HPC simulations of information propagation over complex networks

Jin, Jiangming

2014

Jin, J. (2014). HPC simulations of information propagation over complex networks.
Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/62061>

<https://doi.org/10.32657/10356/62061>

**HPC SIMULATIONS OF INFORMATION
PROPAGATION OVER COMPLEX
NETWORKS**

**JIANGMING JIN
SCHOOL OF COMPUTER ENGINEERING
2014**

HPC SIMULATIONS OF INFORMATION PROPAGATION OVER COMPLEX NETWORKS

JIANGMING JIN

School of Computer Engineering

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

2014

Acknowledgements

First, I would like to express my sincere thanks to my supervisors Dr. Stephen John Turner and Dr. Bu-Sung Lee for their continuous remarkable and patient guidance and encouragement throughout my five years' study. Their expertise and invaluable advice have left a profound impact on my work, especially in the way of critical thinking and problem solving.

I would like to thank my project colleagues Dr. Howie Kuo Shyh-hao, Dr. Rick Goh Siow Mong, and Dr. Terence Hung from Institute of High Performance Computing (IHPC), A*STAR, for their precious technical discussion and generous resource sharing on HPC simulations of infectious disease. I greatly appreciate Dr. Bingsheng He and Mr. Jianlong Zhong for the precious discussion on the system design and implementation of simulations on General Purpose Graphics Processing Unit (GPGPU) in my research project.

Special thanks should be given to Ms. Ng-Goh Siew Lai, Mr. Fan Hongfei, Mr. Li Yusen, Mr. Hao Jianan, and Mr. Shanjiang Tang, my colleagues of Parallel and Distributed Computing Center (PDCC) for their patient help along the way.

Last, but not least, I would like to thank all of my grandparents and parents for their strong support and consistent help.

Table of Contents

Acknowledgements.....	III
Table of Contents	IV
List of Figures	VII
List of Tables.....	X
Index of Abbreviations.....	XI
Abstract.....	XIII
1 Introduction.....	1
1.1 Background	1
1.1.1 Information Propagation over Complex Social Networks	1
1.1.2 High Performance Computing Architectures	3
1.2 Motivation	4
1.3 Objectives and Methodology.....	6
1.3.1 Performance Modeling and Optimization for Infectious Disease Simulation.....	7
1.3.2 Application to General Models of Information Propagation.....	8
1.3.3 Performance Studies on Multi-core CPU and GPGPU	9
1.3.4 A Case Study of Advertisement Diffusion on a Real World Network	10
1.4 Structure of the Thesis.....	10
2 Literature Review	13
2.1 High Performance Computing Architectures and Programming Models.....	13
2.1.1 Multi-core Architecture and OpenMP.....	13
2.1.2 GPGPU Architecture and CUDA.....	15
2.2 Performance Estimation and Prediction	17
2.2.1 Model-based Approach	17
2.2.2 Data Processing Approach	18
2.2.3 Heuristic Execution Approach	18
2.3 Performance Tuning and Runtime Adaptation	18
2.3.1 Tuning Level	19
2.3.2 Offline/Online Tuning.....	20
2.3.3 Runtime Adaptation	21
2.4 Applications of Information Propagation.....	22
2.4.1 Spread of Infectious Disease.....	22
2.4.2 Information Diffusion and Viral Advertising	24
2.5 High Performance Computing in Simulations of Information Propagation	27
2.5.1 Person-Location Network Approach.....	27
2.5.2 Person-Person Network Approach	28
2.6 Summary	28
3 Motivational Example: Infectious Disease Simulation	30
3.1 Agent-based Discrete-time Simulation Model	30
3.1.1 Two Types of Contact Network Browsing Algorithms	32

