody>
</table>

Table A.1: Solver execution times and speedups in the unsatisfiable instances.
Table A.2: Solver execution times and speedups in the satisfiable instances.

| ID | Optimized  | Simple  | 12 Workers | 24 Workers | 48 Workers | Wall Optimized | Wall Simple | Wall 12 Workers | Wall 24 Workers | Wall 48 Workers |
|----|------------|---------|------------|------------|------------|----------------|-------------|------------------|----------------|----------------|----------------|
| 1  | 0.4        | 0.3     | 0.6        | 0.8        | 1.0        | 0.6            | 0.8         | 0.6              | 0.8            | 0.6            | 0.8            |
| 2  | 0.2        | 0.1     | 0.3        | 0.4        | 0.5        | 0.3            | 0.4         | 0.3              | 0.4            | 0.3            | 0.4            |
| 3  | 0.1        | 0.08    | 0.14       | 0.16       | 0.18       | 0.14           | 0.16        | 0.14             | 0.16           | 0.14           | 0.16           |
| 4  | 0.05       | 0.03    | 0.05       | 0.06       | 0.07       | 0.05           | 0.06        | 0.05             | 0.06           | 0.05           | 0.06           |
| 5  | 0.02       | 0.01    | 0.02       | 0.02       | 0.03       | 0.02           | 0.02        | 0.02             | 0.02           | 0.02           | 0.02           |
| 6  | 0.01       | 0.005   | 0.01       | 0.01       | 0.01       | 0.01           | 0.01        | 0.01             | 0.01           | 0.01           | 0.01           |

Note: The table contains execution times and speedups for different solver configurations and environments.
## A. BENCHMARK RESULTS

### Table A.3: Measured tasks during the satisfiable BIGINT instances execution, using the simple assignment approach (Part 1).
Table A.4: Measured tasks during the satisfiable BIGINT instances execution, using the simple assignment approach (Part 2).
<table>
<thead>
<tr>
<th>ID</th>
<th>Tasks Measured</th>
<th>4 Workers</th>
<th>12 Workers</th>
<th>24 Workers</th>
<th>48 Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Tasks Assigned</td>
<td>Tasks Computed</td>
<td>Tasks Assigned</td>
<td>Tasks Computed</td>
</tr>
<tr>
<td>6</td>
<td>All</td>
<td>7</td>
<td>3</td>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>4</td>
<td>1</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>All</td>
<td>61</td>
<td>40</td>
<td>107</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>28</td>
<td>13</td>
<td>76</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>All</td>
<td>15</td>
<td>9</td>
<td>56</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>9</td>
<td>7</td>
<td>49</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>All</td>
<td>36</td>
<td>27</td>
<td>190</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>27</td>
<td>24</td>
<td>83</td>
<td>27</td>
</tr>
<tr>
<td>10</td>
<td>All</td>
<td>102</td>
<td>91</td>
<td>110</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>18</td>
<td>12</td>
<td>69</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>All</td>
<td>35</td>
<td>26</td>
<td>75</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>28</td>
<td>26</td>
<td>63</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>All</td>
<td>116</td>
<td>92</td>
<td>153</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>59</td>
<td>43</td>
<td>114</td>
<td>37</td>
</tr>
<tr>
<td>13</td>
<td>All</td>
<td>73</td>
<td>47</td>
<td>151</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>55</td>
<td>40</td>
<td>138</td>
<td>25</td>
</tr>
<tr>
<td>14</td>
<td>All</td>
<td>43</td>
<td>34</td>
<td>79</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>27</td>
<td>27</td>
<td>68</td>
<td>21</td>
</tr>
<tr>
<td>15</td>
<td>All</td>
<td>5</td>
<td>2</td>
<td>13</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>2</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>All</td>
<td>43</td>
<td>36</td>
<td>73</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>13</td>
<td>9</td>
<td>44</td>
<td>15</td>
</tr>
<tr>
<td>17</td>
<td>All</td>
<td>72</td>
<td>49</td>
<td>160</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>52</td>
<td>39</td>
<td>135</td>
<td>36</td>
</tr>
<tr>
<td>18</td>
<td>All</td>
<td>36</td>
<td>33</td>
<td>79</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>51</td>
<td>38</td>
<td>110</td>
<td>15</td>
</tr>
<tr>
<td>19</td>
<td>All</td>
<td>31</td>
<td>13</td>
<td>48</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>12</td>
<td>6</td>
<td>36</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>All</td>
<td>72</td>
<td>52</td>
<td>99</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>excl. Lower</td>
<td>30</td>
<td>14</td>
<td>85</td>
<td>14</td>
</tr>
</tbody>
</table>

