Distributed verification of real-time systems

Author: NGUYEN HOANG GIA

Supervisors: 
Assoc. Prof. Étienne ANDRÉ
Prof. Laure PETRUCCI

January 11, 2015
Acknowledgements

This internship could not completed without the assistance of some very important and special people.

First and foremost, I would like to express my sincere gratitude towards Assoc. Prof. Étienne André $^1$, my supervisor, for his continuous support and encouragement. His scholar knowledge, kindness as well as good humour have actually inspired and educated me all the time.

I would like to thank to Assoc. Prof. Camille Coti $^2$ and to Assoc. Prof. Sami Evangelista $^3$ for their generous help.

I greatly appreciate Prof. Laure Petrucci $^4$ for her comments during my internship.

Finally, I would like to thank to all Professors at PUF and Assoc. Prof. QUAN Thanh Tho$^5$ and his SAVE group for their support.

$^1$http://lipn.univ-paris13.fr/~andre/
$^2$http://lipn.univ-paris13.fr/~coti/
$^3$http://lipn.univ-paris13.fr/~evangelista/
$^4$http://lipn.univ-paris13.fr/~petrucci/
$^5$http://www.cse.hcmut.edu.vn/~qttho/
Abstract

The goal of verification is to make sure that software or hardware system fully satisfies all the expected requirements. Unfortunately, the challenge for verification is always state space explosion problem. This is the explosion of the number of possible states to consider in order to be able to verify a system formally. A variety of techniques has been proposed to alleviate this problem by focusing on main algorithms which can be slightly faster. In fact to verify faster and in order to take advantage of the capabilities of current distributed architectures nowadays, the verification algorithm ought to redefined to be adapted to the distributed case. The goal is, for a system fetching \( n \) cores, that the improved algorithm be (almost) \( n \) times faster than on a mono-core processor – and similarly for clusters. The main goals of my internship are to improve existing algorithms and develop new efficient parallel distributed algorithms for parameter synthesis, and implement these algorithms in the Imitator tool [AFKS12a].
4 Dynamic Subpart Distribution Scheme
4.1 Main Objective ........................................... 23
4.2 General Idea .............................................. 23
4.3 The Master Algorithm ..................................... 27
  4.3.1 General Idea ............................................. 27
  4.3.2 Initial - Static Splitting ............................... 29
  4.3.3 Dynamic Splitting ..................................... 31
4.4 The Worker Algorithm .................................... 32
  4.4.1 General Idea ............................................. 32
  4.4.2 Worker Violation Detection – Check Covered Pi0 .... 34
4.5 Master Worker Interaction ................................. 37
  4.5.1 Master Worker Sequence Diagram ................... 37
  4.5.2 Check Covered pi0 Sequence Diagram ............... 40

5 Experimental Validation ................................. 41
  5.1 Case Studies .............................................. 41
  5.2 Behavioral Cartography Experiment .................... 42
  5.3 Algorithm Speed Up and Efficiency .................... 49

6 Conclusion and Perspectives .......................... 52

A Pseudo Code and Explanation .......................... 58
  A.1 Master Pseudo Code .................................... 58
    A.1.1 Initial Splitting Pseudo Code ..................... 62
    A.1.2 Dynamic Splitting Pseudo Code .................. 65
  A.2 Worker Pseudo Code ................................... 65
    A.2.1 Worker Violation Detection – Check Covered Pi0 Pseudo
         Code .................................................. 68

B Algorithm Speed Up and Efficiency .................. 69
## List of Figures

2.1 Example of Timed Automata ............................................. 11
2.2 Example of Parametric Timed Automata .............................. 12
2.3 Example of parametric timed automata .............................. 12
2.4 Example of symbolic run of parametric timed automata .......... 13
2.5 Example of a trace associated with a symbolic run of a parametric
timed automata ............................................................. 13
2.6 Example of a trace Set of parametric timed automata ............. 14
2.7 Logo of Inverse Method .................................................. 15
2.8 IM synthesizing example ............................................... 15
2.9 IM behavioral tile example ............................................. 16
2.10 Functional view of IMITATOR ......................................... 16
2.11 an examples behavioral cartography in 2 dimensions (rectangle) 17
2.12 an examples behavioral cartography in multi dimensions (Poly-
hedron) ................................................................. 17
3.1 Challenges for a multi-cluster behavioral cartography ............ 22
4.1 Split Subparts Example ................................................... 25
4.2 Master State Diagram ................................................... 28
4.3 Initial Splitting Example ............................................... 31
4.4 Splitting Problem ....................................................... 31
4.5 Dynamic Splitting process ............................................. 32
4.6 Worker State Chart Diagram ......................................... 33
4.7 Violation Detection – Check Covered pi0 ........................... 35
4.8 Worker State Chart Diagram IM ..................................... 36
4.9 Master Sequence Diagram ............................................ 39
4.10 Check Covered pi0 Sequence Diagram ............................. 40
5.1 Flip Flop experiment (Size of V0: 3082 Constraints: 8) .......... 42
5.2 Flip Flop BC (From left to right: Subpart, Sequential, Random20) 42
5.3 Flip Flop 4D experiment (Size of V0: 386400 Constraints: 182) 43
5.4 Flip Flop 4D BC (From left to right: Subpart, Sequential, Ran-
dom20) ............................................................... 43
5.5 RCP experiment (Size of V0: 3050 Constraints: 19) ............. 44
5.6 RCP BC (From left to right: Subpart, Sequential, Random20) 44
5.7 Sched 3 experiment (Size of V0: 286 Constraints : 59) . . . . . . . 45
5.8 Sched 3 BC (From left to right : Subpart, Sequential, Random20) 45
5.9 Sched 3 Large V experiment (Size of V0: 14746 Constraints : 71) 46
5.10 Sched 3 Large V BC (From left to right : Subpart, Sequential,
                          Random20) ........................................... 46
5.11 Sched 3 Large V 3D experiment (Size of V0: 530856 Constraints : 388) ........................................... 47
5.12 Sched 3 Large V 3D BC (From left to right : Subpart, Sequential,
                          Random20) ........................................... 47
5.13 Sched 5 experiment (Size of V0: 1681 Constraints : 177) . . . . 48
5.14 Sched 5 BC (From left to right : Subpart, Sequential, Random20) 48
5.15 Overall BC experiments ........................................... 49
5.16 SpeedUp chart ........................................... 50
5.17 Efficiency chart ........................................... 50

B.1 Subpart Algorithm Speed Up .................................... 69
B.2 Subpart Algorithm Efficiency .................................... 70
B.3 Sequential Algorithm Speed Up .................................. 70
B.4 Sequential Algorithm Efficiency .................................. 71
B.5 Shuffle Algorithm Speed Up ..................................... 71
B.6 Shuffle Algorithm Efficiency ..................................... 72
# List of Algorithms

1. Master Algorithm ................................................. 59
2. Master Algorithm (Receive Pull tag message) ............... 60
3. Initialize Subpart Algorithm ................................. 62
4. Split At Longest Subpart Dimension Algorithm ............ 63
5. Split Algorithm .................................................. 64
6. Dynamic Splitting Algorithm ................................ 65
7. Worker Algorithm ............................................... 66
8. Worker Covered Pi0 Checking Algorithm ................. 68
Chapter 1

Introduction

1.1 Context of the Internship

1.1.1 PUF

PUF\(^1\) is a learning center belonging to French Universities at Ho Chi Minh city, Viet Nam. It was established on July 20\(^{th}\) 2006 after an agreement between Vietnam and France governments, and has a collaboration network of more than 20 French universities of high reputation, for example, Paris 6, Paris 7, Lyon 2, Toulouse 1, Bordeaux to provide high-quality education environment for Vietnamese students as well as studying in France.

There are two main training centers of PUF, which are located in Ho-Chi-Minh-City and Hanoi. All courses of PUF are taught by French professors to ensure that PUF’s students have an equivalent knowledge as French students.

As a master student in Information Technology at PUF, I must have an internship to do a research project. I choose LIPN laboratory because I am interested in their research areas and I desire to study further after finishing my Master degree.

1.1.2 LIPN Laboratory

LIPN\(^2\) is an abbreviation of Laboratoire d'Informatique de Paris-Nord, located in Paris 13 University. LIPN has been associated with CNRS since January 1992 and is a mixed research unit since January 2001 (UMR 7030). Research at LIPN is carried out in its major areas of expertise, in particular Combinatorics, Combinatorial Optimisation, Algorithmics, Logic, Software, Natural Language, Machine Learning.

There are 5 main teams of LIPN’s research fields:

---

\(^1\)http://pufhcm.edu.vn/

\(^2\)http://lipn.univ-paris13.fr/fr/laboratoire
In this internship, I have a chance to work with the Logic, Computation and Reasoning team as my main team and Combinatorial Optimisation team.

1.2 The Internship

This internship is done under the instructions of Assoc. Prof. Étienne André during a period of five months (from 08-July-2014 to 30-December-2014), with the occasional co-supervision of Prof. Laure Petrucci and Assoc. Prof. Camille Coti.

1.2.1 Objectives

Increasing complexity in real-time and critical systems require formal verification and model checking techniques to automate and support their design. In fact, the model is very hard to design correctly. Thus, formal methods are very important in design and verification of the models and reduce the amount of resources and time spent on it. With formal methods, the errors could be detected early to correct.

Verifying concurrent real-time systems is an extremely difficult problem. The correctness of such real-time systems usually depends on the values of timing delays; thus, it is interesting to consider the delays as parameters (unknown constants) and synthesize a constraint guaranteeing a correct behavior. This is known as the parameter synthesis problem [AHV93]. From infinite timing value, a desired range of discrete values will be picked up to synthesize. The correctness of system will depend on number of timing delay values in the range. In fact, it consumes a lot of time for checking each value at a time. Parallel computing may help to check more than one value at a time.

Inverse method will check each value in a huge set of concrete values to synthesize all constraints. In practice, it takes time and we should distribute the computation to many processes to force it synthesizes faster.

\[3\text{http://lipn.univ-paris13.fr/~andre/}\]
As a part of Imitator project, the general objectives of this internship are as follows:

- The first objective is doing a survey to get background and deeply clarify various techniques of Imitator project.
- The next objective is to propose a new algorithm in order to apply Imitator on distributed case efficiently.
- The last one is to implement this algorithm and conduct experiments, finally make a comparison among all algorithms.