3.1.2	Data Structures.....	33
3.1.3	Interchangeability of Algorithms	34
3.1.4	Component-based Design	36
3.2	Performance Modeling in a Serial Execution Environment	38
3.2.1	Performance Monitoring	38
3.2.2	Algorithm Analysis	39
3.2.3	Serial Performance Models	42
3.2.4	Validation of Performance Models.....	44
3.3	Optimized Kernel Adaptation Strategy.....	45
3.4	Performance Modeling on Multi-core Architectures	48
3.4.1	Multi-threaded Parallel Processing Models	48
3.4.2	Validation of Multi-threaded Parallel Processing Models.....	50
3.5	Optimized Kernel Adaptation Strategy on Multi-core Architectures	52
3.6	Results and Analysis.....	53
3.6.1	Runtime Kernel Adaptation for Multi-threaded Parallel Processing.....	54
3.6.2	Parallel Speedup and Parallel Efficiency	55
3.6.3	Discussion on OpenMP Overhead	59
3.7	Summary	59
4	General Information Propagation Models and Optimized Simulation Strategies	61
4.1	Two Basic Information Propagation Models.....	61
4.1.1	Independent Cascade Model	62
4.1.2	Linear Threshold Model.....	63
4.2	General Models and Equivalence	64
4.2.1	General Cascade Model	65
4.2.2	General Threshold Model.....	66
4.2.3	Equivalence of General Models	66
4.3	Simulation Models in Practice.....	68
4.4	Relation to Infectious Disease Simulation.....	69
4.5	Optimized Strategies in Modeling and Simulation for Information Propagation.....	71
4.5.1	Vertex-Oriented and Edge-Oriented Approaches	72
4.5.2	Data Structures for Simulations of Networks.....	74
4.5.3	Optimized Simulation Strategies.....	76
4.6	Summary	79
5	Simulation of Information Propagation on GPGPU and Multi-core CPU	81
5.1	Simulation of Information Propagation with Medusa	81
5.1.1	System Design with Medusa.....	81
5.1.2	Experiments and Results	85
5.2	Optimized Algorithmic Adaptation Strategies for Different Networks	90
5.2.1	Random Networks.....	90
5.2.2	Scale-free Networks	92
5.3	Experiments and Results with Scalable Networks	95
5.4	Summary	98
6	Simulation of Information Propagation: Performance Studies on Multi-GPU.....	99
6.1	System Design of Multi-GPU Simulation	99

6.1.1	Network Partitioning	100
6.1.2	Distributed GPU Simulation and Synchronization	102
6.2	Performance Evaluation with Random Networks	106
6.2.1	Case Study with a 1m256m Random Network	106
6.2.2	Performance Studies with Scalable Networks.....	110
6.2.3	Discussion of Simulation Performance of Random Networks on Multi-GPU.....	116
6.3	Performance Evaluation of Scale-free Networks	117
6.4	Summary	122
7	Simulation Studies of Viral Advertisement Diffusion over a Real World Social Network	123
7.1	Background	123
7.2	Agent-based Advertisement Diffusion Model	124
7.3	Simulations Methods and Performance	126
7.3.1	Simulations with Subsequent Propagation Attempts.....	126
7.3.2	Simulations without Subsequent Propagation Attempts	129
7.3.3	Replication Ratio on Multi-GPU	132
7.3.4	Performance of Random Number Generation.....	133
7.4	Diffusion Behaviors using Different Initial Nodes Selection Policies	136
7.4.1	Simulation with Subsequent Propagation Attempts	137
7.4.2	Simulation without Subsequent Propagation Attempts	139
7.5	Discussion of Viral Advertising Strategies	142
7.6	Summary	144
8	Conclusions and Future Work.....	145
8.1	Conclusions	145
8.2	Future Work.....	148
8.2.1	Hybrid Multi-GPU Processing Scheme	148
8.2.2	Dynamic Network Structures on Multi-GPU	150
8.3	Summary	152
	Appendix A. Detailed Error Rate Analysis	153
	Appendix B. Regression Method.....	156
	Appendix C. Serial Performance Models including CCA Overhead.....	158
	Appendix D. Optimized Kernel Adaptation Strategy on Multi-core Architectures.....	161
	Author's Publications.....	163
	Bibliography	164

