**NANYANG
TECHNOLOGICAL
UNIVERSITY**

**Parallelizing Model Checking Algorithms
Using Multi-core and Many-core Architectures**

Zhimin Wu

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

2017

Parallelizing Model Checking Algorithms Using Multi-core and Many-core Architectures

Zhimin Wu

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Doctor of Philosophy

2017

Abstract

The reliance on the functioning of software systems is growing with the development of IT industry. The complexity and scalability to design these systems are increasing, especially for concurrent systems with concurrent executions, communications and resource distributions, e.g., classical cryptography protocols and quantum communication systems. Thus, the reliability of these systems becomes crucial. Model checking is an automated verification approach to validate these complicated systems, which has been proved to be effective in discovering errors.

Model checking works through the exploration of all possible system states in a brute force manner. This technique checks whether a system model meets a given specification, and it consists of system modeling, requirement specification, and verification. Compared to other traditional approaches, e.g., simulation, testing, and deductive verification, model checking has many advantages for its automatic and exhaustive searching. However, model checking is not feasible for systems with infinite state space, since the data size would be unbounded or the number of processes would be infinite. For finite state space systems, there are also two major problems or obstacles that limit the usage of model checking techniques in practical industrial systems. Firstly, model checking suffers from the infamous state space explosion problem. Secondly, the efficiency of algorithms on model checking is always subject to the size of state space. It is challenging to handle the examination of largest possible state space efficiently with limited processors and memories.

By now, many techniques are proposed to handle the state space explosion problem. Symbolic model checking approaches reduce the cost of storage to store the system model by representing the state space implicitly with the structure called binary

decision diagram (BDD). Partial order reduction is designed to deal with the state space explosion in concurrent systems with asynchronous components. It builds a reduced state space which eliminates the additional interleaving sequences for executions. On-the-fly model checking avoids generating the complete state space by conducting the verification during the process of state space generation. Symmetry Reduction, which is based on group theory, reduces the size of the state space by identifying the isomorphic states in the state space. Besides these typical smart techniques, parallel computing has recently been garnering much attention in the model checking community. Parallel computing is primarily implemented to extend the available computation resources to handle larger systems. The performance improvement gained from the parallelized generating and analyzing of large state spaces is promising. Thus, we conducted our research on the parallelization of model checking problems.

The existing research on the parallelization of model checking problems reflects the idea that model checking problems can be transferred to some graph exploration problem. So far, various parallel computing platforms have been utilized, e.g., the multi-core CPU, many-core GPU and even the cloud environment. Based on the survey on previous research, the targets of our research are: 1) parallelizing sub-problems in model checking algorithms, 2) improving the parallelization in existing approaches by utilizing the latest techniques in different parallel computing platforms, and 3) combining parallel computing with the existing techniques like on-the-fly model checking for the state space explosion problem. To this end, we analyze the feasibility of algorithms in model checking and start with the algorithms related to LTL model checking. We handle the state space generation, counterexample generation and the on-the-fly strongly connected components (SCC) detection process, to build approaches which optimize the original algorithm via parallelization in different platforms. Then we work on the problem related to probabilistic model checking – the computation of the reachability probabilities, which is also an important subroutine for determining approximately optimal policies of the typical probabilistic model – Markov decision processes (MDPs). We propose a GPU-accelerated value iteration to compute the

reachability probabilities of MDPs. Our parallelization of the algorithms achieves expected performance improvement.

Besides the algorithm-level parallelization, many existing research present the integration of the parallelized algorithms into many famous model checking tools, so as to prove their applicability. These applications have significantly improved the performance of their original sequential version of algorithms. Therefore, it is promising to extend our research from algorithm level to the application level. Our work has contained the extension of the existing PAT model checker with the concurrent LTL model checking algorithm. We aim at developing a new application which can both cover the model checking techniques, and contribute to the latest realistic problems which have not been covered by the applications mentioned above. To this end, quantum communication attracts our attention and is suitable to be the application domain. To the best of our knowledge, there is no tool support for the formal verification of quantum communication systems, including both the general model checking and the automatic quantum bisimilarity checking. Thus, we design and implement a tool-Qubet1.0, a model checking, bisimulation checking and emulation tool for quantum communication systems. Based on this essential tool, our algorithms-level parallelization is integrated with promising results.

To sum up, our research consists of both algorithm-level parallelization and application-level work. Five works are involved in the thesis: we investigate the existing solutions to state space explosion problem and the existing optimizations for model checking algorithms, as well as the latest techniques in different parallel computing platforms, in order to get a better understanding of our targets and the potential solutions. Firstly, we focus on three LTL model checking related problems; we make use of the many-core GPU to construct the parallel on-the-fly state space generation for reachability checking. We gain significant performance speedup (average 50x and up to 100x) compared with the traditional sequential algorithms; we make use of the multi-core CPU to build the concurrent on-the-fly SCC detection for automata-based model checking with fairness assumption. We achieve around 2x

speedup for the complete SCC detection in large-scale system models compared to the sequential on-the-fly model checking in PAT. Besides, our parallel on-the-fly fairness checking approach speeds up fairness checking around 2x-45x; Also, we exploit the latest architecture of many-core GPU and build the GPU-accelerated Counterexample Generation algorithm, which is proved to be effective and scalable; Secondly, we propose the GPU-based computation of the reachability probabilities. We gain around 10x speedup compared to the sequential version of VI, and around 1.5x-17x speedup compared to the topological value iteration. Finally, we design and implement an applicable tool QUBET1.0, and present that our algorithm-level research works well when being applied to the application. Our application work is significant to the designers of quantum communication systems.

Acknowledgements

My deepest gratitude goes first and foremost to my supervisor Prof. Yang Liu, for his constant encouragement and guidance. He has provided me great supports throughout my Ph.D. study on developing new research ideas, critical thinking and technical writing. Besides, he has walked me through all the stages of the writing of this thesis. His consistent and illuminating instruction really helps me to shape my research.

Secondly, I am grateful to Prof. Binsheng He and Prof. Rong su, who were in my Thesis Advisory Committee, for their valuable suggestions and comments to my research works.

Thirdly, I want to show my gratitude to Prof. Yuan Feng, Prof. Jun Sun, Prof. Yun Liang and Prof. Shengchao Qin. They have always given me instructions on my research publications.

In addition, I am grateful to Dr. Jianqi Shi and Dr. Fuyuan Zhang for their valuable suggestions and brainstorming sessions, which really did great help to my research progress. I would like to thank all my friends and teammates at the Cyber Security Lab (CSL) for their encouragement and help, especially, Dr. Fuyuan Zhang, Dr. Akin Gunay, Dr. Tim Muller, Dr. Bihuan Cheng, Hao Xiao, Dongxia Wang, Guozhu Meng, Xiaofei Xie, Junjie Wang, Xiaoning Du, HongXu Chen, Zhengzi Xu, Ruitao Feng, Yuekang Li, and Yuan Zhou. I also want to thanks our laboratory executive Tan Suan Hai for his technical support at CSL.

Finally, I want to express my utmost gratitude to my families, including my parents and my wife, for their unconditional support, understanding and trust. Their meticulously care and guidance will continue as one of my biggest sources of inspiration. Their care and company are precious treasures in my life.

Contents

Abstract	i
Acknowledgements	v
Contents	vi
List of Figures	ix
List of Tables	xi
1 Introduction	1
1.1 Motivations and Goals	2
1.1.1 Algorithm-level Parallelization: the Optimization of Model Checking Algorithms	2
1.1.2 Application-level Parallelization: Tool Support	6
1.2 Summary of Contributions	11
1.3 Thesis Outline	14
1.4 Publication List	16
2 Background	18
2.1 Model Checking	18
2.2 System Modeling	19
2.3 Property Specification and Verification	21
2.3.1 Safety properties	21
2.3.2 Liveness Properties and Fairness Assumptions	21
2.3.3 LTL Model Checking with Fairness Assumption	23
2.3.3.1 Counterexample Generation	26
2.3.4 Probabilistic Model Checking	26
2.3.4.1 Markov Decision Processes and Value Iteration	27
2.4 Model Checking Tools	29
2.5 The Architecture of Parallel Computing Platforms	30
2.5.1 Multi-core CPU Preliminaries	31
2.5.2 GPU Preliminaries	31
2.5.3 Other Platforms	35
3 GPU Accelerated On-the-fly Reachability Checking	37

3.1	Introduction	37
3.2	Related Work	38
3.3	GPURC Overview	39
3.4	GPU Accelerated On-the-fly State Space Generation for Reachability Verification	42
3.4.1	System Encoding in GPU	44
3.4.2	State Space Generation in GPU	46
3.4.3	Supporting Global Variables and Large Number of Concurrent Components	48
3.4.4	Algorithms	51
3.5	Evaluation	52
3.5.1	Performance Evaluation	54
3.5.2	Discussion	57
3.6	Conclusion	59
4	Concurrent On-the-fly SCC Detection for Automata-based Model Checking with Fairness Assumption	60
4.1	Introduction	60
4.2	Related work	62
4.3	Concurrent on-the-fly SCC detection for Model Checking Under Fairness Assumption	65
4.3.1	Approach Overview	66
4.3.2	Data distribution	69
4.3.3	Concurrent State Space Generation	70
4.3.4	On-the-fly Parallel Fairness Verification	72
4.3.5	Algorithm	74
4.3.6	Complexity	76
4.4	Implementation and Evaluation	77
4.4.1	Evaluation on SCC Detection for Model Checking	78
4.4.2	Performance of On-the-fly Parallel Fairness Verification	79
4.5	Conclusion	82
5	GPU Accelerated Counterexample Generation in LTL Model Checking	83
5.1	Introduction	83
5.2	Related Works	85
5.3	CUDA Accelerated Counterexample Generation	85
5.3.1	Detailed Approach	87
5.3.2	Dynamic Three-level Queue Management	92
5.3.3	Dynamic Hierarchical Task Schedule	93
5.3.4	Dynamic Duplicate Eliminated Path Recording	95
5.4	Experiments and Evaluation	97
5.4.1	Performance Analysis	98
5.4.2	Evaluation and Limitation	99
5.4.3	Optimization Options	101
5.5	Conclusion	101

6	A GPU-based Value Iteration Algorithm to Accelerate the Computation of Reachability Probabilities in MDPs	103
6.1	Introduction	103
6.2	Related Work	104
6.3	GPU Accelerated Value Iteration For Computing Reachability Probabilities	106
6.3.1	Parallelization	108
6.3.2	Compact Data Structure and Data Accessing	109
6.3.3	Parallel Backups and Convergence Reduction	111
6.3.4	Algorithm	112
6.3.5	Extension for Approximately Optimal Policies	115
6.3.6	Potential Extension for POMDPS	116
6.4	Implementation and Experiments	116
6.4.1	Evaluation of Performance	117
6.4.2	Analysis of Configurations and Parameters	119
6.4.3	Comparison with other GPU-based VI	120
6.5	Conclusion	121
7	Qubet: A Model Checking, Bisimulation Checking and Emulation Tool for Quantum Communication Systems	123
7.1	Introduction	123
7.2	Related Work	124
7.3	Basic Notions from Quantum Information Theory	125
7.3.1	Basic Linear Algebra	125
7.3.2	Basic Quantum Mechanics	127
7.3.3	QCCS and The Bisimulation between qCCS processes	129
7.4	The Architecture of Qubet	130
7.5	Description of Quantum Protocols in Qubet	131
7.5.1	Description	131
7.5.2	Grammar Checking	134
7.5.3	Simulation	136
7.6	Bisimilarity Checking between Quantum Processes	136
7.7	Demonstration	137
7.8	The Application of Parallelized Model Checking Algorithms	140
7.9	Conclusion	142
8	Conclusion	144
8.1	Thesis Summary	144
8.2	Future work	146

List of Figures

1.1	The Main Works in this Thesis	8
1.2	Thesis Outline	14
2.1	The Architecture of Multi-core CPU	31
2.2	The Architecture of SMX	32
2.3	The Memory Hierarchy	32
2.4	Dynamic Parallelism	33
2.5	Cuckoo Hashing	33
2.6	The Architecture of Cluster	34
2.7	The Architecture of Cloud	35
3.1	Execution Model in Fermi and Kepler	42
3.2	GPUDV with Dynamic Parallelism	43
3.3	Component LTS Encoding	44
3.4	Collaborative Synchronization	49
3.5	VariablesUpdating	50
3.6	Conflict Release R/W	50
3.7	The Manual System Model for Evaluation	54
4.1	Execution Process in One Iteration	65
4.2	Blocking Cycle	65
4.3	Overall Process	68
4.4	TarjanNode Synchronization	72
4.5	SuspendedMap Synchronization	72
4.6	Block and Unblock GSearch Synchronization	72
4.7	Parallel On-the-fly Fairness Checking	77
5.1	Overall Design	86
5.2	Dynamic Three-level Queue Management	92
5.3	Structure of WarpQueue, GQueue and VGQueue	94
5.4	Dynamic Hierarchical Task Schedule-block and grid level	95
5.5	Block-level Path Recording	97
5.6	Distribute of cost in three path generation	99
6.1	Example of MDP	105
6.2	Parallelization	107
6.3	Parallelization	110

6.4	Vector Distribution	111
6.5	Performance Evaluation	119
6.6	Performance with varying nr. of blocks	120
7.1	The architecture of Qubet	130
7.2	Source code for the teleportation protocol	132
7.3	Source code for the Superdense coding protocol	135
7.4	The startup window of Qubet	138
7.5	The grammar checking in Qubet	139
7.6	The startup window of simulation in Qubet	140
7.7	The result of simulation in Qubet	141
7.8	The window of bisimilarity checking result	142

List of Tables

2.1	Some Model Checking Tools	30
3.1	Features of Test Models	55
3.2	Performance Evaluation for Reachability Verification(time in sec)	56
3.3	Performance Evaluation for Complete State Space Generation (time in sec)	56
3.4	Performance Evaluation for Systems with Different Features (time in sec)	57
4.1	Evaluation of LTL Model Checking (time in sec)	78
4.2	Testing Model for Fairness Checking	79
4.3	Evaluation of ESF&EWF Checking (time in ms)	80
4.4	Evaluation of PSF&PWF Checking (time in ms)	81
4.5	SGF Checking (time in ms)	81
5.1	Parameters in the Algorithms	98
5.2	Performance Analysis	99
6.1	Features of MDPs (SCC with $ S > 1$)	118

Chapter 1

Introduction

The four principal validation methods for complex concurrent systems include testing, simulation, deductive verification, and model checking. Model checking is an automatic approach for the verification of finite-state systems. It is entirely algorithmic [58]. The time complexity of model checking algorithm depends on the property to be checked and the size of the transition systems. The size of transition systems may be exponential to the number of concurrent components, program variables or channels, etc. This is called the state space explosion problem, which is the primary challenge in model checking. Many pieces of research work on this problem [155, 201], e.g., on-the-fly model checking [28, 74, 97], symbolic model checking [61, 65, 91, 157], partial order reduction [22, 49, 76] and symmetry reduction [32, 52, 62]. However, the majority of research fights for the problem with sequential algorithms, their performance is also restricted to the size of state space. Since the algorithms for model checking problems can be regarded as some graph exploration problems, there are some research that works on the parallelization of model checking algorithms [16, 17, 54]. These parallelization works have been successfully implemented into many existing model checking tools. However, there are still many more algorithms in model checking that own high potential to be optimized with the parallel computing techniques. In addition, with the development of IT industry, many rising concurrent systems have been developed. It brings new requirements to the tool support. Therefore, both algorithm-level research and application-level development are expected to be challenging and attractive.

1.1 Motivations and Goals

Model checking is a fully algorithmic technique, which has attracted research persistently in the last two decades. A complete process of model checking consists of system modeling, requirement specification and verification. We study the pieces of literature that work on model checking techniques. The research of system modeling concentrates on the development of formal modeling languages and notations, e.g., [12, 33, 75, 96, 115, 126, 156, 166, 188, 212, 215]. The research of requirement specification focuses on the general expression of liveness and safety properties, e.g., [1, 21, 45, 82, 90, 153, 175, 192]. The research of verification, which is the core research in model checking, focuses on two lines. One of them is to develop new algorithms for the verification of new systems or properties, e.g., [13, 39, 46, 80, 87, 171, 181, 193]. The other one is the optimization of existing algorithms to deal with the state space explosion problem, as well as to improve the performance of existing model checking algorithms. e.g., [15–17, 48, 54, 86, 102, 111, 127, 191]. In these works, the state space explosion and the efficiency of algorithms are two major concerns which limit the practical usage of model checking techniques in IT industry. In addition to the research on algorithms, the development of effective model checking tools is another focus of model checking community. It requires researchers to follow the updates of algorithm-level research all the time, which indicates that the development of tools are strongly connected to the algorithm development and optimization. Therefore, our research is inspired by the requirements from both of these two aspects. In particular, according to our investigation, we propose the problems we solve in this thesis below.

1.1.1 Algorithm-level Parallelization: the Optimization of Model Checking Algorithms

Based on our investigation, LTL model checking [11, 59, 84, 105, 107, 110, 173, 176, 210, 216] and probabilistic model checking [5, 94, 95, 106, 116, 179, 180] can cover the verification of most concurrent systems. Checking whether a given system model M satisfies an LTL property is equivalent to the non-emptiness checking of the product

between M and the Büchi Automaton $B_{\neg\phi}$ [142], where Büchi Automaton expresses the negation of the property. This process can also be regarded as the combination of many graph exploration processes. The other problems, such as the reachability verification, the security verification based on the state space, etc., can also be regarded as the graph exploration problems. This feature of LTL model checking indicates that we can potentially utilize the optimization for graph related algorithms to optimize model checking algorithms. For probabilistic model checking, Markov Decision Process (MDP) [130] is an important model to capture systems with probabilistic behaviors. The numerical computation for probabilistic model checking on MDPs always refers to the matrix operation, e.g., matrix-vector multiply in the value iteration problem in MDP, which also has many related optimization research.

On the other hand, model checking suffers from the infamous *state space explosion* problem, which means the number of states may be exponential to the number of concurrent components, program variables or channels. On-the-fly verification [210, 211] is to verify whether the property is being satisfied during the product to generate the state space. It can avoid generating the complete state space. Thus, it can solve the state space explosion problem. Both partial order reduction technique [48, 54, 78, 108, 127, 193] and symbolic representation [112, 195–197] target on the reduction of state space that needs to be explored. Partial order reduction exploits the commutativity of concurrently executed transitions [60], which result in the same state. Thus, this reduction technique is best suited for asynchronous systems¹. There are also some other techniques to deal with the state space explosion such as symmetry reduction [174]. They can handle the problem, but the performance of verification is still restricted to the size of state space.

Parallel and distributed computing are widely used in accelerating the computation in many areas. Transferring the sequential algorithm to parallel, distributed or even cloud-based version can sometimes optimize its performance significantly. There exist many pieces of research on parallelizing the graph exploration-based algorithm and the matrix operations, e.g., [55, 70, 79, 147, 150, 189, 190]. There also exist many pieces of

¹In synchronous systems, concurrent transitions are executed simultaneously rather than being interleaved.

research that utilize parallel and distributed computing to deal with the model checking problem, e.g., parallel nested depth-first search (NDFS) algorithms [7, 168, 170], external memory model checking with parallel delayed duplicate detection [98], accepting cycle detection algorithms MAP [123] and OWCTY [81], parallel BFS-based LTL model checking [15] etc. All these works have achieved great speedup. In parallel and distributed environments, e.g., Multi-processors, GPU, Cluster and Cloud, we can utilize the high computation capability and the potential large memory size to support a larger size of state space, and improve the performance of graph exploration or matrix operation at the same time. Therefore, we choose this topic as our algorithm-level research. Our target is to find the algorithms in LTL model checking and probabilistic model checking that can be optimized by parallelization. We take our research in different platforms based on the specific requirements of different problems.

Based on our survey, to achieve our target on algorithm-level research, we face several technical challenges as shown below:

1. The construction of the efficient data structure for system models to save memory cost and to achieve efficient parallel accessing.
2. The development of efficient execution models with high computation locality. For property verification, it is necessary to verify the correctness of the parallel and distributed property verification process.
3. Implementations should follow the up-to-date features of different platforms.
4. It should be able to integrate the existing solutions for the state space explosion problem, e.g., on-the-fly verification, into the parallelized model checking algorithms.

The contents mentioned above are the general challenges in both our works and other research on the parallelization of model checking algorithms. Since we specifically focus on problems related to LTL model checking and probabilistic model checking, we conduct the investigation and summarize the detailed problems as follows:

LTL model checking: the topic LTL model checking algorithms contains following steps: space exploration, reachability checking or SCC detection, counterexample

generation. Besides, fairness checking is always conducted together with the LTL model checking [208]. To optimize the algorithms in model checking via parallel computing, the first step is to choose the optimized traditional sequential model checking algorithms which are the mainstream in model checking community. Since there exist some works on this topic, e.g., [19, 67, 148], how to compete with them is the primary challenge.

1. For the LTL model checking on concurrent systems, the state space exploration is a Breadth-First Search (BFS)-based or Depth-First Search (DFS)-based process. There are existing research works on parallelizing the exploration process. However, to overcome the state space explosion problem, the exploration process of LTL model checking is always based on the on-the-fly generated state space, which is a process of parallel composition among all components and the Büchi automata. This process is a BFS-based process. Thus, the parallelization based on on-the-fly generated state space is expected. There exists work [19] that focuses on utilizing GPUs to construct the state space on-the-fly. However, it does not support system models with global variables. The computation model in this work is based on a CPU-GPU collaboration, which does not use the latest feature of GPU thus is not suitable to our third requirement mentioned above. Besides, the reachability checking can be easily included in the on-the-fly state space exploration process. There is no work at that time which involves this topic.
2. SCC detection is a key subroutine in LTL model checking. Tarjan's algorithm [164] and Nested Depth-First Search (NDFS) [73] are the two main approaches. Fairness checking is necessary for modeling the concurrent systems faithfully [21]. Based on our survey, fairness assumption can be expressed in the form of an LTL property and we can conduct fairness checking together with the SCC detection process. Thus, we should firstly choose from either Tarjan's algorithm or NDFS based on which one can better handle this problem. Then Tarjan's algorithm is the basis for parallelization. There are many sequential implementations of SCC detection for automata-based LTL model checking with fairness assumptions. However, besides the state space explosion problem [63], the sequential implementations of automata-based LTL model checking with

fairness do not scale well for systems with a large number of SCCs. Also, existing concurrent Tarjan’s algorithm is based on the complete state space, which is not suitable to on-the-fly verification.

3. We conduct the counterexample generation after the above processes. A counterexample is an execution path that violates the properties. To generate such a path, some algorithms like [143, 154] work on DFS-related solution with high complexity currently. Some works on the BFS-related solution, such as [207], which focus on building the minimal-size counterexample to deal with the memory constraint. There does not exist a work to utilize GPU to accelerate the BFS-based counterexample generation, which has potential to improve the performance.

Probabilistic model checking: Markov decision processes (MDPs) are often used in probabilistic model checking as the semantic model to describe both stochastic and uncertain (nondeterministic) behavior concisely. The computation of reachability probabilities [9] in MDPs is an important routine to find the optimal policy of MDPs and its variants. Value iteration [38] is a well-known technique, which is costly and thus often a bottleneck. In the literature, there are two major approaches to improve the efficiency of value iteration method. The first approach is utilizing reachability information and heuristic functions to avoid the storage of the complete state space of MDPs in value iteration. The second approach is to reorganize the structure of an MDP to simplify the problem or to optimize value iteration based on the graphical feature of MDP. However, to the best of our knowledge, there does not exist a complete work to utilize the parallel computing to optimize value iteration, which may be more efficient and general without simplifying any information in value iteration process. Besides, the performance may not be so sensitive to the structure of MDPs.

1.1.2 Application-level Parallelization: Tool Support

Model checking is an automatic verification technique. The development of tools is a necessary work in the model checking community. We studied a lot of algorithm-level

research and found that almost all novel algorithms have been integrated to existing or new tools, such as a multi-core extension of the model checker SPIN [72], the LTSmin tool [7] with a multi-core version of the NDFS, a dedicated multi-core branch of parallel model checker DiVinE [99, 100], and a GPU-Prism which is a many-core extension of PRISM [51]. It not only improves the applicability of these algorithms, but also makes the corresponding works more convincing and announced. Thus, the development of an application is also necessary to us.

However, we are not satisfied with just extending existing model checking tools such as PAT [109], which supports the verification of common concurrent systems. We want to look into novel applications of model checking. Therefore, we aim at the quantum communication systems [131], which are the novel concurrent systems. Quantum communication systems involve a lot of quantum information theories, which are much different to our classical information theories. Thus, an application to this kind of systems consists of many challenges as following [218]:

1. The development of a formal language to describe quantum communication systems exactly, including both the syntax and semantics of the language.
2. Quantum communication systems are also concurrent systems. Thus, both the classical model checking techniques and the quantum specific verification algorithm (quantum bisimilarity checking) should be supported.

To deal with the challenges and problems mentioned in Section 1.1.1 and 1.1.2, the main works in our thesis are shown in Figure 1.1. We present five works in the following:

GPU-accelerated On-the-fly Reachability Checking We start our research to deal with the LTL model checking related problems. The first step in LTL model checking is state space generation. Thus, we propose an approach, named GPURC to utilize Graphics Processing Units (GPUs) to speed up the reachability verification. The key idea is to achieve a dynamic load balancing so that we can fully utilize the many cores in GPUs during the state space exploration. To this end, we firstly construct a compact data encoding of the input transition systems to reduce the memory cost and fit the

FIGURE 1.1: The Main Works in this Thesis

calculation in GPUs. To support a large number of concurrent components, we propose a multi-integer encoding with a conflict-release accessing approach. We then develop a BFS-based state space generation algorithm in GPUs, which makes full use of the GPU memory hierarchy and the latest dynamic parallelism feature in CUDA to achieve a high parallelism. GPURC also supports a parallel collaborative event synchronization approach and integrates a GPU hashing method to reduce the cost of data accessing. The experiments show that GPURC can give a significant performance speedup (average 50x and up to 100x) compared with the traditional sequential algorithms.

Concurrent On-the-fly SCC Detection and Fairness Checking After the state space generation, another important step is the detection of strongly connected component (SCC) and fairness checking, which is the key point to verify whether the model can satisfy the given properties. Tarjan's algorithm for detecting SCCs is a widely used depth-first search procedure for automata-based LTL model checking. It works on the SCC detection on-the-fly with the composition of transition systems and Büchi Automaton (state space generation), which has been deployed as sequential implementations in many tools. However, these implementations suffer from large time cost for systems which involve a significant number of SCC explorations. To

address this issue, we develop a concurrent SCC detection approach for the on-the-fly generated state space in LTL model checking by expanding the existing concurrent Tarjan’s algorithm. Besides, we extend our approach to support fairness checking. The SCC detection is a DFS process. Thus, the parallelized process contains a lot of interaction communication. The high parallelism is not required and also not available for this condition. Thus, we choose the multi-core platform instead of GPU platform for this work. Different from the previous work in [208], which performs fairness checking after the generation of a complete SCC, in our approach we perform fairness checking during the SCC generation to improve efficiency. We firstly implement this approach in PAT model checker. Our experimental results show that our approach achieves up to 2x speedup for the complete SCC detection in large-scale system models compared to the sequential on-the-fly model checking in PAT. Besides, our parallel on-the-fly fairness checking approach speeds up fairness checking around 2x~45x.

GPU-accelerated Counterexample Generation In model checking, if the verification result is negative, most model checkers will generate a counterexample. As we mentioned in the previous paragraph, SCC-based searching is one of the main stream LTL model checking algorithms. When an SCC is huge, the counterexample generation process can be time-consuming, especially when dealing with fairness assumptions. In this work, we propose a GPU accelerated counterexample generation algorithm, which improves the performance by parallelizing the Breadth First Search (BFS) used in the counterexample generation. BFS work is irregular, which means it is hard to allocate resources and may suffer from the imbalanced load. To handle this problem, we make use of features in the latest CUDA compute architecture-*NVIDIA Kepler GK110* to achieve dynamic parallelism and memory hierarchy. We develop a dynamic queue management scheme, a task scheduler and a parallel path recording scheme such that the counterexample generation process can be completely finished in GPU without involving CPU. We have conducted the evaluation of the proposed approach via PAT model checker. Our experiments show that our approach is efficient and scalable.

GPU-accelerated Value Iteration After considering the problems in LTL model checking. We studied the speed up of probabilistic model checking algorithms. We focus on the numerical calculation (Value Iteration) in the conventional probabilistic

model-Markov Decision Processes (MDPs). Value iteration (VI) [38] is a well-known computation method to deal with the computation of reachability probabilities, which is an important subroutine for determining approximately optimal policies of MDPs. However, a single threaded sequential implementation of this method is computationally expensive in terms of time and memory. Hence, the development of novel techniques to improve the computational efficiency of value iteration method is a key research challenge. To this end, we propose a highly parallel version of value iteration to solve general MDPs utilizing the graphics processing unit. Our approach explores the algebraic features (e.g., matrix structure) of MDPs, and uses action-based matrices to achieve massive parallelism for efficient value iteration. We empirically evaluate our approach on several case studies, and compare it with the sequential and topological value iteration implementations. Our results and analysis show that we outperform previous approaches [148] in many cases and support a wider range of MDPs than these specific methods.

A Tool for the Model Checking and Bisimilarity Checking of Quantum Communication Systems Application development is an important work in model checking community, which is strongly connected to the algorithms for model checking problems. In the previous algorithm-level research, we have tried to integrate our algorithms into the existing PAT model checker. We target on an entirely new requirement- the verification of quantum communication systems, which is a rising concurrent system. Firstly, we work on the tool construction. Designing quantum programs and protocols is notoriously difficult. QCCS [213], amongst other quantum process algebras, has been proposed for formal description and verification of quantum communicating systems. In this part, we propose Qubet, an analysis tool to support the model checking, emulation and automatic check of bisimilarity between quantum processes described in qCCS. In addition, since our implementation of QCCS is oriented from CSP. The verification of traditional CSP model is also supported. We present several verification samples of quantum communication systems to prove the correctness of Qubet. Secondly, we integrate our previous algorithm-level research into Qubet, which helps Qubet to gain the expected performance improvement like our algorithm-level research benefits. This tool is useful to the designer of quantum

communication systems. The more important thing is that our application-level research and algorithm-level research are strongly connected.

To sum up, two lines of work are included in this thesis: algorithm-level works and application-level works. In algorithm-level works, we conduct the research on the parallelization of algorithms in LTL model checking and probabilistic model checking. In application-level works, we develop a novel application and integrate our algorithm-level works. The experiments of algorithm-level works prove the performance improvement. The application we develop is significant to the designers of quantum communication systems. It is also an important complement to our algorithm-level research.

1.2 Summary of Contributions

The primary achievements in this thesis consist of both algorithm-level works and application-level works. We summarize our contributions as follows:

1. **GPURC: GPU-accelerated On-the-fly Reachability Checking:** as the first step of LTL model checking, we propose a GPU-accelerated state space generation process.

GPURC involves the redesign of the data structure to store the concurrent system which is described with a number of concurrent LTS. Our data structure is a compact encoding, which also supports the encoding of global variables and event synchronization. For systems with a large size state space, we can support by using a multi-integer encoding.

Our on-the-fly state space generation is a parallel GPU-based BFS process. Our design divides the tasks of successor generation to small parallel units called workgroup. For reachability checking, our searching algorithm works on GPU with dynamic load balancing without the CPU involvement. Our approach incorporates an efficient hierarchical hash structure to store the state space and uses parallel state space generation to achieve event synchronization. A conflict-release accessing model is used to support the multi-integer encoding. The

evaluation indicates that our approach can achieve up to around 100x speedup on benchmark examples compared with the sequential BFS-based and DFS-based algorithm. When exploring the complete state space is necessary, our approach can yield up to 8x speedup.

2. **Concurrent On-the-fly SCC Detection for Automata-based Model Checking with Fairness Assumption:** the second step of LTL model checking is SCC detection and fairness checking. Based on the fact that this process is a DFS-based process with a low locality, we propose a concurrent version in a multi-core CPU environment. This work expands Lowe's Tarjan's algorithm, which is based on the complete state space, to fit the on-the-fly generated state space. Based on it, we build the parallel approach to cover the features of on-the-fly SCC-based LTL model checking.

The data distribution rules for the on-the-fly generated state space is proposed. We design and develop an efficient on-the-fly parallel fairness checking approach, which performs the fairness checking during the generation of SCC instead of performing it separately after the generation like [208] does. Our implementation of the approach is firstly conducted in the Process Analysis Toolkit (PAT) and it can work on a broad range of system models. Our evaluation shows that after integrating our approach, it can achieve a 2x performance improvement in LTL model checking which involves the exploration of a vast number of SCCs. And 2x-45x speedup for fairness checking.

3. **GPU Accelerated Counterexample Generation in LTL Model Checking:** this work involves the construction of the GPU-based counterexample generation, which is the last step after the SCC being detected. We utilize the latest Kepler GPU architecture at that time to build a CPU-free GPU program. Existing related works allocate the GPU resources in a static way. The resources can be reallocated only by the CPU when the execution of a kernel ends and launches a new kernel. For irregular graphs, it is costly and not flexible. Our approach presents a runtime resource adjustment approach for BFS and can be tailored for model checking problems.

Following the parallelization rule, we propose the corresponding way to construct dynamic parent-child relationship and do the dynamic hierarchical task scheduler with dynamic load balancing. For the data structure, we propose a three-level queue management to fit the dynamic parallelism and dynamic BFS layer expanding. Based on it, we propose a dynamic path recording approach, which helps with the duplicate elimination in BFS at the same time. Hierarchical memory structure of GPU is fully utilized for data accessing. The experiments prove the performance and scalability of our approach.

4. **A GPU-based Value Iteration Algorithm to Accelerate the Computation of Reachability Probabilities in MDPs:** besides the LTL model checking related problems, we propose a GPU-based value iteration, which is an important numerical computation method for probabilistic model checking on MDPs. Compared with previous TVI which based on graphical feature (SCC) of MDPs, we take advantage of the algebraic structure of MDPs to define action-based matrices and corresponding data structures for efficient parallel computation of reachability probabilities on GPUs.

Our design of efficient parallel value iteration algorithm for computing reachability probabilities fully utilizes the features of modern GPUs, such as dynamic parallelism and memory hierarchy. The experiments prove the advantage of our approach. We also present an extension of our approach to find the approximately optimal policy for standard MDPs and its variants. We discuss a potential extension to POMDPs. Our evaluation shows that our approach gains around 10x speedup compared to the sequential version of VI, and around 1.5x-17x speedup compared to the topological value iteration. We also discuss that we support a wider range of MDPs than the current topological value iteration.

5. **Qubet: A Model Checking, Bisimulation Checking and Emulation Tool for Quantum Communication Systems:** This work proposes the development of a new tool Qubet, which is specific for the rising concurrent system-quantum communication system. Our tool is the first one to cover the model checking, bisimulation checking and emulation of quantum communication systems. This

FIGURE 1.2: Thesis Outline

is a meaningful application-level research besides the previous algorithm-level works.

Our work involves the development of the editor to recognize and edit the system model written with quantum formal language QCCs, which strictly follows the expression in quantum information theories. We also support the classical CSP. Several sample quantum communication protocols are integrated. Besides, we integrate our algorithm-level works into Qubet so as to improve their applicability and to extend our application at the same time. We make our algorithm-level works and application-level works strongly connected.

1.3 Thesis Outline

We present the outline of our research work in Figure 1.2. Each chapter presents the solution of one problem mentioned in Section 1.1. This roadmap also reveals the progress of our research work from algorithm-level to application-level. The remaining of this thesis is organized as below:

Chapter 2 presents the background and preliminaries of model checking, including the basics of model checking, specification, verification, LTL model checking, part of probabilistic model checking and some popular model checking tools. We also present the preliminaries of parallel computing platforms.

In Chapter 3, we present our work on the GPU-accelerated on-the-fly reachability checking, in which the core problem is a GPU-accelerated state space generation in model checking. We present our design of the compact data structure to store the state space, and the design to make full use of the GPU memory hierarchy and the latest dynamic parallelism feature in CUDA to achieve a high parallelism. We also present a parallel collaborative event synchronization approach and integrate a GPU hashing method to reduce the cost of data accessing.

In Chapter 4, we build a concurrent SCC detection approach for the on-the-fly generated state space in LTL model checking, which is based on the extension of a concurrent Tarjan's algorithm. We also develop the on-the-fly fairness checking. This work is conducted on the multi-core CPU platform. We integrate it into PAT model checker for testing.

Chapter 5 presents our work on the parallelization of counterexample generation in LTL model checking. In this chapter, we present how we utilize the latest features of GPU architecture to work out a GPU-based counterexample generation.

Chapter 6 focuses on the problem related to probabilistic model checking. We construct a GPU-accelerated value iteration approach for the computation of reachability probabilities. In this chapter, we describe the features in value iteration algorithm that we utilize to do parallelization. We also build a compact data structure and utilize the optimization techniques in GPU to improve the efficiency of data access.

In Chapter 7, we construct an application Qubet, which is the first tool to cover the automatic verification (including model checking and bisimilarity checking) of quantum communication systems. Qubet is our application-level research. Besides the design of Qubet, we also present how we integrate our previous algorithm-level research in Chapter 3, 4, 5 and 6 into our application.

Chapter 8 concludes our research in this thesis, and discuss the potential future research directions.

1.4 Publication List

Most of our works presented in this thesis are published in the international conferences.