1.2.2 Contributions

Imitator is based on the Inverse Method (section 2.6) and the Behavioral Cartography (section 2.8) for parametric timed automata. Behavioral Cartography algorithm which covers a bounded parameter domain with tiles (parametric zones of uniform discrete behavior). Behavioral Cartography algorithm simply calls “Inverse Method” for all values of parameter domain (reference valuations or points) one by one, then a set of tiles will be returned and each tile might contains many points and indicates a discrete system behavior. In real scale model with a huge parameter domain, exploring this is very consuming time. Thus, to speed up Behavioral Cartography algorithm, the power of distributed systems such as clusters is taken into account in order to parallel execution on several cores or processes. Unfortunately, it is not easy to efficiently distribute Behavioral Cartography since finding the right values (points) on parameter domain because of tile overlapping which leads to redundant computations and slow down overall performance. In this internship, we will introduce a new exploration approach, a tile overlapping detection mechanism and a valuation of all distribution algorithms. The algorithms we talk about are also the first distributed algorithms for the synthesis of timing parameters and their promising results which is given latter will open perspectives toward verification of large scale models.

1.2.3 Organization of the document

This report is structured as follows. In Chapter 2, we talk about the necessary preliminaries. In Chapter 3 we present Behavioral Cartography and previous works in more details. In Chapter 4, we propose our works in the internship. For comparison purposes in Chapter 5, we conduct experiments then valuate our new algorithm and previous algorithms. We finally conclude and present directions of future research in Chapter 6.
Chapter 2

Background

This chapter introduces the necessary preliminaries used in this report.

2.1 Formal Methods

Formal methods [HBH+99], a particular kind of mathematically based techniques for the specification, development and verification, that methods use to check the correctness of hardware and software. While traditional testing and simulation which could prove the presence of bugs (errors) with finite test cases, thus it is apparent that they do not guarantee the absence of bugs. In contrast, formal methods based on mathematics could prove the absence of bugs. In general, formal design can be seen as a three step process:

- Formal Specification: to describe the system.
- Formal Verification: to prove the correctness of system.
- Implementation: to converse the specification into code after formal verifying.

2.2 Model Checking

Model checking [BK08, Boo99] is a kind of formal verification which is based on the state space exploration (explicitly or symbolically) of the system. In this technique, the system which will be verified, is modeled as a finite state machine which describes all possible states of the system and known as transition system. In addition, the property to be checked in the system model is expressed in a formal language such as temporal logic. Later on, the transition system and the property which is described in a formal language are applied as input to a model checker. After that, the model checker begins exploring all reachable states of the transition system to check whether the property to be checked is true. The set of all states is called state space. If a property is not valid, a
counter-example is returned. Based on this counter-example, users can detect the errors in their model and correct. However, a main obstacle of this method is state space explosion: the transition graph usually become exponentially larger based on the size of the system which cause the exploration of the whole state space becomes impracticable.

Given a model of a system and a specification of the desired properties, a model-checker start determine the specification satisfaction automatically and exhaustively. A diversity of properties can be verified by model-checking, such as deadlock freedom, invariants, request-response properties, etc. These properties can be categorized into five classes: reachability, safety, liveness, deadlock-freeness, and fairness properties.

- A reachability property asserts that some certain situation can be reached.
- A safety property states that, under particular conditions, an event never occurs.
- A liveness property assures that, under particular conditions, some event will ultimately occur.
- Deadlock-freeness is a special property, expressing that the system can never be in a situation in which no progress is possible.
- A fairness property states that, under certain conditions, an event will occur (or will fail to occur) infinitely often.

Note that all these properties (except fairness) can be verified using IMITATOR (see Section 2.7).

In recent years, model-checking has become a widespread technique for verifying finite or infinite systems. Many effective algorithms were proposed with respect to different logics. However, the critical obstacle of applying Model checking to verify real systems is the combinatorial explosion of states. In order to avoid this problem, algorithms have been proposed to avoid exhaustive state space exploration, such as: symbolic representation of the states space using Binary Decision Diagrams [BCM+92], application of partial order methods which eliminates unnecessary interleavings of transitions, or by application of abstractions and symmetries (e.g., [DGG93, CGL94, Val92, GW91, CJEF96]).

### 2.3 Temporal Logic

Temporal logic [Pnu77] is a a form of logic which is dedicated for statements and reasoning that include the notion of order in time. In 1977, A. Pnueli first used it to specify behavior properties of systems.

Readers can consider temporal logics as extensions of classical propositional and first-order logic. Propositional temporal logics are propositional logic extension with temporal operators. Similarly, first-order temporal logics are first-order logic extension with temporal modalities. Temporal logics are also special
type of modal logics, where statements are evaluated on “worlds” which represent time instants.

Linear Temporal Logic (LTL) [Pnu86, BK08]. Linear Temporal Logic is formula defined by \( \varphi ::= \bot | \top | a | \neg \varphi | \varphi \land \varphi | \varphi \lor \varphi | G\varphi | F\varphi | U\varphi | X\varphi \) where \( a \) is an atomic proposition, Greek letter \( \varphi \) denote formulas, \( G, F, U \) and \( X \) denote the always, eventually, until and next operators respectively.

Computation Tree Logic (CTL) [CE81, BK08]. Computation Tree Logic or branching-time logic is formula defined by \( \varphi ::= \bot | \top | a | \neg \varphi | \varphi \land \varphi | \varphi \lor \varphi | AG\varphi | AF\varphi | AU\varphi | AX\varphi | EG\varphi | EF\varphi | EU\varphi | EX\varphi \) where \( a \) is an atomic proposition, Greek letter \( \varphi \) denote formulae, prefixes \( A \) denote for all paths and \( E \) denote there exists a path. \( G, F, U \) and \( X \) denote the always, eventually, until and next operators respectively.

### 2.4 Timed Automata

![Timed Automata Diagram](image_url)

Figure 2.1 – Example of Timed Automata

Timed automata [AD90, AD94] are an extension of finite state automata with a finite set of real-valued clocks, that increase all with the same speed. Along the transitions of the automaton, clock values can be compared with constants in constraints that allow (or not) stay in a location (“invariants”) or to take a transition (“guards”). Further, some of clocks can be reset. Timed automata are a sub-class of hybrid automata [Hen96]. This formalism allows the parallel composition of several timed automata, which behave like a single automata; this provides the designer with a powerful and intuitive way to represent timed systems. It is important to note that the model of timed automata is very sensitive to the size of the automata and the number of automata in parallel, thus often leading to the state-space explosion problem.

Recently, there exists a variety of powerful model checkers tools such as UPPAAL [LPY97], Kronos [BDM+98], and the schedulability analyser TIMES [AFM+02]. These tools are becoming more and more mature, but are still all academic research tools.


2.5 Parametric Timed Automata

![Parametric Timed Automata](image)

Figure 2.2 – Example of Parametric Timed Automata

Parametric timed automata (PTA) are an extension of class of timed automata to parametric case. Parametric timed automata allow within guards and invariants the use of parameters in place of constants [AHV93]. This model is interesting when one does not only want to check that a system is correct for one value of the constants, but for a whole dense set of values. The model of parametric timed automata is also interesting to synthesize parameters for which a given property is satisfied.

Unfortunately, for most interesting problems, parametric timed automata lose the decidability results proved for timed automata. In particular, the reachability of a state is not decidable (although semi-algorithms do exist, i.e., if the algorithm terminates, then the result is correct). Moreover, parametric timed automata are even more sensitive to the state space explosion problem, because of the addition of the parameters.

2.6 The Inverse Method

Before introducing Inverse Method (IM), it is needed to introduce the run, trace and trace set of semantic of Parametric Timed Automata.

![Parametric Timed Automata](image)

Figure 2.3 – Example of parametric timed automata
2.6.1 The Run of Parametric Timed Automata

Given a parametric timed automata as figure 2.3 above, its symbolic runs [AS13] in figure 2.4 are represented under the form of a branch in which states are depicted within nodes containing the name of the location and the constraint on the clocks and the parameters, and transitions are depicted using edges labeled with the name of the action.

![Figure 2.4 – Example of symbolic run of parametric timed automata](image)

2.6.2 The Trace of Parametric Timed Automata

With each concrete run [AS13] we also have equivalent trace which is an abstract part of a system's behavior. In fact, the trace is built from a run by removing the valuation of the clocks, and thus can be seen as a “time abstract” run. As figure 2.5, we depict traces of parametric timed automata 2.3 under a form using nodes labeled with location and arrows labeled with actions.

![Figure 2.5 – Example of a trace associated with a symbolic run of a parametric timed automata](image)

2.6.3 The Trace Set of Parametric Timed Automata

A trace set [AS13] of a parametric timed automata refer to the set of traces associated with the runs of this automata. For describing a trace sets, we will
not depict each trace as the trace above separately, but depict it under the form of a tree or a graph. It could be seen as simplicity of representation of possible traces, and does not contain any information on the possible branching behavior of the system. For instance, given figure 2.6 is depicted a trace set of parametric timed automata 2.3.

![Figure 2.6 – Example of a trace Set of parametric timed automata](image)

### 2.6.4 The Good Parameters Problem

This project come from we are interested in finding correct values for the parameters of parametric timed automata and synthesizing of good parameters corresponds to the good parameters problem [AS13]:

Given a concurrent real time system and a rectangular parameter domain $V_0 \subseteq R^M_+$, what is the largest set of parameter values within $V_0$ for which the system is safe?

### 2.6.5 Inverse Problem

In this section we would like to introduce inverse problem [AS13] which is a subproblem of the good parameters problem 2.6.4 above:

Given a parametric timed automaton $\mathcal{A}$ and a reference valuation $\pi_0$, find a constraint $K_0$ on the parameters such that:

- $\pi_0 \models K_0$, and
- for all $\pi \models K_0$ the trace sets of $\mathcal{A}[\pi_0]$ and $\mathcal{A}[\pi]$ are the same.