List of Figures

Figure 2.1. Layout of the Intel X5450 Quad-core Processors	14
Figure 2.2. Nvidia Fermi GPU Architecture (1 SM)	16
Figure 2.3. Tuning Levels of Existing Projects	19
Figure 2.4. Work Flow of Offline/Online Tuning	20
Figure 2.5. Runtime Resource/Configuration Adaptation Framework	21
Figure 3.1. Infectious Disease Spread Flow based on <i>SEIaIsR</i> Model.....	31
Figure 3.2. Description of I-Loop and S-Loop	33
Figure 3.3. Data Structure of Contact Network	34
Figure 3.4. Data Structure of State Arrays	34
Figure 3.5. State Differences between I-Loop and S-Loop	36
Figure 3.6. Layout of Component-based Infectious Disease Simulator	37
Figure 3.7. Serial Execution Time of I-Loop and S-Loop	39
Figure 3.8. Algorithms for I-Loop and S-Loop	41
Figure 3.9. Estimated and Measured Performance of Serial Execution	44
Figure 3.10. Optimized Kernel Adaptation Strategy	47
Figure 3.11. Performance of Serial Executions	47
Figure 3.12. Evaluation of Multi-threaded Parallel Performance Models.....	51
Figure 3.13. Algorithm for Kernel Swapping.....	53
Figure 3.14. Runtime Kernel Adaptation.....	55
Figure 3.15. Performance of Multi-threaded Parallel Processing.....	56
Figure 3.16. Measured and Estimated Parallel Speedup.....	57
Figure 3.17. Measured and Estimated Parallel Efficiency.....	57
Figure 4.1. Network Layout with the Cascade Model	62
Figure 4.2. Network Layout with the Threshold Model	63
Figure 4.3. Threshold Model Used to Derive Cascade Model	67
Figure 4.4. Simulation Algorithms.....	74

Figure 4.5. Two Types of Data Structures for Network Simulation	74
Figure 4.6. Simulation Algorithm of Adaptive A-Loop	78
Figure 5.1. Overview of Simulation	85
Figure 5.2. Network Structures (adopted from [119])	86
Figure 5.3. Random Network with 1M8M	87
Figure 5.4. Power-Law Network with 1M8M	89
Figure 5.5. Algorithmic Adaptation	91
Figure 5.6. R-MAT Graph Generation Model.....	93
Figure 5.7. Calculation of value of k	94
Figure 5.8. Performance Summary of Different Algorithms	96
Figure 5.9. Parallel Speedup of E-Loop and A-Loop	97
Figure 6.1. Partitioning with Vertex-Oriented Structure.....	101
Figure 6.2. Partitioning with Edge-Oriented Structure	102
Figure 6.3. Boundary and Replica Tables	102
Figure 6.4. Boundary / Replica Synchronization for 3 GPUs.....	104
Figure 6.5. Redundancy Table for Partition 1	104
Figure 6.6. Synchronization Algorithm	105
Figure 6.7. Detailed Synchronization on CPU.....	105
Figure 6.8. State Results during Simulation	107
Figure 6.9. Vertex Processing on 1m256m Random Network.....	107
Figure 6.10. Edge Processing on 1m256m Random Network.....	108
Figure 6.11. Total Execution Time of 1m256m Random Network.....	110
Figure 6.12. Total Execution Time.....	111
Figure 6.13. Total Computation Time	114
Figure 6.14. Maximum Synchronization Time	114
Figure 6.15. Average Data Transfer Time.....	115
Figure 6.16. Replication Ratio of Scalable Networks.....	116
Figure 6.17. Replication Ratio of RMAT Scale-free Networks vs. Random Networks ...	118
Figure 6.18. Cost of Data Transfer and Data Synchronization (1m128m)	118