The list of publications for the Ph.D. candidate are shown below:

1. **Zhimin Wu**, Yang Liu, Yun Liang and Jun Sun, GPU Accelerated Counterexample Generation in LTL Model Checking. 16th International Conference on Formal Engineering Methods (ICFEM 2014), Springer International Publishing, pp. 413-429, November, Luxembourg, 2014.
2. **Zhimin Wu**, Yang Liu, Jun Sun, Jianqi Shi and Shengchao Qin, GPU Accelerated On-the-Fly Reachability Checking. 20th International Conference on Engineering of Complex Computer Systems (ICECCS 2015), IEEE, pp. 100-109. December, Gold Coast, Australia, 2015.
3. **Zhimin Wu**, Parallel and Distributed Algorithms for Model Checking Problems (Doctoral Consortium). 20th International Conference on Engineering of Complex Computer Systems (ICECCS 2015), IEEE, pp. 210-213, December, Gold Coast, Australia, 2015.
4. **Zhimin Wu**, Ernst Moritz Hahn, Akin Gunay, Lijun Zhang and Yang Liu, GPU-accelerated Value Iteration for the Computation of Reachability Probabilities in MDPs. 22th European Conference on Artificial Intelligence (ECAI 2016), IOS press, pp. 1726-1727, August, The Hague, Holland, 2016.
5. **Zhimin Wu**, Akin Gunay, Yang Liu and Shengchao Qin, Concurrent On-the-fly SCC Detection for Automata-based Model Checking with Fairness Assumption. 21th International Conference on Engineering of Complex Computer Systems (ICECCS 2016), IEEE, pp. 135-144, November, Dubai, UAE, 2016.

6. **Zhimin Wu**, Yuan Feng, Fuyuan Zhang, Yang Liu and Mingsheng Ying. Qubet: A Bisimulation Checking and Emulation Tool for Quantum Communication Systems. (Under Submission)

Chapter 2

Background

This chapter presents the background of model checking and the preliminaries of parallel computing.

2.1 Model Checking

Model checking [59] is an automatic technique for the verification of finite-state systems. Given a system model, model checking verifies whether this system fits any specification. Model checking suffers from the infamous state space explosion problem. The size of state space in concurrent systems would be enormous, which exponentially increases via the increment of the interactive components or data values. It is tough to handle the vast state space with limited memory and computing resources. The state space explosion problem has attracted considerable research works in the past decades.

A complete process of model checking is made up of several tasks: (1) the formalization of the system design, which should be recognized by model checking tools. (2) The abstraction of the system requirement, which is expressed as logic specifications, e.g., temporal logic, which can describe the assertion on the behavior evolvement of the system over time. The verification process is always conducted automatically, which justifies whether a system model meets the specification. If the verification result is negative, a witness trace or a counterexample should be

generated, which can help the user with the analysis of the system design and conduct the modifications to the model. The model checking process can be repeated. We explain the details of each tasks, namely system modeling, property specification and verification, in the following sections.

2.2 System Modeling

The foundation of model checking is system modeling. The examined system should be transformed into a formalism representation in order to be recognized by a model checking tool. It is difficult and critical to do the formal system modeling since we should consider how to describe the system exactly. Sometimes, with limited time and memory, we should construct an abstraction for the system. To this end, we should concern seriously on which parts should be kept in the abstracted model. Relevant and important parts should be fully represented and unnecessary parts should be eliminated, e.g., the important points for reasoning the communication protocol are the action of messages exchanging, instead of the contents of the messages. A system model is constructed with states and transitions. State (or configuration) is used to represent the snapshot or instantaneous description of the system, which can reflect the program counter and the values of the variables at a particular instant of time [208]. Transitions describe the actions of the system which result in the state transformation, which consists of a source state, a target state and an action (or event) that links them. A calculus of a system is expressed in a trace, which consists of a finite or infinite sequence of states. Each state in this trace is generated from the previous state via some transition. We always use Kripke structure [69] to handle the formal system modeling, which is a state transition graph.

Definition 2.1. A Kripke Structure is a four-tuple structure $M = (S, I, R, L)$ where S is a finite set of states; $I \subset S$ is the set of initial states, $R \subseteq S \times S$ is the transition set and L is a labeling function: $L : S \rightarrow 2^{AP}$, where AP is a set of atomic propositions.

In a Kripke structure, all reachable states of the system are represented as the nodes in a graph, and the transitions between states are represented as edges. Each node also

consists of the properties that hold in each state. In this thesis, a transition in a Kripke structure is labelled with an event (denoted as $e \in \Sigma$) that links the source state and target state.

Nowadays, distributed and concurrent systems are becoming commonplace, which include network applications, data communication protocols, multithreaded code, and client-server applications, etc. However, they are notoriously difficult to develop. The common flaws in developing concurrent systems consist of under-specification¹, over-specification², the violations of safety properties, and the violations of liveness properties. In order to deal with these problems, we can model and analyze the system design mechanically. Model checking is commonly utilized in this area.

There are many specification languages to handle the system modeling of concurrent systems. In this thesis, we focus on high level languages like CSP (Communicating Sequential Processes) [33] and CCS (Calculus of Communicating Systems) [165], which use mathematical objects as abstractions to represent systems or processes [109]. The behaviors of concurrent systems are expressed in process expressions, which integrate a lot of hierarchical operators, e.g., parallel composition, deterministic or non-deterministic choices. These operators are correlated to the elegant algebraic laws for system analysis.

Many model checking tools mentioned in Table 2.1 have supported CSP and CCS. In this thesis, we do not concern about the model checking on traditional concurrent systems. With the development of concurrent systems, new requirements come out. Quantum communication systems, which harness modern physics through state-of-the-art optical engineering to provide revolutionary capabilities [194], have attracted researchers for a long time. General CSP or CCS are not capable of describing quantum communication systems. We will introduce a new system modeling language in Chapter 7.

¹The system design is incomplete, imprecise or allows behavior that should not be allowed.

²The system design disallows behavior that should be allowed, that is, model is too restrictive.

2.3 Property Specification and Verification

With a system model, the specifications are required to describe the properties that the system design must satisfy [208]. We use temporal logics to state the properties, which can specify how the system evolves over time. There are two kinds of property: safety and liveness properties.

2.3.1 Safety properties

A safety property states that something bad never happens, which indicates that there is no deadlocks or states that can result in the crash of system. For sequential programs without endless loops, there exists a terminal state with no outgoing transitions. However, the computation of concurrent systems always do not terminate, which indicates that there are system errors if any terminal state is detected. Such errors would probably result in deadlocks. To define whether a deadlock occurs, we always check the system status in a global view, which means that the complete system terminates, e.g., each component is waiting for other components to progress. Generally, we state safety property with a logic formula of atomic propositions, which indicates the safety property for mutual exclusion problems. For verification of safety properties, we always conduct the reachability checking, which is based on the depth-first search (DFS) [164] or breadth-first search (BFS) of the state space. If we detect a undesirable state, we can conclude that the safety properties cannot be satisfied. A counterexample would be generated. Our work in Chapter 3 is based on this. We also note that deadlock-freedom is sometimes treated as a liveness property other than a safety property in the event-oriented world [30]. But in this thesis, we consider it as a safety property.

2.3.2 Liveness Properties and Fairness Assumptions

Liveness properties state that something good will eventually happen. It describes the requirement that a process makes progress toward a specific goal. Fairness and Liveness are related notions. Both of them describe which of a collection of concurrent processes

make progress in their execution. Fairness is not a property to be checked. But the achievement of the liveness property depends on the fairness of the system. If a process would never be executed, it definitely cannot reach its goal. Thus, the verification of liveness properties is often based on fair paths.

We present different types of fairness assumptions by presenting some basic definitions in advance. Given an LTS $M = (S, s_0, \rightarrow)$ with event set Σ we first provide the following definitions of events and processes, where α is a typical element of Σ : (1) $enabledEvt(s)$ is the set of enabled events at $s \in S$ i.e., defined as $\{\alpha \mid (s, \alpha, s') \in \rightarrow\}$. (2) $enabledProc(s)$ is the set of enabled processes at s . That is, if a process $p \in enabledProc(s)$, it can progress in the system state s . (3) $engagedEvt(s, \alpha, s')$ indicates the engaged event α (4) $engagedProc(s, \alpha, s')$ indicates the set of processes which progress via the transition $(s, \alpha, s') \in \rightarrow$.

Given an execution R_i^j ,

$$R_i^j = ((s_0, b_0), \alpha_0, \dots, (s_i, b_i), \alpha_i, \dots, (s_j, b_j), \alpha_j, (s_{j+1}, b_{j+1}))$$

where $s_j \in S$ and $b_j \in A_{-\varphi}$. There is an SCC in this execution which indicates $s_i = s_{j+1}, b_i = b_{j+1}$. If $(b_0, b_1 \dots b_k \dots)$ is accepting to $A_{-\varphi}$, R_i^j is accepting. With specific fairness assumptions, R_i^j is fair if and only if $(s_0, \alpha_0, s_1, \alpha_1 \dots s_k, \alpha_k \dots)$ is fair. Some definitions of the sets used in fairness checking are shown: (1) $alwaysEvt((R_i)^j) = \{\alpha \mid \forall k : \{i..j\}, \alpha \in enabledEvt(s_k)\}$ (2) $alwaysPro((R_i)^j) = \{p \mid \forall k : \{i..j\}, p \in enabledPro(s_k)\}$. (3) $onceEvt((R_i)^j) = \{\alpha \mid \exists k : \{i..j\}, \alpha \in enabledEvt(s_k)\}$. (4) $oncePro((R_i)^j) = \{p \mid \exists k : \{i..j\}, p \in enabledPro(s_k)\}$.

Here, set $alwaysEvt((R_i)^j)$ and set $alwaysPro((R_i)^j)$ are the set of events and processes which are always enabled during the loop. Set $onceEvt((R_i)^j)$ and set $oncePro((R_i)^j)$ are the set of events and processes which are enabled at least once during the loop [208].

We consider five categories of fairness assumptions: (1) *Strong Global Fairness (SGF)*: An execution satisfies SGF if and only if for all $(s, \alpha, s') \in \rightarrow$, if $s = s_i$ for infinitely many i , $s_i = s$ and $\alpha_i = \alpha$ and $s_{i+1} = s'$ for infinitely many i . (2) *Event-level Strong Fairness (ESF)*: An execution R satisfies ESF if and only if for all events

α , if α is infinitely often enabled, then $\alpha_i = \alpha$ for infinitely many i . (3) *Event-level Weak Fairness (EWF)* [119]: An execution R satisfies EWF if and only if for all α , if α finally becomes enabled forever in R , then $\alpha_i = \alpha$ for infinitely many i . (4) *Process-level Strong Fairness (PSF)*: An execution satisfies PSF if and only if for all processes p , if p is infinitely often enabled, then $p \in \text{engagedProc}(s_i, \alpha_i, s_{i+1})$ for infinitely many i . (5) *Process-level Weak Fairness (PWF)*: An execution E satisfies PWF if and only if for every processes p , if p eventually becomes enabled forever in E , then $p \in \text{engagedProc}(s_i, \alpha_i, s_{i+1})$ for infinitely many i .

Normally, we use temporal logics to describe the liveness properties, e.g., Computation Tree Logic (CTL) [57] and Linear Temporal Logic (LTL) [2]. In this thesis, we focus on LTL and the LTL model checking, which will be detailed in the following section.

2.3.3 LTL Model Checking with Fairness Assumption

The Linear Temporal Logic (LTL) is a modal temporal logic with modalities referring to time [205]. LTL formulae can describe the properties that indicate the future of paths, e.g., safety properties or liveness properties indicate that a condition will eventually happen.

Definition 2.2. Given a set of propositions AP and a set of events Σ , an LTL formula is,

$$\phi ::= p | \alpha | \neg\phi | \phi \wedge \psi | X\phi | G\phi | F\phi | \phi \cup \psi \text{ where } p \in AP \text{ and } \alpha \in \Sigma.$$

In LTL, X means next, $\cup$ means until, e.g., ϕ holds until ψ , G means always, F means eventually. $\neg$ and $\wedge$ are two logical operators.

LTL model checking is the model checking of a property expressed as an LTL formula. For explicit model checking, the Kripke structure is represented as a graph. The system model is transferred into an automaton M . In this thesis, we always use the *labelled transition system* (Definition 2.3) to express the automaton.

Definition 2.3. Given a set of invisible and visible events Σ , a *labelled transition system* (LTS) is a 3-tuple $M = (S, s_0, \rightarrow)$ where S is a set of states, $s_0 \in S$ is the initial state, $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation.

A transition system is *finite* if S , $\rightarrow$, and AP are finite. For convenience, we write $s \xrightarrow{\alpha} s'$ instead of $(s, \alpha, s') \in \rightarrow$ where $s, s' \in S$ and $\alpha \in \sigma$. A concurrent system may consist of multiple components running in parallel, each of which is an LTS component. The behavior of the composed system can be represented by the parallel composition of all component transition systems.

Definition 2.4. Given two LTSs $M_i = (S_i, Act_i, s_0^i, \rightarrow_i, AP_i, L_i)$ for $i \in \{1, 2\}$, the *parallel composition* of them is an LTS: $M_1 \parallel M_2 = (S_1 \times S_2, Act_1 \cup Act_2, (s_0^1, s_0^2), \rightarrow_1 \cup \rightarrow_2, AP_1 \cup AP_2, L)$ such that $L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$ where $s_1 \in S_1$, $s_2 \in S_2$. The transition relation $\rightarrow$ is the smallest transition relation which satisfies the following:

$$\left\{ \begin{array}{ll} (s_1, s_2) \xrightarrow{\alpha} (s'_1, s'_2) & \text{if } s_1 \xrightarrow{\alpha} s'_1 \wedge s_2 \xrightarrow{\alpha} s'_2 \\ (s_1, s_2) \xrightarrow{\alpha} (s'_1, s_2) & \text{if } s_1 \xrightarrow{\alpha} s'_1 \wedge \alpha \notin Act_2 \\ (s_1, s_2) \xrightarrow{\alpha} (s_1, s'_2) & \text{if } s_2 \xrightarrow{\alpha} s'_2 \wedge \alpha \notin Act_1 \end{array} \right.$$

The process of generating the global state space is to compute the parallel composition of all the components.

With the global state space, then given the property expressed in LTL specification φ , we always use the negation of φ to conduct the verification. $\neg\varphi$ is expressed in an equivalent Büchi automaton $B_{\neg\varphi}$ (Definition 2.5).

Definition 2.5. A Büchi Automaton is a tuple $A = (\Sigma_B, S_B, \rho, b_0, F)$, where Σ is an alphabet, S_B is a set of states, $\rho : S_B \times \Sigma \rightarrow S_B$ is a nondeterministic transition function, $b_0 \in S_B$ is an initial state, and $F \subseteq S_B$ is a set of accepting states.

LTL model checking is to verify $M \models \varphi$, which is the emptiness checking of the product (Definition 2.6) between M and $B_{\neg\varphi}$. If the product is empty, it generates a counterexample, which indicates that the model does not fit the property. The detection of strong connected components (SCC) (Definition 2.7) is a key subroutine in the emptiness checking process. SCC detection is originally based on the complete state space. Therefore, LTL model checking also suffers from the state space explosion problem. However, a clever technique, on-the-fly verification, can efficiently handle this problem. In many cases, we can avoid generating the complete state space of the

model since it checks the emptiness while constructing the product state space. Only part of the state space is generated before the verification process gets the result.

Definition 2.6. Given an LTS representing the system model $M = (S, s_0, \rightarrow_M)$ and a set of alphabets (events) Σ , a Büchi Automaton $B_{\neg\varphi}$, the *composition* of M and $B_{\neg\varphi}$ is a transition system: $M \times B_{\neg\varphi} = (S \times B, (s_0, b_0), \rightarrow)$. The corresponding set of events is $\Sigma \cup \Sigma_B$. The transition relation after the parallel composition is the smallest transition relation which satisfies the following:

1. $((s_1, b_1), \alpha, (s'_1, b'_1)) \in \rightarrow$ if $(s_1, \alpha, s'_1) \in \rightarrow \wedge \dots \wedge (s, \alpha, b' \in \rightarrow)$.
2. $((s_1, b_1), \alpha, (s'_1, b_1)) \in \rightarrow$ if $(s_1, \alpha, s'_1) \in \rightarrow_M \wedge \alpha \notin \Sigma_B$
3. $((s_1, b_1), \alpha, (s_1, b'_1)) \in \rightarrow$ if $(b_1, \alpha, b'_1) \in \rightarrow_{\neg\varphi} \wedge \alpha \notin \Sigma$.

Definition 2.7. Given a directed graph $G = (V, E)$, let $s \rightarrow_G^* t$ mean that there is a path from node s to t in G . A strongly connected component (SCC) is a maximal subgraph S of G , such that for all $s, t \in G, s \rightarrow_G^* t$ and $t \rightarrow_G^* s$. An SCC is non-trivial if it contains at least one edge.

For LTL model checking with fairness assumption, we always build the LTL formula with fairness constraints. But the size of the Büchi automaton is exponential to the scale of the LTL formula, which makes it infeasible to handle large formulas. e.g., formulas with many fairness constraints. Thus, in this thesis, our sequential SCC-based LTL model checking with fairness algorithm adopts the solution of PAT [109] and handles the fairness checking on the generated SCC as follows. Given an SCC S in the product of M and $A_{\neg\varphi}$ and fairness assumption F , if there is no such S that S is accepting and:

1. $\text{onceEvt}(S) \subseteq \text{engagedEvt}(S) \iff M \text{ satisfies } F_{ESF}$.
2. $\text{alwaysEvt}(S) \subseteq \text{engagedEvt}(S) \iff M \text{ satisfies } F_{EWF}$.
3. $\text{oncePro}(S) \subseteq \text{engagedPro}(S) \iff M \text{ satisfies } F_{PSF}$.
4. $\text{alwaysPro}(S) \subseteq \text{engagedPro}(S) \iff M \text{ satisfies } F_{PWF}$.

For all these sets mentioned above, We have introduced their definitions in Section 2.3.2.

Algorithm 1: Sequential Counterexample Generation Algorithm

Input: $init, SCC, \rightarrow$
Output: π_{ce}
 1 $\pi_{ce} \leftarrow Init2SCCBFS(init, SCC, \rightarrow);$
 2 $\pi_{ce} \leftarrow \pi_{ce} \frown Path2AccBFS(\pi_{ce}, SCC, \rightarrow);$
 3 $\pi_{ce} \leftarrow \pi_{ce} \frown SelfLoopBFS(\pi_{ce}, SCC, \rightarrow);$

2.3.3.1 Counterexample Generation

During the LTL model checking, if the verification result is negative, counterexample generation process starts to produce a trace to reflect the errors in the model.

There are many counterexample generation algorithms [143, 144, 185], mostly using BFS searching to find the shortest counterexamples. Therefore, a way to accelerate BFS, combined with counterexample generation requirement, can work for these solutions. In this thesis, we choose the counterexample generation algorithm (Algorithm 1) as the basis for our design. There are three inputs, $init$ is an initial state in P ; SCC is a list that contains all nodes belonging to the SCC; $\rightarrow$ is the outgoing transition relation of each node in P . Strictly speaking, this transition relation is made up of the current explored transitions during the SCC searching process. Algorithm 1 contains three steps. (1) *Init2SCCBFS* is to find the path from $init$ to any state in the SCC using BFS. (2) *Path2AccBFS* is to find the path from the SCC state found with *Init2SCCBFS* to the nearest accepting state. (3) *SelfLoopBFS* is to find a loop that starts from the accepting state. π_{ce} is the returned counterexample run, which is the concatenation of the three paths during the process. All these three steps are BFS based. In this thesis, we will deal with accelerating these three steps in GPU with CUDA and merging them into one algorithm.

2.3.4 Probabilistic Model Checking

Probabilistic Model Checking is a formal verification technique for modeling and analyzing systems that exhibit probabilistic behavior [154], e.g., randomized algorithms, communication and security protocols, computer networks, biological signaling pathways, and many others. The formalization of a system model is the construction of an automaton with probabilistic behavior, for which we always use the

variants of Markov Chain, e.g., discrete-time Markov chains (DTMCs), continuous-time Markov Chains (CTMCs), and Markov decision processes (MDPs) [207]. The properties can also be formally expressed in probabilistic temporal logics, e.g., PCTL, CSL, LTL, PCTL* etc. The algorithms to conduct probabilistic model checking consists of four lines [128]:

- 1 graph-theoretical algorithms for reachability checking or identifying strongly connected components.
- 2 numerical computation for linear equation systems, linear optimization problems, uniformisation, and shortest path problems. It consists of iterative methods and direct methods.
- 3 automata for regular languages.
- 4 sampling-based (statistical) for approximate analysis.

The output of probabilistic model checking varies in different problems and algorithms we take, which includes True or False, Quantitative results or the counterexample when the verification result is negative. There are a lot of research points in probabilistic model checking. In this thesis, we focus on the probabilistic model expressed in MDPs and the reachability related specifications, for which we use the numerical computation. We present the details in the following section.

2.3.4.1 Markov Decision Processes and Value Iteration

Markov decision processes (MDPs) have been extensively studied in the literature for planning and decision making under uncertainty [141]. The minimal (maximal) probability to reach a set of target states (reachability probabilities) is an important specification in probabilistic model checking based on MDPs.

Definition 2.8. An *MDP* is a tuple $M = (S, s_{init}, Act, P, R)$, where S is a finite set of *states*, $s_{init} \in S$ is the *initial state*, and Act is a finite set of *actions*. The (partial) *transition probability function* $P: S \times Act \rightarrow Dist(S)$ where $Dist(S)$ is the set of discrete probability

distributions over the set S assigns probability distributions to combinations of states and actions. The *reward function* $R: S \times Act \rightarrow Dist(R)$ assigns a numeric reward to each state/action pair.

Given an probabilistic system model expressed in MDP M , The computation of reachabilities probabilities for a set of target states $T \subseteq S$ in an infinite horizon is defined formally: $P_{\min}(s, T) = \inf_{\alpha \in Adv} Prob^{\alpha}(s, T)$. Here, Adv is the set of all *schedulers*, which choose the action to be performed in a state depending on the sequence of states and actions seen so far. Here, $Prob^{\alpha}(s, T)$ is the probability of reaching T when starting from state s and following the scheduler α . Computation of reachability probabilities can be formalized as the fixed-point equation $P_{\min}(s, T) = \lim_{n \rightarrow \infty} x_s^n$. The computation process is defined by the following equation:

$$x_s^n = \min_{\alpha \in Act(s)} \sum_{s' \in S} P(s' | s, a) \cdot x_{s'}^{(n-1)} \text{ for } s \notin T, n > 0 \quad (2.1)$$

Another core problem is to find the *reward-optimal* policy π , that is to find the scheduler which maximizes the expected accumulated reward. The Bellman equation [163] defines the optimal value function to calculate the expected value of a policy π :

$$V^{\pi}(s) = \max_{a \in Act(s)} \sum_{s' \in S} P(s' | s, a) \cdot (R_a(s, s') + V(s')) \quad (2.2)$$

$$\pi(s) = \arg \max_{a \in Act(s)} \sum_{s' \in S} P(s' | s, a) \cdot (R_a(s, s') + V(s')) \quad (2.3)$$

The computation of reachability probabilities is especially important for two major MDP variants. The first variant is *MAXPROBMDPs* [10], which are constructed by assigning a reward of zero to all state-action pairs and assigning a reward of one for reaching a set of target states. The second variant is risk-sensitive MDP (*RS-MDP*) [186], which introduces a cost threshold θ . Given a policy π_i , the reachability probabilities are represented by $P^{\pi_i}(s, T, \theta)$, solving an RS-MDP means to find an optimal a policy π which maximizes the reachability probabilities while satisfying that the accumulated cost remains under θ .

Value iteration [162] is a general dynamic programming method to solve MDPs. It is an iterative numerical computation process to update the value function of every state. The process is terminated when one assumes that convergence is reached, which is usually the case once the residual between the reachability probabilities or value functions in two consecutive iterations is less than a threshold. The rule of updating follows Equation 1 for the computation of reachability probabilities and the Bellman equation for the approximately optimal policies.

Partially observable Markov decision processes (POMDPs) in Definition 2.9 are a generalization of MDPs in which the state space is only partially observable [121]. In a POMDP, the current state cannot be determined exactly. Instead, there is a probability distribution over the possible current states, which is modeled by a belief-state b where $b(s)$ denotes the probability that the current state is s . Therefore, the optimal policy for a POMDP has to take into account all actions performed so far as well as *observations* [31], which complicates its computation.

Definition 2.9. POMDP A *POMDP* is a tuple $POM = (S, b_{init}, Act, P, R, O, Z)$, where S , P , and R represent the same structures as in an MDP. O is the set of *observations*. Z is the *observation function* where $Z(a, s, o)$ is the probability that state s obtains observation o after action a was performed and the agent reaches state s . b_{init} is the initial *belief-state*.

2.4 Model Checking Tools

Model checking is an automatic verification technique. In the previous sections, we have mentioned many model checking techniques, for which there exist many efficient implementations. Based on these techniques, many research groups have developed a number of model checking tools. Different tools support different formal languages for the system modeling, different verification algorithms, and have different user-friendly GUI. We present some famous tools in classification in Table 2.1 [206].

These tools are widely used to handle the verification of system design. Many tools are even employed in top companies to deal with critical problems. e.g., the use of

TABLE 2.1: Some Model Checking Tools

Name	Model Checking			GUI		
	Plain, Probabilistic, Stochastic, ...	Modelling language	Properties language	GUI	Graphical Specification	Counterexample visualization
BANDERA [172]	Code analysis	JAVA	CTL,LTL	YES	YES	YES
BLAST [200]	Code analysis	C	Monitor automata	NO	NO	NO
DIVINE [99]	Plain	C/C++(via LLVM bitcode), DVE input language, Timed automata	LTL, Assertions, Memory safety	Yes	No	Yes
LTSA [125]	Plain	FSP	LTL	YES	NO	YES
LTSmin [199]	Plain, Real-time	Promela, μ CRL, mCRL2, DVE Input Language	μ -calculus, LTL, CTL*	NO	NO	NO
NuSMV [66]	Plain	SMV	CTL, LTL, PSL	NO	NO	NO
PAT [185]	Plain, Real-time, Probabilistic	CSP#, Timed CSP, Probabilistic CSP	LTL, Assertions	YES	YES	YES
PRISM [129]	Probabilistic	PEPA, PRISM language, Plain MC	CSL, PLTL, PCTL	YES	NO	NO
SPIN [71]	Plain	Promela	LTL	YES	NO	YES
UPPAAL [113]	Real-time	Timed automata, C subset	TCTL subset	YES	YES	YES
TLC [120]	Plain	TLA+, PlusCal	TLA	YES	YES	NO
IMITATOR [8]	Real-time	Parametric TA, parametric Time Petri Nets	reference valuations for timings	NO	NO	YES

formal methods at Amazon web services [37]. All these demonstrate the importance of model checking techniques, as well as the necessity of model checking tools.

These tools are also continuously in development to catch up with the development of computer science. During last decades, the complexity of systems is increasing (state space explosion problem). Therefore, these tools are extended with new techniques to improve their performance, e.g., SPIN, DIVINE, LTSmin and PAT with parallel implementations. Besides, some new types of system are presented, e.g., quantum communication systems. There are still limited research on the tools to cover these problems, which is one of our research points.

2.5 The Architecture of Parallel Computing Platforms

Parallel and distributed computing [6] are widely used in accelerating the computation in many areas. There exist many pieces of research on parallelizing graph exploration-based algorithms and the matrix operations, from which significant speedup is gain.

FIGURE 2.1: The Architecture of Multi-core CPU

There are several types of platforms to conduct parallel computing, e.g., the Multi-core CPU, many-core GPU, Cluster, and Cloud. We can utilize the high computation capability and the potential large memory size to deal with a larger size of state space, and improve the performance of graph exploration or matrix operation at the same time. In our research, we have taken advantage of the techniques in the multi-core CPU and GPU. Therefore, we present the preliminaries of these platforms in this section.

2.5.1 Multi-core CPU Preliminaries

The multi-core CPU exists from laptop to servers, which is a single computing component that consists of two or more independent actual processing units (called cores) [146]. Cores are the units for reading and executing program instructions. We briefly show the architecture in Figure 2.1. A multi-core platform can consist of many multi-core CPU which constitute ten to hundreds of cores. It has the features that communication cost among threads in these cores is low, and it has high logic control capability. But the performance of applications on this platform is always restricted to memory bandwidth and parallelism.

2.5.2 GPU Preliminaries

GPUs have been widely used to accelerate scientific, engineering and industry computations [104]. CUDA is a parallel computing platform and programming model

FIGURE 2.2: The Architecture of SMX

FIGURE 2.3: The Memory Hierarchy

for the NVIDIA GPUs. In this thesis, we utilize the GPUs with compute capability no less than 3.5, e.g., Nvidia Geforce Titan with *Kepler GK110* architecture [137]. Generally, GPUs are made up of a fixed number of streaming multiprocessors (SMX) e.g., 15 SMX in a full version *Kepler GK110*. Figure 2.2 shows the architecture of one SMX, each of which contains a fixed number of streaming processors (*C:core* in Figure 2.2). The execution model for these streaming processors is called single instruction multiple data (SIMD) [47], which makes GPUs have simple control hardware but imposes heavy cost in flow control.

One of the most important features of GPU is the memory hierarchy [139], shown in Figure 2.3. The hierarchical memory structure consists of Global Memory (GM), Constant Memory (CM), Texture Memory (TM), Shared Memory (SM) and Local Memory (Registers). The access performance of these memories in the descending

FIGURE 2.4: Dynamic Parallelism

Note: 2 hash functions are used in this example. It is not static in cuckoo hashing

FIGURE 2.5: Cuckoo Hashing

order are as follows: $GM < CM/TM < SM < LM$. SM is independent for each SMX. It is shared among all stream processors in a SMX. GM is the large size board memory. e.g., 5GB in Geforce Titan. The slow access to the GM is always the major aspect that affects the performance of GPU computation. The most effective global memory access could be achieved when the same instruction for all threads in a warp accesses global memory locations that are physically adjacent. In this case, the hardware coalesces all memory accesses into a consolidated one to consecutive DRAM locations [83].

In software perspective, the CUDA architecture defines three levels of threads organizing units: *Grid*, *Block* and *Warp*. A warp consists of 32 threads, which is the basic scheduling unit. Threads inside a warp are synchronized. A block contains a limited number of threads (e.g., 1024) and can only execute in a SMX. Grid is the organization of blocks. Threads inside a warp or block communicate through SM and

FIGURE 2.6: The Architecture of Cluster

threads in different blocks communicate through GM. The application running on GPUs is called *Kernel*. Note that the divergence execution in multiple threads inside a warp may follow different paths of execution, and all these paths are executed sequentially instead of in parallel, which is called *Warp Divergence* [41].

Since *Kepler GK110*, GPUs with compute capability no less than 3.5 support the new feature: *Dynamic Parallelism*. Different from the previous Fermi architecture, it gives kernels the ability to launch new tasks in GPU from itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths. This feature makes the GPU computation fully independent from the CPU. In CUDA 5+ architecture, the dynamic parallelism is described as in Figure 2.4. The kernel launched from the CPU is defined as *Parent Kernel*, while the corresponding execution environment is *Parent Grid*. The kernel launched from parent kernel is *Child Kernel*, with the execution environment *Child Grid*.

Defining a parallel-friendly data structure is an important GPU research topic [3]. The most important aspect for designing this kind of data structure should be the efficient update and access with millions of elements. In the CPU, hash table is the most widely used data structure to fit these requirements. However, hash table based techniques used in the CPU often cannot be translated directly in the GPU [43]. Recently, *Cuckoo hashing* has been adopted, e.g., [19, 43, 44], by integrating multiple hash functions to provide each element a fixed number and random store positions. It avoids collisions by moving elements around instead of fixing them in their initial

FIGURE 2.7: The Architecture of Cloud

positions. This technique can easily fit the memory access requirements in GPU. As it just takes few steps to read or update the elements, the number of uncoalesced memory accesses are reduced. An example is presented in Figure 2.5 to describe the process of cuckoo hashing.

2.5.3 Other Platforms

Besides the multi-core CPU and GPU platform, cluster and cloud are the other two important platforms for high-performance computing. A computer cluster consists of many computers which are usually connected to each other through fast local area networks (“LAN”). The architecture is shown in Figure 2.6 [204]. Cloud is an internet-based computing platform that provides shared computer processing resources and data to computers and other devices on demand [203]. The architecture is shown in Figure 2.7. Cluster differs from Cloud in that a cluster is a group of computers connected by a LAN, whereas cloud and grid are more wide scale and can be geographically distributed. A cluster is tightly coupled, whereas a cloud is loosely coupled. In addition, clusters are made up of machines with similar hardware, whereas

clouds and grids are made up of machines with possibly very different hardware configurations.

Both cluster and cloud can provide huge computation capability and large memory resources, which is feasible to the computation-intensive work. Nowadays, cloud has been taken as the major computation architecture worldwide. There is also a few research in model checking area that takes use of cloud computing techniques to optimize the model checking algorithms. e.g., [26, 27, 160]. The challenge in these works is the way to distribute (map) the tasks of model checking in different computer nodes. The communication via network is necessary for these platforms, which is costly. It is not stable since we also have to consider about the fault tolerance when there is something wrong with the network. Considering the complexity and economic efficiency, in our research, we only focus on using the multi-core CPU and many-core GPU platform to do the parallelization of model checking algorithms.

Chapter 3

GPU Accelerated On-the-fly Reachability Checking

3.1 Introduction

Model checking is an automatic technique for the verification of finite state systems. As the number of state variables or processes increases, the size of the state space grows exponentially and results in the *state space explosion* problem [63]. On-the-fly verification is one of the most widely used approaches to deal with the state space explosion problem [88]. Traditionally, on-the-fly verification is DFS-based for its memory efficiency. However it is known that DFS is hard to be parallelized. BFS is widely used in the design of multi-core and many-core based verification algorithms due to the fact that the state space can be easily partitioned and distributed independently. Because the state space is unknown during the verification process, the key challenge in these researches is to achieve a full distribution of the state space so that the different cores can be fully utilized for maximum parallelization.

GPUs have been widely applied to accelerate the computation in many areas, including model checking problems [17, 53, 103]. The challenges of effectively utilizing GPU for model checking are the redesign of the data structure and the algorithm mechanism to fit the architecture and computation model for GPUs (e.g.,

memory hierarchy and single instruction multiple data (SIMD)¹). In this chapter, we propose an on-the-fly reachability checking approach in GPU, which is realized in a tool named GPURC. The core algorithm is a parallel GPU accelerated BFS-based state space generation algorithm performed on the compacted GPU encoding of system models. The parallelism during the state space generation process is dynamically adjusted so as to fully utilize the many cores resource of GPUs to improve the performance. The execution process can be flexibly adjusted according to different system features, e.g., global variables and event synchronization.

Our key contributions of this work are as follows. 1) We propose a compact GPU encoding of concurrent system models with the support of global variables and event synchronization. Our approach can support systems with a large number of concurrent components using a multiple integer encoding. 2) We develop a BFS-based searching algorithm in GPU with dynamic load balancing without the CPU involvement, using the latest dynamic parallelism feature of GPU Kepler architecture. 3) Our approach incorporates an efficient hierarchical hash structure to store the state space and uses parallel state space generation to achieve event synchronization. A conflict-release accessing model is used to support the multi-integer encoding. 4) Experimental results show that our approach can achieve up to around 100x speedup on benchmark examples compared to the sequential BFS-based and DFS-based algorithm. When exploring the complete state space is necessary, our approach can achieve up to 8x speedup.

3.2 Related Work

GPUs have already been used for solving model checking problems, e.g., state space exploration and duplicate elimination problems. [183] presents a smooth interplay of a bitvector state space representation and the GPU accelerated BFS based on bitvector. It also integrates perfect hashing, but it is not suitable for an on-the-fly verification process. [184] accelerates the state space exploration for explicit-state model checking by utilizing GPU to do the breadth-first layered construction. [64] proposes how to use

¹It describes devices with multiple processing elements that perform the same operation on multiple data points simultaneously.

the GPUs in the SPIN model checker. The closest work to ours is [19], which focuses on utilizing GPUs to construct the state space on-the-fly, and their journal version [18], which expands [19] for safety verification. It does not support system models with global variables. The computation models in [19, 183, 184] are all based on the CPU-GPU collaboration, which are only for state space generation instead of verification, while we support the compute model that is purely based on GPU, which is also achieved by our previous work [222] for the GPU-based counterexample generation.

3.3 GPURC Overview

This section presents the overview of GPURC design, which aims to achieve the following goals in parallelizing the reachability verification in GPU: 1) Be able to design an efficient algorithm to fully being parallelized in the GPUs. 2) Be able to reduce the memory cost for large scale state space. 3) Be able to support LTS models with different features.

The core of GPURC is a BFS-based on-the-fly state space generation algorithm, as shown in Algorithm 2. We parallelize the BFS process in GPU by distributing the expanded states to a large number of threads in different blocks. The whole verification process is on the fly such that if a target state (a goal state in terms of the property to check) is found by any thread in any block, the verification terminates.

The input of the algorithm is a succinct representation of the global transition system M and the reachability condition ϕ . The output is the reachability verification result. A number of blocks¹ are started on GPUs and each block has a number of concurrent threads. In this work, we introduce the notion of *thread group* [19] as the logic grouping of threads for the purpose of concurrent generation of the outgoing transitions. Algorithm 2 is executed on all the threads in GPU concurrently. The corresponding block ID (*bid*), thread group ID (*gid*) and thread ID (*tid*) are identified in line 1. The number of threads in each thread group is equivalent to the number of component LTS in M . The number of thread groups is the thread number per block divided by the

¹Grid is another way to organize threads for matrix computation, which is not used in this thesis.

thread group number in a warp. The synchronization of threads inside a warp can be maintained all the time. For example, if a warp consists of 32 threads, the number of components is $|M_i| = 5$, and a block has 512 threads, then the number of thread groups is $(512 \div 32) \times (32 \div |M_i|) = 80$.

Each block has a global working list Ω_{bid} , which is stored in GM and can be accessed by all other blocks. Ω_{bid} is initially empty for all blocks, except block 0. Block 0 starts the BFS from the initial state s_0 ; all other blocks are waiting until some states are inserted in their global working list.