Table A.5: Measured tasks during the satisfiable BIGINT instances execution, using the optimized assignment approach (Part 1).
<table>
<thead>
<tr>
<th>4 Workers</th>
<th>12 Workers</th>
<th>24 Workers</th>
<th>48 Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tasks</td>
<td>Assigned</td>
<td>Computed</td>
<td>Assigned</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.6: Measured tasks during the satisfiable BIGINT instances execution, using the optimized assignment approach (Part 2).
<table>
<thead>
<tr>
<th>ID</th>
<th>4 Workers</th>
<th>12 Workers</th>
<th>24 Workers</th>
<th>48 Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Simple Assignment</td>
<td>Optimized Assignment</td>
<td>Simple Assignment</td>
<td>Optimized Assignment</td>
</tr>
<tr>
<td>6</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>11</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>13</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>14</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>15</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>16</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>17</td>
<td>0.01</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>18</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>19</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>20</td>
<td>0.00</td>
<td>0.01</td>
<td>0.00</td>
<td>0.03</td>
</tr>
<tr>
<td>21</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>22</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>23</td>
<td>0.47</td>
<td>2.41</td>
<td>0.18</td>
<td>0.09</td>
</tr>
<tr>
<td>24</td>
<td>0.15</td>
<td>0.92</td>
<td>0.05</td>
<td>0.41</td>
</tr>
<tr>
<td>25</td>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>0.06</td>
</tr>
<tr>
<td>26</td>
<td>0.21</td>
<td>0.25</td>
<td>0.10</td>
<td>0.27</td>
</tr>
<tr>
<td>27</td>
<td>0.41</td>
<td>5.83</td>
<td>0.15</td>
<td>2.89</td>
</tr>
<tr>
<td>28</td>
<td>0.11</td>
<td>0.69</td>
<td>0.02</td>
<td>0.11</td>
</tr>
<tr>
<td>29</td>
<td>0.00</td>
<td>0.06</td>
<td>0.00</td>
<td>0.08</td>
</tr>
<tr>
<td>30</td>
<td>0.11</td>
<td>5.96</td>
<td>0.12</td>
<td>1.44</td>
</tr>
<tr>
<td>31</td>
<td>0.06</td>
<td>4.18</td>
<td>0.14</td>
<td>2.58</td>
</tr>
<tr>
<td>32</td>
<td>0.03</td>
<td>2.78</td>
<td>0.01</td>
<td>0.97</td>
</tr>
<tr>
<td>33</td>
<td>0.18</td>
<td>2.31</td>
<td>0.17</td>
<td>1.38</td>
</tr>
<tr>
<td>34</td>
<td>0.42</td>
<td>5.45</td>
<td>0.12</td>
<td>2.82</td>
</tr>
<tr>
<td>35</td>
<td>1.31</td>
<td>0.43</td>
<td>0.45</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table A.7: Wall times spent, in seconds, in the task assignment mechanism, during the satisfiable benchmark tests.
Table A.8: Average wall time (in seconds) spent by the workers in idle state during the satisfiable instances execution.

<table>
<thead>
<tr>
<th>Workers</th>
<th>Simple Assignment</th>
<th>Optimized Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>24</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>48</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>96</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>192</td>
<td>0.05</td>
<td>0.02</td>
</tr>
<tr>
<td>384</td>
<td>0.07</td>
<td>0.03</td>
</tr>
<tr>
<td>768</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1536</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>3072</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>6144</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>12288</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>24576</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>49152</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>98304</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>196608</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>393216</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>786432</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1572864</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>3145728</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>6291456</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>12582912</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>25165824</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>50331648</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>100663296</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