Figure 6.19. Total Execution Time of Scale-free Networks	119
Figure 6.20. Total Execution Time of Scale-free Networks (95% nodes convergence) ...	120
Figure 7.1. Simulation with Varying Threshold (with subsequent attempts).....	127
Figure 7.2. Simulation Performance on Multi-GPU System (with subsequent attempts)	128
Figure 7.3. Parallel Speedup of Network Browsing Algorithms (with subsequent attempts)	129
Figure 7.4. Simulation with Varying Threshold (without subsequent attempts)	129
Figure 7.5. Simulation Performance on Multi-GPU System (without subsequent attempts)	130
Figure 7.6. Parallel Speedup of Network Browsing Algorithms (without subsequent attempts).....	130
Figure 7.7. Detailed Simulation Performance of Threshold = 0.9 on Multi-GPU.....	131
Figure 7.8. Detailed Simulation Performance (T = 0.9, T = 0.7).....	132
Figure 7.9. Replication Ratio	132
Figure 7.10. Performance of Random Number Generation	134
Figure 7.11. Total Parallel Speedup	135
Figure 7.12. Simulation with Different Initial Selected Nodes (with subsequent attempts)	139
Figure 7.13. Simulation with Different Initial Selected Nodes (without subsequent attempts)	141
Figure 7.14. Ratio of Final Informed Nodes with Different Initial Selected Nodes.....	142
Figure 8.1. Hybrid Multi-GPU Processing Scheme.....	149
Figure 8.2. Network Dynamics by Boolean Index	151
Figure A.1. Error Rate Analysis of I-Loop and S-Loop.....	154
Figure B.1. Ratio of $N_I(S)$ to Total Population Size	156
Figure B.2. Comparison of Regression Functions	157

List of Tables

Table 3.1.	Simulation Parameters	38
Table 3.2.	Hardware Specifications (One Computing Node)	39
Table 3.3.	Details of Error Rate	45
Table 3.4.	Ratio of $OP(i)$	51
Table 3.5.	Error Rate	58
Table 3.6.	Fraction of Non-Parallel Processing (f and OP).....	58
Table 5.1.	System Provided APIs and Parameters in Medusa	83
Table 5.2.	Simulation Parameters	86
Table 6.1.	Performance Improvement over Single GPU (Vertex-Oriented Processing)...	121
Table 6.2.	Performance Improvement over Single GPU (Edge-Oriented Processing).....	121
Table 7.1.	Simulation Parameters for Varying Thresholds	126
Table 7.2.	Time Cost of Random Number Generation (millisec)	133
Table 7.3.	Simulations of Different Initial Nodes	136
Table B.1.	Comparison of Regression Methods	157
Table C.1.	Error Rate of Serial Performance Models with CCA Overhead.....	160

Index of Abbreviations

AEOS	Automated Empirical Optimizations of Software
A-Loop	Simulation Algorithm for the Adaptive Loop
ATLAS	Automatically Tuned Linear Algebra Software
C-Loop	Simulation Algorithm for the Cascade Model
CCA	Common Component Architecture
CUDA	Compute Unified Device Architecture
CHiLL	Composable High-Level Loop Transformation and Parallelization Framework
FFTW	Fastest Fourier Transform in the West
FSB	Front Side Bus
GPGPU	General Purpose Graphics Processing Units
GSL	GNU Scientific Library
HPC	High Performance Computing
I-Loop	A Specific Case of the C-Loop, using in Chapter 3
IBP	Infect Browsing Phase
IC	Integrated Circuit
LLNL	Lawrence Livermore National Laboratory
MCH	Memory Control Hub
MPI	Message Passing Interface
OMP	OpenMP
OSKI	Optimized Sparse Kernel Interface
PHiPAC	Portable High Performance ANSI C

PAPI	Performance Application Programming Interface
RNG	Random Number Generation
RSS	Residual Sum of Squares
S-Loop	A Specific Case of the T-Loop, using in Chapter 3
SEIR	Susceptible, Exposed, Infectious, Removed
SEIaIsR	Susceptible, Exposed, Infectious-asymptomatic, Infectious-symptomatic, Removed
SIDL	Scientific Interface Definition Language
STP	State Transition Phase
SNP	Social Networking Potential
SNS	Social Network Service
T-Loop	Simulation Algorithm for the Threshold Model

Abstract

Simulation provides a flexible and valuable method to study the behavior of information propagation over complex networks. High Performance Computing (HPC) is a technology that may be used to apply optimized algorithms on powerful new hardware resources for accelerating execution performance. With the increased computing resources required for simulation of large-scale networks, it is attractive to apply HPC techniques to promote the simulation performance both in the temporal and spatial domains.