This problem considers an equality of trace sets between $\mathcal{A}[\pi_0]$ and $\mathcal{A}[\pi]$, and thus guarantees a “time-abstract” equivalence between the behavior of $\mathcal{A}[\pi_0]$ and $\mathcal{A}[\pi]$.

### 2.6.6 The Inverse Method

Given a parametric timed automaton $\mathcal{A}$, the Inverse Method algorithm [AS13, ACEF09a] takes as input a parametric timed automata $\mathcal{A}$ and a reference valuation $\pi_0$ of the parameters that one wants to generalize. It synthesizes a constraint $K_0$ as a behavioral tile (which is a parameter zone with an uniform
time abstract behavior on the parameters) that corresponds to a set such that, for all valuations $\pi_0$ of parameters in this set, the trace sets of $\mathcal{A}[\pi_0]$ and $\mathcal{A}[\pi]$ are equal. The IM algorithm starts with $K = true$, then it iteratively computes a growing set of reachable states. When $\pi_0$ incompatible state $J$ is encountered then $\neg J$ is added to $K$. Later on, the procedure will start with this new $K$, and so on, until no new state is computed. Finally, the behavioral tile (intersection of the projection onto the parameters of all the constraints associated to the reachable states) will be returned as output. In practice, according to given property we want to check, the IM will return 3 kind of tiles such as bad, good and unknown tile.

For more understanding, we give a simple example of procedure of synthesizing constraint in figure 2.8. It starts from state $q_0$ then $q_1$, finally it adds a negative constraint of $q_3$. And an example is given at figure 2.9. After synthesizing constraints with this $\pi_0$, a blue tile (constraint $K_0$) will be returned.

In this section we only give a general idea of Inverse Method. If you are interested in Inverse Method for Parametric Timed Automata, you could find more in the Inverse Method book [AS13].
2.7 IMITATOR

IMITATOR \cite{AFKS12a, And09} stand for Inverse Method for Inferring Time AbstracT behaviOR is an implementation of algorithm InverseMethod, described in \cite{ACEF09b} and a tool for efficient synthesis for Timed Automata. It is based on the inverse method \cite{AFKS12b} and the behavioral cartography for PTA \cite{AF10}. IMITATOR takes as input a network of PTA and a reference valuation $\pi_0$ it synthesizes a constraint $K_0$ solving the inverse problem. In particular, it generalizes a concrete behavior of a PTA, by synthesizing constraints on the parameters. For more understanding, a figure 2.10 is given as below:

![Figure 2.10 – Functional view of IMITATOR](image)

This tool has been originally developed by Étienne André, in collaboration with various contributors.

2.8 Behavioral Cartography (BC)

The behavioral cartography \cite{AF10} BC relies on the idea of covering the parameter space within a real valued parameter domain $V_0$. By iterating the inverse method on the integer points of the parametric domain $V_0$, we are able to decompose the parametric space into behavioral tiles, i.e., parameter zones with a uniform time abstract behavior. Then, according to a property on traces that one wants to check, it is easy to partition the parametric space into a subset of “good” tiles and a subset of “bad” ones (which correspond to good and bad behaviors) as example in figure 2.11. Each parameter corresponds with a dimension of BC, and with $n$ dimension we will have the polyhedron as figure 2.12.

\footnote{The behavioral cartography picture in this section is provided by Étienne André}

\footnote{The behavioral cartography picture in this section is provided by Étienne André}
Parallel Computing

Parallel computing [B⁺10]: is a form of computation in which many calculations are carried out simultaneously by parallel systems.

Parallel system: is a system having more than one processors with parallel processing abilities. That systems are based on the connections between processors and memories, SIMD and MIMD are two main kind of parallel system which are from four classifications defined by Flynn:

\footnotesize{
\textsuperscript{3}The polyhedron picture in this section is obtained from : http://http://commons.wikimedia.org/wiki/File:Uniform_polyhedron-53-s012.png
}
• Single Instruction Single Data Stream (SISD): traditional sequential architecture. At one time, one instruction operates on one data.

• Single Instruction Multiple Data Stream (SIMD): multiple data streams are executed by only one instructions.

• Multiple Instruction, Single Data stream (MISD): multiple instructions operate on a single data stream. This architecture which is often used for fault tolerance.

• Multiple Instruction Multiple Data Stream (MIMD): multiple processors execute different instructions on different data streams, which leads to the flexibility in parallel processing, distributed systems are recognized as MIMD architectures. In this project we used this system.

Parallel Programming Model: is programming model could be executed on parallel system that includes design, coding application. The purpose of this is to parallelize a sequence program brings about an ability to solve a big complex problem, or reduce execution time or both.

The main idea or parallel programming is to split a problem into a number of smaller problems, then assign them to processors, and finally collect them to get final result.

There are several parallel programming models in common use, for instances:

• Shared Memory: multiple processors can communicate together by reading and writing from a shared memory. It means data in shared memory can be accessed by all the processors.

• Distributed Memory: unlike shared memory systems, each processor has access to its own memory space.

Knowing them clearly could help to build the right algorithm. For our new algorithm we choose distributed memory model, and processes communicate with each other using message passing.

2.10 Message Passing Interface - MPI

MPI [Bar, GFB+04] stand for Message Passing Interface is a standardized and portable message-passing system designed by a group of researchers from academia and industry to supply programmers programming on a variety of parallel platforms.

The goal of the Message Passing Interface is to establish a portable, efficient, and flexible standard for programming parallel applications. MPI provides language bindings for C, C++, Fortran-77 and Fortran-90, non-official bindings exist for python and Ocaml. There exist several open-source implementations but the most widely used are being Open-MPI and MPICH2.

IMITATOR is written in Ocaml; Hence, we used “OcamlMPI” which is implementation of Open-MPI for this project.
It is necessary to familiar with basic MPI definitions which are used in next chapters:

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other

- Rank: is unique, integer identifier for a process assigned by the system when the process initializes and used by the programmer to specify the source and destination of messages. Ranks are contiguous and begin at zero.

- Group: is an ordered set of processes, has its own unique identifier, associated with a communicator.

- Communicator: defines the group which may communicate, possesses its own unique identifier with each other. Most MPI subroutines require you to specify the communicator as an argument.

2.11 Ocaml

OCaml [Har05, MMH13] programming language is developed at INRIA in France and since its first release in 1996. It has excelled in a variety of application domains and supports functional, imperative and object-oriented styles of programming which ease the development of flexible and reliable software. Thus Ocaml has become one of the most popular functional programming languages in the world, which has many powerful features were adapted such as functional programming, pattern matching, type inference, garbage collection, object-oriented programming, safety, strongly and statically typed, type inference, polymorphism, modules programs, separate compilation, etc. It is clear that OCaml is a good choice for systems where resource usage, predictability, and performance. Until now, Ocaml language improvements have been frequently added to support the growing commercial and academic purposes.

The IMITATOR is also written in Ocaml so that we will develop new algorithm using Ocaml.
Chapter 3

Behavioral Cartography of Timed Automata

We have given an overview about BC at section 2.8. For now, we will focus on approaches covering BC. This chapter's outlines:

- Section 3.1: recall the non-distributed approach for BC, that can be run on single process.
- Section 3.2: recall the master-worker parallelization scheme as main parallel solution for all distributed algorithms before and during this internship.
- Section 3.3: list all previous distributed algorithms together with their obstacles.

3.1 The Non-distributed Behavioral Cartography

In this section, we introduced the non-distributed behavioral cartography algorithm for parametric timed automata. As the name implies, a term “Non-distributed” means that the algorithm works with only one process. Given a bounded parameter domain $D$, finding the set of integer points which is not covered by a list of polyhedron or tiles is a problem that has no known efficient practical solution, such as the white areas on figure 4.4. Hence, there is no other choice than enumerating all the points one by one. In the non-distributed cartography, an empty set of tiles $S$ is first created. Then, each point in $D$ is sequentially checked to see if it is covered by any tile in $S$. If not, IM is called on this point, and the result is added to $S$. When all the points have been enumerated, the algorithm returns the set of tiles [ACE14].
3.2 Master-Worker Parallelization Scheme

The Master-Worker scheme [SBW00] is a well-known scheme which is a fundamental and widely used approach for parallel and distributed applications. In this scheme, a single master process has responsibility to distribute small works from a big work to a set of worker processes. We could consider that the main functions of master process are to send the work and collect result from a set of worker processes and we require only communication between the master process and individual worker processes.

In this project, we used a pull-based algorithm where workers will pull work from master and master roles are to distribute works to the workers and synthesize results from workers. This scheme may lose its efficiency due to the fact that the master process is not calling IM itself; however this shall be compensated for a large number of processes. Since workers pull work when they are done with their previous computation, this parallel computation scheme balances the load between workers automatically. Even if a worker takes a long time to compute its work, the other workers will compute several works or tiles in the meantime. Consequently, pull-based master-worker parallelization schemes are particularly interesting for BC [ACE14].

In fact that there is no communication between workers but only master and workers by sending MPI tag messages (Distributed memory) and all workers only use IM to call points on parameter domain (SNMP). We can imply this scheme is hybrid parallel programming model that is talked about in section 2.9.

3.3 Towards Distributed Behavioral Cartography

We would like to introduce previous three pull-based distributed algorithms to execute BC on a set of processes running on a cluster, implemented in IMITATOR using the message passing interface MPI. Distributing BC is easy: it is trivial that two executions of IM from two different parameter valuations can be performed on two different nodes. However, distributing it efficiently is challenging: calling two executions of IM from two contiguous integer points has a very large probability to yield the same tile in both cases, and hence to result in a loss of time for one of the two nodes. Hence, the critical question is how to distribute efficiently the point (reference valuations) on which to call IM. We proposed a master-workers scheme at section 3.2, where a master distribute the points to the workers, using three point distribution algorithms:

**Sequential Algorithm** [ACE14] A sequential point enumeration: each integer point not covered yet by any tile is sent to a worker sequentially, i.e., (0, 0), then (0, 1) and so on (in two dimensions). This algorithm suffers from the problem of close points, but still performs not too badly.