*Note: The table continues with more rows.*
<table>
<thead>
<tr>
<th>ID</th>
<th>4 Workers</th>
<th>12 Workers</th>
<th>24 Workers</th>
<th>48 Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average Peak Memory</td>
<td>Higher Peak Memory</td>
<td>Average Peak Memory</td>
<td>Higher Peak Memory</td>
</tr>
<tr>
<td>6</td>
<td>73</td>
<td>147</td>
<td>24</td>
<td>146</td>
</tr>
<tr>
<td>7</td>
<td>146</td>
<td>146</td>
<td>146</td>
<td>146</td>
</tr>
<tr>
<td>8</td>
<td>147</td>
<td>147</td>
<td>147</td>
<td>147</td>
</tr>
<tr>
<td>9</td>
<td>146</td>
<td>147</td>
<td>147</td>
<td>147</td>
</tr>
<tr>
<td>10</td>
<td>149</td>
<td>150</td>
<td>149</td>
<td>151</td>
</tr>
<tr>
<td>11</td>
<td>146</td>
<td>146</td>
<td>146</td>
<td>146</td>
</tr>
<tr>
<td>12</td>
<td>147</td>
<td>147</td>
<td>147</td>
<td>147</td>
</tr>
<tr>
<td>13</td>
<td>151</td>
<td>155</td>
<td>151</td>
<td>155</td>
</tr>
<tr>
<td>14</td>
<td>155</td>
<td>157</td>
<td>152</td>
<td>156</td>
</tr>
<tr>
<td>15</td>
<td>37</td>
<td>149</td>
<td>12</td>
<td>149</td>
</tr>
<tr>
<td>16</td>
<td>280</td>
<td>332</td>
<td>322</td>
<td>340</td>
</tr>
<tr>
<td>17</td>
<td>174</td>
<td>182</td>
<td>174</td>
<td>183</td>
</tr>
<tr>
<td>18</td>
<td>282</td>
<td>362</td>
<td>224</td>
<td>246</td>
</tr>
<tr>
<td>19</td>
<td>167</td>
<td>178</td>
<td>163</td>
<td>179</td>
</tr>
<tr>
<td>20</td>
<td>258</td>
<td>298</td>
<td>225</td>
<td>282</td>
</tr>
<tr>
<td>21</td>
<td>79</td>
<td>157</td>
<td>26</td>
<td>157</td>
</tr>
<tr>
<td>22</td>
<td>120</td>
<td>163</td>
<td>40</td>
<td>163</td>
</tr>
<tr>
<td>23</td>
<td>186</td>
<td>233</td>
<td>164</td>
<td>169</td>
</tr>
<tr>
<td>24</td>
<td>355</td>
<td>483</td>
<td>296</td>
<td>483</td>
</tr>
<tr>
<td>25</td>
<td>164</td>
<td>165</td>
<td>165</td>
<td>165</td>
</tr>
<tr>
<td>26</td>
<td>327</td>
<td>492</td>
<td>265</td>
<td>368</td>
</tr>
<tr>
<td>27</td>
<td>248</td>
<td>290</td>
<td>245</td>
<td>371</td>
</tr>
<tr>
<td>28</td>
<td>253</td>
<td>285</td>
<td>179</td>
<td>240</td>
</tr>
<tr>
<td>29</td>
<td>164</td>
<td>165</td>
<td>165</td>
<td>165</td>
</tr>
<tr>
<td>30</td>
<td>305</td>
<td>358</td>
<td>265</td>
<td>303</td>
</tr>
<tr>
<td>31</td>
<td>259</td>
<td>290</td>
<td>264</td>
<td>294</td>
</tr>
<tr>
<td>32</td>
<td>256</td>
<td>290</td>
<td>216</td>
<td>281</td>
</tr>
<tr>
<td>33</td>
<td>282</td>
<td>328</td>
<td>328</td>
<td>489</td>
</tr>
<tr>
<td>34</td>
<td>256</td>
<td>289</td>
<td>226</td>
<td>283</td>
</tr>
<tr>
<td>35</td>
<td>294</td>
<td>412</td>
<td>185</td>
<td>206</td>
</tr>
</tbody>
</table>

Table A.9: Peak Memory obtained in the satisfiable instances, using the optimized task assignment approach.
<table>
<thead>
<tr>
<th>Workers</th>
<th>Average Peak Memory</th>
<th>Higher Peak Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>523</td>
<td>549</td>
</tr>
<tr>
<td>12</td>
<td>523</td>
<td>549</td>
</tr>
<tr>
<td>24</td>
<td>523</td>
<td>549</td>
</tr>
<tr>
<td>48</td>
<td>523</td>
<td>549</td>
</tr>
</tbody>
</table>

Table A.10: Peak Memory obtained in the satisfiable instances, using the simple task assignment approach.
### A. BENCHMARK RESULTS