This thesis describes a simulation methodology that can facilitate simulations of information propagation in HPC environments. The novelty of our approach comes from the way we provide optimized simulation strategies and integrate such strategies with emerging computing architectures, including both Multi-core CPU and Many-core GPGPU for parallel performance acceleration.

As a motivational example, an agent-based infectious disease simulation is used to illustrate the detailed performance modeling and runtime algorithmic adaptation during simulation, both in a serial processing environment and in a Multi-core CPU parallel processing environment. Our observations indicate that a significant performance gain can be achieved based on algorithmic adaptation at runtime. However, the experimental results show that the number of CPU cores limits the scalability of parallel processing, and the OpenMP overhead in creating and destroying threads is a major factor in the bottleneck of parallel execution.

Inspired by the motivational example, we can construct a general simulation methodology with optimized execution strategies. We introduce two general simulation algorithms derived from two widely used information propagation models. In addition, two types of data structure can be defined for a given network, named the Vertex-Oriented Structure, and the Edge-Oriented Structure. According to these two data structures, we can design

two types of execution approaches: Vertex-Oriented Processing and Edge-Oriented Processing. With vertex-oriented processing, we can implement an adaptive simulation algorithm which enables different algorithms to be interchanged at runtime in order to obtain execution performance benefits. Using performance modeling and prediction, we are able to derive accurate rules for the runtime algorithmic adaptation.

According to the proposed simulation strategies, we implement the general simulation algorithms for information propagation on both Multi-core CPU and Many-core GPGPU. We adopt identical data structures and the same random number generation library on GPU and CPU platforms in order to compare and analyze the performance. Both vertex-oriented processing and edge-oriented processing are comprehensively investigated on both Random and Scale-free networks. The performance benefits of vertex-oriented and edge-oriented processing are compared and analyzed. Because of the limited size of GPU memory, the single GPU execution algorithms are further extended to multiple GPU devices. The system design is illustrated in detail, including the network partitioning and replication strategy, and the data synchronization scheme. Furthermore, the performance of distributed simulation across multiple GPUs is also investigated in detail.

Finally, as a case study, this thesis describes studies of viral advertisement diffusion using our simulation strategies on CPU and multiple GPUs. The advertisement diffusion behaviors influenced by varying thresholds and different initial nodes selection policies are investigated. According to the experimental results, we can observe that the largest degree selection policy and a large number of initial selected nodes can help in maximizing the diffusion effect in the beginning and early stages of advertisement diffusion. However, the influence of a good node selection policy in advertisement diffusion is not that crucial compared to reducing the threshold. The conclusions given by the simulation studies are comparable with real cases in marketing studies.

Introduction

1.1 Background

1.1.1 Information Propagation over Complex Social Networks

Information propagation in an information theoretical context refers to the transfer of information from one entity to another entity in a given system. Some of the earliest work in this area was by Rogers [118], who developed a theory known as the diffusion of innovations. According to this theory, there are four main elements that influence the spread of a new idea: the innovation, communication channels, time, and a social system. The social system is defined as a set of interrelated units that have certain behaviors in sociology [118]. If the information is adopted, it spreads via various communication channels. During communication, the process of information propagation occurs over time. Finally, the social system determines the consequences of information diffusion via changing social structures, roles of different entities, as well as types of policy.

Social networks play an important role in information propagation. A social network refers to a specific social structure which consists of entities (such as individuals or organizations) and the patterns/implications of the relationships between these entities [137]. A social network can be employed to model the structure of a social group, or can

be used to explain how this social structure influences the behaviors among the entities over time. In conventional social network studies using mathematical models, the individual is essentially ignored. However, many different types of relationships or attribute combinations usually form a complex network structure. These complex social networks with patterns of connection between their elements are neither purely regular nor purely random. Such features include a heavy tail in the degree distribution, a high clustering coefficient among vertices, community structure, and hierarchical structure [26].

There are two well-known and most studied complex networks: scale-free networks [15] [119] and small-world networks [126]. Both are characterized by specific structural features: power-law degree distributions for the former and short path lengths and high clustering for the latter. However, as the study of complex networks has continued to grow in importance and popularity in social and economic research issues, many other aspects of network structure have attracted attention as well. It is therefore necessary to apply advanced modeling and simulation methods to analyze the information propagation behaviors over complex social networks.