**Random Algorithm** [ACE14] A random point distribution followed by a sequential enumeration: points a selected randomly and, when points is not
yet covered by any tile become scarce, the master switches to sequential point enumeration to ensure that all the integer points are covered. The fact that the points are not covered by any tile become scarce is detected after the number of unsuccessful attempts to randomly choose an uncovered point goes beyond a certain threshold number (e.g., 20). This algorithm performs better than the first one because of fewer redundant computations caused by close points.

**Shuffle Algorithm** A sequential point enumeration on shuffle array: this is similar to the first Sequential Algorithm. But the difference is that it must shuffle all points, then store them in array. Then master sends them to the master sequentially. This gave better performance but its drawback is memory usage (all points must be explicitly stored and shuffled) and not efficient in large scale models. (This algorithm has yet to be published.)

![Figure 3.1 – Challenges for a multi-cluster behavioral cartography](image)

We introduced previous works, in the next chapters we will introduce our fourth distribution algorithm. This splits the parametric domain $V$ dynamically into “subparts”: when a worker has completed the covering of its subpart, the master splits another subpart into two parts, and assigns one of two to that worker. The new algorithm comes with the aim of solving two problems of point distribution algorithms when extending the cartography of multi-cluster:

- **Problem 1:** Choosing point to call by IM: Workers that work on points that near to each other may be in the same tile (duplicated work) as figure 3.1. A solution to this problem is proposed in section 4.2.
- **Problem 2:** Decide whether an execution of IM on given point should be stopped or not, whenever this point has been just covered by another tile. And we also create a new mechanism to solve this which we talk about in section 4.4.2.

---

1The behavioral cartography picture in this section is provided by Étienne André
Chapter 4

Dynamic Subpart Distribution Scheme

4.1 Main Objective

We recall here our internship main objectives:

- Propose a new algorithm in order to apply IMITATOR on distributed case efficiently.
- Implement this algorithm, conduct experiments and make a comparison among all algorithms.

In this chapter, we will focus on new algorithm ideas and how to implement them. This chapter is structured as follows:

- Section 4.2: first present our general distribution idea.
- Section 4.3: present master algorithm and its function.
- Section 4.4: present worker algorithm and its function.
- Section 4.5: give an overview of interaction between master and worker through sequence diagrams.

All pseudo codes and their explanation are put in appendix sections.

4.2 General Idea

A solution could have been to split the big subpart (polyhedron) $D$ into $N$ smaller subparts (polyhedra), and then ask each process to handle its own subpart independently from each other. This domain decomposition method is
often popularly used for regular data distributions, efficiently, where all sub-parts require the same processing time and preferably on domain shapes such as rectangles, hypercube or polyhedron, which can easily be mapped on a grid of processes. Unfortunately, due to the unknown “shape” of the cartography, this becomes difficult. Indeed, it often happens that some tiles use most of the space in $D$, whereas other tiles can be concentrated in a very small area, the re-partition of the tiles is totally unknown before computing the cartography so that we proposed a load balancing mechanism which could dynamically share workload between processes.

In previous work, master–workers distribution scheme model is efficient for synthesizing tile. Hence, we reused this scheme for new algorithm. A difference is that master will assign subpart as a set of points to worker. After receiving subpart, workers can call $IM$ by themselves and return tile results to master. This still is a pull-based algorithm, since workers pull work from the master.

Since the main idea of this algorithm is to distribute a set of points as “subpart” equally to all processes as the name implies (subpart distribution), then each process can work independently. Hence reducing the load on the master process and the risk of bottle neck created by the fact that all the workers interact with it. In the next sections we will present two primary splitting approaches of subpart distribution scheme:

- **Initial Splitting or Static Splitting**: this function will split the subpart (parameter domain $D$) into almost equal $n − 1$ subparts which corresponds to $n − 1$ number of worker processes. By iteratively splitting the subpart into 2 smaller almost equal subparts until $n − 1$ subparts have been split. This approach is used for initial stage when master process begin to split the subpart (parameter domain) which has all points in it are uncalled.

- **Dynamic Splitting**: in order to share work between workers the second splitting algorithm was created. This function will determine called and uncalled points in a subpart (parameter domain $D$) then a set of uncalled points will be shared between workers. To do so, the master must first determine the current point (the point that worker is calling on) of the worker to know how many points have already been called in its subpart. After that, the master will get a set of uncalled $IM$ points from this current subpart for sending to the free worker process. Then the remainder of the current subpart (modified parameter domain $D$) after splitting will be sent back later to owner worker for updating.

In practice, time for each worker finish its subpart is variable (time to call $IM$ on each point depends on: trace sets in subpart, processors speed, latency, violation (covered point calling) etc. ). Thus, all workers do not always finish at the same time and Dynamic Splitting is the function that has responsibility to force all workers to finish at the same time.

Although each worker works on a particular subpart but in some cases there still has a problem if more than one worker work on the same tile since the tile can be located on two subparts like rectangles of processes 2, 5, 7, 8, 10, 11
in figure 4.1. Hence, worker processes need a mechanism that can detect this problem which is also called violation by themselves, section 4.4.2 will describe clearly this mechanism.

For an instance of violation, both $P_{10}$ and $P_{11}$ processes can possibly call points in a common blue tile. Actually they do not know they are synthesizing the same tile until they are done. And the violation mechanism simply checks if one of them finishes the tile first, the tile must be sent to the other worker processes. Consequently, the other workers will realize the violation then immediately stop synthesizing and they will move to next point to synthesize other tile (red, yellow, gray).

**List of tags used**

Since master and workers communicate with each other by messages tagging these messages is an approach to identify the types of messages. Here we made a table 4.1 to describe all the message types that we used in new algorithm and the tag which is associated to each type.

Upon reception of a message, the process that receives the message can check what its tag is and then act according to the corresponding type.
<table>
<thead>
<tr>
<th>Tag</th>
<th>Sender</th>
<th>Receiver</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subpart</td>
<td>Master</td>
<td>Worker</td>
<td>subpart</td>
<td>Subpart is a set of points which worker needs to call IM.</td>
</tr>
<tr>
<td>TileList</td>
<td>Master</td>
<td>Worker</td>
<td>list of tile(Ks)</td>
<td>Ks is a set of tiles that worker needs to update for avoiding violation.</td>
</tr>
<tr>
<td>Continue</td>
<td>Master</td>
<td>Worker</td>
<td>null</td>
<td>Continue() tag is used to replace null data message.</td>
</tr>
<tr>
<td>Terminate</td>
<td>Master</td>
<td>Worker</td>
<td>null</td>
<td>'Terminate() tag is used to terminate worker.</td>
</tr>
<tr>
<td>Pull</td>
<td>Worker</td>
<td>Master</td>
<td>null</td>
<td>Pull() tag is used for pulling work from master.</td>
</tr>
<tr>
<td>Tile</td>
<td>Worker</td>
<td>Master</td>
<td>tile(K)</td>
<td>K tag is a tile result for synthesizing at master side.</td>
</tr>
<tr>
<td>NotifyPi0</td>
<td>Worker</td>
<td>Master</td>
<td>pi0</td>
<td>Pi0 is useful for master to determine which point is being computed at worker side.</td>
</tr>
<tr>
<td>CheckCover</td>
<td>Worker</td>
<td>Master</td>
<td>null</td>
<td>CheckCover() tag is used to check violation at worker side by pulling list of new tile from master, then checking violation within new tiles take place.</td>
</tr>
<tr>
<td>UpdateSubpart</td>
<td>Worker</td>
<td>Master</td>
<td>null</td>
<td>UpdateSubpart() tag allows worker to update last change of its subpart after dynamic splitting.</td>
</tr>
</tbody>
</table>

Table 4.1 – Tag table
4.3 The Master Algorithm

4.3.1 General Idea

We describe here an “abstract” algorithm for the master, which is created as an operator which could resolve a large amount of tag messages from worker processes. Hence, it is very important that tag messages are proceed quickly for workers to operate smoothly and efficiently in real scaled model. Master roles include:

- Distributing subparts to workers.
- Synthesizing tiles from workers.
- Splitting subparts.
- Communicate with workers.
- Load balancing between the workers.

I then beginning of master starts by creating an empty set of tiles and splitting the main subpart in \( n \) smaller subparts if \( n \) is the number of worker processes. Before receiving a subpart, workers send pull requests to mean that they are ready to receive some work. The master then finds a subpart in repository and sends it to each worker. In case the master’s repository has no subpart left, it looks for an incomplete one from busy workers. If all busy workers are calling the last point of their subpart and could share any point, master will send a message to terminate it and also check if all workers are terminated, the master would be terminated immediately.

Suppose worker received a subpart. Later on, the worker will have to notify master which point it will call \( \text{IM} \) on. The main purpose of getting the current point \( \pi_0 \) is to determine remain of uncalled points in subpart at master side. The next step, worker will call \( \text{IM} \) with current point. In each step of this process, the master will be asked for new tiles by sending the “CheckCover” tag. Then master will send list of new tiles to worker or send “Continue” tag as null message. Assume that worker received list of tiles and its current point is covered by the tiles, then worker must stop to move to the next point. If the point is not covered in every step, finally worker will send a tile result that is returned by \( \text{IM} \) to master.

Notice that worker will not know about change with its subpart. Thus, after calling \( \text{IM} \) master will update its subpart by sending “UpdateSubpart” tag message. In this situation, master will check change of the worker’s subpart to decide send “Subpart” or “Continue” tag message.

Since master received the new tile result, master will prepare buffer for notifying the new tile to other workers later or ignore if this tile is already in repository.

Master State Chart Diagram
Figure 4.2 – Master State Diagram
In this subsection we attempt to express master algorithm with state chart diagram.

State Diagram figure 4.2 in order to give a view of detailed operations of master. For now, we can easily see how the master resolves each tag message one by one. From the starting point, the master has to initialize subparts for all workers (initial splitting at section 4.3.2). Afterward, master gets the message of worker from buffer. In the buffer, master could receive one of five tag messages corresponds with five transitions which are “Pull”, “Tile”, “NotifyPi0”, “CheckCover” and “UpdateSubpart” tag. We will begin with each tag below:

- For the first “Pull” tag message, in the beginning we check for available subpart. First case, master will send the subpart to worker if its repository has an available subpart. Conversely, it get the uncalled part from other worker by trying to dynamic split their subpart. In case no worker has points to split (splittable subpart), this worker will be terminated, after that master will be terminated if all workers are terminated.