Each thread group has a private local queue Ω in SM, which is initialized using one state in Ω_{bid} as shown in line 5. If Ω_{bid} has more states than the maximum number of thread groups in a block, the extra states remain in Ω_{bid} for next round execution. From lines 6 to 18, each thread group performs the BFS searching for reachability detection. Here we use a global hash table (refer to Section 3.4.2) to check whether a state has been visited. The hash table is stored in GM and can be accessed by all blocks. Each thread in the thread group is in charge of one LTS in M for the outgoing transition generation. This is the reason why the number of threads in each thread group be equivalent to the number of component transition systems in M^1 . At line 14, each thread generates transitions for the corresponding LTS based on the transition relations. The event synchronization is handled differently as explained in Section 3.4.2.

This BFS is an iterative process. If the global working list Ω_{bid} is not empty and the local working list Ω has space, states are transferred to Ω . In line 19, when both Ω_{bid} and Ω become empty, we define a Boolean array *Status* in GM, and the running block marks *Status*[*bid*] as true. Then in line 20, the block reduces all elements in *Status* to find if there is a false value. The whole algorithm terminates when all elements in *Status* are true, otherwise the block continues to wait for new states.

During the BFS-based process, the workload is changing all the time, so we adopt a dynamic BFS process to adjust the parallelism such that we can help make full use of the many cores in GPU. To fit both Fermi and Kepler architecture, we integrate the BFS-based process in two execution models. Figure 3.1(a) shows the normal execution

¹For systems with concurrent components more than 32, each thread takes charge of the successor generation of more than one component LTS.

Algorithm 2: PBFS: Breadth-First Search for GPU

input : $M = (S, Act, s_0, \longrightarrow, AP, L)$: the global transition system. ϕ : the reachability condition
output: Yes/No

- 1 Let bid , gid and tid be the current block ID, thread group ID and thread ID respectively;
- 2 Let Ω_{bid} be a working list, which contains the initial state s_0 for block with $bid = 0$;
- 3 **while true do**
- 4 Let Ω be the thread group queue for breadth-first search;
- 5 Initialize Ω using the states in Ω_{bid} based on gid ;
- 6 **while Ω is not empty do**
- 7 Remove a state s from Ω ;
- 8 **if s is not visited then**
- 9 Mark s as visited;
- 10 **else**
- 11 continue;
- 12 **if $s \in \phi$ then**
- 13 **return Yes**;
- 14 **foreach state s' such that $s \xrightarrow{\alpha}_{tid} s'$ and s' is not visited do**
- 15 **if $s' \in S^\phi$ then**
- 16 Insert s' into Ω ;
- 17 **else**
- 18 continue;
- 19 Mark Ω_{bid} as empty: Define $Status[]$ and $Status[bid] = true$;
- 20 $Status[0..|M|]$: break when no element in array is *false*;
- 21 **return No**;

model in Fermi, which is a CPU-GPU collaborative process and is the most widely used in current research. Figure 3.1(b) shows the execution model in Kepler, which utilizes the new dynamic parallelism feature to be a GPU-pure process without the involvement of CPU, which integrates the concept of Parent Kernel and Child Kernel. Details are shown in the first paragraph of Section 3.4.

To fit the computation in GPUs, it requires us to build a compact encoding for both the input models and the generated state space. During the state space generation process, there are a large number of memory accesses that affect the performance. We integrate the efficient GPU hash structures such that we can access the location to store the state within a fixed number of steps. Duplicate states elimination can also be handled by hash, mentioned in Algorithm 2. In addition, we integrate an efficient data transfer approach for the process to transfer data together with the adjustment of parallelism. Details are shown in Section 3.4.2.

FIGURE 3.1: Execution Model in Fermi and Kepler

3.4 GPU Accelerated On-the-fly State Space Generation for Reachability Verification

In this section, we explain the GPU searching process in Algorithm 2. Our searching process can dynamically adjust the parallelism, which can be integrated into two different execution models as shown in Figure 3.1. In this work, we focus on the integration of our approach to the one in Figure 3.1(b) with the latest GPU technique. We present the GPU computation process in Figure 3.2, which is a two-level scheduling model based on the dynamic parallelism of CUDA and GPU memory hierarchy. Both the parent kernel (presented in Algorithm 4) and the child kernel (presented in Algorithm 5) are the extended runtime implementation of the BFS searching process in Algorithm 2.

Different types of memory in GPU hierarchical memory structure have differences in access rate and size. Therefore, we propose to use a hierarchical hash structure for the data accessing and storage. As shown in Figure 3.2, *GlobalHash* and *LocalHash* are created in GM and SM as the GPU implementation for Ω_{bid} and Ω in Algorithm 2, respectively. The details about how they work and the hash function are introduced in Section 3.4.2.

The GPU computation starts from ①. The parent kernel is launched from the CPU to start the verification with the initial state s_0 . parent kernel is concurrently executed by many thread groups. In ②, each thread group independently proceeds the successor generation and synchronization. The generated new states are stored in *LocalHash*. If a collision happens, defined as the overflow of *LocalHash*, a child kernel is launched from the parent kernel to allocate more computation resource, as shown in

FIGURE 3.2: GPUDV with Dynamic Parallelism

③. Unvisited states are transferred from *LocalHash* of the parent kernel to *GlobalHash* so as to be transferred to the child kernel, shown in ⑧. The child kernel performs the same computation as the parent kernel, as shown in ④, which is also executed by many thread groups. In runtime, when a target state is detected, all thread groups in the same block are notified through a *bool* mark in the SM, and these thread groups terminate. If there is no target state, an array in SM works for the status recording of all thread groups, which has the same function as *Status[]* defined in line 19 of Algorithm 2. During the execution, parent kernel terminates after all child kernels finish their execution, which occurs in three conditions, as shown in ⑤: a) a collision happens in *LocalHash*, b) a target state is found and c) no more new state, i.e., the state space being completely generated and there is no target state. In ⑥, parent kernel continues if child kernel terminates in condition a), and finishes its execution if child kernel terminates in condition b) and c).

In our approach, we bring in an efficient GPU hashing design and combine it with the latest features in the GPU architectures. We build a parallel collaborative

FIGURE 3.3: Component LTS Encoding

synchronization and data transferring approach to improve the performance of the state space generation. Note that our approach is a BFS-based, but the search is not strictly layer by layer. We describe more details in Section 3.4.2. These approaches are independent to the GPU execution models.

3.4.1 System Encoding in GPU

The performance of graph traversal in GPU is highly affected by the memory access pattern. Each component LTS can be considered as a sparse graph with imbalanced structure. Hence adjacency matrix is not the suitable data structure for GPU computation. A compact encoding to represent the LTS is necessary. To this end, we build a minimal bit-cost encoding for LTSs.

Assume the global state space is the parallel composition of n LTSs denoted as $M_1 \parallel M_2 \parallel \dots \parallel M_n$, and $M_i = (S_i, Act_i, s_0^i, \rightarrow_i, AP_i, L_i)$ for $1 \leq i \leq n$. We encode the global state space as the composition of all component LTSs M_i , which is a four-layer integer array as explained in the following. The states in the global state space are in the form of $S_1 \times S_2 \times \dots \times S_n$, which is encoded as a state vector. If the system model contains global variables, we encode them into the state vector with additional bits. An intuitive view on the four-array encoding can be observed in Figure 3.3.

Firstly, the encoding for the global state space of M is composed of multiple encoded M_i , where M_i represents the component LTS. We encode the outgoing transitions in global state space based on all states in S_i and all events in Act_i , where S_i and Act_i are the state space and events set of M_i . An outgoing transition of a state is encoded as the montage of the enabled event in Act_i and its *to state*, which is made up of the state from each S_i . They are compactly encoded with a minimum number of bits. An event in Act_i is also encoded with a fixed number of bits, which is equivalent to $\log_2 |Act|$, where $|Act|$ is the number of all events in M . A state in S_i is encoded with $\log_2 |S_i|$ bits, where $|S_i|$ is the number of states of M_i . The encoding of a transition should be aligned to a fixed number of bytes, shown in *Part B* in Figure 3.3. In this way, an integer can be used to encode multiple transitions. Two types of transitions, *transitions* and *transitions with synchronized events* are encoded separately in array *LocalTransitions* and *SyncTransitions*. There is an integer index from *LocalTransitions* to *SyncTransitions* in order to build the complete list of outgoing transitions from a state. Encoded transitions in *SyncTransitions* should be in order by the event ID of the transition.

Array *StatesIndex* is introduced to record the starting offset for the outgoing transitions of a state in array *LocalTransitions*. Array *LTSIndex* is used to record the starting offset for each M_i in array *StatesIndex*. The values in these four encoding arrays are static during the verification and can be binded to *TM* for fast random access.

Secondly, a state vector is encoded with a fixed number of 64bits integers. Global variables are encoded together with the state vector, which locates at the head of each state vector, the number of bits required to encode global variables is based on the number of variables and the range of their value, shown in *Part C* in Figure 3.3. Different types of global variables are transferred to integers. e.g., for Boolean type, 1 means true, 0 means false.

We introduce the concept of *thread group* in Section 3.3, which consists of threads inside the same warp. As the synchronization of threads inside a warp can be maintained all the time, and tasks are independently proceeded inside a thread group, the runtime execution has *Coarse-grained* parallelism, which reduces the cost for synchronization.

3.4.2 State Space Generation in GPU

Based on the proposed system encoding, we explain the state space generation process in this section.

State Space Hashing and Duplicate Elimination For on-the-fly verification, the size of the state space is unknown. It is important to find an effective way to store state vectors. In our previous work [222], we build arrays in both GM and SM to store data. However, it is hard to define the size of the arrays with the load balancing problem among all threads, which always results in a sparse storage and is also not efficient for random access. In this work, we adopt hash tables to solve this problem.

Figure 5.1 shows that our hash structures consist of *LocalHash* in SM and *GlobalHash* in GM. The hash methods for the two hash structures are cuckoo hashing [159] combined with linear probing hashing. Generated state vectors are firstly stored in *LocalHash*. There are two global hash tables inside *GlobalHash*: *GlobalVisitedHash* and *GlobalOpenHash*. *GlobalVisitedHash* stores the visited state vectors, which is used in line 8, 9 and 14 in Algorithm 2. *GlobalOpenHash* is used to store the generated but unvisited state vectors, i.e., Ω_{bid} in Algorithm 2. Cuckoo hashing in our approach is described in Figure 2.5. It uses multiple hash functions with the form: $hash(k) = (a * k + b) \% P \% TableSize$, where P is a prime number. a and b are a set of values which are generated randomly.

The duplicate elimination works on the *LocalHash* and *GlobalOpenHash* to avoid storing duplicated unvisited state vectors and works on *GlobalVisitedHash* to avoid the successor generation for a visited state vector. As each state vector has its own hash value, the same state vectors have the same hash value. However, the hash value is not unique to a state vector, which means two different state vectors may also have the same hash value. These can cause more work to do duplicate detection as we need to compare the value of state vectors instead of just comparing their hash value. We also integrate the linear probing if there is no available hash position to store the state vector. With these, the duplicate elimination can completely avoid duplicates. It should be noticed that the duplicate detection results can be *true-negative* but never be *false-positive*. So there is no missing state vector.

State Space Generation with Dynamically adjusted Parallelism The state space generation is a BFS searching process based on the compact system encoding. Each thread group generates the successor states by decoding the transition relations in the four encoding arrays. Newly generated state vectors are stored in *LocalHash*, in which each thread group gets new state vectors to handle. Once a thread group finishes the successor generation, the state vector handled by it is stored in *GlobalVisitedHash*. Based on our design in Section 3.3, successor generation process is scheduled dynamically and non-deterministically: 1) The parallelism for successor generation is dynamically adjusted based on the collision circumstance shown in Figure 3.2. 2) The successor generation is not restricted to a BFS. As generated states are randomly distributed in the *LocalHash*, there is no guarantee that a thread group gets states layer by layer by following the BFS mechanism. The benefit is that it can work like a DFS to some extent. It has potential to reach the target (e.g., deadlock state) faster.

No matter which execution model we use, i.e., Figure 3.1(a) or Figure 3.1(b), data transferring occurs with the parallelism adjustment. For *Read* operation from *GlobalOpenHash* to the local working list of thread groups, i.e., Ω in Algorithm 2, all threads access the *GlobalOpenHash* in GM in order for coalesced access, then use atomic operation to fill up the local working list for each thread group. For *Write* operation from *LocalHash* to *GlobalVisitedHash* and *GlobalOpenHash*, we use the hash operation mentioned previously.

Collaborative Synchronization The synchronization operation occurs during the interleaving state space generation process. Different from [19], which uses a central mode - one thread sorts all the time to synchronize and other threads just wait based on the warp divergence of GPU, we design a parallel collaborative synchronization approach. Instead of using several bits to mark the synchronized events, we encode all transitions with synchronized events in the same array, mentioned in Section 3.4.1. The process is shown in Algorithm 3. We define the shared space intra a thread group a tuple (*SyncEInterC*, *SyncSInterC*) in line 1 in Algorithm 3, which stores a state's outgoing transition: the ID of event in Act_i and its tostate in S_i . The synchronization process starts with the ordered *SyncTransitions*, where the ID of event stored in *SyncEInterC* represents the smallest event ID. It is an iterative process that all threads are involved

Algorithm 3: Collaborative Synchronization

```

Input: SyncTransition
1 Define Shared : SyncEInterC[], SyncSInterC[], SyncMark[];
2 index = 0, m = numof(Mi), tgid = threadindexinthreadgroup;
3 while true do
4   GetMinSyncT(&SyncEInterC[], &SyncSInterC[]);
5   if SyncEInterC[] = 0 then
6     break;
7   if SyncEInterC[tgid] ≤ SyncEInterC[0...tgid - 1, tgid + 1...m] then
8     leqthanall ++;
9   if leqthanall = m - 1 then
10    while i < m do
11      if SyncEInterC[tid] = SyncEInterC[i], i ≠ tid then
12        SyncSVec(SyncSInterC[tgid], SyncSInterC[i]);
13        SyncMark[tgid] := true;
14    Eliminate duplicate sync result;
15    index ++;

```

in the judgement whether their event ID stored in *SyncEInterC* is the smallest among all threads, shown in line 8. If so, the thread try to find the transition to synchronize by searching each threads' *SyncEInterC* in line 9 to 13. It may generate several same state vectors but we guarantee that only one is stored in *LocalHash* in line 14. After that, each thread reads the next outgoing transition to *SyncEInterC* and *SyncSInterC*. For a intuitive view, we also propose the structure of collaborative synchronization in Figure 3.4.

The synchronization process in Algorithm 3 occurs only inside each thread group. Thread synchronization among all threads in a thread group is guaranteed. Each thread in a thread group owns a shared space for communication with other threads in the same thread group. Each thread reads the synchronization event in ascending order, it guarantees that the events with a minimum ID appear at the same iteration. So the synchronization works in an ascending order with the event ID. The conditions for the process to move to next event are that 1) no synchronized event exists in other components 2) it finishes the synchronization based on current event.

3.4.3 Supporting Global Variables and Large Number of Concurrent Components

For the system model with global variables, the value set of the global variables is updated during the state space generation process. The value set updates based on

FIGURE 3.4: Collaborative Synchronization

the alphabets of the transition systems and the updating rules vary from different systems models. For general usage, we build the interface to support global variables independently to the state space generation process.

We define the structure for global variables as a tuple $GVS = (V, E, L, R, AR)$ where V is a set of global variables, E is a set of alphabets (the event ID) for all LTSs M_i . R is a set of formulas that define the rules to update the value of variables based on alphabets. AR is named from abstracted R , which is a set of symbols and encoded with a number of bits. The update of the value set of global variables V' for the successor is to perform mapping and analyzing operation, $V' = L(V) : AR \leftarrow E, R$. The mapping and analyzing operation L is specific to different system models. Then we make our GPURC work as a common framework and supply two interfaces to support global variables: 1) *EncodeRule*. We mention in Section 3.4.1 that the value set of global variables is encoded at the head of each state vector. This interface gets V as the input and calculates the number of bits needed to encode each global variables and builds an array to store the information for GPURC. 2) *VariablesUpdating*. Shown in Figure 3.5, this interface performs the function of L . It decodes AR and works on the current value set of global variables to output the new value set. This is completely language or platform independent. The key idea is to construct AR , which is built by simplifying the formula of R , which represents the value change of variables with alphabets, to a fixed number of symbols so that it is easy to store and transfer to GPU for execution. e.g., $S_i \xrightarrow{a} S_j, V_i = V_i + 1, (V_i \leq Max)$ is abstracted as i, a, Max , and in *VariablesUpdating* we recognize these symbols as variables should be more close to Max , then the variables should plus one to finish this operation. In addition, for systems in which the event ID can be matched to specific rules regularly, the construction of AR can be simplified

FIGURE 3.5: VariablesUpdating

FIGURE 3.6: Conflict Release R/W

more by updating global variables just based on event ID. e.g., *ReaderWriter* model in Section 3.5.1, in which the events with $eid \% 2 == 0 / eid \% 2 == 1$ indicate the same rules to update global variables.

Besides supporting system models with global variables, the other challenge is to support a large number of concurrent components, which means the number of components exceeds a threshold that a single 64bits integer is not enough to encode a state vector. Thus, multiple integers are required. However, encoding a state vector with more than one 64bits integer means the hash store operation cannot finish in one step as an atomic operation can handle at most a 64bits integers at a time. During the GPU state space generation process, the high parallelism results in a random write/read order. If we write/read operate on multiple integers with the same order as we do on single integer, it has the high probability to cause conflict and inconsistency. We deal with this problem by two approaches: 1) Read and Write operate in the opposite order with *atomicCAS* operation, shown in Figure 3.6. Only when the atomic operation succeeds in reading/writing the position to get/store the last/first integer of the state vector, the read/write operation continues. 2) The hashing method we used originally, the cuckoo hashing, should be changed to a double hashing together with linear probing. This change is to get rid of the inconsistency caused by the data exchange operation in cuckoo hashing, which requires several numbers of exchanging as a state vector is encoded in more than one integer.

3.4.4 Algorithms

In this section, we present the algorithms for the process in Figure 5.1. We describe the algorithm of *Parent Kernel* and *Child Kernel* based on dynamic parallelism. In fact, both of the parent kernel and child kernel handle the same work. Their relationship is based on the parallelism adjustment. And the algorithms shown can be easily transferred to the execution model in Fermi in Figure 3.1(a).

Parent Kernel is described in Algorithm 4 and *Child Kernel* is described in Algorithm 5. Note that we use *bid*, *gid* and *tid* as the block ID, thread group ID and the thread ID respectively as in Algorithm 2. We define *warpid* to represent the thread ID inside a *Warp*, and define *tgid* as the thread index inside a thread group.

In Algorithm 4, the input is the encoded transition system M . s_0 is the set of initial state vectors. $|M_i|$ is the number of LTSs. In line 2, *GroupStore*[] is the local working list of *thread group* to store the state vector it works on. The first thread in each thread group x transfers an unvisited state vector from I to its *GroupStore*[x]. Lines 3 to 28 are the major process. In line 4, each thread y in a thread group decodes and gets the state S of LTS_y based on *GroupStore*[x] and its *tgid*. In line 5, the function *GetAllSucc* initializes *SuccIdx*, which is a tuple (*LTbeginInt*, *LTendInt*, *STbeginInt*, *STendInt*) to index all outgoing transitions of S in *LocalTransitions* and *SyncTransitions*. In line 7, the first thread in a thread group detects if the *GroupStore*[x] has been visited by accessing *GlobalVisitedHash* with *cuckoo hashing*. Then it enters the process in line 9 and line 10 if necessary, which is the process mentioned in Section 3.4.2 and 3.4.3.

As the size of the *LocalHash* is limited, during the process of successor generation, collisions happen when the saturation of the *LocalHash* overflows, which means the *LocalHash* cannot hold more insertions. If there is no collision, the process continues from line 26 to 20. All threads attend to get new state vectors randomly from *LocalHash* to fill up the *GroupStore* for all thread groups, and start a new iteration.

If collisions have happened, no matter which execution model is integrated, all data in *LocalHash* is hashed to *GlobalOpenHash* in line 14. The current grid works as a parent grid. In the loop from line 16 to 20, the first thread in the block launches the child grid with more blocks and distributed data in *GlobalOpenHash* to the child kernel.

In dynamic parallelism, the CUDA interface *cudaDeviceSynchronize* is used to synchronize between the parent kernel and the child kernel. In line 18, the child kernel finishes its execution and synchronizes with parent kernel for the information about 1) *IfTargetDetected*. 2) Whether the collision happens in child kernel. 3) Whether there is no unvisited state vector. Based on this information, a parent kernel decides to either exit or calculate and allocate new size of resource to launch a new child kernel. The interface in line 19 is integrated from [139], which is a high performance reduction approach. We use it to calculate the number of unvisited state vectors in *GlobalOpenHash*.

The function and process of Algorithm 5 are similar to Algorithm 4. Thus, we ignore the duplicate description in line 7. The differences between Algorithm 4 and Algorithm 5 are: 1) In line 5, each thread group gets state vectors from *GlobalOpenHash*. 2) In lines 9 to 10, after transfer data back to *GlobalOpenHash*, the child kernel exits execution. 3) In line 11, if no more new state vectors exist, child kernel exits the execution.

It can be shown from the algorithms that the parent kernel needs to iteratively launch the child kernel and should not terminate until all child kernels terminate. The parallelism of parent kernel is static while it is flexible for child kernel to adjust parallelism. These motivate us that in our BFS-based state space exploration, we allocate little scale parallelism for parent kernel and let the child kernel achieve the high parallelism to finish the tasks as soon as possible.

3.5 Evaluation

GPURC is developed in CUDA C++ with two variants: 1) GPURC-GC: implemented in the CPU-GPU collaborative execution model in Figure 3.1(a). 2) GPURC: implemented in the GPU-pure execution model in Figure 3.1(b). It is the implementation of Algorithm 4 and Algorithm 5. We evaluate the performance of GPURC-GC and GPURC by comparing them with the sequential state space generation in PAT for deadlock verification algorithm, which is implemented based on the PAT model

Algorithm 4: Parent Kernel Algorithm

Input: Compact Encoding of M , s_0 , $|M_i|$, GVS

```

1  if  $tgid = 0$  then
2     $GroupStore[gid] = s_0$ ;
3  while  $\neg ifDs$  do
4     $S \leftarrow GetStateinV(tgid, GroupStore[gid])$ ;
5     $ifanyOutgoing \leftarrow GetAllSucc(S, \&SuccIdx)$ ;
6    if  $tgid = 0$  then
7       $ifdup \leftarrow DuplicateElimination(GroupStore[gid])$ ;
8    if  $ifanyOutgoing$  and  $\neg ifdup$  then
9      Successor generation, Collaborative Synchronization  $\rightarrow ifcollision$ ;
10     OPTION: GVS for system models with global variables.
11    $IFTargetDetected() \rightarrow ifDs$ ;
12   if  $\neg ifDs$  then
13     if  $ifcollision$  then
14       cuckoo hash store&linear probing:  $LocalHash \rightarrow GlobalOpenHash$ ;
15       if  $tid = 0$  then
16         while  $\neg ifDs$  do
17            $LaunchChildKernel$ ;
18            $CUDA-API: CudaDeviceSynchronize()$ ;
19            $cudaHighPerformanceReduce \rightarrow NoS$   $CUDA-API: CudaDeviceSynchronize()$ ;
20           Adjust Parallelism based on the  $NoS$ : number of state vectors in  $GlobalOpenHash$ ;
21     else
22       break;
23    $CUDA-API: \_syncthreads()$ ;
24    $index = 0$ ;
25   if  $LocalHash[tid] \neq NULL$  then
26      $atomicAdd(Index)$ ;
27      $Index < |threadgroups| ? LocalHash[tid] \rightarrow GroupStore[Index] : donothing$ ;
28    $Index = 0 ? GlobalOpenHash \rightarrow GroupStore : continue$ ;
```

Algorithm 5: Child Kernel Algorithm

Input: $GlobalOpenHash$, Compact Encoding of M , $|M_i|$, GVS

```

1   $S \leftarrow Read(GlobalOpenHash[tgid + |threadgroups| * bid])$ ;
2   $Index = 0$ ;
3  if  $S \neq NULL$  then
4     $atomicAdd(Index)$ ;
5     $Index < |threadgroups| ? S \rightarrow GroupStore[Index] : donothing$ ;
6  while  $\neg IFTargetDetected$  do
7    The process is same to Parent Kernel;
8    if  $\neg ifcollision$  then
9      cuckoo hash store&linear probing:  $LocalHash$  to  $GlobalOpenHash$ ;
10     "return" to parent kernel;
11  Get new state vector from  $LocalHash$ , same to Parent Kernel  $Index = 0 ? GlobalOpenHash \rightarrow GroupStore : exit$ ;
```

checker [109] and named as **DFS** for DFS-based algorithm and **BFS** for BFS-based algorithm. **SPUP** means the speedup.

Our experiments are conducted on a PC with two Intel(R) Xeon(R) CPU E5 – 2670, 2.60GHz, 16GB RAM and *Geforce Titan Black* GPU with 13SMX, 6GB GM and 48KB SM in each SMX. The compute capability of the GPU is 3.5 based on Kepler GK110 architecture. The execution model in Fermi in Figure 3.1(a) is common for all GPU.

FIGURE 3.7: The Manual System Model for Evaluation

3.5.1 Performance Evaluation

We take three sets of experiments. All experiments are taken more than 10 times and the **Speedup** means the average speedup. In the experiments, we assign 512 threads per block. Note that the parallelism (the number of GPU blocks) of GPURC is dynamically adjusted during the execution. We define **MPblocks** as the maximum parallelism, i.e., the maximum number of blocks in GPU that can be allocated during the execution, which in all experiments is set to 5000 blocks. We build two types of system models manually: *ManualDeadlock* (**MDL**), which consists of two types **MDL-1** and **MDL-2**, and *ManualNoDeadlock* (**MNoDL**), separately shown in Figure 3.7. Both of them are composed of multiple component LTSs. Component LTS M_i in *ManualDeadlock* contains a deadlock state at the bottom layer. Component LTS M_i in *ManualNoDeadlock* is a circle structure. The **features** of all models for the experiment are shown in Table 3.1.

In the first set of experiments, we take deadlock state as the target and all models have deadlock state. We use different sizes of *Dinning Philosopher (DP)* model and our two types of *ManualDeadlock* models. During the execution, the parallelism is adjusted dynamically based on the unvisited state vectors in *GlobalOpenHash*. The realistic parallelism started in this set of experiments ranges from 1500 to 5000. Experiment results are shown in Table 3.2, where we can see that in this set of experiments, compared with the traditional BFS-based sequential algorithm, our approach can give up to 70x speedup. Based on the result of *MDL-2*, we can see that compared with DFS-based sequential algorithm, our approach can bring up to 20x speedup.

The results show that the GPU based approaches can quickly spread the threads to search for different parts of the state space, and hence give good speedup compared

TABLE 3.1: Features of Test Models

Model	Proc	Ints	States	Trans
DP	13	2	$1.1 * 10^6$	$7 * 10^6$
DP	14	2	$3.42 * 10^6$	$2.34 * 10^7$
DP	15	2	$1.06 * 10^7$	$7.7 * 10^7$
MDL-1	8	1	$1.67 * 10^6$	$1.34 * 10^7$
MDL-1	9	1	$1.01 * 10^7$	$9.07 * 10^7$
MDL-1	10	1	$6.04 * 10^7$	$5.23 * 10^8$
MDL-2	8	1	$1.67 * 10^6$	$1.34 * 10^7$
MDL-2	9	1	$1.01 * 10^7$	$9.07 * 10^7$
MDL-2	10	1	$6.04 * 10^7$	$5.23 * 10^8$
MDL-2	11	1	$3.62 * 10^8$	—
DP-Free	11	2	$5.1 * 10^5$	$3.6 * 10^6$
DP-Free	12	2	$1.68 * 10^6$	$1.29 * 10^7$
DP-Free	13	2	$5.56 * 10^6$	$4.62 * 10^7$
MNoDL	9	1	$1.95 * 10^6$	$2.1 * 10^7$
MNoDL	10	1	$9.76 * 10^6$	$1.17 * 10^8$
MNoDL	11	1	$4.88 * 10^7$	$5.25 * 10^8$
RW	100	4	$5.4 * 10^5$	$1.6 * 10^6$
RW	120	4	$9.3 * 10^5$	$2.7 * 10^6$
RW	140	5	$1.46 * 10^6$	$4.3 * 10^6$
RW	160	6	$2.16 * 10^6$	$6.4 * 10^6$
RW	180	6	$3.06 * 10^6$	$9.09 * 10^6$
SP	12	1	$3.6 * 10^5$	$2.1 * 10^6$
SP	13	1	$1.06 * 10^6$	$6.6 * 10^6$
SP	14	1	$3.05 * 10^6$	$2.05 * 10^7$

to the traditional approach. This phenomenon comes from the high parallelism in the GPU and the random data accessing in our approach which is not restricted to layer by layer. On the other hand, *MDL-1* and *MDL-2* contain the same number of states and transitions. But the cost of DFS-based algorithm on *MDL-1* can be ignored. We can see that the performance of DFS on reachability verification depends on the structure of models. The good performance on both *MDL-1* and *MDL-2* shows our approach is not restricted to the structure. GPURC-GC and GPURC give similar results because the number of the CPU-GPU coordination is small and gives less overhead. Further, in the last row of Table 3.2, we start 8000 blocks for the experiments. The BFS/DFS gets *out of memory exception* that cannot handle the *MDL-2* with 11 processes while the GPU approach works, which is based on the compact encoding and with high parallelism for exploration, we could reduce the size of state space required to generate before reaching the target states.

In the second set of experiment, we want to see the speedup of GPURC when the complete state space is explored. We use different sizes of *Dinning Philosopher Deadlock Free (DP-Free)* model and our *ManualNoDeadlock* model. During the process of evaluation, the parallelism always reaches 5000. Results are shown in

TABLE 3.2: Performance Evaluation for Reachability Verification(time in sec)

Model	GPURC-GC	GPURC	BFS	DFS	SPUP(B/D)
DP/13	2.5	2.2	25	-	11X/-
DP/14	3.4	3	104	-	35X/-
DP/15	68	62	367.7	-	60X/-
MDL-1/8	1.2	1.4	15.3	-	12X/-
MDL-1/9	3.1	2.8	112.8	-	40X/-
MDL-1/10	9	8.3	623.3	-	70X/-
MDL-2/8	1.25	1.5	13.9	7.2	11X/5X
MDL-2/9	5.2	5.7	102.3	61.1	20X/11X
MDL-2/10	26.1	24.3	604	474	30X/20X
MDL-2/11	15.5/8000	17.5/8000	EX	EX	-/-

TABLE 3.3: Performance Evaluation for Complete State Space Generation (time in sec)

Model	GPURC-GC	GPURC	BFS	SPUP
DP-Free/11	10.1	9.5	11.4	1.2X
DP-Free/12	25	22	50.1	2X
DP-Free/13	48	42	192.5	5X
MNoDL/9	10.3	8.4	24.9	3X
MNoDL/10	44	36.3	145.4	4.5X
MNoDL/11	89	72	538.4	8X

Table 3.3. Compared with the traditional BFS-based algorithm, our approach can achieve up to 8x speedup. We can conclude that our approach is also available for the complete state space generation. But compare with the results in Table 3.2, we can conclude that the biggest advantage of GPURC is to handle the on-the-fly state space generation for the target searching. Note that $6GiB \approx 6.44 \times 10^9 bytes$ memory can store approximately 8.05×10^8 64bit integers. In our experiments, at most 6 integers are used to encode a state vector, which means at least 1.34×10^8 state vectors can be stored in GPU.

In the third set of experiment, we aim to test the performance of GPURC for supporting global variables and large numbers of concurrent components. We use different sizes of *ReaderWriter* (**RW**) model and *Semaphore* (**SP**) model in PAT. Both of them contain global variables and there is no deadlock state. We do *non-executive* verification with **RW** and do the *RechabilityTest* verification with **SP**. *non-executive* verification requires to search the complete state space. *RechabilityTest* verification is to verify if the target state vector **RW** contains a large number of concurrent components. As shown in Table 3.4, We use *Ints* to represent the number of 64bit integers required to encode the state vector. For the *non-executive* verification, each state vector is encoded with multiple integers so during the state space generation process, we need to access the memory for multiple times to finish the read/write operation, which is costly and

TABLE 3.4: Performance Evaluation for Systems with Different Features (time in sec)

Model	GPURC-GC	GPURC	BFS	DFS	SPUP(B/D)
RW/100	33	32	36.9	-	1.1X/-
RW/120	57	55.1	72.2	-	1.5X/-
RW/140	63	61.3	112	-	2X/-
RW/160	71.5	69	177	-	3X/-
RW/180	88	83.2	234	-	3X/-
SP/12	0.25	0.22	15.6	2	70X/9X
SP/13	0.6	0.56	50.5	6	100X/10X
SP/14	1.21	1.1	151.3	14	150X/11X

affects the performance. We mention that a thread group cannot exceed a warp. In this experiment, the whole warp is a thread group. One thread in a thread group needs to take charge of several component LTSs, which is a sequential process in all threads. For the *ReachabilityTest*, we always finish the searching with *Parallelism* $\approx 450blocks$. We can see compared with **SeqB**, our approach can reach up to 150x speedup. Compared with **SeqD**, the speedup can reach up to around 10x. It can be concluded our approach can handle models with different features well.

Finally, the change of performance with different size of state space in GPU can be concluded from Table 3.3. We can see that for sequential algorithm, the time cost increases almost linearly with the increasing size of state space. But for GPU algorithm, it is not increasing as fast as the sequential algorithm, which reflects that we can get more speedup for a larger state space. On the other hand, we can see the increment rate of speedup is decreasing, which is the feature of memory intensive GPU algorithm as the benefit of parallelism can be neutralized by the costly memory access.

Based on all our experiments, we can conclude that GPURC is efficient for the reachability test during the on-the-fly state space generation, which can be applied in safety verification. Our approach supports different types of system models and can be generally integrated to deal with other state space searching problems. Furthermore, in our experiments, for some system models with larger size than what we show in tables, the sequential algorithm results in *OutOfMemoryException* while our approach can handle. This benefits from our compact encoding in Section 3.4.1.

3.5.2 Discussion

Four points in our approach should be noticed:

- The parallelism utilized in GPU. Theoretically, the value of *MPblocks* defines the maximum parallelism and may have impact on the performance. In fact, we test different parallelism and find there is no much difference among different *MPblocks*. The reason is the huge I/O cost neutralize the benefit of a larger parallelism.
- The size of hash table used in our approach, which is restricted to the GPU memory hierarchy. When the state vector is encoded with multiple integers, the size of *LocalHash* should not exceed a threshold as the SM that available for a block is limited. e.g., $1536 \times 64\text{bit}$ integers for state space encoded with single 64bit integer, $256 \times 64\text{bit}$ integers for state vector encoded with > 4 64bit integers.
- Our approach works well with both the CPU-GPU collaborative and the GPU-Pure execution models. But the performance gap between utilizing this two execution models here is not too obvious. Based on researches [89] on the *dynamic parallelism*, the *Clustering* problems with high data dependency during the iterations can benefit much from the GPU-pure execution model as the large size of data can be directly used inside the execution in the GPU to avoid the cost of copy all data back to the CPU memory. The observation in our approach is because the size of data required to cluster for launching a new kernel is small. But GPU-Pure execution model makes the algorithm development more flexible. e.g., in our approach, we can use the data in global memory directly for the resource reallocation to launch a new kernel (child kernel) instead of copying some data back to the CPU to calculate. And with dynamic parallelism, we can allocate the resource to launch a new kernel based on runtime results at any threads, without terminating all threads to return the control to CPU. In addition, based on [187], with dynamic parallelism, *nested parallel* problem can also be handled completely in GPU and the solution for recursive problems with dynamic parallelism shows the complexity simplify.
- There is a limitation in the approach. Although we support a large number of concurrent components, we do not support the event synchronization with more than 32 components, which is due to that our synchronization occurs inside a

warp (32 threads). This can be solved with our current work by integrating a sort operation for each thread to sort all synchronized events. The rest process can be similar to our existing algorithm. We plan to expand our approach with this in future work.

3.6 Conclusion

In this chapter, we propose an approach to accelerating the reachability verification with support to LTS Models with different features in GPU. We propose the compact encoding that supports global variables, efficient hierarchical hash structure and parallel state space generation with collaborative synchronization, which can be generally used in other state space exploration problems. Our approach supports system models with global variables and large number of concurrent components. Experiments have shown that the design of GPURC significantly enhances the performance of dealing with the on-the-fly reachability verification problem. Meanwhile, our approach is flexible and scalable according to the evaluation results.

Chapter 4

Concurrent On-the-fly SCC Detection for Automata-based Model Checking with Fairness Assumption

4.1 Introduction

Automata-based LTL model checking is the emptiness checking of the composition of a transition system M and a Büchi Automaton $B_{\neg\varphi}$, which represents the negation of an LTL property φ [38]. The composition process indicates that the state space is unknown in advance, which is on-the-fly generated. The idea of the emptiness checking is to search the on-the-fly generated state space to find an execution path that is accepted by the Büchi automaton. *Strongly Connected Component* (SCC) based model checking that uses Tarjan's algorithm [164] for SCC detection, is a well-known approach for LTL model checking. In this approach, the problem of LTL model checking is converted to the detection of an infinite path that is accepted by the Büchi Automaton. The infinite path contains SCCs with accepting cycles. It uses the depth-first search to explore the state space to detect SCC.

Fairness and liveness are two essential notions for faithfully modeling the execution progress of a process in a collection of concurrent processes [21]. Fairness constraints

can be expressed in LTL. Thus, fairness checking can be integrated into the SCC-based LTL model checking, which expands the original process to the verification of the fairness assumption's satisfaction in all detected SCCs.