The simulation of information propagation is used to investigate the interaction behaviors between *Active* nodes and *Inactive* nodes within a given complex social network. Currently, the *Independent Cascade Model* and the *Linear Threshold Model* are widely used in studying the behaviors of information propagation over networks. In the independent cascade model, we define an initial set of active nodes A_0 at step 0. The information propagation unfolds in discrete time steps: at step t , the newly active node v_i has a single chance to activate its inactive neighbor u with an independent probability $p_{(v_i, u)} \in [0, 1]$. If v_i succeeds in activating u , u will transit its state from inactive to active at step $t + 1$ [57]. Such a process continues until no more activations are possible. In the linear threshold model, each node on the network is randomly assigned a threshold $T_u \in [0, 1]$ with a specific probability distribution (usually a

uniform distribution). At step t , each inactive node is influenced by its active neighbors (a set A_t , where A_t is $\emptyset$ if no active neighbor exists). The influence weight between the active node v_i and the inactive node u can be expressed as a probability $w(v_i, u)$. Thus, node u 's influence weight from its active neighbors can be represented by the sum of the influence from its active neighbors. If the influence weight is greater than its threshold, u 's state will transit from inactive to active at step $t + 1$.

The simulation of information propagation over complex networks has become a very important method to discover the microscopic individual behaviors over time. However, large-scale network simulations involving information propagation often require a large amount of computation and memory resources. It is therefore necessary to develop performance-oriented simulation techniques and map these optimized simulation methods onto High Performance Computing platforms.

1.1.2 High Performance Computing Architectures

Moore's Law [99] states that the number of transistors on an Integrated Circuit (IC) can be doubled approximately every two years. Such a statement has been considered as a trend of development of computer architectures since 1965. However, the trend cannot be continued forever because the exponentially increasing number of transistors in a single CPU may not significantly improve the performance due to limitations such as power dissipation. Currently, the industrial solution is not to increase the number of transistors per CPU core, but rather to increase the number of cores in a chip. The Multi-core approach significantly increases compute power through core-level parallelism. Typically, Multi-core CPUs have become the mainstream processors in the market. Although increasing the number of cores might be able to meet the increased demand for compute capacity, the technical challenges in chip design make it very expensive. In this context, heterogeneous computing systems have been popularized and are widely used in practice. In these systems, emerging accelerators such as General Purpose Graphics Processing Units (GPGPU) coordinated by a CPU have become attractive because of their

computational power in various applications. Thus, the computing landscape has undergone transformation from simply adding more transistors per single processor to a new generation of HPC architectures such as Multi-core CPU, Many-core streaming processing General Purpose GPU (GPGPU), etc.

While the technological advancement of HPC platforms is exciting, software engineers have to face the challenges of migrating legacy applications to a new programming model in order to fully exploit the capacity of these powerful new architectures. In general, different architectures have different programming languages and programming models. The OpenMP model can be used to parallelize programs on a Multi-core CPU and the GPU applications can be implemented using the CUDA programming model. To accelerate an application using an HPC system, programmers need to first understand the program behaviors of the given application. Secondly, programmers should carefully divide the program into different segments according to whether it is suitable for multi-threaded processing on the Multi-core CPU or on GPGPU accelerators. Thirdly, each segment needs to be implemented by the corresponding HPC programming model. Finally, all segments need to be integrated together to form a complete executable program.

1.2 Motivation

Simulations of large-scale networks require powerful computing resources, as well as professional programming techniques to develop and maintain the program. With the increasing problem size and complexity, the large computing demand of simulations may saturate limited computing resources, including execution time, memory usage, network bandwidth, etc. To improve computational efficiency, it is necessary to take advantage of new generation hardware such as Multi-core, GPGPU, as well as heterogeneous architectures. However, different platforms have varying architecture characteristics, especially in the design of processing units as well as hierarchical memory. It is an open research question how to execute such complex applications efficiently on these

architectures in order to reap the full benefits of these powerful new computer systems.