- The second tag is “Tile” tag message. Any completed tile of the worker must be sent to master. Later on, master will store it in buffer then send to other workers later or ignore if this tile is in repository already. Notice that we avoid sending immediately, which could cause flush of bandwidth.

- The most simple tag is “NotifyPi0”, master will store $\pi_0$ in its repository when it receives this tag from a worker.

- “CheckCover” tag message. As its name implies, this tag is used for check covered point purpose. But for the first stage after receiving this tag, master may send list of tiles from buffer to the worker or send “Continue” tag if master has not new tiles. Assuming that worker received list of tiles, then it will check covered point by itself.

- Last one is ”UpdateSubpart” tag message, master will determine change of its subpart which is caused by splitting. Then master will decide to send Subpart or Continue tag.

We give in Appendix A.1 the whole pseudo code for the master.

4.3.2 Initial - Static Splitting

This section will propose an approach to split a polyhedron into several smaller polyhedrons. In practice, a polyhedron is described as multi-dimensional array with each dimension might contain thousand points. It is quite difficult to store and manipulate with over thousand points multi-dimensional array because of memory consumption problem.

To resolve this problem above, we would like to introduce a specific array we used to store polyhedron. Instead of storing it as a multi-dimensional array, which consumes a lot of memory, we used two-dimensional arrays with size...
\((n \times 2)\) to describe fully multi-dimensional array where \(n\) is number of parameters. The 2 integers of each element indicate the range of length each parameter since the integer increased steadily (for an array \([0][1][2][3][4][5]\) we could also describe it as an array \([0][5]\)).

For instance, a BC \(V_0\) with 3 parameters each dimension has integer range from 1 to 4, which is expressed as table 4.2 below.

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
1 & 2 & 3 & 4 \\
\end{array}
\]

Table 4.2 – Normal array

As table 4.2 above, we can also describe it in smaller size array as table 4.3 below which consumes less memory.

\[
\begin{array}{cc}
1 & 4 \\
1 & 4 \\
1 & 4 \\
\end{array}
\]

Table 4.3 – New array

In explanation parts and figures we would not use this smaller array because of incomprehension. Return to the main intents of initial splitting:

- To decrease the traffic between workers and master since workers receive subparts(set of points) from the master rather than single point.
- To reduce probability of calling points in same tile (many processes yield the same work or tile) because each worker will call all points on its own subpart.

The initial splitting is made by a function that splits a polyhedron (subpart) into smaller polyhedrons of equal size (subparts). An example is given in figure 4.3. Given a big subpart as 2 dimensional array, with Initial Splitting this subpart could be split into 2 smaller and 3 smaller subparts for 2, 3 worker processes respectively.

From figure 4.3 and due to our algorithm 3, it appears that number of processes equals \(x = 2^n + 1\) to get highest efficient, but actually this can be true only if each subpart is processed in the same time as the other ones. Thus this is a lower bound to keep all the workers busy, but not an optimum. With \(2^n\) is number of worker processes that means the number of worker processes will be even number (2, 4, 6, 8, 16, .etc). Finally, it needs 1 more processes for master. Thus, \(x\) should be odd(3, 5, 7, 9, 17, .etc) to get highest performance.

Even with odd number of worker processes as figure 4.3(3 processes), number of points is not directly proportional to finished time which means big subpart
could be done faster than small one. Consequently, we proposed a mechanism to resolve this problem in Section 4.3.3.

We give in Appendix A.1.1 the whole master initial-splitting pseudo code.

### 4.3.3 Dynamic Splitting

As problems we mentioned in section 4.3.2, splitting the rectangle D into N subparts, it often happens that some tiles use most of the space in D, whereas other tiles can be concentrated in a very small area. Or some tiles costly than other tiles because size of trace set, some processors faster than others etc, thus some workers will finish faster than the others. Hence, dynamic load balancing among worker processes will be taken into account. Consider Figure 4.4: we split a rectangle into two smaller rectangle S1 and S2. Assume each tile has equal calling time (same processor speed and trace set) then the S2 process time will be the overall time and it wastes resource of S1.

In previous point distribution scheme, master has to calculate the next point for each worker. In this scheme master will send subpart to the worker then it will compute the next point itself.

Dynamic splitting is intended to split uncalled points part of worker’s subpart. For example, consider Figure 4.5 where red points represent called points and green points are uncalled points. To split subpart it need to determine
uncalled point area as arrow in figure then it will split this area in two new subpart. It could split until this subpart contains only one point. Notice that called part will be eliminated while splitting

4.4 The Worker Algorithm

4.4.1 General Idea

We describe here an “abstract” algorithm for the Worker, main intents of worker are:

- Getting subparts from the master.
- Notify point information to the master.
- Updating its subpart after dynamic splitting.
- Calling IM each point in its subpart.
- Checking violation – covered $\pi_0$.
- Sending tile results back to the master.

Firstly, before receiving a subpart, each worker pulls a subpart from the master. Later on, worker can work independently. To call IM each point in subpart, worker has to notify the point it want to call to the master first. When master begins with each step of IM, worker will check if this point is violated at this time by pulling the latest tiles from master. Assume that worker finished all of points in it subpart, it will send pull tag message again, then master will send a new subpart by splitting any subpart of other worker. Finally, when all workers have not enough points in subpart to split, the worker will be terminated.

Worker Statechart Diagram
Figure 4.6 – Worker State Chart Diagram
State Diagram figure 4.6 in order to illustrate operations of workers. Before receiving subpart, worker must to send “Pull” tag message to the master, later on worker will receives one of two tag messages below:

From the starting point, the worker has to initialize subparts for all workers (initial splitting at section 4.3.2). Afterward, master gets the message of worker from buffer. In the buffer, master could receive one of five tag messages corresponds with five transitions which are “Pull”, “Tile”, “NotifyPi0”, “CheckCover” and “UpdateSubpart” tag. We will begin with each tag below:

- The context is that worker received “Subpart” tag message. Next step, it has to scan for uncovered $\pi_0$ in its subpart and there is two cases here:
  - Basically, with receiving new subpart worker will detect uncovered point in the first time. To call IM on the uncovered $\pi_0$, firstly worker will notify the point to master (For dynamic splitting purpose). After calling IM, worker could receive “tile” or “KillIM” messages which are described in next sub-section 4.4.2. Assume that worker receive “KillIM” message then worker will enter directly to “Waiting for tag message” state after sending “UpdateSubpart” tag message. If not, worker will receive returned tile result, later on this tile will be stored in repository and send to master then it goes to “Waiting for tag message” state. In “Waiting for tag message” state after sending “UpdateSubpart” tag message, worker will receive one of two tag message, depend on its subpart has changed for dynamic splitting before, worker will receive “Subpart” or “Continue” tag message conversely.
  - In case worker found uncovered point that means this point is covered by previous finished tile or tiles from other workers. Thus worker need to find a next point. Note that worker will scan sequentially.

- Besides that, “Uncovered $\pi_0$ is not found” state could be entered then return back to the beginning to pull new subpart. This means this point is covered by some finished tiles previously or tiles which are received from other workers.

We give in Appendix A.2 the whole worker pseudo code.

4.4.2 Worker Violation Detection – Check Covered Pi0

As we mentioned the problem in section 3.3 which 2 more processes or nodes could yield the same tile and hence waste of time. For real scale model this will be huge problem.

In this section we will introduce a new solution which solves this problem and improves performance for all the currently used distributed IM algorithms.

In previous algorithms, when more than one process call points on a same tile as first case in figure 4.7, 2 processes call points on same tile. After process 1 finished its tile, process 2 works until it finishes its tile. Thus it wastes resources
and time and this is one of disadvantages of recent distributed IM algorithms that we could not avoid.

To solve this problem, we proposed a new mechanism to detect early the violation. Each step of IM worker or process has to check if its point is covered or not. If the point is covered then worker has to stop immediately then move to next uncovered point.

**Worker Violation Detection Statechart Diagram**

To understand how workers could detect violation, we will give a legible state chart diagram in figure 4.8 with explanation. Violation detection procedure will be occurred in “Calling IM” state in 4.6 so that it also is a part of worker state chart diagram.

In beginning “CheckCover” tag will be sent firstly, then worker could receive one of two messages master:

- If worker receives “Continue” tag, it will goes to next step, or final step with returning tile to master then finally terminate.

- Conversely, master will receive “TileList” tag message from master. Tile list is new tiles of other workers need to update for the worker’s repository, after updating master will detect if its current is covered or not. Suppose that maser detected π₀ is not covered then it will take the “Do next step” transition such as receiving “Continue” tag. Otherwise, master must be terminated if π₀ is covered.
Figure 4.8 – Worker State Chart Diagram IM
We give in Appendix A.2.1 the whole worker violation detection pseudo code.

4.5 Master Worker Interaction

In the previous sections, we have introduced master and worker separately. For now, we sketch simple sequence diagrams which are more understandable to determine how the worker and the master work together and how they interact with each kind of tag message.

4.5.1 Master Worker Sequence Diagram

An abstract model describes interactions between master and worker is needed to know how all kind of tags work. This sequence diagram includes:

- Master: stands for the master process with rank 0 in default.
- Worker1: stands for the worker process with rank 1.
- InverseMethod: is an algorithm that each worker has to call.

Assume that message sending between the master and the worker always succeeds. In order to make more succinct model, we will not describe how processes store their messages in their buffer and get their messages out of their buffer. Besides that, we also eliminate unnecessary messages.

Message 1: In the beginning, the master initializes subparts for each worker process. Simultaneously, all the workers send “Pull” tag message to the master.