There are many sequential implementations of SCC detection for the automata-based LTL model checking with fairness assumption. However, besides the state space explosion problem [63], it is also necessary and meaningful to improve the performance of the sequential implementations for automata-based LTL model checking with fairness when it works on large-scale systems with a large number of SCCs. In automata-based LTL model checking with fairness, the verification process is overlapped with the state space generation, which is the product of M and $B_{\neg\phi}$. The verification process may immediately report failure if it detects a fair SCC. However, sometimes the verification process has heavy costs, since the fair SCC does not exist or appear after a large number of SCCs being detected. In this chapter, we extend the concurrent implementation [67] of Tarjan's algorithm based on known state space to a concurrent SCC detection for LTL model checking with on-the-fly generated state space. We aim to improve the verification performance. Furthermore, we develop an efficient approach for the parallel fairness checking.

The main challenges of utilizing parallel computing to accelerate the automata-based LTL model checking with fairness are: (1) Tarjan's algorithm is a DFS process, which has attracted many types of research on its parallelization. However, all research works on the complete state space. It is challenging and significant to convert a parallel Tarjan's algorithm to be available for on-the-fly generated state space. 2) An efficient data distribution approach is necessary since the state space is unknown in advance. It is also important to maintain data consistency. 3) The sequential algorithm mentioned in [208] verifies the fairness after the generation of a complete SCC, which can be regarded as an independent part of the verification process. A better way for fairness checking is promising.

To solve these challenges, we expand the concurrent Tarjan's algorithm from Lowe [67] to fulfill the automata-based LTL model checking for SCC detection on on-the-fly generated state space. Our key contributions in this chapter are as follows: 1) Lowe's Tarjan's algorithm depends on the complete state space (unrooted mode).

Although he mentioned rooted mode in both [68] and [67], there is no details or experiments for this. We expand the algorithm to fit the on-the-fly state space generation. Based on it, we build the parallel approach to cover the features of on-the-fly SCC-based LTL model checking. 2) We build our data distribution rules for the on-the-fly generated state space. 3) We design and develop an efficient on-the-fly parallel fairness checking approach, which performs the fairness checking during the generation of SCC instead of performing it separately after the generation like [208] does. 4) We implement our approach in the Process Analysis Toolkit (PAT) [109] and make it work on a broad range of system models. Our evaluation shows that after integrating our approach, we achieve a 2x performance improvement in LTL model checking which involve the exploration of a large number of SCCs, and 2x~45x speedup for fairness checking.

The structure of this section is as follows: In Section 4.2, we introduce the related work, including the concurrent Tarjan’s algorithm from Lowe [67]. In Section 4.3, we present our design of the concurrent on-the-fly SCC detection for automata-based (LTL) model checking with fairness assumption. In Section 4.4 we present our experiments and evaluation. Finally, we introduce the conclusion and future work in Section 4.5.

4.2 Related work

LTL model checking does the SCC exploration via Tarjan’s algorithm. Thus, we firstly show the sequential implementation of this process in Algorithm 6, which is the version in PAT [208]. It use Tarjan’s algorithm for the SCC detection on on-the-fly generated state space. After the process finds an SCC, it does the fairness checking on this SCC by option and outputs the counterexample.

Parallel computing has been widely used to deal with model checking problems. [19] presents the GPU accelerated state space generation. Our previous work [222] presents the GPU-based counterexample generation for LTL model checking. [221] presents the GPU-based on-the-fly reachability checking. [168] and [170] present a multicore NDFS algorithm for LTL model checking. [209] proposes a parallel LTL

Algorithm 6: Sequential on-the-fly LTL Model Checking

```

Input:  $M, \mathcal{B}, s_0, b_0$ 
1  Define  $OGTrans, StepStack, TaskStack, SCCSet$ ;
2   $Product : S_0 = GenerateIniS(s_0, b_0, M, B)$ ;
3   $Add(OGTrans, S_0), TaskStack.push(S_0)$ ;
4   $i = 0, preorder = \{\}, visited = \{\}$ ;
5  while  $TaskStack \neq EMPTY$  do
6       $S = TaskStack.peek()$ ;
7      if  $S \notin preorder$  then
8           $preorder[S] = i, i++$ ;
9       $done = true$ ;
10     if  $S \in visited$  then
11          $S'_n = visited[S]$ ;
12         forall the  $S_i \in S_n$  do
13             if  $S_i \notin preorder$  then
14                 if  $done$  then
15                      $TaskStack.push(S_i), done = false$ ;
16     else
17          $s'_n = (S.s_n).MakeOneMove(M)$ ;
18          $Product : S'_n = GlobalSuccessor(s'_n, S.b_n)$ ;
19         forall the  $S'_i \in S'_n$  do
20             if  $S'_i \notin preorder$  then
21                  $OGTrans[S].add(S'_i)$ ;
22                 if  $done$  then
23                      $TaskStack.push(S'_i), done = false$ ;
24          $visited.add(S)$ ;
25     if  $done$  then
26          $lowlinkS = S.lowlink, preorderS = lowlinkS$ ;
27         forall the  $S_i \in OGTrans[S]$  do
28             if  $S_i \notin SCCSet$  then
29                 if  $preorder[S_i] > preorderS$  then
30                      $S.lowlink = Min(S.lowlink, S_i.lowlink)$ ;
31                 else
32                      $S.lowlink = Min(S.lowlink, preorder[S_i])$ ;
33         if  $lowlink == preorder$  then
34              $SCCSet.add(S)$ ;
35             backtrack  $Stepstack \rightarrow SCCSet$ ;
36             Optional: Fairness Checking; if  $ISFair$  then
37                 Record Result, Generate Counterexample, report Counterexample;
38             forall the  $S'_i \in SCCSet$  do
39                  $visited.remove(S_i); OGTrans.remove(S_i)$ ;
40         else
41              $StepStack.push(S)$ ;
42  return Result;

```

model checking algorithm which starts multiple threads to generate the SCC when it is detected, and the fairness checking occurs in the thread for SCC generation. [169] presents a state compression and reconstruction approach. It builds a state space exploration algorithm based on the shared memory multi-core architecture. [202] presents a novel emptiness checking approach for LTL model checking, which is based on SCC enumeration and support TGBA. Its key feature is the usage of a global union-find data structure. [161] presents a parallel SCC decomposition based on set-based

SCC algorithms instead of Tarjan’s algorithm. [67] introduces some concurrent DFS-based algorithms, such as concurrent Tarjan’s algorithm for SCC detection. It is the major related work for our approach. We describe it in detail below.

Concurrent Tarjan’s Algorithm: Lowe [67] designs a concurrent Tarjan’s algorithm for SCC detection. Given a system model M with the complete state space, the SCC exploration starts on multiple different states. He defines an object *Search* as the unit for exploration and an object *Scheduler* for the arrangement of searches to threads. Compared to traditional Tarjan’s algorithm, Lowe’s approach differs in three parts: 1) If a state is visited, it should record the ID of the corresponding *Search*. A state can only be visited by one search. 2) If a search i explores a state that records other *Search*’s ID j , i is suspended to j . 3) The suspended relation can be broken when the status of the state becomes *completed*, which means the state has been detected to be in one SCC. One iteration of the overall process is shown in Figure 4.1: Each *Search* has its own *Taskstack* and *Stepstack*. It follows the sequential Tarjan’s algorithm to explore the successors (child) of its initial state. Then based on the description above, they introduce a *SuspendingRelation* set. Searches need to check if there is a cycle in the *SuspendingRelation*, shown in *part A* in Figure 4.2. If so, the states in the Searches that related to this cycle are transferred to one single *Search* to block this cycle, shown in *part B*, Figure 4.2. This cycle is also an SCC. More details can be found in [67]. When any search finds an SCC, all states in SCC should be marked as completed to activate the Searches which are suspended.

The difference between our work and Lowe’s is that we expand the concurrent SCC detection for automata-based LTL model checking with fairness assumption based on the concurrent Tarjan’s algorithm. We support the concurrent on-the-fly SCC detection, in which the state space is unknown in advance. Although Lowe mentioned this mode (rooted mode) in both [67] and [68], he doesn’t supply any detail or experiment result. We also support the parallel on-the-fly fairness checking.

FIGURE 4.1: Execution Process in One Iteration

FIGURE 4.2: Blocking Cycle

4.3 Concurrent on-the-fly SCC detection for Model Checking Under Fairness Assumption

The core in the emptiness checking of automata-based model checking is the SCC detection. Thus, our approach is based on Lowe's concurrent Tarjan's algorithm (Section 4.2). Our target is to expand it to construct a concurrent version for the on-the-fly generated state space. To this end, we first present several key challenges,

which indicate the efforts on making Lowe’s concurrent Tarjan’s algorithm feasible for automata-based model checking with on-the-fly generated state space: (1) in this context, the state space is generated by the composition of transition system and Büchi automaton. Thus, the state space is unknown in advance, and the simple predefined ID for each state in Lowe’s concurrent Tarjan’s algorithm is not available. (2) In Lowe’s concurrent Tarjan’s algorithm, if a state is explored in a search, there is no duplicate state being explored in any other searches since the state space is known in advance. However, duplicate elimination is an important problem for concurrent on-the-fly generated state space. If there are duplicate states, the judgment on the suspension of a search cannot be made correctly. (3) Preservation of data consistency in the synchronization process is another challenge for the concurrent SCC detection on on-the-fly generated state space, which requires the upgrading of the execution process and data structures.

In this section, we present how to solve the above challenges in our approach in detail.

4.3.1 Approach Overview

We introduce the overall execution process of our approach in Figure 4.3. For readability, the figure shows only the main procedure of our approach. We present the complete algorithm later in this section.

First, we describe some essential concepts in our approach: (1) we define *GSearch* as our scheduling unit. Different to *Search* in Figure 4.1, *GSearch* is a combination of state space generation and exploration. Each *GSearch* has a *GSID* to identify itself. Each *GSearch* has its own *TaskStack* and *StepStack*. A *GSearch* also needs to record the status of itself, e.g., whether the *GSearch* is suspended to any state. (2) A *Thread* is the physical execution unit. A thread can handle multiple *GSearches* based on scheduling. (3) We define *TarjanNode* to indicate the state generated on the fly by *GSearch*, which is a meta-state. In *TarjanNode*, we define *SID* and *GSID*. *SID* is the identification of *TarjanNode* in global state space, which is constructed by the string concatenation of $S_1S_2 \dots S_nS_A$. *GSID* represents the *GSearch* that explores

this *TarjanNode*. We also define the *status*, *suspendlist*, which refer to the concurrent Tarjan's algorithm. *enableEvt* and *ParticipatingProcesses* are defined for the fairness checking. (4) We define *SearchParams* as the shared space among all *GSearches*. It consists of *StealingList*, *PendingList*, *BlockList*, *SCCList*, *SuspendMap* and the generated state space *OGTrans*. *StealingList* is to store all the generated *TarjanNodes*. It is a unique hash structure for duplicate elimination since *TarjanNodes* are generated concurrently in all *GSearches*. *BlockList* stores all *GSearches* that have been suspended. *PendingList* stores the resumed *GSearch*, which is waiting for *Scheduler* to allocate the free thread for it. *SuspendMap* stores the suspended relationship, e.g., *GSearch*₁ is suspended to *TarjanNode*₂. The *visited* set is also contained in *SearchParams*, which indicates the states that have been expanded. (5) We integrate the *Scheduler* from Lowe. It takes the charge of creating new *GSearch* and assigning it to a free thread. A thread is regarded as free if the *GSearch* it handles is suspended to any state. If there is no *GSearch* in *PendingList*, new *GSearches* are created by assigning new initial states. *Scheduler* maintain the *GSID*. Within the multi-core environment, we start multiple *GSearches* for each thread at the beginning. Thus, we can see in Figure 4.3, the startup sequential state space generation process is to generate an initial set of states for each *GSearch*. This is a difference from the concurrent Tarjan's algorithm with known state space. Figure 4.3 shows the execution process of one *GSearch*. With initial *TarjanNode*, *GSearch* initiates the *preorder* for it and generates the successors by the parallel composition. The synchronization operation is needed if synchronized events exist. The generated successors should be transferred to the array of *TarjanNode SuccTN[]*. We initialize the *SID* and other variants in *TarjanNode*. *SuccTN[0]* is the next *TarjanNode* to be expanded based on the rule of DFS. Here we give the example in the first iteration, hence we regard *SuccTN[0]* as unexpanded. Thus, *GSearch*₁ should judge whether *SuccTN[0]* has been seen in other *GSearches* by accessing the *StealingList*. If not, *SuccTN[0]* is marked with *GSearch*₁'s *GSID*. If *SuccTN[0]* is not in *StepStack*, it is pushed to the *TaskStack*. Only *SuccTN[0]* belongs to *GSearch*₁. Other *TarjanNodes* in *SuccTN[]* should be directly transferred to *StealingList*. Duplicate elimination is handled with the unique *SID*. It should be noted that when a *GSearch* writes its *GSID* to *SuccTN[0]*, it should also update the *StealingList* if *SuccTN[0]* exists

FIGURE 4.3: Overall Process

in *StealingList* without the mark of any *GSID*. Then the initial *TarjanNode* should also be pushed to *StepStack*.

Branch A in Figure 4.3 presents that if *GSearch*₁ detects that *SuccTN*[0] is marked with other *GSearch*'s *GSID*, it checks the status of *SuccTN*[0]. If *SuccTN*[0] is not completed, *GSearch*₁ is suspended to *SuccTN*[0]. It updates the *SuspendMap* and the *BlockList*, and detects if a blocking cycle exists. If this is the case, it blocks the cycle by transferring all related *TarjanNode* to another one *GSearch* (refer to Figure 4.2 in Section 4.2). *GSearch*₁ becomes blocked and waits to resume. When a blocked *GSearch*

is resumed, it is transferred to *PendingList*. *Scheduler* works on two tasks: (1) when a *GSearch* is suspended and waiting for resuming, the corresponding thread becomes free. If the *PendingList* is empty, it creates a new *GSearch* for the free thread. 2) If the *PendingList* is not empty, it gets *GSearch* from *PendingList* and pushes it to the free thread.

Branch B in Figure 4.3 is the common *lowlink* updating process and the condition that a cycle being detected. In our approach, the fairness checking starts concurrently with the generation of the complete SCC. After generating the complete SCC, the suspended relationship should be updated. All *GSearches* which are suspended to *TarjanNode* in the SCC are resumed and continue their exploration.

4.3.2 Data distribution

The data distribution in our approach for each search consists of two parts: startup distribution and runtime distribution.

Firstly, in the on-the-fly LTL model checking, the state space is not known in advance. It is generated during the product between M and $B_{\neg\phi}$. M is also the parallel composition of component LTSs. So at the beginning, we always have only one initial state. Based on Section 4.2, in that algorithm, fixed number of threads are started for concurrent searches at the beginning, which start in different states to follow different traces to increase the possibility to find SCC. For our approach, we should also start multiple *GSearches* for high efficiency. The difference is that for on-the-fly generated state space, we first start the state space generation for several iterations to generate partial state space. Then we base on the number of cores in the working machine to start the corresponding number of threads and *GSearches*. Thus, at the beginning, the parallelism can be fully utilized.

Secondly, during the concurrent search process in our approach, the data distribution is based on the state space generation. When *GSearch* is started, each *GSearch* marks its startup state with its *GSID*. Then each *GSearch* stores all generated successor states in its *Stepstack* and *DFSstack*. It should be noticed that as the state space generation is

a concurrent process in many threads, it is impossible to config the global ID to identify each successor state as [67] does. In our approach, each state is indeed composed of many states from component LTSs and the automaton, thus, we can use a *string* to config the generated states, which is the montage of each states' ID in the original component LTS or the Büchi Automaton. As *GSearch* works on a DFS process, each *GSearch* only marks the front state in the *DFSstack* with its *GSID*, then all successors are copied to the *StealingList* and the outgoing relations are copied back to the table *OutgoingTrans*. The key point is that before marking any successor state, the *GSearch* should visit the *StealingList* to check if this successor state is visited or marked with other *GSID*, and then decide if the search should continue, backtrack or suspend.

4.3.3 Concurrent State Space Generation

In our approach, the state space is generated concurrently by all active *GSearches*. Each *GSearch* handles the successor generation independently. During the successor generation, event synchronization operations are involved in both the parallel interleaving of M_i and the composition between M and $B_{-\varphi}$. We allocate a private array for each *GSearch* to store the generated successors temporarily. Event synchronization also works on this array. When we convert each successor state (s_i, b_i) to *TarjanNode*, we need to explore all s_i to fill up the information for fairness checking. These are all independently handled by each *GSearch*. The key point is, the state space generation in the concurrent environment refers to a lot of operations on shared space. e.g., the *SearchParam*. Thus, it is important to handle the concurrent access to prevent any data inconsistency.

To ensure the consistency of shared data, we use the *lock* throughout the program. In general, each shared variable should have a separate lock to make sure that at one time, only one *GSearch* can access it. *SearchParams*, as we have mentioned in Section 4.3.1, is the structure to store shared variables among *GSearches*. In order to ensure synchronization¹, most collection variables in *SearchParams* are created as the concurrent collection data type, which is provided by C#.NET. For other shared

¹Different from event synchronization, this represents the synchronization of data.

variables, the *SearchParams* has static locks to ensure their synchronization. To ensure synchronization and correctness of the program, we present some details of locks below, which indicate how errors occur without these locks.

- *TarjanNode.visitLock*: A *TarjanNode* is a shared variable. During the exploration of the state space, each *GSearch* checks the *TarjanNode* via the following steps: (a) Checks the status. (b) Checks whether it is the first unexpanded successor in *GSearch*. If yes, the *GSearch* visits the node. These steps are supposed to be atomic. Otherwise, two *GSearches* may take the same *TarjanNode* and cause errors, as shown in Figure 4.4. Two threads handle two *GSearches*. The first *GSearch* is taking the *TarjanNode* and hasn't finished, while the other *GSearch* is also checking the ownership of the node. Two *GSearches* take the same node and none of them being suspended. Thus, *visitedlock* is necessary to lock these three steps.
- *Suspendmap.Lockitself*: The updating and accessing of the *SuspendedMap* should be synchronized, otherwise blocking cycle may be missed. For example, in Figure 4.5, one *GSearch* is checking whether blocking cycle existed or not. When it gets the conclusion that no cycle exists, and it has not added new suspension into the map, other *GSearch* starts to check the *SuspendedMap* to get the path. Both of the *GSearches* think that there is no blocking cycle and add the suspension into the map. At this condition, blocking cycle appears but no one handles it. Then deadlock occurs.
- *TarjanNode.block()* & *TarjanNode.unblock()*: Two or more *GSearches* may become blocked on the same *TarjanNode* concurrently. Moreover, the *TarjanNode* may be detected in an SCC at the same time. Shown in Figure 4.6, when the first *GSearch* is updating the status of the *TarjanNode* but not yet finished, the second *GSearch* is checking *TarjanNode*'s *GSID*. The second *GSearch* thinks that the *TarjanNode* is incomplete and starts to block itself on this node. However, the first *GSearch* sets the node as complete and unblocks all blocked searches on this node. In this situation, the status of this node has already

FIGURE 4.4: TarjanNode Synchronization

FIGURE 4.5: SuspendedMap Synchronization

FIGURE 4.6: Block and Unblock GSearch Synchronization

been completed and cannot go through the unblocking process again. The first *GSearch* cannot get a chance to be active again.

4.3.4 On-the-fly Parallel Fairness Verification

We introduced the definition of fairness assumption and our major efforts on the on-the-fly parallel fairness checking in previous sections. Our fairness checking is based on the exploration of SCCs. When the concurrent LTL model checking algorithm detects the existence of an SCC, it needs to generate the complete SCC. Shown in Figure 4.7, our key idea is to overlap part of the fairness checking process with the SCC generation process and do on-the-fly parallel checking so as to increase the performance. For

instance, we overlap the generation of the sets *enabledEvt*, *enabledPro*, *engagedEvt* and *engagedPro* which base on the complete exploration of the SCC. Then we do parallel exploration to deal with these sets to generate the verification result. If any thread finds the condition that makes the SCC fair/unfair, the checking process terminates. In specific, given an SCC *Scc*, we present the detail design for *ESF/PSF* as a sample. We also specify the *SGF*.

Sequential Algorithm: The algorithm explores every component s_i in *Scc*. For each state, the algorithm gets its engaged event list and add to the *engagedEvt/Pro(Scc)*. Then it explores each component again to find which component has any event $\alpha \in \text{enabledEvt/Pro}(s_i)$ but $\notin \text{engagedEvt/Pro}(Scc)$. These states are regarded as bad states. Then the algorithm removes these bad states from *Scc* to get *Scc'*, and calls the modified Tarjan's algorithm again to check whether there is an SCC in the *Scc'*. If so, *Scc* is fair. Otherwise it is not fair.

Parallel On-the-fly Approach: *engagedEvt/Pro(Scc)* are generated during the generation of *Scc*. Bad states are generated and removed from the SCC concurrently with the *lock*. Each thread takes charge of several components of *Scc*. The new SCC is used to call the modified Tarjan's Model Checking algorithm. The modified Tarjan based model checking algorithm is not the concurrent version mentioned in Section 4.2 because the overhead generated by concurrency may be more expensive than the advantages provided as the size of a single SCC may be limited.

For *SGF*, the sequential algorithm explores every component s_i in *Scc*. When s_i is being explored, it generates the *enabledEvt(s_i)*, in which it contains a set of event ID. Then the algorithm explores all successors of s_i . For successors that also $\in Scc$, in the corresponding events that lead to the transition to these successors, if the ID of any event $eid \in \text{enabledEvt}(s_i)$, *Scc* is fair. For *SGF*, there is no need to generate a set that based on the complete exploration of *Scc*, so this algorithm doesn't contain the on-the-fly part. In the parallel algorithm, shared memory is used to store *Scc*, and a fixed number of threads start to check all components. Checked components are marked. If any thread finds the component that makes *Scc* fair, it broadcasts to other threads and all threads terminate.

4.3.5 Algorithm

In this section, we present the algorithm of the concurrent on-the-fly SCC detection for LTL model checking with fairness assumption synthesizing our descriptions in the preceding sections. We show them in Algorithm 7, Algorithm 8 and Listing 4.1.

We describe the startup of our approach in Algorithm 7. *SearchParams* is defined in line 1. Line 2 is a *Sequential* state space generation process. This process generates some states which are stored in the *StealingList*. It should be guaranteed that no duplicates exist. *Scheduler* starts in Line 3 to create *GSearch*. The number of threads is the same as the initial number of *GSearches*. All *GSearches* work concurrently.

We present the algorithm of *GSearch* in Algorithm 8 and Listing 4.1, which is the key algorithm of our approach. All operations are based on *TarjanNode* instead of the original state. In Algorithm 8, all *GSearches* access the shared memory to visit M , $\mathcal{A}_{\neg\varphi}$ and *SearchParams*. All key differences compared to sequential algorithm are shown with the *underline*. Each *GSearch* has its own *TaskStack* and *StepStack*. In line 1, *GSearch* gets its initial state and converts it to an initial *TarjanNode*. The iteration for SCC detection starts in line 3. Line 4 is to set the local *preorder* of *TarjanNode* in a certain *GSearch*. Line 5 and 10 lead to two conditions shown below:

(1) *From line 5 to 9.* If the *TarjanNode* T is in *SearchParams.visited*, it has been expanded. Then in lines 6 to 9, *GSearch* gets and chooses the available successor to push into *TaskStack*, which is handled in the module *Add New Task*. We present that in Listing 5.1. We judge whether the successor *TarjanNode* belongs to other *GSearches* in line 1. If not, we just mark the first available successor with *GSID* and push it into *TaskStack* in line 3 to 6. If the successor already exists in *SearchParams.StealingList* but not been occupied by any *GSearch*, we should update its *GSID*. If the successor belongs to another *GSearch* and hasn't being completed, *GSearch* is suspended to this successor in lines 11 to 14. It updates the *SuspendedMap* and detects whether this operation generates a blocking cycle. If so, it breaks this cycle, refers to Figure 4.2 in Section 4.2.

(2) *From line 10 to 17.* If the visiting *TarjanNode* T is not expanded, *GSearch* generates successors based on T by the interleaving between M and $B_{\neg\varphi}$ (line 11). It

Algorithm 7: Startup of Concurrent on-the-fly SCC detection for Automata-based Model Checking

Input: M, B, φ
1 Define Shared *SearchParams*, Local *StepStack*, *TaskStack*;
2 Duplicate Elimination: Sequential LTL Model Checking $\rightarrow$ *SearchParams.StealingList*;
3 Start a thread for *Scheduler*; *Scheduler*: create $GSearch(i) \leftarrow SearchParams.StealingList[i]$;
4 Start n threads;
5 *Scheduler*: allocate $GSearches$ to all threads to work concurrently;

updates the $OGTrans[T]$ without duplication (line 13). This operation should use the *lock* to avoid conflicts. For each *TarjanNode* in $SuccTN[]$, *GSearch* synchronizes the value of it with the data in *StealingList* (line 14). It checks whether the *TarjanNode* has been expanded before or been occupied by other *GSearch*. If it is a new *TarjanNode* without being marked with any *GSID*, the module *Add New Task* works as mentioned. Line 17 marks T as expanded. During the process from line 5 to 17, if all successors of T have been expanded, it comes to line 19. It is to update the *lowlink* of T and is same to the sequential algorithm.

```

1  if ( $T_i.GSID = -1$ ) {
2      if (done) {
3           $T_i.mark(GSID)$ ;
4          lock: Update SearchParams.StealingList;
5          TaskStack.push( $T_i$ );
6          done = false; break;
7      }
8  }
9  else {
10     if ( $T_i.status \neq completed$ ) {
11         lock: Update SearchParams.SuspendedMap;
12         Check Block Cycle, lock:break cycle;
13         lock: Update SearchParams.BlockList;
14         Wait for Resume;
15     }
16 }
```

LISTING 4.1: Add New Task

GSearch generates an SCC in lines 20 to 26. The status of all *TarjanNodes* in SCC is marked as *completed*. All *GSearches* that being suspended to these *TarjanNodes* are resumed to be ready for scheduling. If fairness checking is required, the concurrent on-the-fly fairness checking works in line 27. It should be noted that in line 22, the fairness checking has started and overlaps with the generation of SCC. This is part of our concurrent on-the-fly fairness checking, which can improve the performance of

Algorithm 8: GSearch-Concurrent on-the-fly SCC detection for Automata-based Model Checking

```

Input:  $M, B_{-\varnothing}, SearchParams$ 
1   $T = TarjanNode(SearchParams.StealingList[GSID]);$ 
2   $TaskStack.push(T \leftarrow Mark(T, GSID));$ 
3  while  $\neg TaskStack.Empty$  do
4      Get  $T$ , Update  $T.preorder$ ;  $done = true$ ;
5      if  $T \in SearchParams.visited$  then
6           $T'[] = SearchParams.visited[T];$ 
7          forall the  $T_i \in T'[]$  do
8              if  $T_i.preorder = false$  then
9                  Module:Add New Task;
10         else
11              $succTN[] = T.MakeOneMove(M, A_{-\varnothing});$ 
12             forall the  $T'_i \in succTN[]$  do
13                 lock and AvoidDup:  $SearchParams.OGTrans[T].add(T'_i)$ ;
14                 lock: Sync  $T'_i$  with  $SearchParams.StealingList$ ;
15                 if  $T'_i.preorder = false$  then
16                     Module:Add New Task;
17              $SearchParams.visited.add(T);$ 
18         if  $done$  then
19             update  $lowlink$ ;
20             if  $lowlink == preorder$  then
21                  $SCCSet.add(T);$ 
22                 backtrack  $Stepstack \rightarrow SCCSet \& Fairness$ ;
23                  $SearchParams.SCCList.Add(SCCSet);$ 
24                  $T'_i.complete = true$ ;
25                 Resume  $GSearches$  suspended to  $T'_i \in SCCSet$ ;
26                 lock: Update  $SearchParams.PendingList$ ;
27                 Option: Concurrent On-the-fly Fairness Checking;
28                 if  $ISFair$  then
29                     Record Result; Generate Counterexample;
30                 forall the  $T'_i \in SCCSet$  do
31                     lock:  $SearchParams.visited.remove(T_i)$ ;
32                     lock:  $SearchParams.OGTrans.remove(T_i)$ ;
33             else
34                  $StepStack.push(T);$ 

```

fairness checking. If the detected SCC satisfies the fairness assumption, on-the-fly LTL model checking enters the counterexample generation process. Line 31 and 32 work as we always need to generate all SCCs. In line 34, if the SCC is not detected, *GSearch* pushes T to the local *StepStack*.

4.3.6 Complexity

To compare with the sequential Tarjan's algorithm and Lowe's concurrent version with the unrooted mode, we discuss the complexity of concurrent on-the-fly SCC detection without the complexity of state space generation. Given a transition system with N

FIGURE 4.7: Parallel On-the-fly Fairness Checking

nodes and E edges, the sequential Tarjan's algorithm shows the complexity of $O(N + E)$ and Lowe shows the complexity of concurrent Tarjan's algorithm is $O(N^2 + E)$. Our concurrent on-the-fly SCC detection has additional node transfers for the construction of global outgoing transition relationships and the StealingList mentioned before. The complexity of these part is $O(N)$ since we use lock and hash table to avoid duplicates. Thus, the complexity of our concurrent on-the-fly SCC detection without the complexity of state space generation is also $O(N^2 + E)$.

4.4 Implementation and Evaluation

We implement our approach using $C\sharp$ in Process Analysis Toolkit (PAT) [109]. We call it *PAT-C*. We evaluate the performance of *PAT-C* by comparing it to the original PAT with sequential on-the-fly LTL model checking and fairness checking, which we call *PAT-O* and *PAT-OF*. We conduct our experiments on a laptop with an Intel(R) Xeon(R) CPU E5-1650, 3.2GHZ, 12 cores, 16GB RAM.

In our experiments, $|S|$, $|T|$ and $|SCCs|$ represent the number of states, the number of transitions and the number of SCCs, respectively. $LSCC$ and $SSCC$ represent the size of the largest SCC and the smallest SCC. SP means the speedup. For the testing models, *MsS* represents the *Miners Scheduler*. *CC* represents the *Consensus with*

TABLE 4.1: Evaluation of LTL Model Checking (time in sec)

Model	Proc	S	T	SCCs	PAT-O	PAT-C	SP
MsS	1000	$9 * 10^3$	$1.8 * 10^4$	0	222.6	207.3	1.1
MsS	1200	$2.6 * 10^4$	$5.3 * 10^4$	0	611.9	523.4	1.2
MsS	1500	$3.3 * 10^4$	$6.7 * 10^4$	0	1104	626	1.76
CC	4	$1.5 * 10^4$	$7 * 10^4$	$5.2 * 10^3$	1.05	1.57	0.67
CC	5	$5 * 10^5$	$3.3 * 10^6$	$1.6 * 10^5$	49.7	31.9	1.6
ABP	100	$2.1 * 10^5$	$7 * 10^5$	1	10.8	23.2	0.5
ABP	200	$8 * 10^5$	$2.7 * 10^6$	1	53	92	0.58
ABP	300	$1.8 * 10^6$	$6 * 10^6$	1	150.2	247	0.61
Lift.1	3,2	$3.2 * 10^5$	$7.2 * 10^5$	1349	39.6	24.7	1.6
Lift.2	3,2	$3.8 * 10^5$	$8.7 * 10^5$	1569	45.8	33.4	1.4
Lift.3	3,2	$4.6 * 10^5$	$1.5 * 10^6$	983	55.6	32.6	1.7

Crashes. *DP* represents the *Dining Philosophers*. *ABP* represents the *Alternation Bit Protocol*. *KvR.1/2* represents the *K-valued Register.12*. *MLS.12* represents the *Multiple Lift System 12*. *TBM* represents the *DBM Testing*.

4.4.1 Evaluation on SCC Detection for Model Checking

We conduct the performance evaluation of multiple LTS models on a range of state space. We initialize 12 *GSearches*. We start 12 threads in parallel to handle these *GSearches* concurrently since our machine owns 12 cores. It should be noted that to thoroughly evaluate the performance of our approach, in our experiments we find all SCCs instead of just one. We involve four models in this part: *MsS*, *CC*, *ABP* and *Lift*. We show the experimental results in Table 4.1: (1) the product between *MsS* model and the Büchi Automaton generates no SCCs. Hence, our approach works as a concurrent complete state space generation process. We use 1000, 1200 and 1500 processes for our experiments. From Table 4.1, we can observe: (1) Our approach gains up to 2x speedup compared to the sequential LTL model checking, which is significant since the original execution cost is fairly large. 2) The product between *CC&Lift.1/2/3* model and the Büchi Automaton can generate a large number of small SCCs. Results in Table 4.1 show that with a large number of small SCCs, our approach can also gain around 2x speedup compared to the sequential LTL model checking in PAT. 3) The product between *ABP* model and Büchi Automaton generates just one large size SCC. Results in Table 4.1 show that the performance with these kind of models is expected to be even slower than the sequential LTL model checking in PAT.

TABLE 4.2: Testing Model for Fairness Checking

Model	Proc	$ S $	$ T $	$ SCCs $	LSCC	SSCC
DP	8	$1.4 * 10^5$	$1.08 * 10^6$	51	$3 * 10^4$	1
DP	9	$5.3 * 10^5$	$4.5 * 10^6$	66	$1.1 * 10^5$	1
DP	10	$1.9 * 10^6$	$1.8 * 10^7$	83	$4.2 * 10^5$	1
Peterson	4	$5.1 * 10^4$	$2.1 * 10^5$	5	$4.5 * 10^4$	1
Peterson	5	$1.4 * 10^6$	$7.2 * 10^6$	17	$1.3 * 10^6$	1
CC	4	$1.2 * 10^4$	$5.4 * 10^4$	$3.8 * 10^3$	2	1
CC	5	$3.8 * 10^5$	$2.5 * 10^6$	$1.2 * 10^5$	2	1
KvR.2	4,3	$6.2 * 10^5$	$2.5 * 10^6$	1042	$4.1 * 10^5$	104
KvR.2	5,3	$3.7 * 10^6$	$1.5 * 10^7$	2748	$2.3 * 10^6$	256
KvR.2	4,4	$3.8 * 10^6$	$1.8 * 10^7$	6342	$2.4 * 10^6$	104
MLS.1	2,2,3	$3.8 * 10^5$	$1.9 * 10^6$	940	$2.7 * 10^5$	4
MLS.1	3,2,2	$1.2 * 10^6$	$5.1 * 10^6$	1416	$9.8 * 10^5$	4
MLS.1	2,2,4	$1.5 * 10^6$	$9.3 * 10^6$	3243	$1.1 * 10^6$	4
MLS.2	2,2,3	$6.5 * 10^5$	$3.5 * 10^6$	3029	$3.5 * 10^5$	1
MLS.2	3,2,2	$2.2 * 10^6$	$9.5 * 10^6$	5457	$1.5 * 10^6$	1
MLS.2	2,2,4	$2.7 * 10^6$	$1.6 * 10^7$	8345	$1.4 * 10^6$	1
DBM	2,4	$4.2 * 10^4$	$1.7 * 10^6$	1	$1.4 * 10^4$	-
DBM	2,5	$9.5 * 10^4$	$4.5 * 10^6$	1	$3.2 * 10^4$	-
DBM	2,6	$1.9 * 10^5$	$1.1 * 10^7$	1	$6.3 * 10^4$	-
PMC	50,1000	$1 * 10^6$	$2 * 10^6$	51	$2 * 10^4$	$2 * 10^4$
PMC	60,1000	$1.2 * 10^6$	$2.4 * 10^6$	61	$2 * 10^4$	$2 * 10^4$
PMC	80,1000	$1.6 * 10^6$	$3.2 * 10^6$	81	$2 * 10^4$	$2 * 10^4$

In conclusion, our approach can generally improve the performance of automata-based LTL model checking. The performance is better for the larger state space. The performance decreases at *ABP* model since it has only one large SCC in the state space. Thus, based on our approach, all *GSearches* easily encounter *TarjanNode* in other *GSearches*. According to the cycle breaking rules, our approach works like a sequential algorithm as all data is transferred to one *GSearch*. The parallelism cannot be well utilized and a lot of time is consumed at transferring nodes. Finally, the concurrent on-the-fly model checking is more suitable to models with a large amount SCCs and trivial average SCC size so as to reduce the cost of suspending, which is also indicated in [67].

4.4.2 Performance of On-the-fly Parallel Fairness Verification

We conduct the experiments on fairness checking using different models with different SCC numbers and SCC sizes. For each type of fairness, we choose three models, and we perform the experiments independently. We show the features of test models in Table 4.2. These models differ in the number and size of SCCs, which can help reflect the features of our approach. In this part, the parallelism started is less than 11 threads, which is adjusted dynamically based on the scheduling of the Microsoft .Net platform.

TABLE 4.3: Evaluation of ESF&EWF Checking (time in ms)

Model	Proc	ESF Checking			EWF Checking		
		PAT-OF	PAF-C	SP	PAT-OF	PAF-C	SP
DP	8	$1.1/1.5 * 10^3$	48/304	25/5	-	-	-
DP	9	$1.5/1.6 * 10^4$	81/1218	188/13	-	-	-
Peterson	4	29	30	0.97	-	-	-
Peterson	5	$1.1 * 10^3$	267	4.3	-	-	-
CC	4	3	3	1	3	95	—
CC	5	137	157	0.87	157	3415	—
KvR.2	4,3	-	-	-	119	4	29.7
KvR.2	5,3	-	-	-	716	16	44.8
KvR.2	4,4	-	-	-	750	38	19.7

In our experiments, we measure the time cost in milliseconds. Firstly, we compare our approach to the sequential fairness checking in PAT. We show the results of *ESF*, *EWF*, *PWF* and *PSF* checking in Table 4.3. We can see that our approach for fairness checking can gain significant speedup for most models¹. The performance improvement is more visible when the state space consists of a lot of SCCs and the average SCC size is large. We observe a performance decrease at the *CC* model because all SCCs in the state space are minuscule, with the size of 1 or 2. For *Peterson*, where the state space consists of just 5 SCCs, the performance is also decreased. Under this condition, the overhead of parallelism is more dominant than the advantage of parallelism. Secondly, as we mentioned in Section 4.3.5, our approach generates the sets for fairness checking together with the SCC generation, so it introduces costs to the SCC generation process. We count the cost of the entire SCC generation and Fairness checking for some models, which is shown on the right side of “/” in Table 4.3. We can see that our approach can still gain significant speedup. The reason why the speedup is not as high as the results on the left side of “/” is that the SCC generation cost a lot compared with the fairness checking. We can also conclude that the parallelization of fairness checking can bring more performance improvements for weak fairness type than strong fairness type. The reason is that the weak fairness checking does not have to update any shared variables (bad states), therefore *lock* operation is not necessary. In contrast, in strong fairness checking, a *lock* exists. During the execution, some threads need to wait for other threads for the right to access.