#### Table A.11: Execution times obtained in the pwbo, MPBO and MiniSat+ solvers, in all SMALL-INT instances.

<table>
<thead>
<tr>
<th>ID</th>
<th>CPU Time (s)</th>
<th>Wall Time (s)</th>
<th>Wall Time (s)</th>
<th>$T_{pwbo}/T_{MPBO}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
<td>8.63</td>
<td>74.77</td>
<td>8.38</td>
<td>8.93</td>
</tr>
<tr>
<td>37</td>
<td>9.39</td>
<td>83.95</td>
<td>11.24</td>
<td>7.47</td>
</tr>
<tr>
<td>38</td>
<td>10.98</td>
<td>384.20</td>
<td>12.95</td>
<td>29.67</td>
</tr>
<tr>
<td>39</td>
<td>11.68</td>
<td>64.67</td>
<td>19.70</td>
<td>3.28</td>
</tr>
<tr>
<td>40</td>
<td>40.09</td>
<td>265.50</td>
<td>59.23</td>
<td>4.48</td>
</tr>
<tr>
<td>41</td>
<td>50.59</td>
<td>578.22</td>
<td>188.91</td>
<td>3.47</td>
</tr>
<tr>
<td>42</td>
<td>67.34</td>
<td>152.70</td>
<td>93.96</td>
<td>2.05</td>
</tr>
<tr>
<td>43</td>
<td>111.37</td>
<td>297.80</td>
<td>97.67</td>
<td>3.05</td>
</tr>
<tr>
<td>44</td>
<td>242.74</td>
<td>1167.49</td>
<td>341.53</td>
<td>3.42</td>
</tr>
<tr>
<td>45</td>
<td>326.37</td>
<td>992.40</td>
<td>352.56</td>
<td>2.81</td>
</tr>
<tr>
<td>46</td>
<td>340.86</td>
<td>198.00</td>
<td>85.88</td>
<td>3.91</td>
</tr>
<tr>
<td>47</td>
<td>713.26</td>
<td>314.89</td>
<td>134.67</td>
<td>2.32</td>
</tr>
</tbody>
</table>
Appendix B

Benchmark Instances

<table>
<thead>
<tr>
<th>Benchmark ID</th>
<th>PB Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb/trendy/misc2010/datasets/caixa/normalized-658.cudf.trendy.opb</td>
</tr>
<tr>
<td>2</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/ftp.netlib.org/lp/data/normalized-mps-v2-20-10-bore3d.opb</td>
</tr>
<tr>
<td>3</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/ftp.netlib.org/lp/data/normalized-mps-v2-20-10-bandm.opb</td>
</tr>
<tr>
<td>4</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/MIPLIB/normalization-normalized-mps-v2-20-10-momentum1.opb</td>
</tr>
<tr>
<td>5</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/ftp.netlib.org/lp/data/normalized-mps-v2-20-10-icosm.opb</td>
</tr>
<tr>
<td>7</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/www.csit.fsu.edu/~burkardt/datasets/mps/normalized-mps-v2-20-10-testprob.opb</td>
</tr>
<tr>
<td>9</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/MIPLIB/normalization-normalized-mps-v2-20-10-misc01.opb</td>
</tr>
<tr>
<td>10</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/plato.asu.edu/pub/fctp/normalized-mps-v2-20-10-bk4x3.opb</td>
</tr>
<tr>
<td>11</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/MIPLIB/normalization-normalized-mps-v2-20-10-misc02.opb</td>
</tr>
<tr>
<td>12</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/www.csit.fsu.edu/~burkardt/datasets/mps/normalized-mps-v2-20-10-marce.opb</td>
</tr>
<tr>
<td>13</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/MIPLIB/normalization-normalized-mps-v2-20-10-fluggi.opb</td>
</tr>
<tr>
<td>14</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/MIPLIB/normalization-normalized-mps-v2-20-10-misc03.opb</td>
</tr>
<tr>
<td>16</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/plato.asu.edu/pub/mlplib/normalized-mps-v2-20-10-nug08.opb</td>
</tr>
<tr>
<td>17</td>
<td>normalized-PB06/OPT-BIGINT/mps-v2-20-10/ftp.netlib.org/lp/data/normalized-mps-v2-20-10-africo.opb</td>
</tr>
<tr>
<td>18</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb/paranoid/misc2010/random/rand-smallset/normalized-rand59f7dd.cudf.paranoid.opb</td>
</tr>
<tr>
<td>19</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb/paranoid/misc2010/random/rand-smallset/normalized-rand396c68.cudf.paranoid.opb</td>
</tr>
<tr>
<td>20</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb/paranoid/misc2010/random/rand-smallset/normalized-rand496e42.cudf.paranoid.opb</td>
</tr>
</tbody>
</table>

Table B.1: Full name of the benchmark instances (Part1).
<table>
<thead>
<tr>
<th>Benchmark ID</th>
<th>PB Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>22</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>23</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>24</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>25</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>26</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>27</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>28</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>29</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>30</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>31</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>32</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>33</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>34</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>35</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>36</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>37</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>38</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>39</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>40</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>41</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>42</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>43</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>44</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>45</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>46</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
<tr>
<td>47</td>
<td>normalized-PB10/OPT-BIGINT-LIN/leberre/opb-trendy/misc2010/random/rand_smallist/normalized-rand3e123454567890.cudf.trendy.opb</td>
</tr>
</tbody>
</table>

Table B.2: Full name of the benchmark instances (Part2).