Loop 1: Later, master will enter the main While loop with a first IF statement which includes two branch conditions:

- Alt 2.1 first one, the master will checks for an available subpart then send the subpart to worker in case it has. The purpose of first condition is that the master will send all the subparts which are split in initializing subpart phase. Later on, a loop and statment conditions will be occurred as below:
  - Loop 3: After sending subpart, the worker starts with a loop as the first main loop to traverse all points in the subpart one by one.
  - Try/Catch statement: we used try-catch statement simply to interrupt 3rd loop if the worker has just detected violation.
  - Message 4: Before calling IM on any point, the worker need to notify its current point to the master. This is necessary for the master knows where the worker is calling on its subpart.
  - Messages 5, 11, 12: After receiving “NotifyPi0” tag message at master side. Afterward, the worker tries to call IM on the current point. In the beginning of every IM’s steps, worker musts pull new tiles from the master to update its repository to check whether the point
is covered at that time. If the current point is covered by new tile, the worker has to stop then move to the next step (13th message). Conversely, it will do until tile result is finished, then the worker sends it to the master (11th, 12th messages).

– Messages 13, 14, 15, 16 and Alt 6.1, 6.2: Before the worker move to the next point, it has to update its subpart, because its subpart might be split by master in the pass. If subpart is changed, master sends “Subpart” tag message to the worker to update its subpart. Otherwise, the master will send continue tag message to inform that its subpart is not changed.

– Alt 2.2: in contrast with Alt 2.1 at 4.5.1, this statement condition indicates there is a worker \( w \) which finishes calling all points on its subpart sooner than the others. Thus the master has to find another subpart then splits a small part from this subpart of other worker to send this small part to the \( w \) worker. The main purpose of this case is to make all processes as busy as they can. Actually, it depends on the uncalled points remaining on subpart so that two possible situations can be happened:

– Alt 7 and message 17, 18 if there has a splittable supart of any worker, master will split this subpart into 2 then send the uncalled points part to the worker.

– Alt 8, 9 and message 19, 20: alt 8 is contrary of alt 7 at 4.5.1 that all workers have not any single uncalled point to split so that master will send “Terminate” tag message to the worker. Finally at alt 9, the master will check whether this worker has just been terminated is the last worker then master considers to terminate itself.

For convenience, we describe the function “Check Covered pi0” which is a part of whole diagram in a separate diagram in section 4.5.2.
Figure 4.9 – Master Sequence Diagram
We give in Appendix A.1.2 the whole dynamic-splitting pseudo code for the master.

4.5.2 Check Covered $\pi_0$ Sequence Diagram

The “Check Covered $\pi_0$” function idea is introduced in previous section 4.4.2. For now, a sequence diagram of this function is given to describe how it works. In fact, this diagram is a part of the large previous “Master Worker” sequence diagram 4.9. In the beginning, message 1 of this model is the same with message 6 in “Master Worker” model. Note that checking covered $\pi_0$ will be occurred in each step of calling $IM$. After the Worker1 has received message 1, Worker1 sends “CheckCovered” tag message to the Master. Then it depends on new tiles in the Master’s repository, which is needed for the Worker1 is available or not. Hence there are two cases can be occurred:

- Alt 1.1: is a condition that Master has no new tiles for sending to Worker1. Simultaneously, “Continue” tag message is sent to Worker1.

- Alt 1.2: in contrast with Alt 1.1, the Master has new tiles for the Worker1 and sends them to the Worker1 as message 5. After that, Worker1 has responsibility to check whether $\pi_0$ is covered, Worker1 will set $KillIM$ variable to true if $\pi_0$ is covered as message 7 and vice versa as message 8.

Finally, Worker1 finishes calling $IM$, it will check whether “killIM” variable is true or not.

Figure 4.10 – Check Covered $\pi_0$ Sequence Diagram
Chapter 5

Experimental Validation

For conducting experiments, all of results in this chapter are run on Magi, a real cluster of Paris 13 University. Magi cluster contains 16 nodes with:

- 12 nodes with 12 cores each (2 GB of memory per core, therefore 24 GB in total)
- 5 nodes with 8 cores each (2 GB of memory per core, therefore 16 GB in total)
- one node with 40 cores and 512 GB of memory in total

5.1 Case Studies

Before going to experiment results, we would like to introduce briefly about three problems in practice which are used in our experiment:

- Flip Flop, Flip Flop 4D (4 dimensions) are model of asynchronous circuits case studies.
- RCP stands for Root Contention Protocol of the IEEE 1394 (FireWire) High Performance Serial Bus. This protocol is part of a leader election protocol in the physical layer of the IEEE 1394 standard.
- Sched 3, Sched 3 Large V, Sched 3 Large V 3D (3 dimensions) and Sched 5 are five scheduling problems on a single processor. The goal is to synthesize task parameter valuations guaranteeing that every task meets its relative deadline.

In fact, there are more than 3 and for more information about these case studies and other case studies, reader could find in the book Inverse Method [AS13].
5.2 Behavioral Cartography Experiment

The bar charts below compare the time in seconds from synthesizing constraints for 5 different distributed algorithms and the last (non-distributed) is for non-distributed fashion. To check correctness of different algorithms, a figure of BC results was given at each case study. Besides that, we only present the BC in 2 dimensions for convenience.

Figure 5.1 – Flip Flop experiment (Size of V0: 3082 Constraints: 8)

Figure 5.2 – Flip Flop BC (From left to right : Subpart, Sequential, Random20)
Figure 5.3 – Flip Flop 4D experiment (Size of V0: 386400 Constraints: 182)

Figure 5.4 – Flip Flop 4D BC (From left to right: Subpart, Sequential, Random20)
Figure 5.5 – RCP experiment (Size of V0: 3050 Constraints : 19)

Figure 5.6 – RCP BC (From left to right : Subpart, Sequential, Random20)
Figure 5.7 – Sched 3 experiment (Size of V0: 286 Constraints : 59)

Figure 5.8 – Sched 3 BC (From left to right : Subpart, Sequential, Random20)
Figure 5.9 – Sched 3 Large V experiment (Size of V0: 14746 Constraints : 71)

Figure 5.10 – Sched 3 Large V BC (From left to right : Subpart, Sequential, Random20)
Figure 5.11 – Sched 3 Large V 3D experiment (Size of V0: 530856 Constraints: 388)

Figure 5.12 – Sched 3 Large V 3D BC (From left to right: Subpart, Sequential, Random20)
Figure 5.13 – Sched 5 experiment (Size of V0: 1681 Constraints: 177)

Figure 5.14 – Sched 5 BC (From left to right: Subpart, Sequential, Random20)
7 experiments of 5 algorithms were taken. In experiments 5.1 and 5.5, we tried to use number of processes larger than number of tiles to see how it react to the results. In 5.1 results it is not really efficient which is contrast to 5.5 results. Thus, It is hard to predict the results if we use number of processes over the number of tiles.

Even with 5.3, 5.7, 5.9, 5.11 and 5.2 results, reader could realize it is hard to indicate which algorithm better than the other. It totally depends on which example is used with particular algorithm.

Finally, we computed average of 7 case studies for each algorithm. And each algorithm will run with 4, 12, 32, 64 processes in the last figure above. As the figure Subpart with KillIM slightly gave better results than the other algorithms.

5.3 Algorithm Speed Up and Efficiency

From Amdahl’s law we tried to sketch the SpeedUp and Efficiency charts. However, Sequential algorithm and Distributed version of it is different.

For SpeedUp of Sequential results formula: \( S = \frac{T_s}{T_p} \)

For Efficiency results formula: \( E = \frac{T_s}{P \times T_p} = \frac{S}{P} \)

With \( T_s \) is run time with single process(sequential), \( T_p \) is run time with multi-processes(parallel) and \( P \) is number of processes. Since each processor have many cores so that we use process term to indicate each core of processor.
Figure 5.16 – SpeedUp chart

Figure 5.17 – Efficiency chart
We chose the previous fastest Shuffle, the slowest Sequential and the newest Subpart algorithms for comparison purposes. 5.16 and 5.17 figures above compare speed and efficient of 3 algorithms with different number of processes. In first figure, the speedup of Subpart algorithm increases impressively and faster than the others although Shuffle and Subpart are quite equal with 4 processes. Hence, no doubt the efficiency of Subpart is also better than other algorithms. It is also noticeable that in efficiency line chart all algorithms decrease slightly while the number of processes increases steadily, there is some loss of algorithms because of violation of processes which is mentioned in previous sections. It is clear that the Subpart is the most efficient recently, but it has not reached limit of speedup and efficiency so that we are attempting to find another approach to get there although it is really difficult in theory.

All values of figures above are average values which are computed from 7 test cases in previous section 5.2. For each algorithm line chart, reader could find out in appendix section B.
Chapter 6

Conclusion and Perspectives

In conclusion, we proposed a new efficient distributed algorithm for BC (Behavioral Cartography), a specific solution for particular MPI design problem. In fact, we used 2 main approaches to solve 2 critical problems below:

- Choosing on which points IM should be called.
- Deciding whether an execution of IM on a given point should be stopped or not, whenever this point has been just covered by another tile.

Solutions:

- To solve first problem, we proposed a new subpart distribution scheme that each worker will pick $\pi_0$ in its subpart. Thus this solution could prevent more than one process worker yielding a same tile and reduce traffic between master and workers since each worker pulls a subpart as a set of points.

- Since all algorithms must call all points in BC to be sure covered 100% BC. Hence, it should be stopped if any point has been covered by another tile. The last problem was resolved by implementing new violation detection mechanism that could avoid yielding a same tile of more than one worker process.

In fact, there still remain few problems that with $n$ processes, the algorithm could not get $n$ times faster, one of main reasons is that one process plays as the Master role to compute splitting subpart and it does not call IM, only $n - 1$ worker processes call IM. Thus it hardly compensates for master process. Beside that, we could not allocate exactly each tile for each process. Hence, there are more than one Worker which calls IM on the same tile result, this problem consumed time even though we proposed a violation detection mechanism.
Let us do a brief overview of all algorithms. Before we present Subpart algorithm, Shuffle algorithm is the fastest but its drawback is to require huge memory to shuffle a point array in beginning phrase. Random algorithm is very fast, consumes less memory and 100% BC covered. Since, both of them call randomly points, which is hardly to splitting thus Subpart algorithm is based on Sequential algorithm which is easy for splitting, less memory consumption and covers 100% BC but it is also the slowest algorithm.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Sequential</th>
<th>Random</th>
<th>Shuffle</th>
<th>Subpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less Memory</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100% BC Covered</td>
<td>√</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Violation Detection</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>√</td>
</tr>
<tr>
<td>Number of MPI Tags</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6.1 – Feature table (N: Not implemented yet)

**Future Works**  In the future, we will attempt to combine Subpart and Shuffle algorithm to design a new algorithm. Since Shuffle is very fast but its shortcoming is that it has not enough memory to shuffle huge point array in beginning. Thus, Subpart algorithm has a responsibility to split the point array into smaller arrays which could fix in the memory. Besides that, with integrating new violation detection feature which could make it even faster. Therefore, next algorithm will be more simple, less memory and faster.