We show the results of SGF checking in Table 4.5. For SGF checking, as we mentioned in Section 4.3.5, the complete checking process works in parallel after the

¹Here all models represent the state space generated from the product between the model and the Büchi Automaton.

TABLE 4.4: Evaluation of PSF&PWF Checking (time in ms)

Model	Proc	PWF Checking			PSF Checking		
		PAT-OF	PAF-C	SP	PAT-OF	PAF-C	SP
DP	8	50	1	50	1327/1705	14/364	95/5
DP	9	224	1	224	1.7/1.8 * 10 ⁴	48/1497	364/11
MLS.1	2,2,3	153/1728	4/1270	38/1.4	187	71	2.6
MLS.1	3,2,2	892/6100	3/5100	298/1.2	617	161	3.8
MLS.1	2,2,4	733/7256	9/5453	81/1.35	819	288	2.9
MLS.2	2,2,3	257/2797	4/2019	64/1.3	340	219	1.6
MLS.2	3,2,2	1432/12601	19/10613	77/1.2	1.1 * 10 ³	475	2.4
MLS.2	2,2,4	1290/13382	16/11800	80/1.2	1.4 * 10 ³	833	1.7

TABLE 4.5: SGF Checking (time in ms)

Model	Proc	PAT-OF	PAT-C	SP
DBM	2,4	1.7 * 10 ³	353	4.8
DBM	2,5	5.4 * 10 ³	1.1 * 10 ³	4.95
DBM	2,6	1.5 * 10 ⁴	2.9 * 10 ³	5.1
PMC	50,1000	1.1 * 10 ³	370	3.2
PMC	60,1000	1.7 * 10 ³	453	3.7
PMC	80,1000	1.9 * 10 ³	605	3.1
CC	4	1	83	-
CC	5	77	2.7 * 10 ³	-

generation of SCC. Hence, we just record the time of fairness checking and compare it to the sequential version. We can see besides the *CC* model with very small SCC, our approach gains around 5x speedup for the *DBM* and around 4x speedup for the *PMC*.

Finally, we compare our experimental results with the previous results of Liu, Sun and Dong [209], which also works on the parallel on-the-fly LTL model checking in PAT. In [209], the major Tarjan process is taken in one thread and SCC generation is taken in parallel by forking new threads to handle it. The performance of both the approach in [209] and this chapter depends on the ratio of SCCs and the average SCC size. The differences are that the on-the-fly LTL model checking in [209] works better than sequential algorithm only with the large number of large size SCCs. Our approach can also gain performance improvement without any SCC, which is not possible for the approach in [209]. Large number of small size SCCs may influence the fairness checking process in both of these approaches. But for large sized SCCs, the approach shown in this chapter may have better performance improvement than PAT as we start the exploration from different directions and the process is on-the-fly. In conclusion, the approach in this chapter can be suitable for much more types of models and gain more performance improvement than the approach in [209].

4.5 Conclusion

In this chapter, we expand the concurrent Tarjan's algorithm based on the complete state space and develop a concurrent on-the-fly SCC detection for the automata-based (LTL) model checking with fairness checking. To this end, we build a novel abstract data structure for the concurrent LTL model checking taking data consistency into account. Besides, we develop a parallel on-the-fly fairness checking approach for different types of fairness assumptions. We also implement our proposed approach in the practical model checker PAT. Our experiments show that our approach can improve the performance of both the SCC detection and the fairness checking compared to the sequential version in PAT model checker.

Chapter 5

GPU Accelerated Counterexample Generation in LTL Model Checking

5.1 Introduction

The LTL model checking problem is known as the emptiness checking of the product between M and $B_{\neg\varphi}$, where M represents the system model and $B_{\neg\varphi}$ represents the Büchi automaton that expresses the negation of an LTL property φ . The emptiness checking detects whether there exists an execution path in the product that can be accepted by the Büchi automaton. There are two main streams of LTL model checking approaches: Nested Depth First Search (NDFS) and Strongly Connected Component (SCC) search, where the latter one is more suitable to handle fairness assumptions.

SCC based verification algorithms aim to find an SCC with at least one accepting state. If such SCC exists, it means that there is a run that can be accepted by the $B_{\neg\varphi}$, i.e., the violation of the LTL property φ . To generate a counterexample in such cases is to produce an infinite path $\pi = \rho_1\rho_2\rho_3$, which consists of three parts: a path ρ_1 from the initial state to a state s in the SCC, a path ρ_2 from the s to an accepting state a in the SCC and a loop ρ_3 that starts and ends at a . To generate such a path, currently, some algorithms [144, 185] work on DFS-related solution with high complexity. Some work on BFS-related solution, e.g., [143], which focus on building the minimal size counterexample to deal with the memory constraint.

In this chapter, we propose an approach that has the potential to accelerate the counterexample generation process using GPU. The problem here is equivalent to building a solution to improve the performance of BFS with path recording. Compared to the multi-core CPU architecture, GPU typically has a lot more cores and high memory bandwidth, which potentially provides high parallelism. Because the number of nodes in each layer of BFS is changing, it makes the resource allocation and the load balancing a challenging task in GPU-based BFS searching. In CUDA programming model, CPU launches the kernel in GPU with static grid and block structures, which result in the lack or waste of compute resources. In previous research such as CUDA IIIT-BFS [145], it is necessary to launch the kernel each time when the BFS starts a new layer. It is costly and even slower than the CPU-BFS in some cases. To deal with this problem, CUDA UIUC-BFS [122] has been proposed based on a hierarchical memory management solution. It builds a three-level queue for BFS to avoid consequent kernel launching, which offers certain speedup. But it is still a static method that cannot adjust according to the task size. Furthermore, there is no load balance approach in it.

In this work, we propose an almost CPU-free BFS based path generation process by leveraging on the new dynamic parallelism feature of CUDA. The key problem addressed is the number of tasks during BFS based path generation is dynamically changing. In this chapter we propose four contributions. (1) Compared to related works of parallelizing BFS for model checking problems, our approach is totally CPU-Free. Existing related works allocate GPU resources in a static way. The resources can be re-allocated only by CPU when the execution of a kernel ends and launches a new kernel. For irregular graphs, it is costly and not flexible. Our approach presents a runtime resource adjustment approach for BFS and can be tailored for model checking problems. (2) We propose an approach to build a dynamic parent-child relationship scheduler and a dynamic hierarchical task scheduler for dynamic load balancing. (3) We develop a three-level queue management to fit the dynamic parallelism and dynamic BFS layer expanding. Based on it, we propose a dynamic path recording approach, which helps with the duplicate elimination in BFS at the same time. The hierarchical memory structure of GPU is fully utilized for data accessing. (4) We implement our approach in PAT model checker and evaluate them to show the effectiveness of our

approach.

5.2 Related Works

In the area of model checking, as the verification problem can be transformed to a graph search problem, there exist many works on accelerating model checking algorithms with CUDA. [182] focuses on the duplicate detection in external memory model checking. It utilizes the GPU to accelerate sorting process in duplicate detection in BFS and builds a delayed duplicated detection on GPU. In [101], the authors propose a design of maximal accepting predecessors algorithm for accelerating LTL model checking in GPU. [184] accelerates the state space exploration for explicit-state model checking by utilizing GPU to do the breadth-first layered construction. [103] shows how the LTL model checking algorithms can be redesigned to fit on the many-core GPU platforms so as to accelerate LTL model checking. [19] focuses on the on-the-fly state space exploration in GPU and proposes several options to implement this. All these research has proved that CUDA compute architecture can be well utilized in solving model checking problems. In this chapter, different to previous research of which most are based on a static way to allocate computing resource in advance and involve the CPU frequently, we build an approach for counterexample generation which can completely put the work in GPU and dynamically fit the feature of BFS. Then the dynamic parallelism and memory hierarchy from the latest CUDA architecture-*Kepler GK110* and its corresponding GPU device serve as the basis of our design.

5.3 CUDA Accelerated Counterexample Generation

The overall design of our approach is presented in Figure 5.1. *Host (CPU) counterexample generation* represents the process in Algorithm 1. We build a general path generation approach to reach the target of function *Init2SCCBFS*, *Path2AccBFS* and *SelfLoopBFS* in Algorithm 1 based on different input. Our approach for handling the BFS based path generation is presented as *CUDA (GPU) Path Generation*. The

FIGURE 5.1: Overall Design

complete counterexample generation process in Algorithm 1 can be replaced by executing GPU-based BFS for three times.

Our approach consists of two parts: *Parent Kernel* and *Child Kernel*. The overall process is described as follows. CPU launches the parent grid to execute the parent kernel. The parent kernel starts the BFS based path generation to generate one or more new layers of tasks. When the task size exceeds the number of threads in the parent grid, it launches the child grid to execute the child kernel. Then the child kernel starts to do path generation and records path data. After generating a layer, the task scheduler checks whether any warp or block being overload. We define *overload* as the number of tasks exceeds the thread number in the GPU or no more tasks can be added to the BFS queue. Tasks rescheduling starts to do load balancing within the child kernel. If the whole child grid is *overloaded*, it returns to the parent. The child kernel stops running. The resources of the child grid are released. Then the parent kernel reallocates tasks, launches a new child grid to execute the child kernel and distributes tasks to it. The process continues until the “goal” being reached. The “goal” means terminating condition. The relationship between the parent grid and the child grid is dynamically adjusted according to the number of tasks. In order to maximize the parallelization, by default, each thread is asked to do the BFS and path recording for one state in BFS queue. It means that the number of tasks in each layer of BFS decides the number of threads needed, so as to decide the structure of the child grid. This dynamic relation ends at the time the process of our approach ends.

The dynamic parallelism is used to deal with the dynamic task size so as to make the execution flexible. Other features of CUDA programming model and *Kepler GK110*, such as the hierarchy memory, are integrated into each part. Our solution utilizes the latest GPU features to provide a novel counterexample generation solution.

5.3.1 Detailed Approach

We present Algorithm 9 and Algorithm 10 in this section for the parent kernel and the child kernel based on the process in Figure 5.1. They follow the CUDA dynamic parallelism programming model presented in *pages 141 to 159* in [138]. Note that in CUDA, there are build-in objects *blockIdx* and *threadIdx* to record the ID of block and the ID of thread in each block. But there is no object to represent the ID of threads in warp. It can be calculated directly as the warp is built in sequence, meaning that threads with index $0 \sim 31$ belong to warp 1.

To simplify the presentations of the two algorithms, we abstract the common part in both of two algorithms in Listing 5.1. It corresponds to line 10 in Algorithm 9 and line 8. The details are introduced together with the algorithms.

Algorithm 9 corresponds to the parent kernel executed in the parent grid, named by *CudaParentCounterexampleGeneration*. It focuses on the task schedule and parent-child relation management. In Algorithm 9, the input variable *init* means the initial state. *TerminatingCondition* is a Boolean function which decides whether the algorithm should terminate at the current state. The condition in our approach means that the path generation process reaches any target state in the target states set, which can be an SCC or an accept state list based on the input of each process in Algorithm 1. Line 1 presents two types of thread ID mentioned above. Line 2 presents the two types of queues used in the algorithm. *WarpQueue* is an array of queues that represents the task queue for each thread. *WarpPathQueue* has the same structure, which is to record the path to the target state. They are all allocated dynamically in SM. The details structure and operation rules can be referred to Section 5.3.2 and 5.3.4. Lines 3 and 5 are the first step of path generation. The initial state and initial path record are added to the queue of the first thread in the block. Here the path record is a tuple with two components:

Algorithm 9: Cuda Parent Counterexample Generation Algorithm

```

Input: init, TerminatingCondition,  $\rightarrow$ 
1 inblocktid = threadIdx.x; inwarptid = inblocktid%32;
2 Define: WarpQueue, WarpPathQueue in SM;
3 if inblocktid = 0 then
4   | WarpQueue[0].enqueue(init);
5   | WarpPathQueue[0].enqueue((-1, init));
6 CUDA-API: __syncthreads();
7 while TRUE do
8   | if WarpQueue[inwarptid]  $\neq \emptyset$  then
9   |   | S  $\leftarrow$  WarpQueue[inwarptid].Dequeue();
10  |   | Shared Code with MemoryOption = SM
11  | if inwarptid = 0 then
12  |   | if |WarpQueue| > WARPQUEUE_SIZE then
13  |   |   | Intra_Warp_task_transfer;
14  | if inblocktid = 0 then
15  |   | if |TasksInBlock| > InitialT then
16  |   |   | break;
17  |   | else
18  |   |   | Inter_Warps_task_transfer;
19 CUDA-API: __syncthreads();
20 if  $\neg$ TerminatingCondition(anyState) then
21   | if inblocktid = 0 then
22   |   | ChildSizeCalculation(EXPAND_LEVEL);
23   |   | write WarpPathQueue to GM;
24   |   | write WarpQueue to GM with Duplicate Elimination;
25   | while  $\neg$ TerminatingCondition(anyState) do
26   |   | if inblocktid = 0 then
27   |   |   | Generate Tasks Distribution Offset;
28   |   |   | Launch ChildKernel, Transfer tasks in GM to Child Grid;
29   |   |   | CUDA-API: cudaDeviceSynchronize(); //If Child returns to Parent;
30 CUDA-API: __syncthreads();

```

(*Predecessor*, *StateID*). The function shown in lines 6, 19 and 30 is CUDA build-in API for intra-block synchronization [138]. The loop from line 7 to line 18 is the major path generation process in the parent kernel. The condition to break the loop is that the parent grid being overloaded. Line 8 means that the thread works when its queue is not empty. In line 9, the thread gets the task *S* from its queue, then line 10 mentions the *Shared Code*, which is the abstraction of the BFS and counterexample generation related work. The *Shared Code* is presented in Listing 5.1.

In Listing 5.1, line 1 is the target state detection. When the path generation of any thread reaches any state in the target states set, path records stored in *WarpPathQueue* in SM are copied back to GM (using atomic operation *atomicAdd*) in line 2 and this information is broadcasted through *MemoryOption* in line 3. For Algorithm 9, *MemoryOption* is set to *SM*. Then other threads stop running and the thread which detects the terminating condition backtracks to generate the full path in line 4.

```

1      if (TerminatingCondition(anyState)) {
2          write WarpPathQueue[inwarptid] to GM;
3          broadcast to other threads through MemoryOption;
4          Iterativebacktracking  $\rightarrow$  FullPath;
5          break;
6      }
7      Snew = NewLayerTaskGeneration(S);
8      if (|WarpPathQueue[inwarptid]| = WARPPATHQUEUE_SIZE) {
9          write WarpPathQueue to GM;
10         WarpPathQueue[inwarptid].enqueue({S, Snew});
11     }
12     WarpQueue[inwarptid].enqueue(Snew);
13     if (inwarptid = 0) {
14         Transfer tasks among queues in WarpQueue;
15     }
16

```

LISTING 5.1: Shared Code

Successors generation in line 7 is based on S and $\rightarrow$. In line 12, new successors S_{new} are added to the queue of corresponding thread in *WarpQueue*. Lines 8 to 10 are to record path information. Path records are stored in *WarpPathQueue* in SM firstly, when the element number in the queue exceeds a constant *WARPPATHQUEUE_SIZE*, it is copied back to GM (using atomic operation *atomicAdd*). The constant *WARPPATHQUEUE_SIZE* means the configured size of each queue in the array *WarpQueue*. The record in GM can also work as the preparation for future duplicate elimination, which is detailed in Section 5.3.4. At the beginning, only thread 0 has tasks in its queue. So lines 13 and 14 are to involve other threads in the same warp by transferring tasks to the threads with empty queue, which is done in the central mode by the first thread in a warp.

Back to Algorithm 9, lines 11 to 13 perform load balancing within a warp. Lines 14 to 18 are the inter-warps load balancing and the process to check whether the parent grid is overloaded. We define a constant *INITIAL_T* to represent the number of threads in parent grid. Lines 21 to 24 work on the calculation of the size of child grid firstly. Then it transfers data from SM to GM in order to transfer data from parent kernel to child kernel. Note that line 24 shows that a duplicate elimination approach takes an action when copying back the content in task queue from SM to GM, which utilizes

the path record information in GM. Details are also shown in Section 5.3.4. Line 27 shows that parent kernel needs to calculate the task distribution offset, which records the tasks storage index in GM for each block in the child grid. We define a constant *EXPAND_LEVEL* to represent the growth rate of the parallelism to launch a new child grid, which is shown in lines 28 and 29. The loop from lines 25 to 29 indicates that parent kernel works as a scheduler to reallocate the child grid to execute the child kernel iteratively. This loop breaks only when the path generation detects any target state.

Algorithm 10 corresponds to the child kernel executed in the child grid in Figure 5.1. Functionally, it works on the path generation and the task schedule approach is also implemented in it. In Algorithm 10, variables or functions with the same name as in Algorithm 9 have the same meaning. The tasks in GM and the *Distributionoffset* generated in Algorithm 9 are the inputs. In line 1, *globaltid* represents the first thread among all blocks. Line 2 defines two variables in GM for communication among threads in different blocks. A loop from lines 3 to 32 is the major executing process. The break conditions of the loop are that the path generation detects any terminal states or the child grid being overloaded. Lines 4 and 5 are for each warp to get its own tasks and push to the queue of each thread in balance. This is based on the *Distributionoffset*. Lines 6 to 8 are shared code with *MemoryOption* being *SM+GM*. The full generated path is “returned” to the parent kernel through GM. Lines 10 to 22 are the intra warp and inter-warps load balancing. Lines 23 to 32 are the process to check whether the whole child grid is overloaded and whether inter-blocks load balancing is needed. These three load balancing approaches make up the complete hierarchical task scheduling. And they can be regarded as three levels schedule: Warp level, means task adjustment among threads in a warp; Block level, means task adjustment among blocks of the child grid; Grid level means returning the control to parent. Block level and Grid level need to copy the content in task queue to GM with the duplicate elimination approach. It decides at which level task scheduling is taken dynamically. Lines 24 and 32 represent the invocation of the inter blocks synchronization interface. It is not CUDA built-in API. This is described in following parts.

Specifically, Algorithm 9 is designed to be executed among threads in a block, while Algorithm 10 is to be executed among threads in multiple blocks. This is because the

Algorithm 10: Cuda Counterexample Generation Algorithm of the Child Kernel

Input: *Tasks, DistributionOffset, TerminatingCondition, $\neg$*

```

1  globaltid = blockDim.x * blockIdx.x + threadIdx.x;
2  Define WarpQueue, WarpPathQueue in SM Child_return2Parent, ChildSynNeed in GM;
3  while  $\neg$ TerminatingCondition(anyState) or Child_return2Parent do
4      if inwarptid = 0 and interblocktaskschedulehappens then
5          WarpQueue[0...31].enqueue(GetTasks(Tasks, DistributionOffset));
6      while WarpQueue[inwarptid]  $\neq$  0 do
7          S = WarpQueue[inwarptid].dequeue();
8          Shared Code with MemoryOption = SM + GM
9      CUDA-API: __syncthreads();
10     if inwarptid = 0 then
11         if |WarpQueue[0...31]| > WarpQueueSize then
12             InWarpad justment = true;
13         if TasksInWarp > 32 then
14             InBlockad justment = true;
15             Ats = AvailableTaskSize;
16     if inblocktid = 0 then
17         if TasksInBlock > ThreadNumInBlock then
18             ChildSynNeed = TRUE;
19         else if TasksInBlock  $\leq$  ThreadNumInBlock and InWarpad justment then
20             Intra_Warp_task_transfer;
21         else
22             Inter_Warps_task_transfer  $\Leftarrow$  Ats;
23     CudaInterBlocksSyn();
24     if ChildSynNeed then
25         if globaltid = 0 then
26             if TasksInChild > ThreadNumInChild then
27                 Child_return2Parent = TRUE;
28                 write WarpQueue to GM;
29             else
30                 write WarpQueue to GM;
31                 Inter Blocks Task Scheduler;
32     CudaInterblocksSyn();

```

parent kernel focuses on task rescheduling while the child kernel focuses on the path generation.

Some other functions are cited for Algorithm 9 and 10: function *CudaQuicksort* utilizes the dynamic parallelism feature of CUDA [138] to do quick sort for preprocessing the target states set. *CudaInterBlocksSyn* refers to the algorithm mentioned in [177]. It is for inter-blocks synchronization as CUDA does not supply API for this.

Synchronization and Atomic Operation In our algorithm, synchronization happens in each layer expanding by default as the algorithm need to do load balancing. After any task scheduling, synchronization is needed to make sure that each thread gets its own tasks correctly. In previous algorithms, the atomic operation can be used to work

FIGURE 5.2: Dynamic Three-level Queue Management

as the *lock*. When some threads want to write the same memory address at the same time, only the first one which calls the lock gets the access right and others discard their write operations and continue their executing.

5.3.2 Dynamic Three-level Queue Management

As discussed in Section 2.5, GM can be read or written by all blocks running in different SMX, and SM is just available to blocks running in the same SMX. Read or write operations in SM cost much less than operations in GM. But the size of SM is much smaller than GM. Since our algorithm refers to huge data size, we cannot avoid accessing GM. However, as our tasks are distributed to the parallel threads, we can utilize SM to accelerate local data accessing. Considering that our algorithm is building dynamic parent-child relationship, we need a dynamic task distribution. We build a dynamic hierarchical queue to utilize the hierarchical memory. In order to fit our dynamic parallelism design, we build a three-level queue management approach, shown in Figure 5.2. The first level queue is stored in SM, i.e., *WarpQueue* in Algorithm 9 and 10. The second level queue is stored in GM, denoted as *GQueue*. The third level queue is also stored in GM, named Virtual Global Queue, denoted as *VGQueue*. For simplicity, we denote *GQueue* and *VGQueue* as GM in Algorithm 9 and 10.

Here, as there are many threads working together, the problem of read-write conflict, which means parallel threads write or read at the same time, should be considered in the queue structure and the design of task schedule approach. One potential solution is to use the lock or atomic operation to prevent the conflicts, which leads to a huge cost

with frequently write requests at the same time. Another potential solution is to use the lock-free structure. We take two types of lock-free structures into consideration: 1) As mentioned, the Kepler GK110 contains four warp schedulers in a single SMX, i.e., 4 warps can run in parallel. We build a lock-free queue with 4 sub-queues so as to avoid the conflict. However, it is hardly feasible because the warp scheduling in GPU is not visible to us. Therefore we adopt the design as showed in Figure 5.2. In each block, no matter in the parent grid or the child grid, we make the first-level queue in SM a dynamic sub-queue set based on the warp number in one block. As shown in Figure 5.3 part A, each *WarpQueue* consists of 32 queues, which is due to the size of warp so as to make it lock-free. We want to guarantee one thread holds only one expanding task. Thus, if the task size in a block exceeds the number of threads, the tasks are re-scheduled and transferred to *GQueue* in GM.

In Figure 5.2, *GQueue* is built at the first time when the parent grid launches a child grid, it is also a group of arrays shown in Figure 5.3, part B. As the parent grid communicates with the child grid via GM, which is also the way blocks communicate with each other, it is used to transfer tasks to the child grid for the execution of the child kernel. In the following execution, *GQueue* stores the tasks when blocks being overloaded or the task reschedule among the blocks in the child grid is needed. As in the global view, the tasks stored in the *GQueue* is not continuous, *VGQueue*, shown in Figure 5.3, is dynamically built as the third level and it is used for sequential accessing tasks data. This three-level queue follows the rules of dynamic parallelism, aiming at building a flexible way of data accessing and improving the performance. It works for the task schedule and can completely match the *Parent-Child* structure.

5.3.3 Dynamic Hierarchical Task Schedule

As the size of tasks during the execution changes dynamically, unbalanced load or overload happens frequently, especially for an irregular graph. Launching kernel is an expensive work. So we cannot rearrange the structure of the child grid at each time that the unbalanced load happens. Flexible task scheduling methods are necessary. Combined with our path generation problem, there are several conditions that the

FIGURE 5.3: Structure of WarpQueue, GQueue and VGQueue

program needs to do task scheduling in hierarchical level. Algorithm 9 lines 11 to 18 and Algorithm 10 lines 10 to 32 are related to these:

- The first time to launch the child grid from the parent grid. When the parent kernel finishes some layers of BFS-related path generation and makes that the parent grid cannot hold more tasks, the parent grid needs to launch the child grid and the schedules initial tasks to the child grid for the child kernel.
- The inside warp task transfer to make each thread has tasks in its queue. When each warp begins the execution after getting tasks, it needs to guarantee that each thread is involved in the path generation procedure.
- When the whole number of tasks in a warp make a warp overload, it needs to do inter warps task transfer. This is similar to the inside warp data transfer.
- When the number of tasks in a block make it overload, inter blocks task rescheduling occurs.
- When the whole tasks in the child Grid make it overloaded, the child kernel stops executing and the control returns to the parent grid to rearrange the child grid so as to reschedule the tasks. This and the inter blocks one are shown in Figure 5.4. Both the inter blocks schedule and the parent grid schedule utilize *GQueue* and *VGQueue* GM to redistribute tasks. However, inter warps or inside warp schedule is based on SM.

FIGURE 5.4: Dynamic Hierarchical Task Schedule-block and grid level

These make up a hierarchical fine-grained task scheduling. As many steps are in SM, it can make full use of the fast access feature. Only the child grid being overloaded could cause the structure of the child grid to be rearranged. Normally, we arrange the grid size of child large than needed at the beginning. Thus we set a constant *EXPAND_LEVEL* to make the size of grid and block bigger than required, i.e., $INITIAL_T \times EXPAND_LEVEL$ in Algorithm 10. The *EXPAND_LEVEL* is based on the restriction of GPU architecture, which is mentioned in Section 5.4. It is to make a compromise between resource cost and rescheduling cost. As the decision to do which level task rescheduling is due to the runtime task size, our design is a dynamic hierarchical task schedule method.

Note that after each layer of path generation, the overload detection occurs. This, together with the terminating condition detection, work in a central mode. This is to get rid of frequent communication among threads. When the whole block is overloaded and needs to copy tasks in each *WarpQueue* back to *GQueue*, each warp does its own transfer, which makes it a parallel data transfer. Here, the targets of the task scheduling are to balance the workload in each warp/block, as well as to allocate enough resources for future execution.

5.3.4 Dynamic Duplicate Eliminated Path Recording

Our algorithm is to deal with the counterexample generation, where path recording is necessary. Path recording should also be parallelized. As our approach performs BFS,

the counterexample path is updated in each layer. Note that our path recording is to record the visited state ID and its first *Predecessor*. The “first Predecessor” means the firstly recorded predecessor. In fact, our algorithm is to find a path to reach the target set, thus one predecessor for one state is enough to generate a complete path. Take Figure 5.5 as example, record (2,4) and record (3,4) are not recorded together, just (3,4) is recorded as it is reached earlier.

Combined with our previous design, the path recording is happening in two levels: (1) warp level in SM, each warp owns a *WarpPathQueue*, which is mentioned in Section 5.3.1. (2) block level in GM, path recording is taken under three conditions: When the number of records in *WarpPathQueue* exceeds the configured *WARPPATHQUEUE_SIZE*, it is executed independently in each warp and mentioned in line 9 in Listing 5.1. Another two conditions are that path recording is taken before the task being copied back to GM or after the terminating condition being detected, mentioned in lines 23 in Algorithm 9 and line 9 in Listing 5.1. The structure for path recording in this level is two arrays. One is the *path_recording_array*, the index of array represents the ID of state and the value represents the predecessor. The other is the *predecessor_visited_array*, different from the first array, its value represents if the corresponding state is visited. The example of this procedure can be shown in Figure 5.5. When the record (1,2) is copied back to GM, it is recorded as *path_recording_array*[2] is 1 then *predecessor_visited_array*[1] is *true*. For record (n,3), as *predecessor_visited_array*[3] is *true*, this record is discarded. But *predecessor_visited_array*[n] is marked with *true*. Atomic operations are used for writing these two arrays.

We call this approach *Dynamic Duplicate Eliminated Path Recording*. The duplicate elimination here does not mean duplicate path record elimination. It is for duplicate BFS tasks elimination. When the tasks being copied back to *GQueue*, it should first detect if the corresponding value of task state in *predecessor_visited_array* is *true*. If so, this state is not copied back to GM for following task reschedule. So it reaches the duplicate elimination target to some extent. It is mentioned in Algorithm 9 and 10 when the algorithms proceed to write *WarpQueue* to GM.

FIGURE 5.5: Block-level Path Recording

When the terminal states being detected, we need to generate the full path, which is mentioned in line 4 in Listing 5.1. The process starts from the target state reached by path generation process, marked as s . The iteration is started to find predecessor of s by getting value $prec(s) = predecessor_visited_array[s]$. This terminates when $predecessor_visited_array[s] = Init$. We generate the full path by recording each $prec(s)$ during the iteration. Atomic operation is also needed in getting the full path. Overall, our path recording also fits the idea of dynamic parallelism.

5.4 Experiments and Evaluation

We evaluate our algorithms in two aspects. Firstly, we test the performance of our dynamic CUDA counterexample generation with models in different size and structures. Secondly, we analyze the effects of GPU parameters to our algorithms and discuss the limitation of the algorithms. We also propose two optimization options. The basic implementation of our algorithm uses C++. The system model is from PAT model checker [109]. Our experiments are conducted using a PC with Intel(R) Xeon(R) CPU E5-2620 @ 2.00GHz and a Tesla K20c GPU @ 2.6 GHz with 5GB global memory, 13 SMXs and totally 2496 CUDA cores.

TABLE 5.1: Parameters in the Algorithms

Parameter	Meaning	Default Value
<i>INITIAL_T</i>	The thread number of parent	32
<i>WARPQUEUE_SIZE</i>	The length of queue in WarpQueue	32
<i>WARPPATHQUEUE_SIZE</i>	The length of queue in WarpPathQueue	32
<i>EXPAND_LEVEL</i>	The times of thread number to expand compared to statistic requirement	2

5.4.1 Performance Analysis

To analyze the performance, we choose the classic dining philosophers problem (DP) as the input model. We use different number of processes to get different SCC size. The four GPU parameters used in the algorithms and their default value are shown in Table 5.1. The value of the parameters should be controlled in a fixed range based on hardware specifications. Their influence on our task schedule and their restrictions are discussed in next section.

Based on the default configuration, our algorithms succeed in generating the counterexample for the verification of DP model in sizes from 5 to 8. We record the execution time for each process in parent kernel (Algorithm. 9), as well as the execution time of child kernel (Algorithm. 10). Firstly, Figure 5.6 shows the distribution of the execution time for each BFS work in Algorithm. 1. *Init2SCCBFS*, *Path2AccBFS* and *SelfLoopBFS* are the three steps mentioned in Section 2.3.3.1. We can see that the first path generation costs more than the other two. This is because the $\rightarrow$ (outgoing transition table) for the first path generation is large since it contains all transitions generated during the model verification. Thus, the schedule, dynamic expanding and data transfer cost more. When doing $scc \rightarrow acc \rightarrow accloop$, the $\rightarrow$ is much smaller as we are preprocessing to eliminate non-SCC states in the $\rightarrow$.

We choose the data from the execution of parent kernel for the *Init2SCCBFS* path generation, as well as the total cost of the execution of child kernel. We get the results of the execution time percentage of each part, as shown in Table 5.2: *Schedule* means the task schedule; *Search* means the BFS with path recording; *Prepare* means the queue build-up for launching child grid; And *Child* means the execution time of child kernel. We can see that the child kernel takes charge of the highest percentage during the counterexample generation. In Parent, its major cost is on the initial schedule and the preparation for the child expanding. We can see the costs of each part are

FIGURE 5.6: Distribute of cost in three path generation

TABLE 5.2: Performance Analysis

Processes	TotalSize	SCCSize	AccSize	Schedule	Search	Prepare	Child	DataTrans	Total
5	348	120	36	20.7%	20.6%	20.5%	34.8%	0.57	9.3
6	1013	508	112	21.9%	22%	23.3%	30.8%	0.6	8.3
7	3420	2047	365	20.7%	20.6%	22.6%	35.5%	0.64	16.2
8	12339	7980	1195	24.4%	24.2%	25.4%	26.8%	1.01	19.7

balanced among different size of tasks. The experimental results match the design of our algorithms.

5.4.2 Evaluation and Limitation

As shown in Table 5.1, there are four constants which affect the performance, which are mentioned in Section 5.3.1. Firstly, the value of *INITIAL.T* is due to the structure of the state space of the model. If the model's width is always short, setting a large *INITIAL.T* can reduce the chance to launch *Child Grid*. A large value may waste a lot of resources when the Child is working on the major process. Secondly, based on our algorithm design, the hierarchical task scheduler is based on the grid size, means the number of threads. However, as the task size of each layer during the path generation of an irregular graph is unknown, if the size of one layer is larger than the remaining space of the *WarpQueue*, the task reschedule may occur, which is costly. So for the models with irregular state space, *WARPPATHQUEUE.SIZE* affects the performance. Thirdly, as the path records in warp level need to be copied back to GM when the $|WarpPathQueue|$ exceeds the *WARPPATHQUEUE.SIZE*. So set a large value to *WARPPATHQUEUE.SIZE* definitely reduces the cost. At last,

EXPAND_LEVEL, as we mentioned, if we just set the exact size of child grid according to the realistic requirement, it may cause the Child being overloaded soon and the Parent do rescheduling again. *EXPAND_LEVEL* is to make the compromise. It decides how much more resources to be allocated to the Child.

However, the size of the queue needs to be bounded. All above are restricted by the size of SM in each SMX. As described in Section 2.5, the size of the chip memory in each SMX is 64KB. According to the max SM per multiprocessor, only 48KB are available for the SM. Before we launch the kernel, we need to decide how many SMs a block can use. In our algorithms, each item in the queue is an *int*. We represent the total shared memory cost as *MemC*, as defined below:

$$\begin{aligned} MemC = & sizeof(int) \times (|WarpQueue| + |WarpPathQueue|) \\ & \times PG(INITIAL_T) \times EXPAND_LEVEL \end{aligned} \quad (5.1)$$

It requires $MemC < 48KB$. $PG(INITIAL_T)$ denotes the statistic required size of resources (thread number), which starts from the *INITIAL_T*. This is a dynamic variable so we combine the $(PG(INITIAL_T) \times EXPAND_LEVEL)$ to be *MaxWarpsize*. We can learn from Equation 5.1 that these parameters are conditioned by each other. Considering the restriction of CUDA architecture, the available size of SM is set before launching the kernel. If queue size described above is too large, the number of threads in one block is restricted. During the execution, data in the *WarpQueue* is flushed in each layer as old data being visited, and the size of data in the *WarpPathQueue* is increasing all the time. Thus, the constants *WARPPATHQUEUE_SIZE* and *WARPQUEUE_SIZE* decide the extra number of GM accessing. In fact, the value of all these parameters should be decided based on the structure of model.

The values of these parameters are also related to the grid level task scheduling (Parent Grid *launches* Child Grid) in our design. We take the default setting in Table 5.1 as an example. Suppose the total task size currently is T_{total} . The structure of child grid, means blocks number in grid, is marked as B_c . We guarantee that each block in child grid starts with tasks $T_s = 32$, equals to the thread number in a warp. Then $B_c = T_{total} \div T_s (+1)$. With the setting, each block owns $ExpandLevel \times T_s = 64$ threads, means 2 warps. With these, total shared memory cost in a block TC_B

is: $TC_B = 64 \times (WarpQueueSize + WarpPathQueueSize) \times sizeof(int) = 16384bytes$. Compared to *MemC*, it means two more warps can be added in a single block. and $B_c \leq 13(SMX\ number)$. Thus, the maximum number of threads available under this setting is 1664, means if any layer in a graph contains more than 1664 states, the schedule cannot work.

In summary, due to the restriction of SM size in CUDA, our approach does not work well for graphs with large branching nodes. To solve this problem, we use more global memory or build the united memory space with host memory, which has been proposed in the new CUDA 6.0.

5.4.3 Optimization Options

The experiments show that our approach is scalable in dealing with the counterexample generation problem. In our CUDA Dynamic Path Generation algorithm, the task schedule, as well as the queue building, take a substantial on the total cost. Based on this, we present two optimization options as follows. (1) According to [184], GPU works fast on small pieces of data. So building a compact graph representation to capture the model can improve the performance significantly. (2) Reduce the numbers of scheduling and global memory accessing. These can be done by applying a latency task schedule. To make each thread hold more tasks, we perform the load balancing after several layers of expanding. The intra-block and warp level task schedule, which bring the low cost, should be conducted more, which means we should increase the threshold to do inter-blocks or parent level schedule. These potential optimizations are important in the improvement of our algorithm and can be easily supported based on current design.

5.5 Conclusion

In this work, we proposed a GPU-based dynamic counterexample generation approach for SCC-based LTL model checking. We designed the dynamic queue management

scheme, the hierarchical task scheduling scheme, the dynamic parent-child relation scheme, the path recording scheme by adopting the new features of dynamic parallelism of CUDA. The experiments show that our algorithm can be scalable in solving the counterexample generation problem. In future work, we plan to optimize this algorithm to build a compact encoding for the task data and path record data in order to save resources.