We will also attempt to design an autonomous distribution scheme, where each member uses 2 processes, 1 process is used for calling IM and another process is used for sending and receiving tag messages. However, we are afraid it may not be efficient since each member uses 2 processes to communicate and communicating with others in circumstance of BC might flush the bandwidth. Hopefully, in next works we could design a more efficient autonomous algorithm.

Until now, each IM call used 1 process to compute a tile and it also uses BFS (Breadth First Search) to explore a trace set. So that we need a high processor clock to explore faster which recent is limited by physics. In future, it is more efficient if we could allocate each process for each trace set path. Hence, with many processes are assigned to each IM call, synthesizing constraints will be faster and we do not need high clocked processors and could try in GPU’s environment.
Bibliography


[Har05] Jon D Harrop. OCaml for Scientists. Flying Frog Consultancy, 2005. 19


Appendix A

Pseudo Code and Explanation

In the following pseudo codes of appendix A, we would like to describe our implementations. Master pseudos code will be shown firstly at algorithms 1, 2 and its functions at algorithms 3 and 6. The lasts are worker and its function at algorithm 7 and 8, respectively.

A.1 Master Pseudo Code
Algorithm 1: Master Algorithm

**Input**: PTA (Parametric Timed Automata)
D (Parameter Domain)
P (Number of initial processes)

**Output**: S

1. \( S \leftarrow \{ \} \)
2. Type Subpart = Array[[from, to]]
3. workerToSubpart[(n, subpart)] ← \{ \}
4. tile_Buffer[(n, tile)] ← \{ \}
5. InitialSubparts ← Initialize_Subparts(Subparts, P)
6. workingWorker ← true
7. current_Subpart[(n, subpart)] ← \{ \}
8. while workingWorker do
   9. \( n, Tag \leftarrow receiveFromWorker() \)
   10. if Tag = Pull() then
       11. [At algorithm 2]
   12. else if Tag = Tile(K) then
       13. if isNotRedundant(K) then
           14. \( S \leftarrow S \cup \{ K \} \)
       15. foreach worker ≠ n do
           16. \( \text{tile}_\text{Buffer}[\text{worker}] \leftarrow \text{tile}_\text{Buffer}[\text{worker}] \cup \{ \text{Tile}(K) \} \)
       17. else if Tag = NotifyPi0(pi0) then
           18. current_Pi0[(n, pi0)] ← \{ \}
       19. else if Tag = CheckCover() then
           20. if tile_buffer[n] ≠ \( \emptyset \) then
               21. sendToWorker(n, TileList(tile_buffer[n]))
               22. tile_Buffer[n] ← \( \emptyset \)
           23. else
               24. sendToWorker(n, Continue())
           25. else
               26. if n ∈ current_Subpart then
                   27. sendToWorker(n, TileList(current_Subpart[n]))
               28. else
                   29. sendToWorker(n, Continue())
Algorithm 2: Master Algorithm (Receive Pull tag message)

if $n \in \text{workerToSubpart}$ then
  \text{workerToSubpart}[n] \leftarrow \emptyset

if \text{AvailableSubparts()} then
  \text{Subpart} \leftarrow \text{getSubpart(InitialSubparts)}
  \text{sendToWorker}(n, \text{Subpart}(\text{subpart}))
  \text{workerToSubpart} \leftarrow (n, \text{Subpart})

else
  if has a splittable subpart in workerToSubpart then
    \(w, \text{subpart} \leftarrow \text{getSplittableSubpart(workerToSubpart)}\)
    old_sp, new_sp \leftarrow \text{dynamicSplitting(subpart, getCurrent_Pi0(w))}
    \text{sendToWorker}(n, \text{Subpart(new_sp)})
    \text{workerToSubpart}[n] \leftarrow \text{new_sp}
    \text{current_Subpart}[w] \leftarrow \text{old_sp}
  else
    \text{sendToWorker}(n, \text{Terminate()})
    if workerToSubpart[worker] = \emptyset then
      workingWorker \leftarrow \text{false}

Master Algorithm:
Inputs and outputs:

- Inputs of master: a parametric timed automata (PTA), a parameter domain (D) in this case is $V_0$. The final is number of initial processes (P) this value of parameter will be taken from the MPI function.

- Output of master: is a set of tiles within BC.

Master’s body:

- Line 3: $S$ is a set of tiles which will be synthesized from the workers.

- Line 4: $\text{Subpart}$ is an array type, expressed as polyhedron ($V_0$) with each element stores information of polyhedron’s (parameter domain or BC) dimension length.

- Line 6: a main purpose of $\text{tile Buffer}$ is to avoid flooding bandwidth between master and workers. To so so, when Master receives new tiles from any worker then these tiles be must be sent to other workers for updating, this parameter is used as buffer to temporarily store needed tiles of each worker.

- Line 7: $\text{InitialSubparts}$ parameter is created in the beginning to store the subparts of each worker after using “Initial Splitting” function.
Line 8: The `workingWorker` parameter is boolean type which indicates whether there is any worker is doing or not. If value of this parameter is set “false” the master will be terminated immediately.

Line 9: The `current_Pi0` parameter stores the current π₀ of each worker. These current π₀s are very useful for “Dynamic Splitting” function to know where to split afterward.

Line 10: The `current_Subpart` parameter stores the current Subpart of each worker, this list records changes of all worker’s subpart then the changed subparts will be sent to its worker.

Line 11: This While loop is the main loop which is terminated iff all workers finish their tasks.

Line 12: Indicates that master waits for the tag message from workers. From now, master could receive one of 5 tag messages: `Pull`, `Tile(K)`, `NotifyPi0(pi0)`, `CheckCover()`, `UpdateSubpart()`:

Line 13: The first condition is that master received the `Pull()` tag message. At the beginning, master will delete old subpart of this worker from `workerToSubpart` list if it exists. Finally, master has to check whether there is any available subpart in the list. In case of existing an available subpart, master will send this to the worker. On the contrary, the master tries to split subpart from other workers, then 2 cases could occur:

- Line 8 of algorithm 2: if master found a splittable subpart in its `workerToSubparts` list. Afterward, dynamic splitting function will be used to split the subpart of the worker \( w \) into uncalled \( \pi_0 \) part \( \text{new}_sp \) and remaining part \( \text{old}_sp \). The \( \text{new}_sp \) will be sent to the requestor worker and the other will put into `current_Subpart` which will be sent to the worker \( w \) later. Finally, master also updates 2 new subpart changes in the `workerToSubparts` list.

- Line 13 of algorithm 2: in case subparts of all workers could not be split. Master has to send `Terminate()` tag message in order to terminate this worker.

Line 15: Master receives `Tile(K)` tag message from the worker then updates its tile repository \( S \). An important purpose of this tag is to send the tile \( K \) to other workers later. Thus, the master has to temporarily store it in the `tile_Buffer` list as buffer. The workers will receive a set of tiles \( K \)'s after sending `CheckCover()` tag message to the master.

Line 20: After master receives `NotifyPi0(pi0)` tag message. The master will update its `current_Pi0` list with the worker’s(sender) current \( \pi_0 \).

Line 22: Third branch condition is receiving `CheckCover()` tag message of master, every step of IM algorithm, worker sends this tag to master. Then master will send a list of new tiles if it is available in `tile_Buffer[n]`. Conversely, this list is empty then master will send `Continue()` tag message.
• Line 28: final branch condition indicates that master receives *UpdateSubpart* tag message from worker. *current_Subpart* list will be used to store changes of all workers subpart. If the subpart of the worker is changed, master will send a changed subpart to owner for updating at worker side.

### A.1.1 Initial Splitting Pseudo Code

This section is to explain more details about Initial Splitting algorithm 3 and its functions 4 and 5.

```
\begin{algorithm}[H]
\caption{Initialize Subpart Algorithm}
\textbf{Input:} subparts
\textbf{Output:} subparts (List of Supart)
\begin{algorithmic}[1]
\State \textbf{for} \textit{i} = 0 to \textit{n} \textbf{do}
\State $\text{maxPi0s} \leftarrow 0$
\State $\text{subno} \leftarrow 0$
\State \textbf{for} \textit{i} = 0 to \textit{subparts.length}() - 1 \textbf{do}
\State \quad temp $\leftarrow$ \text{getTotalPi0}(\text{subparts.get}(\textit{i}))
\State \quad \textbf{if} $\text{maxPi0s} < \text{temp} \textbf{then}$
\State \quad \quad $\text{maxPi0s} \leftarrow \text{temp}$
\State \quad \quad $\text{subno} \leftarrow \textit{i}$
\State \textbf{if} $\text{maxPi0s} \neq 1$ \textbf{then}
\State \quad \text{newSubpartList} $\leftarrow$
\State \quad \quad \text{splitLongestDimensionSubpart(\text{subparts}(\text{subno}))}
\State \quad \text{subparts} $\leftarrow$ \text{subparts.removeAt}(\text{subno})
\State \quad \text{subparts} $\leftarrow$ \text{newSubpartList}
\State \textbf{else}
\State \quad raise Exception "could not split smaller"
\State \textbf{return} \text{subparts}
\end{algorithmic}
\end{algorithm}
```

Algorithm 3: Inputs of algorithm 3 include a list of subparts and \textit{n} is a number of subparts which equals to \textit{n} processes. Value of \textit{n} will be computed from number of input processes through MPI library. Main idea of algorithm is to get a biggest subpart in the list then the master split it into 2 smaller subparts equally. This procedure will iterate until \textit{n} subparts were split or all dimensions in the list is equals to 1, after that the master will raise the message that could not split anymore.
Algorithm 4: Split At Longest Subpart Dimension Algorithm