Chapter 6

A GPU-based Value Iteration Algorithm to Accelerate the Computation of Reachability Probabilities in MDPs

6.1 Introduction

Markov decision processes (MDPs) have been extensively studied in the literature for planning and decision making under uncertainty [42, 50]. A common problem in MDPs is the computation of reachability probabilities, which is an important subroutine of finding approximately optimal policies of MDPs. Value iteration [163] is a well-known method to solve this problem. However, value iteration is computationally expensive both in terms of time and space [132]. Hence, improving the computational performance of value iteration is a key research challenge.

Nowadays, graphic processing units (GPUs) are widely used to accelerate execution performance of various computational methods in many areas [14, 20, 93, 167, 222]. GPUs have several advantages over CPUs such as high memory bandwidth, computation capability, and massive parallelism. In this chapter, we take advantage

of these features to develop a parallel value iteration algorithm that utilizes GPUs for the efficient computation of reachability probabilities, and further for approximately optimal policies of MDPs. Different from previous research, the main target of our approach is to use the algebraic features of an MDPs' representation matrix to develop a parallel value iteration algorithm that runs more efficiently than sequential value iteration on GPUs. To this end we analyze both the matrix representation of MDPs and the potential of parallelization in the computation process. Then we define the independent sub-matrices according to the different actions of the MDP, build the compact data structure and design the efficient GPU based parallel value iteration procedure, which also decides the convergence of value iteration in a distributed manner.

To evaluate our approach, we compare it with sequential value iteration and topological value iteration (TVI) [148]. The experimental results show that we can achieve a $10\times\sim$ speedup compared to sequential value iteration, and our approach outperforms TVI in most of the cases. For MDPs which do not contain strongly connected components (SCCs) with more than one state, or which contain a small number of large SCCs, our approach achieves up to $17\times$ speedup comparing to TVI. We also illustrate the superiority of our approach compared to the existing GPU-based value iteration methods.

Our main contributions are: (1) We take advantage of the algebraic structure of MDPs to define action-based matrices and corresponding data structures for efficient parallel computation of reachability probabilities on GPUs. (2) We develop an efficient parallel value iteration algorithm for computing reachability probabilities. We utilize features of modern GPUs, such as dynamic parallelism and memory hierarchy. (3) We present an extension of our approach to find approximately optimal policy for standard MDPs and its variants. We also discuss a potential extension to POMDPs.

6.2 Related Work

In the literature there are two major approaches to improve the efficiency of the value iteration method. The first approach is utilizing reachability information and heuristic

FIGURE 6.1: Example of MDP

functions to avoid storage of the complete state space of MDPs in value iteration. *LAO** [77] utilizes heuristic functions to guide the expansion of an explicit graph iteratively based on some best-first strategy. LRTDP [29] expands the Real-Time Dynamic Programming (RTDP) by setting a ‘solved’ tag to a state for which Bellman residuals are smaller than a convergence threshold, and also to the states which are reachable through the optimal policy from this state. Weld and Geffner [10] introduce the generalized stochastic shortest path MDPs and a heuristic algorithm, which supports more general reward structure comparing to many well-established MDP types. Most of these approaches cannot compute the complete optimal policy of MDPs. Furthermore, they do not consider the inherent features of MDPs as we do in our approach. Our approach also does not do any initial estimate for the value function.

The second approach is to reorganize the structure of an MDP to simplify the problem or optimize value iteration based on graphical features of MDP. SPUDD [92] uses ADD structure to represent the MDP and build VI algorithm for ADD. TVI [148] introduces an improvement of VI, which utilizes the graphical features of the strong connected components (SCC) of the MDP to construct an acyclic MDP in order to perform the VI backups in the best order and to only perform backups when they are necessary. TVI may be the most related work to ours as we both base on the value iteration algorithm for complete state space backups. The difference between TVI and our approach is that TVI performs based on the structure of SCCs in MDPs, while our approach utilizes the algebraic feature of MDP that related to the representation matrix of MDP and the matrix-vector multiplication during the Bellman backup process. In addition, our approach does not only work well for MDPs with a certain SCC structure.

There exist several works on the parallel value iteration method. Peng [36] presents a preliminary short report which conducts the parallelization by simply distributing each state to a thread. However, it misses any concepts on the usage of GPU techniques. LVI [117] mentions their usage of parallel value iteration briefly, which is also a state-wise parallelization presented in [24]. Besides, [24] presents another fine-grained parallelization, which is to directly parallel all subtasks in the value iteration process. The deficiency is that it requires frequent synchronization in one iteration, which is costly. It introduces some optimization option while many technical details are missed. By contrast, we presents the formal definitions and complete ideas with up-to-date GPU techniques in this chapter. Our approach analyzes the algebraic features of MDPs. We utilize the GPU optimization techniques to define the reasonable parallelization granularity. Our approach not only fully utilizes the high parallelism of GPU, but also avoids the frequent synchronization among threads. We also construct the new compact data structures for MDPs. We present the detailed comparison in Section 6.4. Archibald [198] addresses the parallelization of value iteration on CPUs by considering load balancing. The paper is outdated and major focuses on the discussion of different algorithms working for different problems. [118] and [135] work on parallel value iteration for POMDPs, which is beyond the main scope of this chapter.

To the best of our consideration, our approach is novel to fully utilize the algebraic features of MDPs and parallel computing techniques in GPU together to deal with the value iteration problem for the computation of reachability probabilities and further find the approximately optimal policies of MDP.

6.3 GPU Accelerated Value Iteration For Computing Reachability Probabilities

The main goal of our approach is to build a GPU accelerated parallel value iteration to compute the reachability probabilities of an MDP, which can significantly improve the efficiency compared to the sequential VI. To transform the sequential VI into an

FIGURE 6.2: Parallelization

efficient parallel algorithm, we take advantage of the algebraic structure (matrix) of an MDP and the matrix operation involved in the Bellman backup process.

In sequential VI, the complete state space should be backed up in each iteration, which requires the exploration of all states sequentially. More specifically, the term $\sum_{s' \in S} P(s' | s, a) \cdot x_{s'}^{(n-1)}$ in (Equation 2.1) requires the sequential VI to compute the reachability probabilities for each state by exploring each enabled action and reachable states. This exploration process creates a bottleneck for sequential VI, since the time required for the exploration process grows exponentially with respect to the state space of an MDP. Hence, if we can accelerate the exploration process, we can significantly improve value iteration's efficiency.

An MDP can be represented using a matrix structure M . For instance, the MDP on the left-hand side of Figure 6.1, which involves two states and two actions, can be represented by a two by two matrix M as we show on the right-hand side of the figure. In M , each row represents the outgoing transitions from a state (e.g., Row₀ shows the transitions from S_0). Each row involves a set of sub-vectors (Sub_{a_0} & Sub_{a_1}), where each sub-vector represents the immediate reachability probabilities of the states with respect to actions. For instance, if action a_0 is taken in state S_0 , the probability of reachability both to S_0 and S_1 is equivalent to 0.5.

In value iteration, the calculation of Equation 2.1 updates x_i by selecting the action a that maximize/minimize the vector to vector multiplication $Sub_{a_i} \times X$. This computation is independent in each state (Row_i). e.g., we calculate $Sub_{a_0} \times X$ in state0 (Row_0), which has no interaction with the calculation in state1(Row_1). Furthermore, we collect the vector to vector multiplication of the complete state space together, and then divide by

the actions. We can find in global perspective that computation is indeed a substantial amount of synchronized sub-matrices to vector multiplications. The sub-matrices are the matrices representation of MDPs by actions. Thus, we define the *Action-based Matrices for MDP*.

Definition 6.1. Action-based Matrices Given a MDP $M = (S, s_{init}, Act, P, R)$ and the representation matrix M , for each action $a \in Act$, an action-based matrix is a tuple $(M_a = \{S_a, S'_a, P(S' | S, a)\})$, where S_a is the set of states in which action a is activated, S'_a is the set of states reached via action a from states in S_a , and $P(S' | S, a) \rightarrow (0, 1]$ is the probability array.

Each M_a is a m_a by m_a matrix, where $m_a = |S_a|$. Intuitively, an M_a represents the transition relationship of the states in S_a with respect to action a . Using the action-based matrices, the value iteration process can be transformed into several interleaving action-based matrix to vector multiplications and subsequent minimisation, where each action-based matrix to vector multiplication is an independent computation. As a result, the action-based matrices and the described partitioning of matrix operations allow us to develop an efficient parallel value iteration that works well on GPUs as we define in the rest of this section.

6.3.1 Parallelization

We present the details of our parallelization process for value iteration in Figure 6.2. We show the *part A* of Figure 6.2 as M' based on the matrix representation M in Figure 6.1. M' is an adjacency list representation for MDPs, where $Succ_a(0)$ is the immediate reachable states. Each list in M' , e.g., $State0 \rightarrow index_a(0) \rightarrow Succ_a(0)$, represents a sub-vector in M , e.g., Sub_{a_0} . According to Definition 6.1, it is indeed mapped to a row in the action-based matrices M'_a .

The overall parallelization is shown in *part B* of Figure 6.2. Given state s , for each $a \in Act(s)$ the sum $\sum_{s' \in S} P(s' | s, a) \cdot x_{s'}^{(n-1)}$ can be computed by the multiplication of each row in M'_a and the vector X in parallel. We call this an action level calculation. The minimisation $\min_{\alpha \in Act(s)}$ is performed along a synchronisation step.

Furthermore, the multiplication between each row in M' and the vector X is also an independent computation. We call this a state level calculation. Accordingly, a two-level (state+action) parallelization can be constructed. It requires the consideration of both the size of state space and the number of active actions in each state, which is due to the dynamic structure of MDPs. In addition, we must achieve a high parallelism so as to fully utilize GPUs.

To this end, we define the concept of *workgroup*. It is the basic runtime threads organization and computation unit in our approach. We build it by grouping a number of threads, which equals to the value of W . Here W means that from each state there are W active actions (e.g., 2 in Figure 6.1). It should be noted that threads inside a workgroup should not exceed a warp. GPU architecture introduces that *no synchronization* operation is required for intra threads in the same warp. Basically, a workgroup handles the state level calculation. Each thread in one workgroup handles the action level calculation, which refers to the list in *part A* of Figure 6.2. We can assign large number of workgroups in runtime based on the number of rows in M' .

The description of parallelization is based on M' while the real computation on GPU is combined with a lot of matrices calculation based on actions. Thus, we should make the data structure flexible for accessing in GPU. On the other hand, the size of GM in GPU is also limited. (e.g., 6GB in our work). A more compact data structure is promising. Section 6.3.2 describes all about the data structure in detail.

6.3.2 Compact Data Structure and Data Accessing

We show the data structure in this section in Figure 6.3 to save the memory cost and make it convenient for GPU computing. In *part A* of Figure 6.2, we present the adjacency list. Accordingly, we present the representation of action-based matrices M'_a in *part A* of Figure 6.3. Then we build the compact data structure for GPU computing in *part B* of Figure 6.3.

Each M'_a is a sparse matrix derived from *part A* of Figure 6.2. It is expressed in the common Compressed Row Storage (CSR) [134]. The first unsigned integer array

FIGURE 6.3: Parallelization

$ArrayS$ directly stores the set S_a , which should be in order of the state id. The value of $ArrayS[i], i \in [0..|S_a|]$ indicates that action a is activated in state $ArrayS[i]$. Thus, M'_a is referred in the update of $x^n_{ArrayS[i]} \in X$. $ArrayIndex$ (integer) stores the indices of successors for each $s \in S_a$, and $ArraySucc_a$ (integer) and $ArrayProb_a$ (double) store the successor states S'_a and probability $P(S'_a|S_a, a)$.

Based on the *part A* of both Figure 6.2 and Figure 6.3, we realise that if we directly do calculation based on the set of action based matrices, we need a large memory space. That is due to that we need to store the index for each action, as well as the ‘attendance’ in each state. To reduce the memory cost, we reorganize the action-based matrices to build the compact structure shown in *Part B* in Figure 6.3. We call it as an *AMDP* (*Action-based structure for MDP*).

The data structure should also be convenient for the accessing from the workgroup. According to the task distribution for a workgroup, the compact structure is organized in rows. It is a collection of the action-based row from all action-based matrices. The number of rows is $|S|$. The first Integer array is the *Rowindex*. The second row is the successors unsigned integer array $ArraySucc$. In each Row_n of the $ArraySucc$, it consists of the row data of action-based matrices which has $ArrayS[i] = n$. We define the action-based row data as an action-based *segment*. To avoid storing the index for each action-based segment, we use *inter-index*, which is encoded in the first several bits of the first element. We store the probability in the third double array, which is a bijection with the $ArraySucc$. *Rowindex* and $ArraySucc$ are static data thus we can use TM for its storage to reduce the accessing cost. It should be noted that in the compact structure, we ignore the identification of actions, which is based on the fact that the number of threads in one workgroup is only related to W and we only need the final result about *max/min*.

FIGURE 6.4: Vector Distribution

With the data structure, we should concern about the data accessing. The AMDP is a static data in the value iteration. In GPU, we can use TM for its storage to accelerate the accessing rate. On the other hand, we previously describe the vector X that needs to be updated in each iteration. It is an array with size $|S|$ and is shared among all threads in GM. As it is frequently accessed, we should try to reduce the accessing cost. To this end, we distribute X in the hierarchical memory. Shown in Figure 6.4, as we refer to the Jacobi method [85] for the backups after each iteration, we introduce V and $TmpV$ in GM. They are the arrays to store the complete vectors for backups. We introduce $PValue$ and $SValue$ in SM. They are the arrays with the partial vector so as to increase the rate of fast local accessing. The length of $PValue$ and $SValue$ is equivalent to the number of workgroups in each block. The operations based on X is discussed in the next part.

6.3.3 Parallel Backups and Convergence Reduction

The vector X is backed up and the convergence judgement occurs in each iteration. Based on the distribution of X in hierarchical memory, we can figure out that all partial vectors in SM are the operation data of different GPU blocks. Hence our approach handles these two operations in parallel. For the backup operation, we conduct it in workgroup-level parallelization. Normally, each workgroup works on backing up one element of X in each iteration. However, with the hardware restriction on the number of threads in each block, the number of workgroups in each block is fairly limited. Hence when $|S|$ is large, the parallel task distribution requires that one workgroup works on backing up multiple elements in each iteration. We introduce the *RowRound* in Figure 6.4 to indicate the tasks for each workgroup. When we update V and $TmpV$, we

Algorithm 11: Host Program for Initialization

Input: $M, T, \text{Bound}, \text{Precision}$

```

1 Define  $\text{RowRound}, \text{Startblocks}, IT$  ;
2 Define  $\text{AM}[], V, \text{TmpV}, \text{AMDP}, \text{BlockStatus}[]$  ;
3  $\text{AM}[] = \text{ReadMDP2AMatrices}(M)$ ;
4  $\text{AMDP} = \text{ConstructAMDP}(\text{AM}[])$  ;
5 Init & Copy data to Device GM;
6  $\text{Calculate}() \rightarrow \text{Startblocks}, \text{RowRound}$ ;
7  $\text{ParentKernel} < 1, 1 >()$ ;
8  $\text{CUDA API: cudaThreadSynchronize}()$ ;

```

read/write these two arrays alternately in any two continuous iterations. It is to avoid the cost of the copying data between them. In order to improve the efficiency, we conduct the coalesced accessing for this operation. For the convergence judgement, we conduct it in both workgroup-level and block-level parallelization. The block does not converge until all workgroups converge. Value iteration terminates when all blocks do not report “un-converged”.

6.3.4 Algorithm

In this section, we present a complete GPU-accelerated value iteration algorithm synthesizing our descriptions in the preceding sections. We build our algorithm using the *dynamic parallelism* [140] feature of CUDA, which we have introduced in Figure 2.4. Accordingly, our algorithms consist of three parts: the *host program* on CPU, the *parent kernel* and *child kernel* on GPU.

We propose the workflow of the host program in Algorithm 11. It constructs the data structure and initializes several parameters for the GPU computation. Parent kernel is launched in the host program. The input M is the original representation of MDP. T is the set of target states. *Bound* and *Precision* are the parameters of convergence for value iteration. We define the data structures required for GPU value iteration (lines 1 & 2), construct the *AMDP* (lines 3 & 4) and initialize their memory space in the GM of GPU (lines 5 & 6). We have introduced the concept of $V, \text{TmpV}, \text{RowRound}$ in the previous section. *BlockStatus*, *Startblocks* and *IT* are defined and initialized for the parent kernel.

We show the workflow of parent kernel in Algorithm 12. It handles the logic control of our GPU-accelerated value iteration. *BlockStatus* in GM indicates the status of *block convergence* for each block, on which child kernel works. *IT* is the count of iteration

Algorithm 12: Parent Kernel for Logic Control

Input: AMDP, V , $TmpV$, $BlockStatus$, $RowRound$

```

1 Define  $Startblocks$ ,  $IT$ ;
2 while TRUE do
3    $IfConv \leftarrow ChildKernel <Startblocks, 512>(BlockStatus)$ ;
4   CUDAAPI:cudaDeviceSynchronize();
5   if Convergence/out of Bound then
6     Output, break;
7    $IT++$ ;
```

Algorithm 13: Child Kernel for Computation

Input: AMDP, V , $TmpV$, $BlockStatus$, $RowRound$, IT

```

1 Define  $wgid$ ,  $wgidinblk$ ,  $tidinwg$ ,  $wgnum1blk$ ,  $Nwg$ ;
2 Define shared  $SValue[]$ ,  $PValue[]$ ,  $Conv$ ,  $Local_{low}$ ,  $Local_{up}$ ;
3 Init:if  $IT \% 2 = 0$  then
4    $PValue[] \leftarrow V$ ;
5 else
6    $PValue[] \leftarrow TmpV$ ;
7 CUDAAPI: syncthreads();
8  $c = wgid, round = 0$ ;
9  $begin = Rowindex[c], end = Rowindex[c + 1]$ ;
10 while  $c < |S|$  and  $round < RowRound$  do
11    $getFirst(begin) \rightarrow interindex, nonzero, prob$ ;
12   while  $tidinwg! = 0$  do
13      $begin++ = interindex$ ;
14      $getFirst(begin) \rightarrow interindex, nonzero, prob$ ;
15     until get its own target,break;
16   for  $j = 0; j < interindex; j++$  do
17     if  $nonzero \% Nwg \in [Local_{low}, Local_{up}]$  then
18        $tmpresult++ = Mul(PValue[], prob)$ ;
19     else
20        $tmpresult++ = Mul(V/TmpV[], prob)$ ;
21      $getNotFirst(++ begin)$ ;
22   AtomicMax(Min)( $SValue[c]$ ,  $tmpresult$ );
23   if  $BlockStatus[blockIdx.x]$  then
24      $Conv \leftarrow PartialConvergenceJudge()$ ;
25    $c++ = wgnum1blk \cdot gridDim.x, round++$ ;
26 CUDAAPI: syncthreads();
27 backup: if  $IT \% 2 = 0$  then
28    $SValue[] \rightarrow TmpV$ ;
29 else
30    $SValue[] \rightarrow V$ ;
31 if  $threadIdx.x = 0$  and  $Conv$  and  $BlockStatus[blockIdx.x]$  then
32   update  $BlockStatus[blockIdx.x]$ ;
33 else
34   report un-converged;
```

times (lines 1). $Startblocks$ is the parallelism of child kernel which is related to the number of workgroups and decide the value of $RowRound$. Parent kernel monitors the convergence status of the child kernel (line 3) in each iteration.

Algorithm 13 describes the work flow of child kernel. It handles the main computation in value iteration. The workgroup is the basic working unit. We define $wgid$ as the workgroup index in a grid, $wgidinblk$ as the workgroup index and

$wgnum1blk$ as the number of workgroups in a block. We also define the thread index in each workgroup- $tidinwg$ (line 1). Nwg is the number of workgroups in a grid¹. In line 2, we introduce $PValue$ and $SValue$ in SM for each block as the storage of partial X that initialized in $V/TmpV$, whose range is based on the workgroups' ID and $RowRound$ in this block and marked by $Local_{low/up}$. $Conv$ is a variable in SM and is delicate in each block to record the block convergence status. $PValue$ is initialized in each block by transferring data from $V/TmpV$ in GM (lines 3 to 6), which is designed as the coalesced accessing. Before the calculation, each workgroup gets the initial index of the row data in AMDP (stored in TM) that works on (line 9).

The calculation process occurs from lines 10 to 25. Each thread gets the initial index of the action-based segment in the row that it works on. This process is in parallel via the *inter-index* decoding function $GetFirst()$ (lines 11 & 14). All active threads² in the active workgroups³ calculate the reachability probabilities according to Equation 1 in a parallel manner as we explained in Section 6.3.1 (lines 16 to 21). Each thread should decide if the *nonzero* element is located in local $PValue$ so as to utilize the fast SM accessing. After generating the temporal result, each thread calls the atomic function (line 22) to backup the new x_s^n in $SValue$. Each workgroup conducts the workgroup-level convergence judgement based on the $SValue/PValue[wgidinblk]$ (line 24). If any workgroup does not converge, $Conv$ is set to false and the block does not converge. At the end of each iteration, each block needs to back up $SValue[wgidinblk]$ to $V/TmpV$ in GM (lines 27 to 30), which is also designed as the coalesced accessing. Finally, each block updates the convergence status of itself in GM (line 32). The control returns to parent kernel after one iteration.

¹In the scope of this thesis, only 1 grid is used, the multi-threading hierarchy is realised in warp and block.

²As we mentioned in Section 6.3.1, one thread in a workgroup handles one segment. Sometimes the number of segments in one row is less than the number of threads in one workgroup, thus only part of the threads are active.

³As we mentioned in Section 2.5, not all threads are executing concurrently, thus at a time, only a certain number of workgroups are active.

6.3.5 Extension for Approximately Optimal Policies

The computation of reachability probabilities can be an important subroutine to find the approximately optimal policies of common MDPs. This can be achieved by extending the value iteration that we explain in Definition 1 as follows: (1) $\pi(s)$ can be calculated by the calculation of $V^\pi(s)$. Therefore, we record the selected action according to the return value of atomic operation in line 22 in Algorithm 13. (2) Compare Equation 2&3 for optimal policy to Equation 1, we should consider the R during the calculation process (line 17 in Algorithm 13). Thus, we encode the value of $R_a(s, s')$ together with the elements in *ArraySucc* in a fixed number of bits in the same integer, where the first bit is the sign bit. Then a decoding operation is added before the calculation. And generating the policy requires the identification of each action.

Furthermore, we can also extend our approach to two important MDP variants whose solution is strongly connected to the reachability probabilities. The first variant is **MAXPROBMDP** [10], in which goal states have the reward of 1 and other states have the reward or 0. This condition is similar to that we described in Section 2.3.4.1. The extension to deal with MAXPROBMDP requires a minor modification by considering the reward for all goal states during the calculation of our GPU-based value iteration.

The second variant is **RS-MDP** [186], for which value iteration follows the equation: $\pi^* = \arg \max_{\pi} P^\pi(s_0, \theta_0)$ to get the approximately optimal policy where $P^\pi(s, \theta)$ is the reachability probabilities. The value iteration process is similar to the Bellman equation, but it is necessary to consider the threshold when selecting the actions for each state. Therefore, we should record the accumulated cost together with backup for each state. Besides the extension described in the first graph of this section, we use a fixed number of bits in each element in *ArrayS* to record the accumulated cost for each state, where $|S| = |\text{ArrayS}|$. However, the threshold can be a constant. We add decoding operation and expand the atomic operation in line 22 in Algorithm 13 to consider the value of θ .

6.3.6 Potential Extension for POMDPS

Our idea can also be applied to parallelize the Point-based value iteration (PBVI) [152], which is essential for solving POMDPs. The goal of POMDP planning is to find a sequence of actions to maximize the expected sum of rewards. PBVI works on a finite set (B) of belief points expanded from the initial belief points. Each point corresponds to an α -vector and value backups update it repeatedly. The overall process is that the belief point expansion process find the available belief points and add it to the set B , value backup works iteratively based on the equation 8, 9, 10 in [152]. The size of B may change during the process. As a result, the number of α -vector also changes.

The dynamic programming process that we define in Algorithms 12 and 13 can be applied to this process. Parent kernel handles the belief points expansion and the child kernel operates on the value backup. Data is shared between the two kernels. Parent kernel can adjust the parallelism of child kernel based on $|B|$. For POMDPs, compared to MDPs, the divergence relies on the interaction of both actions and observations. Therefore, for the value backup process, we can adjust the workgroup size or workload per workgroup to fit the computation. Besides, the information stored in the data structure should also be changed, which is straightforward.

In summary, we conduct our GPU-accelerated value iteration by further analyzing the features of both the MDPs' data structure and the computation process of the sequential value iteration method. In order to build a high efficient approach, we fully utilize the optimization technique of GPU computing to develop both the novel compact structure to represent MDPs and the efficient parallel value iteration with reasonable parallelization granularity. This section has covered all these concepts in detail and readers can reproduce this approach conveniently.

6.4 Implementation and Experiments

In this section, we evaluate the performance of our approach by comparing it with the sequential value iteration (VI) and the topological value iteration (TVI) of Dai et al. [148]. We implemented our approach in CUDA C. We used the general

implementation based on the algorithms in Section 6.3.4 for our experiments. The extension for MAXPROB MDP and RSMDP just requires minor changes in this implementation. We conducted our experiments on a server with two Intel(R) Xeon(R) CPU E5-2670, 2.60GHz, 16GB RAM and a Geforce Titan Black GPU with 13SMX, 6GB GM and 48KB SM in each SMX. The compute capability of the GPU is 3.5 based on Kepler GK110 architecture. We set the parallelism to 512 threads in one block, which can reach 100% occupancy per multiprocessor according to the CUDA Occupancy Calculator [136].

To validate the correctness of our implementation we compared our solutions for all the benchmarks with the solutions computed by PRISM. Our solutions agree with PRISM [129] up to $\varepsilon = 1 \times 10^{-10}$. Our approach always converges in a finite number of iterations.

We evaluated our approach in 3 steps : 1) we evaluate the performance speedup of our approach by comparing it with the sequential value iteration methods, including sequential VI and TVI. 2) We analyze the factors that could affect the performance of our approach, including the parallelism, the features of MDPs and the parameters involved in our approach. 3) We compare our approach with the existing parallel value iteration methods.

6.4.1 Evaluation of Performance

In this part, we compare our approach's performance with VI and TVI. Our approach is independent from the structure and initial state of MDPs. But since TVI is restricted by the structure of MDPs, we use MDPs in two types of structure for the experiments: (1) Most states in the MDPs belong to a small numbers of large SCCs or there is no SCC with more than one state. (2) The MDPs that consist of large number of small SCCs, which can be constructed into an acyclic layered structure. We use three practical problems related to MDPs namely csma¹, coin² and rabin³ as MDPs in type 1 structure,

¹The IEEE 802.3 CSMA/CD protocol is designed for networks with a single channel and specifies the behaviour of stations with the aim of minimising simultaneous use of the channel.

²The shared coin protocol of the randomised consensus algorithm of Aspnes and Herlihy [25]. coin-2/4/8 have the same structure but with different size of state space.

³The choice coordination problem (distributed consensus).

TABLE 6.1: Features of MDPs (SCC with $|S| > 1$)

Model/W/O	$ S $	$ SCC $	SCC-MAX/MIN/AVE	SCC%
csma/3/16	1.4×10^6	7	1114/19/175	0.08%
coin-8/6/2	4.6×10^6	-	-	-
coin-4/6/2	2.4×10^6	665	1.97×10^5 / 186 / 3342	93.5%
coin-2/6/2	1.25×10^6	-	-	-
rabin/4/6	5.2×10^5	0	1/0/0	0%
Layered1/6/3	3.01×10^5	301	1000/1000/1000	99.9%
Layered2/6/3	3.01×10^5	101	3000/3000/3000	99.9%

and follow the domain in TVI [148] to build the layered MDPs as type 2. The features of the MDPs are shown in Table 6.1, where W is the number of possible active actions from each state and O is the number of possible next states for each possible action in a state. The experiment results are shown in Figure 6.5¹, in which the numerical value is the computation time in seconds.

As the results in Figure 6.5 show, our approach achieves around $10x \sim$ speed up comparing to VI in MDPs with different structure. It can be observed from Figure 6.5 that TVI performs slightly worse with MDPs in type 1 structure, since under this condition, TVI cannot reduce the number of backup significantly and the SCC detection brings additional cost. This situation is also addressed by Dai *et al.* [148]. We can see our approach can achieve up to $17x$ speedup compared to TVI under this condition. For the layered MDPs, we consider two strong layered MDPs. *Layered1* only has one transition between any two SCCs. The initial state is a state in the first layer and the goal state is an end state in the last layer. *Layered2* has the same size with *Layered1*. But the number of layers is considerably less than *Layered1*. The experiment results show that for *Layered1*, TVI outperforms our GPU accelerated VI slightly due to the large number of layers. But with *Layered2*, our approach outperforms TVI with around $1.5x$ speedup since the number of layers decreases considerably. Both our approach and TVI outperforms the VI for these two MDPs.

In conclusion, our approach's performance is substantially better compared to VI for MDPs with all types of structures. We also achieve considerable speedup compared to TVI in MDPs which has small number of large SCCs or only SCCs with just one state. Although our approach performs slightly worse than TVI in MDPs with a deep

¹For readability, we divide the cost time of experiments "coin6" by 100.

FIGURE 6.5: Performance Evaluation

layer, we can still conclude from the results that our approach can be more general to a wide range of MDPs.

6.4.2 Analysis of Configurations and Parameters

In this part, we analyze the configurations and parameters that could affect the performance of our approach. Firstly, we discuss the performance with different parallelism (the number of blocks). We conducted the experiments on the “coin6” model. We record the change in the time cost by the number of blocks on a range of state space. Experimental results are shown in Figure 6.6, where x-axis is the number of blocks and y-axis is the cost of time in seconds. Our observations from the results are :

- 1) When the number of parallel blocks is constant, the computation time increases with respect to the size of the state space. It indicates the weak-scalability¹ of our approach.
- 2) To fully utilize all multiprocessors, the number of blocks should be at least as many as the number of SMX of the GPU. It is in accord with the condition that we obtain the largest time cost with ten blocks.
- 3) The decrease in computation time is not always linear to the increase in the number of blocks. As we can observe from Figure 6.5, the reduction in computation time becomes negligible when the number of blocks is increased beyond a certain threshold. This may be due to the rising cost of memory access and block synchronization, which neglects the benefit of increasing parallelism.

¹The scalability definition, can refer to [149].

FIGURE 6.6: Performance with varying nr. of blocks

W and O are the other two factors that affect the performance of our approach. W decides the parallelization based on action-based matrices as it defines the number of threads in each workgroup. $W \cdot O$ decides the computational workload in each iteration. When the iteration times is confirmed, the cost of launching kernel and block scheduling is confirmed. So the higher the computational workload in one iteration, the more benefits we can gain from the massive parallelism.

Finally, the SCC structure of an MDP also influences the performance of our approach. When an MDP has deep layers and the target states are located at the bottom layer, the workgroups that handle the backup of states in the upper layers do not contribute to the convergence of the value iteration. Accordingly, the iteration times increase significantly, and the gains that we achieve through parallelization are neglected. But this factor does not decide the performance of our approach like that happens on TVI.

6.4.3 Comparison with other GPU-based VI

In this part, compare our approach with other GPU based value iterations. Normally, the idea of GPU-based VI is a state-wise parallelization as the calculation in each state is independent of others'. However, if the MDPs are made up of a small state space

while each state has a lot of active actions with a large number of outgoings, the state-wise parallelization results in a low parallelism while each thread gets a high workload. Our approach defines the two-level parallelization based on the action-based matrices, which takes both the state and active actions of each state into consideration. Thus, our approach can be adapted to a wide range of MDP structures.

Another idea of GPU-based VI takes a fine-grained parallelization, which distributes all subtasks so as to ignore the dynamic structure of MDPs. According to the other algorithm presented in [24], the computation process in each iteration should involve four synchronization to finish the convergence judgement. However, in our approach, we define the workgroups which organize the threads inside the same warp as the basic computation unit, both the calculation of bellman updates and convergence judgement are independent in each workgroup. Hence, there is no need to do synchronization among the threads in the same workgroup. We only do a single synchronization before the backup in each iteration. We also provide the compact data structure for the MDPs. Thus we expect our approach to perform better. Unfortunately, [24] is an outdated report and many key details are missing. Thus, we cannot reproduce the algorithm for an experimental comparison. In summary, our approach also has more reasonable design compared to the existing GPU-based value iteration methods.

6.5 Conclusion

We presented a novel GPU accelerated parallel value iteration approach for the efficient computation of reachability probabilities of MDPs. The main idea of our approach is to utilize the algebraic features of MDPs to divide the computation process of reachability probabilities into partitions, which can be computed in a massively parallel manner with an efficient parallelization granularity on GPUs. To achieve this we constructed a compact data structure using action-based matrices, and designed an efficient parallel value iteration algorithm using the dynamic parallelism model of GPUs. We also presented an extension of our approach to find approximate optimal policies of MDPs. We compared the efficiency of our approach with sequential and topological value iteration. Our results show that we achieve $10\times\sim$ speedup compared to sequential,

and up to 17x speedup compared to topological value iteration. We also illustrate the potential advantages of our approach compared with the existing GPU-based value iteration methods.

Chapter 7

Qubet: A Model Checking, Bisimulation Checking and Emulation Tool for Quantum Communication Systems

7.1 Introduction

Quantum cryptography can provide unconditional security; it allows the realization of cryptographic tasks that are proven or conjectured to be impossible in classical cryptography. The security of quantum cryptographic protocols is mathematically provable, based on the principles of quantum mechanics, without imposing any restrictions on the computational capacity of an attacker. The proof is, however, often notoriously difficult, partially due to the fact that a potential quantum attacker could employ *entanglement*, a brand new quantum resource that has no classical counterpart, to enhance the ability of collective attacks. Formal methods, in particular process algebra, turn out to be a valuable approach for analyzing cryptographic protocols in the classical regime. In the past decade, several quantum extensions of process algebra such as QPAlg [151], CQP [178], and qCCS [214] have been proposed for formal description and verification of quantum communicating systems. Remarkably, a bisimulation for

processes in qCCS was introduced and proven to be a congruence with respect to all process constructs [213]. Uniqueness of the solutions to recursive process equations was also established, which provides a powerful proof technique to verify complex quantum protocols.

Based on the survey, there are no tools currently to cover the model checking (including the bisimilarity checking) for quantum communication systems. Thus, we conduct the research on quantum communication systems. In this work, we present Qubet (Model Checking, Bisimulation Checking and Emulation Tool for Quantum Communication Systems), which supports the editing of quantum communication models described in qCCS, the simulation of system processes and their bisimilarity checking. Besides, similar to other famous model checker, e.g, PAT, Qubet also support the model checking algorithms for classical concurrent systems described in CSP. Thus, we also integrate our achievements in the previous research works. Qubet is designed with an extensible architecture and a user-friendly GUI which provides the users an interface to learn and monitor the behaviors of quantum processes. Users can download the installation of Qubet from our website [223].

7.2 Related Work

Software tools based on CQP have been developed in [23] and [56] to check the equivalence between quantum sequential programs as well as concurrent protocols. A semi-automatic tool was implemented by Kubota et al. [114] which is able to check the bisimilarity between quantum processes written in (a subclass of) qCCS. However, user-defined equations on density operators are required, and no GUI is provided in their tool. For the support to classical model checking algorithms, we refer to the famous model checker PAT [109].

7.3 Basic Notions from Quantum Information Theory

In this part, we briefly recall some basic notions from linear algebra and quantum information theory which are needed.

7.3.1 Basic Linear Algebra

A *Hilbert space* $\mathcal{H}$ is a vector space which is complete with respect to an inner product

$$\langle \cdot | \cdot \rangle : \mathcal{H} \times \mathcal{H} \rightarrow \mathbb{C}$$

such that

1. $\langle \psi | \psi \rangle \geq 0$ for any $|\psi\rangle \in \mathcal{H}$, with equality if and only if $|\psi\rangle = 0$;
2. $\langle \phi | \psi \rangle = \langle \psi | \phi \rangle^*$;
3. $\langle \phi | \sum_i c_i |\psi_i\rangle = \sum_i c_i \langle \phi | \psi_i \rangle$,

where $\mathbb{C}$ is the set of complex numbers, and for each $c \in \mathbb{C}$, c^* stands for the complex conjugate of c . For any vector $|\psi\rangle \in \mathcal{H}$, its length $\| |\psi\rangle \|$ is defined to be $\sqrt{\langle \psi | \psi \rangle}$, and it is said to be *normalised* if $\| |\psi\rangle \| = 1$. Two vectors $|\psi\rangle$ and $|\phi\rangle$ are *orthogonal* if $\langle \psi | \phi \rangle = 0$. An *orthonormal basis* of a Hilbert space $\mathcal{H}$ is a basis $\{|i\rangle\}$ where each $|i\rangle$ is normalised and any pair of them is orthogonal.

Let $\mathcal{L}(\mathcal{H})$ be the set of linear operators on $\mathcal{H}$. For any $A \in \mathcal{L}(\mathcal{H})$, A is *Hermitian* if $A^\dagger = A$ where $A^\dagger$ is the adjoint operator of A such that $\langle \psi | A^\dagger | \phi \rangle = \langle \phi | A | \psi \rangle^*$ for any $|\psi\rangle, |\phi\rangle \in \mathcal{H}$. The fundamental *spectral theorem* states that a set of normalised eigenvectors of a Hermitian operator in $\mathcal{L}(\mathcal{H})$ constitutes an orthonormal basis for $\mathcal{H}$. That is, there exists a so-called spectral decomposition for each Hermitian A such that

$$A = \sum_i \lambda_i |i\rangle \langle i| = \sum_{\lambda_i \in \text{spec}(A)} \lambda_i E_i$$

where the set $\{|i\rangle\}$ constitutes an orthonormal basis of $\mathcal{H}$, $\text{spec}(A)$ denotes the set of eigenvalues of A , and E_i is the projector to the corresponding eigenspace of λ_i . A linear

operator $A \in \mathcal{L}(\mathcal{H})$ is *unitary* if $A^\dagger A = AA^\dagger = \mathcal{I}_{\mathcal{H}}$ where $\mathcal{I}_{\mathcal{H}}$ is the identity operator on $\mathcal{H}$. The *trace* of A is defined as $\text{tr}(A) = \sum_i \langle i|A|i \rangle$ for some given orthonormal basis $\{|i\rangle\}$ of $\mathcal{H}$. It is worth noting that the trace function is actually independent of the chosen orthonormal basis. It is also easy to check that the trace function is linear and $\text{tr}(AB) = \text{tr}(BA)$ for any operators $A, B \in \mathcal{L}(\mathcal{H})$.

Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be two Hilbert spaces. Their *tensor product* $\mathcal{H}_1 \otimes \mathcal{H}_2$ is defined as a vector space consisting of linear combinations of the vectors $|\psi_1 \psi_2\rangle = |\psi_1\rangle |\psi_2\rangle = |\psi_1\rangle \otimes |\psi_2\rangle$ with $|\psi_1\rangle \in \mathcal{H}_1$ and $|\psi_2\rangle \in \mathcal{H}_2$. Here the tensor product operator $\otimes$ maps two component vectors into a new vector such that

$$\left(\sum_i \lambda_i |\psi_i\rangle \right) \otimes \left(\sum_j \mu_j |\phi_j\rangle \right) = \sum_{i,j} \lambda_i \mu_j |\psi_i\rangle \otimes |\phi_j\rangle.$$

Then $\mathcal{H}_1 \otimes \mathcal{H}_2$ is also a Hilbert space where the inner product is defined as the following: for any $|\psi_1\rangle, |\phi_1\rangle \in \mathcal{H}_1$ and $|\psi_2\rangle, |\phi_2\rangle \in \mathcal{H}_2$,

$$\langle \psi_1 \otimes \psi_2 | \phi_1 \otimes \phi_2 \rangle = \langle \psi_1 | \phi_1 \rangle_{\mathcal{H}_1} \langle \psi_2 | \phi_2 \rangle_{\mathcal{H}_2}$$

where $\langle \cdot | \cdot \rangle_{\mathcal{H}_k}$ is the inner product of $\mathcal{H}_k$, $k = 1, 2$. For any $A_1 \in \mathcal{L}(\mathcal{H}_1)$ and $A_2 \in \mathcal{L}(\mathcal{H}_2)$, $A_1 \otimes A_2$ is defined as a linear operator in $\mathcal{L}(\mathcal{H}_1 \otimes \mathcal{H}_2)$ such that for each $|\psi_1\rangle \in \mathcal{H}_1$ and $|\psi_2\rangle \in \mathcal{H}_2$,

$$(A_1 \otimes A_2) |\psi_1 \psi_2\rangle = A_1 |\psi_1\rangle \otimes A_2 |\psi_2\rangle.$$

The *partial trace* of $A \in \mathcal{L}(\mathcal{H}_1 \otimes \mathcal{H}_2)$ with respected to $\mathcal{H}_k$, $k = 1, 2$, is defined as $\text{tr}_{\mathcal{H}_k}(A) = \sum_i \langle i|A|i \rangle$ where $\{|i\rangle\}$ is an orthonormal basis of $\mathcal{H}_k$. As for the trace function, also the partial trace functions are independent of the chosen orthonormal basis.

Traditionally, a linear operator $\mathcal{E}$ on $\mathcal{L}(\mathcal{H})$ is called a *super-operator* on $\mathcal{H}$. A super-operator is said to be *completely positive* if it maps positive operators in $\mathcal{L}(\mathcal{H})$ to positive operators in $\mathcal{L}(\mathcal{H})$, and for any auxiliary Hilbert space $\mathcal{H}'$, the trivially extended operator $I_{\mathcal{H}'} \otimes \mathcal{E}$ also maps positive operators in $\mathcal{L}(\mathcal{H}' \otimes \mathcal{H})$ to positive operators in $\mathcal{L}(\mathcal{H}' \otimes \mathcal{H})$. Here $I_{\mathcal{H}'}$ is the identity operator on $\mathcal{L}(\mathcal{H}')$. We always assume complete positivity for super-operators in this chapter. The elegant and powerful

Kraus representation theorem of completely positive super-operators states that a super-operator $\mathcal{E}$ is completely positive if and only if there is some set of operators $\{E_i \mid i \in I\}$ with appropriate dimension such that

$$\mathcal{E}(A) = \sum_{i \in I} E_i A E_i^\dagger$$

for any $A \in \mathcal{L}(\mathcal{H})$. The operators E_i are called Kraus operators of $\mathcal{E}$. We abuse the notation slightly by denoting $\mathcal{E} = \{E_i \mid i \in I\}$. It is worth noting that the set of Kraus operators is not unique and we can always take one such that the number of Kraus operators does not exceed d^2 where d is the dimension of the Hilbert space. A super-operator is said to be *trace-preserving* if $\text{tr}(\mathcal{E}(A)) = \text{tr}(A)$ for any positive $A \in \mathcal{L}(\mathcal{H})$; equivalently, its Kraus operators E_i satisfy $\sum_i E_i^\dagger E_i = I$.

7.3.2 Basic Quantum Mechanics

According to von Neumann's formalism of quantum mechanics, an isolated physical system is associated with a Hilbert space which is called the *state space* of the system. A *pure state* of a quantum system is a normalised vector in its state space, and a *mixed state* is represented by a density operator on the state space. Here a density operator ρ on the Hilbert space $\mathcal{H}$ is a positive linear operator such that $\text{tr}(\rho) = 1$. Another equivalent representation of density operator is a probabilistic ensemble of pure states. In particular, given an ensemble $\{(p_i, |\psi_i\rangle)\}$ where $p_i \geq 0$, $\sum_i p_i = 1$, and $|\psi_i\rangle$ are pure states, then $\rho = \sum_i p_i |\psi_i\rangle\langle\psi_i|$ is a density operator. Conversely, each density operator can be generated by an ensemble of pure states in this way. Let $\mathcal{D}(\mathcal{H})$ denote the set of density operators on $\mathcal{H}$.

The state space of a composite system (for example, a quantum system consisting of many qubits) is the tensor product of the state spaces of its components. For a mixed state ρ on $\mathcal{H}_1 \otimes \mathcal{H}_2$, partial traces of ρ have explicit physical meanings: the density operators $\text{tr}_{\mathcal{H}_1}(\rho)$ and $\text{tr}_{\mathcal{H}_2}(\rho)$ are exactly the reduced quantum states of ρ on the second and the first component system, respectively. Note that in general, the state of a composite system cannot be decomposed into the tensor product of

the reduced states on its component systems. A well-known example is the 2-qubit state $|\Psi_0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. This kind of state is called *entangled state*. To see the strangeness of entanglement, suppose a measurement $M = \lambda_0|0\rangle\langle 0| + \lambda_1|1\rangle\langle 1|$ is applied on the first qubit of $|\Psi_0\rangle$ (see the following for the definition of quantum measurements). Then after the measurement, the second qubit will definitely collapse into state $|0\rangle$ or $|1\rangle$, depending on whether the outcome λ_0 or λ_1 is observed. In other words, the measurement on the first qubit changes the state of the second qubit in some way. This is an outstanding feature of quantum mechanics which has no counterpart in classical world, and is the key to many quantum information processing tasks such as teleportation [35] and superdense coding [34].

The evolution of a closed quantum system is described by a unitary operator on its state space: if the states of the system at times t_1 and t_2 are ρ_1 and ρ_2 , respectively, then $\rho_2 = U\rho_1U^\dagger$ for some unitary operator U which depends only on t_1 and t_2 . In general, the dynamics which can occur in a physical system is described by a trace-preserving super-operator on its state space.

A quantum *measurement* is described by a collection $\{M_m\}$ of measurement operators, where the indices m refer to the measurement outcomes. It is required that the measurement operators satisfy the completeness equation $\sum_m M_m^\dagger M_m = I_{\mathcal{H}}$. If the system is in state ρ , then the probability that the measurement result m occurs is given by $p(m) = \text{tr}(M_m \rho M_m^\dagger)$, and the state of the post-measurement system is $\frac{M_m \rho M_m^\dagger}{p(m)}$.

A particular case of measurement is the *projective measurement* which is usually represented by a Hermitian operator. Let M be a Hermitian operator and

$$M = \sum_{m \in \text{spec}(M)} m E_m \quad (7.1)$$

its spectral decomposition. Obviously, the projectors $\{E_m \mid m \in \text{spec}(M)\}$ form a quantum measurement. If the state of a quantum system is ρ , then the probability that result m occurs when measuring M on the system is $p(m) = \text{tr}(E_m \rho)$, and the post-measurement state of the system is $\frac{E_m \rho E_m}{p(m)}$. Note that for each outcome m , the map $\mathcal{E}_m(\rho) = E_m \rho E_m$ is a super-operator by Kraus Theorem.

7.3.3 QCCS and The Bisimulation between qCCS processes

In this section, we briefly recall the basic definitions of qCCS and the notion of bisimulation between qCCS processes [213]. We have introduced the basic notions of quantum information theory in the previous sections.

The syntax of qCCS is given by the Backus-Naur form as

$$\begin{aligned} t &::= \mathbf{nil} \mid \alpha.t \mid t+t \mid t||t \mid t \setminus L \mid t[f] \mid \mathbf{if} \ b \ \mathbf{then} \ t \\ \alpha &::= \tau \mid c?x \mid c!e \mid c?q \mid c!q \mid \mathcal{E}[\tilde{q}] \mid M[\tilde{q};x] \end{aligned}$$

where c and c are classical and quantum channels, x and q classical and quantum variables, $\tilde{q}$ a set of quantum variables, e a classical expression, τ the silent action, f a relabeling function, L a set of (classical or quantum) channels, b a Boolean expression, and $\mathcal{E}$ and M respectively a super-operator and a quantum measurement applying on the Hilbert space associated with systems in $\tilde{q}$.

Let Con be the set of *configurations* with the form $\langle P, \rho \rangle$ where P is a closed (contains no free classical variables) quantum process and ρ a density operator. The semantics of qCCS is given by a probabilistic labeled transition system $\langle Con, Act, \rightarrow \rangle$ where Act is the set of actions that a quantum process is able to perform, and the transition relation $\rightarrow$ is defined in terms of a set of rules presented in [213].

For a process P , let $qv(P)$ be the set of free quantum variables in P . One of the key notions of qCCS is the bisimulation between quantum configurations defined as follows.

Definition 7.1. Let X be a set of quantum variables. A relation $R \subseteq Con \times Con$ is called an X -strong bisimulation if for any $\langle P, \rho \rangle, \langle Q, \sigma \rangle \in Con$, $\langle P, \rho \rangle R \langle Q, \sigma \rangle$ implies that

1. $qv(P) \cup X = qv(Q) \cup X$, $\text{tr}_{qv(P) \cup X}(\rho) = \text{tr}_{qv(Q) \cup X}(\sigma)$, and
2. whenever $\langle P, \rho \rangle \xrightarrow{\alpha} \mu$, there exists ν such that $\langle Q, \sigma \rangle \xrightarrow{\alpha} \nu$ and $\mu R \nu$;
3. whenever $\langle Q, \sigma \rangle \xrightarrow{\alpha} \nu$, there exists μ such that $\langle P, \rho \rangle \xrightarrow{\alpha} \mu$ and $\mu R \nu$.

FIGURE 7.1: The architecture of Qubet

Two quantum configurations $\langle P, \rho \rangle$ and $\langle Q, \sigma \rangle$ are X -strongly bisimilar, denoted $\langle P, \rho \rangle \sim_X \langle Q, \sigma \rangle$, if there exists an X -strong bisimulation R such that $\langle P, \rho \rangle R \langle Q, \sigma \rangle$.

In the above definition, $\text{tr}_{qv(P) \cup X}(\rho)$ denotes the partial state of ρ on the subsystem excluding $qv(P) \cup X$, and in μRv we lift R to a relation between probability distributions over configurations [217]. We introduce the set X to avoid comparing the states of *local* quantum systems (those systems in X). As we can see from the case studies, this will bring convenience in bisimilarity check in practice.

Similarly, we can define X -weak bisimulation, $\approx_X$, between qCCS configurations by replacing the strong transitions in Clauses 2 and 3 of Definition 7.1 with weak ones. The details are omitted here due to the lack of space.

7.4 The Architecture of Qubet

Since the novelty of Qubet is the concept related to quantum communication systems, in this part, we ignore the module to support classical model checking. The simplified architecture of Qubet is shown in Figure 7.1. We have implemented Qubet in C# based on Microsoft .NET framework, where the Math.NET library [40] was used to implement matrix data structures and the corresponding operations over them such as matrix multiplication, tensor product, and spectral decomposition.

Qubet strictly follows the object-oriented design pattern, which supports easy maintainability, reusability, and extensibility. The Editor is a GUI for users to input and edit the models of quantum communication systems, and specify different bisimilarity options. We create a grammar file for qCCS in the Parser. Then we implement the

semantic rules of qCCS with different classes. Accordingly, each expression of the model described in qCCS is allocated as an object of the corresponding semantic rule class. When the rule is applied, Qubet calls the corresponding action function in the rule class. As a result, we are able to extend the syntax of qCCS and its corresponding semantics conveniently by adding new rule classes if necessary. The Simulator is to generate the complete state space of the input quantum system based on the semantic rules, and display the state graph in a proper way. The algorithms in Bisimilarity Checking Algorithms are, again, implemented in different classes for ease of extension. Although these algorithms are specific to qCCS, we build a common interface to call them. In this way, it is possible to import other algorithms based on the same back-end representation, which is probabilistic Labeled Transitions Systems (pLTS) in Qubet. This means that libraries in some existing model checkers such as Bogor, LTSA, and PAT can be easily adapted in Qubet.

7.5 Description of Quantum Protocols in Qubet

Qubet provides a friendly environment, including multi-documents, auto-completion, keywords highlighting, etc, for users to input and edit the system model with a language similar to qCCS. In this section, we take the teleportation protocol as an example to show the utility of Qubet to describe quantum communication systems.

7.5.1 Description

Quantum teleportation [35] is one of the most important protocols in quantum information theory which can make use of a maximally entangled state shared between the sender and the receiver to teleport an unknown quantum state by sending only classical information. It serves as a key ingredient in many other communication protocols. Let M be a 2-qubit measurement such that $M = \sum_{i=0}^3 i|\tilde{i}\rangle\langle\tilde{i}|$, where $\tilde{i}$ is the binary expansion of i , and σ^i , $i = 0, \dots, 3$ be the four Pauli operators. Then the quantum


```

var x;
qvars q,q1,q2;
evars HD, CN, PX, PY, PZ, SW, ID1;
mvar M;
mstates RHO1, RHO2;
channel c 0;

qvarset(A)={q,q1};
qvarset(B)={q2};
qvarset(Telspec) = {q,q2};
lqset bx={q,q1};

set RHO1 = (1/2)*(|0><0|+|1><1|+|0><1|+|1><0|);
/*Here RHO1 can be any 2 by 2 density matrix*/
set RHO2 = (1/2)*(|00><00|+|11><11|+|00><11|+|11><00|);
set RHO = kron(RHO1, RHO2);
set ID1 = ID(2); /*build-in operator, 2 is dimension*/
set M = |01><01|+2*|10><10|+3*|11><11|;

A = CN(q,q1)->HD(q)->M[q,q1;x]->c!x->Stop;
B = c?y->(if(y==0) {ID1(q2)->Stop}
    [*] if(y==1){PX(q2)->Stop}
    [*] if(y==2){PZ(q2)->Stop}
    [*] if(y==3){PY(q2)->Stop});
Tel = (A||B)\{c};

Telspec = SW(q,q2) -> Stop;
#assertion Tel bisimulates <W,bx> Telspc

```

FIGURE 7.2: Source code for the teleportation protocol

processes participated in teleportation protocol are defined in qCCS as follows:

$$\begin{aligned}
 Alice &= CNOT[q, q_1].H[q].M[q, q_1; x].e!x.\mathbf{nil}, \\
 Bob &= e?x. \sum_{0 \leq i \leq 3} (\text{if } x = i \text{ then } \sigma^i[q_2].\mathbf{nil}), \\
 Tel &= (Alice || Bob) \setminus \{e\}.
 \end{aligned}$$

Let $Tel_{spec} = SWAP[q, q_2].\mathbf{nil}$ be the specification of teleportation protocol, where $SWAP$ is the unitary operator which exchanges the states of the two parameter qubits. Then the correctness of teleportation protocol is verified by showing that for any ρ ,

$$\langle Tel, |\Psi_0\rangle_{q_1, q_2} \langle \Psi_0| \otimes \rho_q \rangle \approx_X \langle Tel_{spec}, |\Psi_0\rangle_{q_1, q_2} \langle \Psi_0| \otimes \rho_q \rangle, \quad (7.2)$$

where $X = \{q, q_1\}$, which means we do not care the final states of q and q_1 .

The source code for the teleportation protocol is listed in Figure 7.2, where `var` and `qvars` are classical and quantum variable (qubit) declarations, `evars` the declaration of unitary operators used in the system model, `mvars` the declaration of quantum measurements (Hermitian operators), and `channel/qchannel` the declaration of classical/quantum channels with the integer in the end denoting the buffer size of the channel, where 0 indicates a synchronized channel. The built-in unitary operators `HD`, `CN`, `PX`, `PY`, `PZ`, `SW` denote respectively the Hadamard H , Control-NOT $CNOT$, Pauli operators σ^i , $i = 1, 2, 3$, and SWAP operator $SWAP$ mentioned above. Furthermore, `qvarset` is used to declare quantum variables contained in a quantum process, `lqset` to set the local quantum variables for bisimilarity checking, and the values of quantum states and quantum measurements are set by the `set` command. Notably, Qubet supports the use of bra-ket notation which is standard for describing quantum states and operators in quantum mechanics.

The property we need to check is the $\{q, q_1\}$ -weak bisimilarity between `Te1` and `Te1spc`, which is also included in the description of the system model. The assertion is shown at the last line of Figure 7.2, where $\langle W, bx \rangle$ is the bisimilarity options in which W means weak bisimulation.

Another protocol-Superdense coding, is a well-known protocol presented by Bennett and Wiesner [34], in which two bits of classical information can be faithfully transmitted by sending only one qubit, provided that a maximally entangled state is shared *a priori* between the sender and the receiver. The superdense coding protocol goes as follows:

1. Alice and Bob prepare a maximally entangled state $|\Psi_0\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$ on q_1 and q_2 . Alice holds q_1 while Bob holds q_2 .
2. Depending on which message among the four possibilities $\{0, 1, 2, 3\}$ Alice wishes to transmit, she applies one of the four Pauli super-operators σ_i , $0 \leq i \leq 3$, on her qubit q_1 , and sends it to Bob.

3. Upon receiving q_1 , Bob performs a Bell-basis measurement $M^B = \sum_{i=0}^3 i |\Psi_i\rangle\langle\Psi_i|$ on q_1 and q_2 , where

$$\begin{aligned} |\Psi_1\rangle &= \frac{|01\rangle + |10\rangle}{\sqrt{2}}, \\ |\Psi_2\rangle &= \frac{|00\rangle - |11\rangle}{\sqrt{2}}, \\ |\Psi_3\rangle &= \frac{|01\rangle - |10\rangle}{\sqrt{2}}. \end{aligned}$$

Bob interprets the measurement outcome as the message sent by Alice.

superdense coding protocol can be defined in Figure 7.3.

$$\begin{aligned} Alice_s &= c?x. \sum_{0 \leq i \leq 3} (\text{if } x = i \text{ then } \sigma^i[q_1].e!q_1.\text{nil}), \\ Bob_s &= e?q_1.CNOT[q_1, q_2].H[q_1].M^B[q_1, q_2; x].d!x.\text{nil}, \\ Sdc &= (Alice_s || Bob_s) \setminus \{e\}. \end{aligned}$$

Let

$$Sdc_{spec} = c?x.d!x.\text{nil}$$

be the specification of superdense coding protocol. Then its correctness can be verified by showing

$$\langle Sdc, |\Psi_0\rangle_{q_1, q_2} \langle \Psi_0| \rangle \approx_X \langle Sdc_{spec}, |\Psi_0\rangle_{q_1, q_2} \langle \Psi_0| \rangle$$

where $X = \{q_1, q_2\}$. **Verification result** The source code for the superdense coding protocol is listed in Figure 7.3, and the property we need to check is

```
#assertion Sdc bisimulates <W,X> Sdc_spc
```

Qubet returns true as expected.

7.5.2 Grammar Checking

Besides the common grammar checking based on the syntax of qCCS, the parser in Qubet also checks whether the value (matrix) of quantum operators/states are VALID,

```

var x;
//var s=<0||1>; represent <0|1>
qvars q1,q2;
evars HD, CN, PX, PY, PZ;
mvars M;
//mixed state declaration: mstates RHO1,RHO2;
//super operator declaration: svars SOP1, SOP2;
//pure state declaration: pstates p1,p2;
//vector declaration: vectors v1,v2,v3;

qchannel c 0;
channel d 1;

qvarset (A)={q1};
qvarset (B)={q2};
lqset bx={q1,q2};

set RHO=(1/2)*(|00><00|+|00><11|+|11><00|+|11><11|);
set M=|01><01|+2*|10><10|+3*|11><11|;
//Init the super operator: set SOP = {PX,PZ,...PM};
//Init the vector: set v1=|0>+|1>; set v2 = |0>|0>; set v3=kron(v1,
    v2);
//can support the initialization of RHO with : set RHO=kron
    (|0><0|+|1><1|, |0><0|+|1><1|);

C = d!1->Stop;
A = d?y->(if (y==0){c!q1->Stop} [*] if (y==1){PX(q1)->c!q1->Stop} [*]
    if (y==2){PZ(q1)->c!q1->Stop} [*] if (y==3){PY(q1)->c!q1->Stop});
B = c?q1->CN(q1,q2)->HD(q1)->M[q1,q2;x]->d!x->Stop;

Sdc = (A||B)\{c}||C;

D = d?y-> d!y ->Stop;
SdcSpec = C||D;

#assert Sdc bisimulates <W,bx> SdcSpec;

```

FIGURE 7.3: Source code for the Superdense coding protocol

e.g., whether a quantum measurement M is set to a Hermitian matrix, and the initial quantum state RHO is set to a positive-semidefinite matrix with the trace being 1, which is the tensor product between a density matrix and the maximally entangled state. All the information is shown in a separate window. Qubet suspends the simulation and verification until the grammar checking succeeds.

Algorithm 14: The decision procedure for the weak bisimilarity

Input: $init_{S1}, init_{S2}$

- 1 Define Equivalence Class: $R = S = S1 \cup S2$;
- 2 $(s, a, \mu_s) = \text{FindSplit}(R)$;
- 3 **while** $s! = NULL$ **do**
- 4 $R = \text{Refine}(R, (s, a, \mu_s))$;
- 5 $s, a, \mu_s = \text{FindSplit}(R)$;
- 6 **if** $init_{S1}, init_{S2} \in R$ **then**
- 7 $Bsimulation = True$;
- 8 **else**
- 9 $Bsimulation = False$;
- 10 **return** $Bsimulation$;

7.5.3 Simulation

Based on the system model and the semantic rules, the simulator of Qubet can present the system behaviors by random simulation, user-guided step-by-step simulation, complete state space generation etc. The state graph is a pLTS shown with graphic view. Users can also see the complete information (e.g., quantum process, quantum state, etc.) in each state by pointing the corresponding node with the mouse.

7.6 Bisimilarity Checking between Quantum Processes

Qubet supports both *X-strong* and *X-weak* bisimilarity check defined in Definition 7.1. We adapt the partition refinement algorithm in Algorithm 14 that orient from Figure 2 of [124] in the following way. Given two quantum configurations to be checked, we firstly generate the full state spaces S_1 and S_2 of them. Then the joint state space $S_1 \cup S_2$ is decomposed into several equivalence classes according to the quantum free variables and quantum states of the environment. To be specific, two configurations $\langle P, \rho \rangle$ and $\langle Q, \sigma \rangle$ are in the same equivalence class if and only if the first clause of Definition 7.1 holds. We take this partition as the initial one for the partition refinement algorithm, instead of the coarsest one $\{S_1 \cup S_2\}$ as in [124].

The rest of our implementation follows the procedure presented in Algorithm 15 and 16 [124]. The key step is to find a splitter, if there exists one for the current partition,

Algorithm 15: The FindSplit algorithm

Input: R

```

1 forall the  $s \in S$  do
2   forall the  $(s, a, \mu_s) \in T$  do
3     forall the  $t \in [s]_R$  do
4       if Does not exist  $t \xrightarrow{a} \mu_t$  that  $\mu_s \mathcal{L} \mu_t$  then
5         return  $(s, a, \mu_s)$ ;
6 return NULL;

```

Algorithm 16: The Refine algorithm

Input: $R, (s, a, \mu_s)$

```

1  $C_s = C_{\neg s} = \emptyset$ ;
2 forall the  $t \in [s]_R$  do
3   if There exist  $t \xrightarrow{a} \mu_t$  that  $\mu_s \mathcal{L} \mu_t$  then
4      $C_s = C_s \cup t$ ;
5   else
6      $C_{\neg s} = C_{\neg s} \cup t$ ;
7 return  $R \setminus \{[s]_R \cup C_s, C_{\neg s}\}$ ;

```

for which the most challenging part is the construction of weak transitions. To solve the Linear Programming (LP) problem induced by the flow network construction presented in [124], we employ the library of Microsoft solver foundation [133]. The partition refinement process repeats until a fixed point (a stable partition without any splitter) is reached.

7.7 Demonstration

We present the demonstration of our tool in three parts: (1) The graphic view of editing quantum communication system model and the grammar checking. (2) How to conduct the simulation. (3) How the verification (bisimilarity checking) works. Here we present another protocol - the superdense coding protocol. The description and implementation of this protocol can be found at our website [223];

Figure 7.4 presents the startup window. We can create a new Model by click “New” or open an existing file of qCCS model. We have integrated several samples in the

FIGURE 7.4: The startup window of Qubet

“Example” menu. Figure 7.4 also shows the editing form, in which the superdense coding protocol is presented with the qCCS in Qubet.

After we finish editing the model of quantum communication systems, we can check whether our model fits the syntax of qCCS. To do this, one can click the “Check Grammar(F5)” button. As a consequence, a separate window will be presented to list the result of grammar checking, shown in Figure 7.5. We also present the result of property checking for each quantum operator and quantum state.

If the grammar checking succeeds, one can continue with the simulation or verification. By clicking “Simulation”, the simulation startup window will be opened, shown in Figure 7.6. One can select which process to simulate at the upper left corner of the window. We support several types of simulation. One can choose the simulation mode by clicking “Simulate,” “Generate Graph” or “Simulate Trace.” Figure 7.7 presents the simulation result after clicking the “Generate Graph”. In the graphic view of the generated pLTS, one can see the detail information contained in each state, as well as the environment information.

FIGURE 7.5: The grammar checking in Qubet

By clicking the “Verification” button in the startup window of Qubet, one can open the verification (bisimilarity checking) form, shown in Figure 7.8. We present the verification engine automatically based on the assertion defined in the model. Qubet suspends the “verify” button until users choose the assertion they want to verify. The result of bisimilarity checking is also shown in this form.

In addition, we support many useful functionalities such as graphic view of model exploration, print of source codes, find and replace in editing etc. We hope Qubet

FIGURE 7.6: The startup window of simulation in Qubet

can offer convenience to researchers working in the design and verification of quantum communication protocols.

7.8 The Application of Parallelized Model Checking Algorithms

In Qubet, the quantum communication protocol is also a concurrent system. The model described in qCCS can be translated to the state space expressed in probabilistic LTS via the semantics of qCCS. In addition, the state space of a quantum communication system can be represented as a probabilistic labeled transition system. Thus, the common model checking algorithms, e.g., LTL model checking, probabilistic model checking etc, is also capable to verify the quantum communication systems. In fact, the process of state space generation, SCC detection, reachability checking and counterexample generation can be similar to the process for standard concurrent systems. Therefore, our previous

FIGURE 7.7: The result of simulation in Qubet

research, which focuses on the parallelization of model checking algorithms [219–222], can be applied into Qubet.

We have mentioned in some of our previous research that we have tried the application in PAT model checker. In common, the GPU-based algorithms are developed on Linux systems. To transfer them to fit Qubet, which is based on the .NET platform in Windows system, we need the support from another framework named **CUDAfy** [4]¹. The detail implementation of these algorithms in Qubet are similar to our earlier implementation in PAT. Currently, we cannot test the performance with quantum communication systems since the state space of our samples has limited size. However, we have tested with the common concurrent systems expressed in CSP and compared with our previous experiments on PAT. The comparison results show that the performance improvement is expected, and the application of our previous research in

¹CUDAfy allows .NET developers to easily create complex applications that split processing cleanly between host and GPU. There are no separate CUDA cu files or complex set-up procedures to launch GPU device functions. It follows the CUDA programming model and any knowledge gained from tutorials or books on CUDA can be easily transferred to CUDAfy, only in a clean .NET fashion.

FIGURE 7.8: The window of bisimilarity checking result

Qubet can gain the similar benefits as we gain in PAT. In summary, we can conclude that the integration of our previous research is meaningful to our application. The application also makes our previous work more convincing.

7.9 Conclusion

In this chapter, we propose Qubet, a model checking, bisimulation checking and emulation tool for quantum communication systems which has an object-oriented architecture with great maintainability, reusability, and extensibility. We also support the classical communication systems expressed in CSP. A user-friendly GUI is also provided which enables the users to learn and monitor the behaviors of quantum

processes easily. Furthermore, we integrate our previous research on the parallelization of model checking algorithms into our tool.

Chapter 8

Conclusion

In this chapter, we summarize the research work in this thesis, and discuss the future research directions.

8.1 Thesis Summary

This thesis investigated the speedup of the model checking algorithms using multi-core CPU and many-core GPU, with the applications in quantum computing. For the algorithm parallelization, we made an attempt to parallelize the various steps in the traditional LTL verification algorithm for both concurrent and probabilistic models. To demonstrate the usefulness of our approach, we used a case study in quantum communication systems with a model checker named Qubet developed. The contribution of this thesis is summarized as follows.

Firstly, we presented an approach to accelerate the reachability verification of concurrent systems modeled using LTS models in GPU platform. The approach includes a GPU-based parallel state space generation process, which is the first step in LTL model checking. We described the design of the data structure to achieve the reduction of storage space, as well as the requirements for parallelization. The evaluation results proved the efficiency of our approach.

Secondly, we showed how to expand Lowe's concurrent Tarjan's algorithm to develop a concurrent on-the-fly SCC detection for automata-based LTL model checking with fairness assumptions. SCC detection is the key subroutine in LTL model checking algorithm. We proposed a specific data structure to support the parallel execution of on-the-fly SCC detection algorithm using multi-core CPU architecture. Our approach has been integrated into PAT model checker. We evaluated the efficiency of our approach with PAT and the results showed the substantial performance improvement of our approach over the sequential model checking algorithm.

Thirdly, we proposed an approach of GPU-based counterexample generation, which is the last step in LTL model checking algorithm. We proposed a design of a dynamic queue management scheme, a hierarchical task scheduler and a parallel path recording scheme by adopting the dynamic parallelism of CUDA. The evaluation of this approach indicated the scalability and performance.

Fourth, we described the last work in the algorithm parallelization. We proposed a GPU accelerated parallel value iteration approach for the efficient computation of reachability probabilities in MDPs. We showed how to use the algebraic feature of MDPs to divide the computation of reachability probabilities into partitions, as well as how to compute the Bellman backup in a massively parallelized manner with an efficient granularity in GPUs. The evaluation results proved the advantage of our approach over the existing topological-based value iteration algorithm.

Finally, we presented a case study in quantum communication systems and showed how to design a model checking, bisimulation checking and emulation tool named Qubet for it. We showed the object-oriented architecture of Qubet, which has great maintainability, reusability and extensibility. We showed that Qubet can effectively help the designer of quantum communication systems to perform systematic analysis of the quantum models.

8.2 Future work

This thesis has presented my research during the past several years. Different model checking algorithms have different features. Therefore, the development of parallelization solutions for them should be specifically to different problems. It is also necessary to select the most suitable parallel computing platforms. Furthermore, tool support is significant to use model checking techniques. Both algorithm development and application development are necessary to the research in model checking area. Besides the contents included in this thesis, there are still a number of problems to be solved. Our future works are presented as follows:

1. We want to use cloud platform to perform the cloud-based model checking. During the past several years, the development of cloud computing techniques has made great contribution to accelerate the computation in the area of science, engineering and industry. It is a potential way to solve the state space explosion problem with its massive storage and computation power. Based on the challenges mentioned in Section 2.5.3, we aim at the development of efficient synchronization mechanics to parallelize model checking problems in cloud platform.
2. We plan to try out features in the latest GPU architecture to handle the state space explosion problem in parallelized model checking algorithms. In the latest GPU architecture, the feature “unified memory” [158] has been supported, which allows the access of CPU memory from GPU kernels. Therefore, we can have more memory space to store the data for the parallelized model checking algorithms in GPU.
3. We are thinking about investigating further on the problem in probabilistic model checking. We intend to research on the MDP related problems with two directions: 1) We aim to finish the development of parallel value iteration for POMDPs and evaluate it on various benchmarks. POMDPs has more widely practical usage than MDPs currently. 2) We aim to build a GPU-accelerated symbolic value iteration and evaluate it further on the tool SPUDD.

4. For the application development, we intend to implement the symbolic bisimilarity checking that presented in [215] in Qubet. This will improve the efficiency of bisimilarity checking for quantum programs and protocols with quantum inputs. A typical example is the teleportation protocol presented in Chapter 7: with the current technique, we have to prove Equation 7.2 for *any* quantum state ρ ; while in symbolic bisimulation, we only need to check a *single* pair. Moreover, we are planning to apply our tool in the analysis of quantum cryptographic systems such as BB84 key distribution protocol.

Bibliography

- [1] A. Prasad Sistla (1994). Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511.
- [2] A. Prasad Sistla and Edmund M. Clarke (1985). The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749.
- [3] Aaron Lefohn, Sengupta Shubhabrata, Kniss Joe, Strzodka Robert and John Owens (2006). Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics (TOG)*, 25(1):60–99.
- [4] Advanced Supercomputing Solutions Ltd, MIT. CUDAFy.NET. <https://cudafy.codeplex.com/>. Accessed Dec 4, 2016.
- [5] Akin Günay, Songzheng Song, Yang Liu and Jie Zhang (2015). Automated analysis of commitment protocols using probabilistic model checking. In *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015)*, pages 2060–2066. AAAI Press.
- [6] Albert Y. Zomaya and others (1996). Parallel and distributed computing handbook.
- [7] Alfons Laarman, Rom Langerak, Jaco Van De Pol, Michael Weber and Anton Wijs (2011). Multi-core nested depth-first search. In *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA 2011)*, pages 321–335. Springer.
- [8] André, É., Fribourg, L., Kühne, U., and Soulat, R. (2012). IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *18th International Symposium on Formal Methods (FM’12)*, volume 7436, pages 33–36, Paris, France. Springer.

- [9] Andreas Kuehlmann, Kenneth L. McMillan and Robert K. Brayton (1999). Probabilistic state space search. In *Proceedings of the 1999 International Conference on Computer Aided Design (ICCAD 1999)*, pages 574–579. IEEE Press.
- [10] Andrey Kolobov Mausam, Daniel S. Weld and Hector Geffner (2011). Heuristic search for generalized stochastic shortest path MDPs. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS 2011)*.
- [11] Anh Tuan Luu, Jun Sun, Yang Liu, Jin Song Dong, Xiaohong Li and Quan Thanh Tho (2012). SeVe: automatic tool for verification of security protocols. *Frontiers of Computer Science in China*, 6(1):57–75.
- [12] Anthony W. Roscoe (2005). On the expressive power of CSP refinement. *Formal Aspects of Computing*, 17(2):93–112.
- [13] Antoine Durand-Gasselín, Javier Esparza , Pierre Ganty and Rupak Majumdar (2015). Model checking parameterized asynchronous shared-memory systems. In *Proceedings of the 47th International Conference on Computer Aided Verification (CAV 2015)*, pages 67–84. Springer.
- [14] Anton Wijs (2015). Gpu accelerated strong and branching bisimilarity checking. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, pages 368–383. Springer.
- [15] Anton Wijs (2016). BFS-based model checking of linear-time properties with an application on GPUs. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2016)*, pages 472–493. Springer.
- [16] Anton Wijs , Joost-Pieter Katoen and Dragan Bošnački (2016). Efficient GPU algorithms for parallel decomposition of graphs into strongly connected and maximal end components. *Formal Methods in System Design*, 48(3):274–300.
- [17] Anton Wijs and Dragan Bošnački (2012). Improving gpu sparse matrix-vector multiplication for probabilistic model checking. In *Proceedings of the 19th International SPIN Workshop on Model Checking Software (SPIN 2012)*, pages 98–116.

- [18] Anton Wijs and Dragan Bošnački (2015). Many-core on-the-fly model checking of safety properties using GPUs. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–17.
- [19] Anton Wijs and Dragan Bošnački (2014). GPUexplore: Many-Core On-the-Fly State Space Exploration Using GPUs. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, pages 233–247. Springer.
- [20] Anton Wijs, Joost-Pieter Katoen and Dragan Bošnački (2014). GPU-based graph decomposition into strongly connected and maximal end components. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*, pages 310–326. Springer.
- [21] Antti Puhakka and Antti Valmari (2001). Liveness and fairness in process-algebraic verification. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, pages 202–217. Springer.
- [22] Antti Valmari (1989). Stubborn sets for reduced state space generation. In *International Conference on Application and Theory of Petri Nets*, pages 491–515. Springer.
- [23] Ardeshir-Larijani Ebrahim, Gay Simon J and Nagarajan Rajagopal (2013). Equivalence checking of quantum protocols. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2013)*, pages 478–492. Springer.
- [24] Ársall pór Jóhannsson (2009). *GPU-based Markov decision process solver*. PhD thesis.
- [25] Aspnes James and Herlihy Maurice (1990). Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461.
- [26] Bellettini Carlo, Camilli Matteo, Capra Lorenzo and Monga Mattia (2013). Distributed CTL model checking in the cloud. *arXiv preprint arXiv:1310.6670*.