**Input:** subpart

**Output:** subparts (List of Supart)

1. \( d \leftarrow \text{subpart.get_dimensions}() - 1 \)
2. \( \text{maxDimensionLength} \leftarrow 0 \)
3. \( \text{maxDimension} \leftarrow 0 \)
4. \( \text{temp} \leftarrow 0 \)
5. for \( i = 0 \) to \( \text{subpart.get_dimensions}() - 1 \) do
6. \hspace{1em} \( \text{temp} \leftarrow \text{s.getMax}(i) - \text{s.getMin}(i) + 1 \)
7. \hspace{1em} if \( \text{temp} >= \text{maxDimensionLength} \) then
8. \hspace{2em} \( \text{maxDimensionLength} \leftarrow \text{temp} \)
9. \hspace{2em} \( \text{maxDimension} \leftarrow i \)
10. \( \text{listSubpart} = \text{split}(s, \text{maxDimension}) \)
11. return \( \text{listSubpart} \)

Algorithm 4: this simple function is used by algorithm 3 to check all dimensions inside the subpart then the master gets a longest dimension to split.
Algorithm 5: Split Algorithm

**Input:** subpart

**nthD**

**Output:** subparts (List of Subpart)

1. `maxlength ← Supart.getMax(nthD) − Supart.getMin(nthD) + 1`
2. `S ← Suparts`
3. **if** `maxlength = 1` **then**
4. **raise Exception** "the length is minimum, could not split smaller"
5. `s1 ← newsubpart`
6. `s2 ← newsubpart`
7. **for** `i = 0` **to** `subpart.getDimensions() − 1` **do**
8. **if** `i == nthD` **then**
9. **if** `maxlength > 2` **then**
10. `s1.setMin(i) ← s.getMin(i)`
11. `s1.setMax(i) ← (s.getMin(i) + (maxlength/2 − 1))`
12. `s2.setMin(i) ← (s.getMin(i) + (maxlength/2))`
13. `s2.setMax(i) ← s.getMax(i)`
14. **else**
15. `s1.setMin(i) ← s.getMin(i)`
16. `s1.setMax(i) ← s.getMin(i)`
17. `s2.setMin(i) ← s.getMax(i)`
18. `s2.setMax(i) ← s.getMax(i)`
19. **else**
20. `s1.setMin(i) ← s.getMin(i)`
21. `s1.setMax(i) ← s.getMax(i)`
22. `s2.setMin(i) ← s.getMin(i)`
23. `s2.setMax(i) ← s.getMin(i)`
24. `subparts ← s1, s2`
25. **return** subparts

Algorithm 5: This function is used by “Initial Splitting” algorithm 3 and “Dynamic Splitting” algorithm 6. The function simply splits the subpart into two smaller almost equal subparts.
A.1.2 Dynamic Splitting Pseudo Code

This section explains more details about Initial Splitting algorithm.

<table>
<thead>
<tr>
<th>Algorithm 6: Dynamic Splitting Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> subpart</td>
</tr>
<tr>
<td><strong>pi0</strong></td>
</tr>
<tr>
<td><strong>Output:</strong> subparts (List of Supart)</td>
</tr>
<tr>
<td><strong>notFound</strong> ← true</td>
</tr>
<tr>
<td><strong>maxDimensionLength</strong> ← 0</td>
</tr>
<tr>
<td><strong>j</strong> ← subpart.dimensions() - 1</td>
</tr>
<tr>
<td><strong>lst</strong> ← ∅</td>
</tr>
<tr>
<td><strong>while</strong> !notFound do</td>
</tr>
<tr>
<td><strong>if</strong> s.getMin(j) &lt; pi0.(j) <strong>then</strong></td>
</tr>
<tr>
<td><strong>s.setMin(j) ← pi0.(j)</strong></td>
</tr>
<tr>
<td><strong>maxDimensionLength</strong> ← s.getMax(j) - pi0.(j) + 1</td>
</tr>
<tr>
<td><strong>if</strong> maxDimensionLength &gt; 1 <strong>then</strong></td>
</tr>
<tr>
<td><strong>lst</strong> ← Split(s,j)</td>
</tr>
<tr>
<td><strong>notFound</strong> ← false</td>
</tr>
<tr>
<td><strong>j</strong> ← j - 1</td>
</tr>
<tr>
<td><strong>return</strong> lst</td>
</tr>
</tbody>
</table>

Algorithm 6: “Dynamic Splitting” function of the master has inputs are a subpart and its current $\pi_0$ 1. From line 7 to 9 are to find a splittable dimension of current subpart and eliminate a part of called IM points. Then the algorithm reuses Split 11 function to split the subpart into two smaller subparts.

A.2 Worker Pseudo Code

Here is Worker Pseudo Code
Algorithm 7: Worker Algorithm

**Input:**
- PTA (Parametric Timed Automata)
- D (Parameter Domain)
- P (Number of initial processes)

**Output:**
- S (Set of tiles)

1. subpart ← {}  
2. S ← {}  
3. ReceivedTerminateTag ← false  

while !ReceivedTerminateTag do

1. sendToMaster(Pull())
   /* get the new subpart */
2. Tag ← receiveFromMaster()
3. if Tag = Subpart(subpart) then

4. subpart ← receiveFromMaster()
5. morePi0 ← true

6. while morePi0 ∈ subpart do

7. sendToMaster(NotifyPi0(pi0))
8. try:
9. K ← IM(pi0)
10. S ← S ∪ {K} sendToMaster(Tile(K))
11. catch KillIM({})
12. sendToMaster(UpdateSubpart())
13. Tag ← receiveFromMaster()
14. if Tag = Subpart(subpart) then

15. updateSubpart(subpart)
16. else /* Tag = Continue */
17. ReceivedContinueTag ← true

18. moveToUncoveredPi0()  // If it’s last point ->
19. morePi0 = false

else /* Worker receive Terminate Tag from Master */

20. ReceivedTerminateTag ← true

21. moveToUncoveredPi0()  // If it’s last point ->
22. morePi0 = false

23. end while

end while
Worker Algorithm:
Algorithms of worker are more simpler than the master.
Inputs and output of the worker are the same with the master. Worker’s body:

- Line 3: subpart parameter store information of current subpart.
- Line 4: $S$ is set of tiles which is same with the master.
- Line 5: ReceivedTerminate is the boolean type parameter which indicates whether this worker has to terminate by master.

Line 6: show a main While loop. Inside the loop has 2 branch conditions, the final at line 26 indicates that the worker receives the Terminate tag message and the other is Subpart tag message that worker receives from master.

Line 7: In beginning of main loop, worker sends Pull tag message to the master for pulling subpart.

Line 9: master received Subpart tag which is attached a subpart. In this stage worker sequentially calls IM for each point in the subpart.

Line 12: 2nd While loop is to check each uncalled point in subpart. In the beginning of this loop, Worker will notify its current $\pi_0$ first to master then worker enters to a new loop. Inside this new loop, worker could receive one of two tags of master. First tag at line 21 is the Subpart tag message which is need for updating. Since the worker sent Notify $\pi_0$ tag message to the master before, the master could check for the remaining(uncalled) point of this worker and share a part of this subpart to the other workers anytime. So that, whether master shares its subpart, the worker always needs to update its subpart to guaranty it does not call $\pi_0$as which are located out of its subpart. And the final branch condition is Continue tag at line 23 for terminating the most inner while loop.

At Line 25: Worker has to move to the uncovered area after updating its tile repository. If worker found new uncovered $\pi_0$, worker will begin to call IM with this new $\pi_0$. Conversely, all points are covered, more $\pi_0$ variable will be set to false and then worker breaks the 2nd while loop.

Line 15: “Try” command indicates that worker will try to call IM with its uncovered $\pi_0$, in each step of IM, worker will detect violation by calling the algorithm 8.

Assumed that worker did not detect any violation in all steps of calling IM. The worker will get the tile result, later on it will send to the master. If there is any violation detected, an exception will be through to terminate the worker. Finally, worker has to move to a new uncovered $\pi_0$ as line 25.
A.2.1 Worker Violation Detection – Check Covered Pi0

Pseudo Code

Algorithm 8: Worker Covered Pi0 Checking Algorithm

\[ \begin{align*}
\textbf{Input:} & \quad \text{PTA (Parametric Timed Automata)} \\
& \quad D (\text{Parameter Domain}) \\
& \quad P (\text{Number of initial processes}) \\
\textbf{Output:} & \quad S (\text{Set of tiles}) \\
1 & \quad \text{sendToMaster}(\text{CheckCovered}()) \\
2 & \quad \text{Tag} \leftarrow \text{receiveFromMaster()} \\
3 & \quad \textbf{if} \quad \text{Tag} = \text{TileList}(Ks) \quad \textbf{then} \\
4 & \quad \quad S \leftarrow S \cup \{Ks\} \\
5 & \quad \textbf{else} \\
6 & \quad \quad \text{// Tag = Continue()} \\
\end{align*} \]

Algorithm 8: A main purpose of algorithm 8 is to detect violation. To do so, in each step of calling IM, worker has responsibility to call this function. At the beginning, worker sends CheckCovered tag message (at line 3) to master. Later on, master will send a set of new tiles which is needed to update (at line 6) and conversely master will send a Continue tag message as line 7 in case there is not any new tile.
Appendix B

Algorithm Speed Up and Efficiency

From 7 test cases at section 5.2, we computed speedup and efficiency for each specific algorithm. Line graphs B.1, B.3, B.5 illustrate speed up and B.2, B.4, B.6 illustrate efficiency of Subpart, Sequential and Shuffle respectively. For SpeedUp and Efficient comparison among all algorithms, reader could turn back to section 5.3.

Figure B.1 – Subpart Algorithm Speed Up
Figure B.2 – Subpart Algorithm Efficiency

Figure B.3 – Sequential Algorithm Speed Up
Figure B.4 – Sequential Algorithm Efficiency

Figure B.5 – Shuffle Algorithm Speed Up
Figure B.6 – Shuffle Algorithm Efficiency