- [27] Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier and Sylvain Hallé (2012). Mapreduce for parallel trace validation of ltl properties. In *Proceedings of the 12th International Conference on Runtime Verification (RV 2012)*, pages 184–198. Springer.
- [28] Biallas Sebastian, Brauer Jörg, Gückel Dominique and Kowalewski Stefan (2011). On-the-fly path reduction. *Electronic Notes in Theoretical Computer Science*, 274:3–16.
- [29] Bonet Blai and Geffner Hector (2003). Labeled RTDP: Improving the Convergence of Real-Time Dynamic Programming. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS 2003)*, volume 3, pages 12–21.
- [30] Bowen Alpern and Fred B. Schneider (1985). Defining liveness. *Information Processing Letters*, 21(4):181–185.
- [31] Braziunas, D. (2003). Pomdp solution methods. *University of Toronto, Technical Report*.
- [32] C. Norris Ip and David L. Dill (1996). Better verification through symmetry. *Formal methods in system design*, 9(1-2):41–75.
- [33] Charles Antony Hoare Richard (1978). Communicating sequential processes. In *The origin of concurrent programming*, pages 413–443. Springer.
- [34] Charles H. Bennett and Stephen J. Wiesner (1992). Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Physical Review Letters*, 69(20):2881–2884.
- [35] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters (1993). Teleporting an unknown quantum state via dual classical and EPR channels. *Physical Review Letters*, 70:1895–1899.
- [36] Chen Peng and Lu Lu (2013). Markov Decision Process Parallel Value Iteration Algorithm On GPU. In *Proceedings of the 14th International Symposium on Computer Architecture (ISCA 2013)*. Atlantis Press.

- [37] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker and Michael Deardeuff (2014). Use of formal methods at Amazon Web Services.
- [38] Christel Baier, Joost-Pieter Katoen and others (2008). *Principles of model checking*, volume 26202649. MIT press Cambridge.
- [39] Christian Eisentraut , Jens Chr Godskesen , Holger Hermanns, Lei Song and Lijun Zhang (2015). Probabilistic bisimulation for realistic schedulers. In *Proceedings of the 20th International Symposium on Formal Methods (FM 2015)*, pages 248–264. Springer.
- [40] Christoph Rüegg. Math.Net Numerical Computing. <http://numerics.mathdotnet.com/>. Accessed April 4, 2016.
- [41] Chulian Zhang (2016). *Inter-warp Divergence Aware Execution on GPUs*. PhD thesis, Technical Report, Northeastern University.
- [42] Craig Boutilier, Thomas Dean and Steve Hanks (1999). Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research (JAIR)*, 11:1–94.
- [43] Dan Alcantara, Sharf Andrei, Abbasinejad Fatemeh, Sengupta Shubhabrata, Mitzenmacher Michael, John Owens and Amenta Nina (2009). Real-time parallel hashing on the GPU. *ACM Transactions on Graphics (TOG)*, 28(5):154.
- [44] Dan Alcantara, Volkov Vasily, Sengupta Shubhabrata, Mitzenmacher Michael, John Owens and Amenta Nina (2011). Building an efficient hash table on the GPU. *GPU Computing Gems Jade Edition*, 2:39–53.
- [45] Dana Angluin and Dana Fisman (2016). Learning regular omega languages. *Theoretical Computer Science*, 650:57–72.
- [46] Daniel Kroening , Matt Lewis and Georg Weissenbacher (2015). Proving safety with trace automata and bounded model checking. In *Proceedings of the 20th International Symposium on Formal Methods (FM 2015)*, pages 325–341. Springer.
- [47] David Nassimi and Sartaj Sahni (1981). Data broadcasting in SIMD computers. *IEEE Transactions on Computers (TC)*, 100(2):101–107.

- [48] Doron A. Peled (1993). All from one, one for all: on model checking using representatives. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV 1993)*, pages 409–423. Springer.
- [49] Doron Peled (1998). Ten years of partial order reduction. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV 1998)*, pages 17–28. Springer.
- [50] Doshi Prashant, Goodwin Richard, Akkiraju Rama and Verma Kunal (2004). Dynamic workflow composition using markov decision processes. In *Proceedings of the 2004 IEEE International Conference on Web Services (ICWS 2004)*, pages 576–582. IEEE.
- [51] Dragan Bosnacki , Stefan Edelkamp , Damian Sulewski and Anton Wijs (2010). GPU-PRISM: An extension of PRISM for general purpose graphics processing units. In *Proceedings of the 2010 9th International Workshop on Parallel and Distributed Methods in Verification, and the 2nd International Workshop on High Performance Computational Systems Biology*, pages 17–19. IEEE.
- [52] Dragan Bošnački (2002). A nested depth first search algorithm for model checking with symmetry reduction. In *Proceedings of the 22nd International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2002)*, pages 65–80. Springer.
- [53] Dragan Bošnački, Stefan Edelkamp, Damian Sulewski and Anton Wijs (2011). Parallel probabilistic model checking on general purpose graphics processors. *International Journal on Software Tools for Technology Transfer (STTT)*, 13(1):21–35.
- [54] Dragan Bošnački, Stefan Leue and Alberto Lluch Lafuente (2009). Partial-order reduction for general state exploring algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(1):39–51.
- [55] Duane Merrill, Michael Garland and Andrew Grimshaw (2015). High-performance and scalable gpu graph traversal. *ACM Transactions on Parallel Computing*, 1(2):14.

- [56] Ebrahim Ardeshtir-Larijani, Simon J. Gay and Rajagopal Nagarajan (2014). Verification of concurrent quantum protocols by equivalence checking. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, pages 500–514. Springer.
- [57] Edmund M. Clarke and E. Allen Emerson (1981). Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of the Workshop on Logic of Programs*, pages 52–71. Springer.
- [58] Edmund M. Clarke, Orna Grumberg and D. Long (1993). Verification tools for finite-state concurrent systems. In *Workshop/School/Symposium of the REX Project (Research and Education in Concurrent Systems)*, pages 124–175. Springer.
- [59] Edmund M. Clarke, Orna Grumberg and Doron Peled (1999). *Model checking*. MIT press.
- [60] Edmund M. Clarke, Orna Grumberg, Marius Minea and Doron Peled (1999). State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):279–287.
- [61] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith (2001). Progress on the state explosion problem in model checking. In *Informatics*, pages 176–194. Springer.
- [62] Edmund M. Clarke, Reinhard Enders, Thomas Filkorn and Somesh Jha (1996). Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104.
- [63] Edmund M. Clarke, William Klieber, Miloš Nováček and Paolo Zuliani (2012). Model checking and the state explosion problem. In *Tools for Practical Software Verification*, pages 1–30.
- [64] Ezio Bartocci, Richard DeFrancisco and Scott A. Smolka (2014). Towards a GPGPU-parallel SPIN model checker. In *Proceedings of the 21th International SPIN Workshop on Model Checking Software (SPIN 2014)*, pages 87–96. ACM.

- [65] Farn Wang (2003). Efficient verification of timed automata with BDD-like data-structures. In *Proceedings of the 4th International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI 2003)*, pages 189–205. Springer.
- [66] FBK-irst, CMU, of Genova, T. U., and of Trento, T. U. NuSMV: a new symbolic model checker. <http://nusmv.fbk.eu/>. Accessed Dec 4, 2016.
- [67] Gavin Lowe (2014). Concurrent Depth-First Search Algorithms. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*, pages 202–216.
- [68] Gavin Lowe (2016). Concurrent depth-first search algorithms based on Tarjan’s algorithm. *International Journal on Software Tools for Technology Transfer (STTT)*, 18(2):129–147.
- [69] George Edward Hughes and Max J. Cresswell (1996). *A new introduction to modal logic*. Psychology Press.
- [70] George Karypis, Kirk Schloegel and Vipin Kumar (1997). Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*.
- [71] Gerard Holzmann (2003). *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional.
- [72] Gerard J. Holzmann and Dragan Bosnacki (2007). The design of a multicore extension of the spin model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674.
- [73] Gerard J. Holzmann, Doron Peled and Mihalis Yannakakis (1996). On nested depth first search. In *Proceedings of the 2nd International SPIN Workshop on Model Checking Software (SPIN 1996)*, volume 32, pages 81–89.
- [74] Girish Bhat, Rance Cleaveland and Orna Grumberg (1995). Efficient on-the-fly model checking for CTL. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS 1995)*, pages 388–397. IEEE.

- [75] Guangdong Bai, Jike Lei, Guozhu Meng, Venkatraman Sai Sathyanarayan, Saxena Prateek, Jun Sun, Yang Liu and Jinsong Dong (2013). AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. *Proceeding of the Network and Distributed System Security Symposium (NDSS 2013)*.
- [76] Guy Gueta, Cormac Flanagan , Eran Yahav and Mooly Sagiv (2007). Cartesian partial-order reduction. In *Proceedings of the 14th International SPIN Workshop on Model Checking Software (SPIN 2007)*, pages 95–112. Springer.
- [77] Hansen Eric A and Zilberstein Shlomo (2001). LAO*: A heuristic search algorithm that finds solutions with loops. *Journal of Artificial Intelligence*, 129(1):35–62.
- [78] Henri Hansen, Shang-Wei Lin, Yang Liu, Truong Khanh Nguyen and Jun Sun (2014). Diamonds are a girl’s best friend: Partial order reduction for timed automata with abstractions. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV 2014)*, pages 391–406. Springer.
- [79] Hsiang-Fu Yu, Hsieh Cho-Jui, Si Si and Dhillon Inderjit (2012). Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *Proceedings of the 12th International Conference on Data Mining (ICDM 2012)*, pages 765–774. IEEE.
- [80] Igor Konnov , Helmut Veith and Josef Widder (2015). SMT and POR beat counter abstraction: parameterized model checking of threshold-based distributed algorithms. In *Proceedings of the 47th International Conference on Computer Aided Verification (CAV 2015)*, pages 85–102. Springer.
- [81] Ivana Cerná and Radek Pelánek (2003). Distributed explicit fair cycle detection. In *Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN 2003)*, volume 2648, pages 49–73.
- [82] Ivaylo Dobrikov, Michael Leuschel and Daniel Plagge (2016). LTL Model Checking under Fairness in ProB. In *Proceedings of the 18th International Conference on Software Engineering and Formal Methods (ICFEM 2016)*, pages 204–211. Springer.

- [83] Jack W. Davidson and Jinturkar Sanjay (1994). Memory access coalescing: a technique for eliminating redundant memory accesses. In *ACM SIGPLAN Notices*, volume 29, pages 186–195.
- [84] Jaco Geldenhuys and Antti Valmari (2005). More efficient on-the-fly LTL verification with Tarjan’s algorithm. *Theoretical Computer Science*, 345(1):60–82.
- [85] James Hardy Wilkinson (1965). *The algebraic eigenvalue problem*, volume 87. Clarendon Press Oxford.
- [86] Jan Friso Groote, Tim W.D.M. Kouters and Ammar Osaiweran (2015). Specification guidelines to avoid the state space explosion problem. *Software Testing, Verification and Reliability*, 25(1):4–33.
- [87] Jean-Claude Fernandez and Laurent Mounier (1990). Verifying Bisimulations” On the Fly”. In *Proceedings of the 3rd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FROTE 1990)*, volume 90, pages 95–110.
- [88] Jean-Michel Couvreur (1999). On-the-fly verification of linear temporal logic. In *Proceedings of the 4th International Symposium on Formal Methods (FM 1999)*, pages 253–271.
- [89] Jeffrey DiMarco and Michela Taufer (2013). performance impact of dynamic parallelism on different clustering algorithms. In *SPIE Defense, Security, and Sensing (SPIE DSS 2013)*, pages 87520E–87520E. International Society for Optics and Photonics.
- [90] Jens Ulrik Skakkebwk (1994). Liveness and fairness in duration calculus. In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR 1994)*, pages 283–298. Springer.
- [91] Jerry R. Burch , Edmund M. Clarke, Kenneth L. McMillan, David L. Dill and Lain-Jinn Hwang (1992). Symbolic model checking: 1020 states and beyond. *Information and Computation*, 98(2):142–170.

- [92] Jesse Hoey, Robert St-Aubin, Alan Hu and Craig Boutilier (1999). SPUDD: Stochastic planning using decision diagrams. In *Proceedings of the Fifteenth conference on Uncertainty in Artificial Intelligence (UAI 1999)*, pages 279–288. Morgan Kaufmann Publishers Inc.
- [93] Jian Liheng, Wang Cheng, Liu Ying, Liang Shenshen, Yi Weidong and Shi Yong (2013). Parallel data mining techniques on graphics processing unit with compute unified device architecture (CUDA). *The Journal of Supercomputing*, 64(3):942–967.
- [94] Jianye Hao, Songzheng Song, Yang Liu, Jun Sun, Lin Gui, Jin Song Dong and Ho-fung Leung (2012). Probabilistic Model Checking Multi-agent Behaviors in Dispersion Games Using Counter Abstraction. In *Proceedings of the 15th International Conference on Principles and Practice of Multi-agent Systems (PRIMA 2012)*, pages 16–30.
- [95] Jianye Hao, Songzheng Song, Yang Liu, Jun Sun, Lin Gui, Jin Song Dong and Leung, Ho-fung (2012). Probabilistic model checking multi-agent behaviors in dispersion games using counter abstraction. In *Proceedings of the 15th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA 2012)*, pages 16–30. Springer.
- [96] Jim Woodcock and Jim Davies (1996). *Using Z: specification, refinement, and proof*, volume 39. Prentice Hall Englewood Cliffs.
- [97] Jiří Barnat , Luboš Brim and Petr Ročkal (2012). On-the-fly parallel model checking algorithm that is optimal for verification of weak LTL properties. *Science of Computer Programming*, 77(12):1272–1288.
- [98] Jiří Barnat, Luboš Brim and Petr Ročkal (2007). Scalable multi-core ltl model-checking. In *Proceedings of the 14th International SPIN Workshop on Model Checking of Software (SPIN 2007)*, pages 187–203. Springer.
- [99] Jiří Barnat, Luboš Brim and Petr Ročkal (2008). Divine multi-core—a parallel ltl model-checker. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis (ATVA 2008)*, pages 234–239. Springer.

- [100] Jiří Barnat, Luboš Brim, Ivana Černá, Pavel Moravec, Petr Ročkai and Pavel Šimeček (2006). Divine—a tool for distributed verification. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, pages 278–281. Springer.
- [101] Jiří Barnat, Luboš Brim, Milan Česka and Tomas Lamr (2009). CUDA Accelerated LTL Model Checking. In *Proceedings of the 14th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pages 34–41. IEEE.
- [102] Jiří Barnat, Luboš Brim, Vojtěch Havel , Jan Havlíček , Jan Kriho , Milan Lenčo , Petr Ročkai , Vladimír Štill and Jiří Weiser (2013). DiVinE 3.0—an explicit-state model checker for multithreaded C & C++ programs. In *Proceedings of the 45th International Conference on Computer Aided Verification (CAV 2013)*, pages 863–868. Springer.
- [103] Jiří Barnat, Petr Bauch, Luboš Brim and Milan Česka (2012). Designing Fast LTL Model Checking Algorithms for Many-core GPUs. *Journal of Parallel and Distributed Computing (JPDC)*, 72(9):1083–1097.
- [104] John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone and James C. Phillips (2008). GPU computing. *Proceedings of the IEEE*, 96(5):879–899.
- [105] Jun Sun, Songzheng Song and Yang Liu (2010a). Model checking hierarchical probabilistic systems. In *Proceedings of the 12th International Conference on Formal Engineering Methods (ICFEM 2010)*, pages 388–403. Springer.
- [106] Jun Sun, Songzheng Song and Yang Liu (2010b). Model Checking Hierarchical Probabilistic Systems. In *Proceedings of the 12th International Conference on Formal Methods and Software Engineering (ICFEM 2010)*, pages 388–403. Springer.
- [107] Jun Sun, Yang Liu, Abhik Roychoudhury, Shanshan Liu and Jin Song Dong (2009). Fair Model Checking with Process Counter Abstraction. In *Proceedings of the Second World Congress on Formal Methods*, pages 123–139. Springer.

- [108] Jun Sun, Yang Liu and Jin Song Dong (2008). Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *Proceedings of the 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer.
- [109] Jun Sun, Yang Liu, Jin Song Dong and Jun Pang (2009). PAT: Towards Flexible Verification under Fairness. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*, pages 709–714. Springer.
- [110] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi and André Étienne (2013). Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Transation on Software Engineering and Methodology (TOSEM)*, 22(1):3:1–3:29.
- [111] Kenneth L. McMillan (1993a). *An Approach to the State Explosion Problem*. PhD thesis, Ph. D. Thesis, Carnegie Mellon University, 1992, CMU-CS-92-131.
- [112] Kenneth L. McMillan (1993b). Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer.
- [113] Kim Guldstrand Larsen and P. Pettersson (1999). Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science*.
- [114] Kubota Takahiro, Kakutani Yoshihiko, Kato Go, Kawano Yasuhito and Sakurada Hideki (2012). Application of a process calculus to security proofs of quantum protocols. In *International Conference on Foundations of Computer Science (FCS)*, page 1. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).
- [115] Kurt Jensen and Lars M. Kristensen (2015). Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems. *Communications of the ACM*, 58(6):61–70.
- [116] Kwiatkowska Marta, Norman Gethin and Parker David (2002). PRISM: Probabilistic symbolic model checker. In *Proceedings of the 12th International*

- Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS 2012)*, pages 200–204. Springer.
- [117] Kyle Hollins Wray, Shlomo Zilberstein and Abdel-Ilah Mouaddib (2015). Multi-Objective MDPs with Conditional Lexicographic Reward Preferences. In *29th Conference on Artificial Intelligence (AAAI 2015)*, pages 3418–3424.
- [118] Lee Taekhee and Kim Yong Jun (2013). Gpu-based motion planning under uncertainties using pomdp. In *Proceedings of the 2013 IEEE International Conference on Robotics and Automation (ICRA 2013)*, pages 4576–4581. IEEE.
- [119] Leslie Lamport (1977). Proving the correctness of multiprocess programs. *The IEEE Transactions on Software Engineering (TSE)*, (2):125–143.
- [120] Leslie Lamport and Yuan Yu (2001). TLC—The TLA+ Model Checker.
- [121] Leslie Pack Kaelbling, Michael L. Littman and Anthony R. Cassandra (1998). Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1):99–134.
- [122] Lijuan Luo, Martin Wong and Wen-mei Hwu (2010). An Effective GPU Implementation of Breadth-first Search. In *Proceedings of the 47th Design Automation Conference (DAC 2010)*, pages 52–55. ACM.
- [123] Luboš Brim, Ivana Černá, Pavel Moravec and Jiří Šimša (2004). Accepting predecessors are better than back edges in distributed ltl model-checking. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2004)*, pages 352–366. Springer.
- [124] Luis María Fioriti Ferrer, Vahid Hashemi, Holger Hermanns and Andrea Turrini (2016). Deciding probabilistic automata weak bisimulation: theory and practice. *Formal Aspects of Computing*, 28(1):109–143.
- [125] Magee Jeff and Kramer Jeff. LTS Analyzer. <https://www.doc.ic.ac.uk/~jnm/book/>. Accessed Dec 4, 2016.

- [126] Mahony Brendan and Jin Song Dong (1998). Blending Object-Z and Timed CSP: an introduction to TCOZ. In *Proceedings of the 20th International conference on Software engineering (ICSE 1998)*, pages 95–104. IEEE Computer Society.
- [127] Manchun Zheng, David Sanán, Jun Sun, Yang Liu, Jin Song Dong and Yu Gu (2013). State Space Reduction for Sensor Networks Using Two-Level Partial Order Reduction. In *Proceedings of 14th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2013)*, pages 515–535.
- [128] Marta Kwiatkowska, Gethin Norman and David Parker (2004). Prism 2.0: A tool for probabilistic model checking. In *Proceedings of the 1st International Conference on the Quantitative Evaluation of Systems (QEST 2004)*, pages 322–323. IEEE.
- [129] Marta Kwiatkowska, Gethin Norman and David Parker (2011). PRISM 4.0: Verification of probabilistic real-time systems. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, pages 585–591. Springer.
- [130] Martin L. Puterman (2014). *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.
- [131] Michael A. Nielsen and Isaac Chuang (2002). *Quantum computation and quantum information*.
- [132] Michael L. Littman, Thomas L. Dean and Leslie Pack Kaelbling (1995). On the complexity of solving Markov decision problems. In *Proceedings of the 11th Conference on Uncertainty in AI (UAI 1995)*, pages 394–402. Morgan Kaufmann Publishers Inc.
- [133] Microsoft. Microsoft solver foundation. [https://msdn.microsoft.com/en-us/library/ff524509\(v=vs.93\).aspx/](https://msdn.microsoft.com/en-us/library/ff524509(v=vs.93).aspx/). Accessed Oct 4, 2016.
- [134] Nathan Bell and Michael Garland (2008). Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation.

- [135] Noer Dennis (2013). Parallelization of the value-iteration algorithm for partially observable markov decision processes.
- [136] NVIDIA (2015). GPU Occupancy Calculator. *CUDA SDK*.
- [137] Nvidia Corporation (2012). NVIDIAs next generation CUDA compute architecture: Kepler GK110. *Whitepaper*.
- [138] Nvidia Corporation (2013). Nvidia CUDA C Programming Guide 5.5.
- [139] Nvidia Corporation (2014). CUDA C PROGRAMMING GUIDE 6.0.
- [140] Nvidia Corporation (2015). Cuda c programming guide 7.0.
- [141] Oguzhan Alagoz, Heather Hsu, Andrew J. Schaefer and Mark S. Roberts (2010). Markov decision processes: a tool for sequential decision making under uncertainty. *Medical Decision Making*, 30(4):474–483.
- [142] Paul Gastin and Denis Oddoux (2001). Fast LTL to Büchi automata translation. In *Proceedings of the 13th International Conference on Computer Aided Verification (CAV 2001)*, pages 53–65. Springer.
- [143] Paul Gastin and Pierre Moro (2007). Minimal Counterexample Generation for SPIN. In *Proceedings of the 14th International SPIN Workshop on Model Checking Software (SPIN 2007)*, pages 24–38. Springer.
- [144] Paul Gastin, Pierre Moro and Marc Zeitoun (2004). Minimization of Counterexamples in SPIN. In *Proceedings of the 11th International SPIN Workshop on Model Checking Software (SPIN 2004)*, pages 92–108. Springer.
- [145] Pawan Harish and P.J.Narayanan (2007). Accelerating Large Graph Algorithms on The GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing (HiPC 2007)*, pages 197–208. Springer.
- [146] Pawel Gepner and Michal Filip Kowalik (2006). Multi-core processors: New way to achieve high system performance. In *Proceedings of the 5th International Symposium on Parallel Computing in Electrical Engineering (PARELEC 2006)*, pages 9–13. IEEE.

- [147] Pellegrini, C. (2008). PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318–331.
- [148] Peng Dai, Daniel S. Weld, Judy Goldsmith and others (2011). Topological value iteration algorithms. *Journal of Artificial Intelligence Research (JAIR)*, pages 181–209.
- [149] Peter Arbenz, Uche Mennel and Marzio Sala (2006). Multi-level μ -finite element analysis for human bone structures. In *International Workshop on Applied Parallel Computing-State of the Art in Scientific Computing (PARA 2006)*, pages 240–250. Springer.
- [150] Petitet Antoine (2004). HPL-a portable implementation of the high-performance Linpack benchmark for distributed-memory computers. <http://www.netlib.org/benchmark/hpl/>.
- [151] Philippe Jorrand and Marie Lalire (2004). Toward a quantum process algebra. In Selinger, P., editor, *Proceedings of the 2nd International Workshop on Quantum Programming Languages*, page 111.
- [152] Pineau Joelle, Gordon Geoff, Thrun Sebastian and others (2003). Point-based value iteration: An anytime algorithm for POMDPs. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2003)*, volume 3, pages 1025–1032.
- [153] Piotr Chrzastowski-Wachtel (1985). Generalizations of Liveness and Fairness Properties.
- [154] Prism Model Checker. Probabilistic Model Checking. <http://www.prismmodelchecker.org/lectures/pmc/>. Accessed Jan 18, 2017.
- [155] Radek Pelánek (2008). Fighting state space explosion: Review and evaluation. In *Proceedings of the 13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, pages 37–52. Springer.
- [156] Rajeev Alur and David L. Dill (1994). A theory of timed automata. *Theoretical computer science*, 126(2):183–235.

- [157] Randal E. Bryant (1992). Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318.
- [158] Raphael Landaverde, Tiansheng Zhang, Ayse K. Coskun and Martin Herbordt (2014). An investigation of unified memory access performance in cuda. In *Proceedings of the 2014 IEEE International Conference on High Performance Extreme Computing Conference (HPEC 2014)*, pages 1–6. IEEE.
- [159] Rasmus Pagh and Flemming Friche Rodler (2004). Cuckoo Hashing. *Journal of Algorithms (JOA)*, 51(2):122–144.
- [160] Rehan Abdul Aziz (2010). Distributed Model Checking Using Hadoop. Technical report, Technical Report, Department of Computer Science, Aalto University, Finland.
- [161] Renault Etienne, Duret-Lutz Alexandre, Kordon Fabrice and Poitrenaud Denis (2015). Parallel explicit model checking for generalized Büchi automata. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, pages 613–627. Springer.
- [162] Richard Bellman (1956). Dynamic programming and lagrange multipliers. *Proceedings of the National Academy of Sciences of the United States of America*, 42(10):767.
- [163] Richard Bellman (1957). A markovian decision process. *Journal of Mathematics and Mechanics*, 6:679–684.
- [164] Robert Tarjan (1972). Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160.
- [165] Robin Milner (1986). *A calculus of communicating systems*. University of Edinburgh, Department of Computer Science. Laboratory for Foundations of Computer Science.
- [166] Roger Duke , Gordon Rose and Graeme Smith (1995). Object-Z: A specification language advocated for the description of standards. *Computer Standards & Interfaces*, 17(5-6):511–533.

- [167] Ron Bekkerman, Mikhail Bilenko and John Langford (2011). *Scaling up machine learning: Parallel and distributed approaches*. Cambridge University Press.
- [168] Sami Evangelista, Alfons Laarman, Laure Petrucci and Jaco Van De Pol (2012). Improved multi-core nested depth-first search. In *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis (ATVA 2012)*, pages 269–283. Springer.
- [169] Sami Evangelista, Lars Michael Kristensen and Laure Petrucci (2013). Multi-threaded explicit state space exploration with state reconstruction. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA 2013)*, pages 208–223. Springer.
- [170] Sami Evangelista, Laure Petrucci and Samir Youcef (2011). Parallel nested depth-first searches for LTL model checking. In *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA 2011)*, pages 381–396. Springer.
- [171] Sánchez, B. (2015). Algorithms for Model Checking HyperLTL and HyperCTL^{*}. In *Proceedings of the 47th International Conference on Computer Aided Verification (CAV 2015)*, pages 30–48. Springer.
- [172] SAnToS Group. Bandera. <http://bandera.projects.cs.ksu.edu/index.shtml>. Accessed Dec 4, 2016.
- [173] Shao Jie Zhang and Yang Liu (2010). An automatic approach to model checking UML state machines. In *Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion (SSIRI-C 2010)*, pages 1–6. IEEE.
- [174] Shao Jie Zhang, Jun Sun, Chengnian Sun, Yang Liu, Junwei Ma and Jin Song Dong (2013). Constraint-based automatic symmetry detection. In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE 2013)*, pages 15–25. IEEE.

- [175] Shao Jie Zhang, Jun Sun, Jun Pang, Yang Liu and Jin Song Dong (2011). On Combining State Space Reductions with Global Fairness Assumptions. In *Proceedings of the 17th International Symposium on Formal Methods (FM 2011)*, volume 6664, pages 432–447. Springer Berlin / Heidelberg.
- [176] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa and Jin Song Dong (2013). A Formal Semantics for Complete UML State Machines with Communications. In *Proceedings of the 10th International Conference on integrated Formal Methods (iFM 2013)*, pages 331–346.
- [177] Shucaï Xiao and Wuchun Feng (2010). Inter-block GPU Communication via Fast Barrier Synchronization. In *Proceedings of the 24th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2010)*, pages 1–12. IEEE.
- [178] Simon J. Gay and Rajagopal Nagarajan (2005). Communicating quantum processes. In Palsberg, J. and Abadi, M., editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 145–157.
- [179] Songzheng Song, Jianye Hao, Yang Liu, Jun Sun, Ho-Fung Leung and Jin Song Dong (2012). Analyzing multi-agent systems with probabilistic model checking approach. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 1337–1340.
- [180] Songzheng Song, Jun Sun, Yang Liu and Jin Song Dong (2012). A Model Checker for Hierarchical Probabilistic Real-Time Systems. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*, pages 705–711.
- [181] Songzheng Song, Yang Liu, Jie Zhang and Jun Sun (2014). An extensive model checking framework for multi-agent systems. In *Proceedings of the 13th International conference on Autonomous agents and multi-agent systems (AAMAS 2014)*, pages 1645–1646. International Foundation for Autonomous Agents and Multiagent Systems.

- [182] Stefan Edelkamp and Damian Sulewski (2008). Model Checking via Delayed Duplicatedetection on The GPU. In *Technical Report 821*. Dekanat Informatik, University.
- [183] Stefan Edelkamp and Damian Sulewski (2009). Parallel state space search on the GPU. In *Proceedings of the 12th International Symposium on Combinatorial Search (SoCS 2009)*.
- [184] Stefan Edelkamp and Damian Sulewski (2010). Efficient Explicit-state Model Checking on General Purpose Graphics Processors. In *Proceedings of the 17th International SPIN Workshop on Model Checking Software (SPIN 2010)*, pages 106–123.
- [185] Stefan Schwoon and Javier Esparza (2005). A Note on On-the-fly Verification Algorithms. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, pages 174–190. Springer.
- [186] Stella X Yu, Yuanlie Lin and Pingfan Yan (1998). Optimization models for the first arrival target distribution function in discrete time. *Journal of Mathematical Analysis and Applications (JMAA)*, 225(1):193–223.
- [187] Stephen Jones (2012). Introduction to dynamic parallelism. In *GPU Technology Conference (GTC 2012)*, volume 338.
- [188] Steve Schneider , Jim Davies , DM. Jackson , George M. Reed , George M. Reed and AW. Roscoe (1991). Timed CSP: Theory and practice. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 640–675. Springer-Verlag.
- [189] Steven Dalton, Luke Olson and Nathan Bell (2015). Optimizing sparse matrix-matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):25.
- [190] Sungpack Hong, Tayo Oguntebi and Kunle Olukotun (2011). Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the 2011*

- International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, pages 78–88. IEEE.
- [191] Thomas Neele , Anton Wijs , Dragan Bošnački and Jaco van de Pol (2016). Partial-Order Reduction for GPU Model Checking. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA 2016)*, pages 357–374. Springer.
- [192] Thomas Wolfgang (1990). Automata on infinite objects. *Handbook of theoretical computer science, Volume B*, pages 133–191.
- [193] Tian Huat Tan, Yang Liu, Jun Sun and Jin Song Dong (2011). Verification of Orchestration Systems Using Compositional Partial Order Reduction. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering (ICFEM 2011)*, pages 98–114. Springer Berlin / Heidelberg.
- [194] Travis S. Humble and Ronald J. Sadlier (2014). Software-defined quantum communication systems. *Optical Engineering*, 53(8):086103–086103.
- [195] Truong Khanh Nguyen, Jun Sun, Yang Liu and Jin Song Dong (2011). A model checking framework for hierarchical systems. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 633–636.
- [196] Truong Khanh Nguyen, Jun Sun, Yang Liu and Jin Song Dong (2012). Symbolic Model-Checking of Stateful Timed CSP Using BDD and Digitization. In *Proceedings of the 14th International Conference on Formal Methods and Software Engineering (ICFEM 2012)*, pages 398–413.
- [197] Truong Khanh Nguyen, Jun Sun, Yang Liu, Jin Song Dong and Yan Liu (2012). Improved BDD-Based Discrete Analysis of Timed Systems. In *Proceedings of the 18th International Symposium on Formal Methods (FM 2012)*, pages 326–340.
- [198] TW Archibald, KIM McKinnon and LC Thomas (1993). Serial and parallel value iteration algorithms for discounted markov decision processes. *European Journal of Operational Research*, 67(2):188–203.

- [199] Universiteit Twente. LTSmin Model Checker. <http://fmt.cs.utwente.nl/tools/ltsmin/>. Accessed Dec 4, 2016.
- [200] University of California, San Diego. Blast Model Checker. <http://cseweb.ucsd.edu/~rjhala/blast.html>. Accessed Dec 4, 2016.
- [201] Vahid Rafe, Mohsen Rahmani and Kianoosh Rashidi (2013). A Survey on Coping with the State Space Explosion Problem in Model Checking. *International Research Journal of Applied and Basic Sciences*, 4(6):1379–1384.
- [202] Vincent Bloemen, Alfons Laarman and Jaco van de Pol (2016). Multi-core on-the-fly SCC decomposition. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2016)*, page 8. ACM.
- [203] Wikipedia. Cloud Computing. https://en.wikipedia.org/wiki/Cloud_computing. Accessed Dec 4, 2016.
- [204] Wikipedia. Computer Cluster. https://en.wikipedia.org/wiki/Computer_cluster. Accessed Dec 4, 2016.
- [205] Wikipedia. Linear temporal logic. https://en.wikipedia.org/wiki/Linear_temporal_logic. Accessed Oct 4, 2015.
- [206] Wikipedia. List of Model Checking Tools. https://en.wikipedia.org/wiki/List_of_model_checking_tools. Accessed Dec 4, 2016.
- [207] Williams J. Stewart (1994). *Introduction to the numerical solutions of Markov chains*. Princeton Univ. Press.
- [208] Yang Liu (2009). *Model checking concurrent and real-time systems: the PAT approach*. PhD thesis.
- [209] Yang Liu, Jun Sun and Jin Song Dong (2009). Scalable multi-core model checking fairness enhanced systems. In *Proceedings of the 11th International Conference on Formal Methods and Software Engineering (ICFEM 2009)*, pages 426–445. Springer.

- [210] Yang Liu, Jun Sun and Jin Song Dong (2010). Developing Model Checkers Using PAT. In Ahmed Bouajjani and Wei-Ngan Chin, editor, *Proceedings of the 8th International Symposium on Automated Technology for Verification and Analysis (ATVA 2010)*, pages 371–377. Springer.
- [211] Yang Liu, Jun Sun and Jin Song Dong (2011). PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers. In *Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering (ISSRE 2011)*, pages 190–199.
- [212] Yi Li, Jing Sun, Jin Song Dong, Yang Liu and Jun Sun (2012). Translating PDDL into CSP# - The PAT Approach. In *Proceedings of the 17th International Conference on Engineering of Complex Computer Systems (ICECCS 2012)*, pages 240–249.
- [213] Yuan Feng, Runyao Duan and Mingsheng Ying (2012). Bisimulation for Quantum Processes. *ACM Transactions on Programming Languages and Systems*, 34(4):1–43.
- [214] Yuan Feng, Runyao Duan, Zhengfeng Ji and Mingsheng Ying (2007). Probabilistic bisimulations for quantum processes. *Information and Computation*, 205(11):1608–1639.
- [215] Yuan Feng, Yuxin Deng and Mingsheng Ying (2014). Symbolic bisimulation for quantum processes. *ACM Transactions on Computational Logic (TOCL)*, 15(2):14.
- [216] Yuanjie Si, Jun Sun, Yang Liu, Jin Song Dong, Jun Pang, Shao Jie Zhang and Xiaohu Yang (2014). Model checking with fairness assumptions using PAT. *Frontiers of Computer Science*, 8(1):1–16.
- [217] Yuxin Deng and Wenjie Du (2011). Logical, metric, and algorithmic characterisations of probabilistic bisimulation. *arXiv preprint arXiv:1103.4577*.
- [218] Zhimin Wu (2015). Parallel and Distributed Algorithms for Model Checking Problems (Doctoral Consortium). In *Proceedings of the 20th International Conference on Engineering of Complex Computer Systems (ICECCS 2015)*, pages 210–213. IEEE.

- [219] Zhimin Wu, Akin Günay, Yang Liu and Shengchao Qin (2016). Concurrent On-the-fly SCC Detection for Automata-based Model Checking with Fairness Assumption. In *Proceedings of the 21th International Conference on Engineering of Complex Computer Systems (ICECCS 2016)*, pages 135–144.
- [220] Zhimin Wu, Ernst Moritz Hahn, Akin Günay, Lijun Zhang and Yang Liu (2016). GPU-Accelerated Value Iteration for the Computation of Reachability Probabilities in MDPs. In *Proceedings of the 22th European Conference on Artificial Intelligence (ECAI)*, pages 1726–1727.
- [221] Zhimin Wu, Yang Liu, Jun Sun, Jianqi Shi and Shengchao Qin (2015). GPU Accelerated On-the-Fly Reachability Checking. In *Proceedings of the 20th International Conference on Engineering of Complex Computer Systems (ICECCS 2015)*, pages 100–109. IEEE.
- [222] Zhimin Wu, Yang Liu, Yun Liang and Jun Sun (2014). GPU Accelerated Dynamic Counterexample Generation in LTL Model Checking. In *Proceedings of the 16th International Conference on Formal Engineering Methods (ICFEM 2014)*, pages 413–429.
- [223] Zhimin Wu, Yuan Feng, Fuyuan Zhang and Yang Liu. The web page of Qubet. <https://sites.google.com/site/panqccs>. Accessed Oct 24, 2016.