As simulations of information propagation over large-scale networks become more and more complex, the network simulation mechanism needs to be carefully designed. For example, proper data structures and efficient network browsing algorithms are very important for performance optimizations. For complex networks, the network structures are usually irregular due to the complicated relationships among the nodes. Such irregularity of the data structure may exhibit poor locality and excessive memory access. Thus the underlying network layout in the memory needs to be carefully optimized. It is a difficult research issue how to optimize the data structure in the memory so that benefits can be obtained in both temporal and spatial domains during simulation. Using the proper data structures, we can further design optimized network browsing algorithms in order to obtain simulation performance gains.

In this research project, we show how to develop an efficient modeling and optimization methodology for applications of information propagation. For example, the spread of infectious disease is a typical case of information propagation. The disease is carried by infectious individuals and propagated to healthy contacts with a certain transmissibility rate. In the case of infectious disease propagation, the disease can be considered as a type of information and the human contacts are the medium for disease propagation. Through the motivational example of infectious disease simulation, we can apply performance modeling to study the performance behaviors of different simulation algorithms in order to develop an optimized strategy. The different performance behaviors during simulation provide us with an opportunity to choose the most efficient algorithm at runtime in order to obtain the optimal simulation performance. In addition, these optimized runtime adaptation techniques can be applied to more general models of information propagation. The model equivalence between different general models is the basis of runtime adaptation for the corresponding simulation algorithms. In order to fulfill the feature of runtime adaptation, the data structure needs to be carefully designed and optimized.

To show the experimental results, we conduct detailed performance studies on different HPC platforms. The motivational example of infectious disease simulation is executed on a Multi-core architecture in one computing node. However, the limited number of processing units constrains the simulation performance. With the emergence of new generations of GPGPU, both the computation capacity and the memory bandwidth have exceeded those of the Multi-core CPUs. The Many-core GPGPU can provide powerful computing resources to process large-scale network simulations with massive parallelism. Significant performance benefits can then be obtained on a single GPGPU such as the Nvidia Fermi architecture. Furthermore, a Multi-GPU system can provide not only more computing resources but also more memory resources for high performance network simulation. However, multiple GPUs also introduce extra communication cost to handle data consistency since the original network is partitioned into different GPU devices. Using systematic studies of simulation performance of large scalable networks on Multi-GPU systems, the factors that influence the performance can be analyzed in detail.

Such an optimized methodology can be applied to a case study of viral advertisement diffusion over a real world social network. We can choose the most effective simulation strategy to simulate how the diffusion behaviors are influenced by different system thresholds and by different initial nodes selection policies. According to the simulation results, we can further discuss how to derive an improved marketing campaign in order to maximize the spread of information.

1.3 Objectives and Methodology

With the motivation discussed in Section 1.2, the following objectives are defined for this project.

- *Performance Modeling and Optimization of Infectious Disease Simulation*

To investigate the feasibility of performance modeling and runtime adaptation to derive optimized simulation algorithms.

- *Application to General Models of Information Propagation*

To investigate whether such optimization strategies can be applied to more general models of information propagation.

- *Performance Studies on Multi-core CPU and GPGPU*

To carry out extensive performance studies to evaluate the benefits of the optimized algorithms on HPC architectures.

- *Case Study of Viral Advertisement Diffusion over a Real World Network*

To apply these techniques to a real-world social network and to compare the simulation results with existing studies as a validation of the methodology.

The methodology adopted to achieve these objectives is discussed in detail in the following sections. Note that all the experiments are conducted over three simulation runs. The data shown in the thesis are the average values of the results over these three runs.

1.3.1 Performance Modeling and Optimization for Infectious Disease Simulation

To investigate the feasibility of runtime adaptation, a component-based simulator is developed for large-scale infectious disease simulation [117]. With the component-based design, two different network browsing algorithms (named the I-Loop and S-Loop respectively) can be designed and encapsulated as independent components enabling dynamic adaptation. Performance modeling is an analytical approach to characterize the workload, and analyze the bottleneck of the application. Using algorithmic analysis, we may construct accurate performance models to estimate the program behaviors such as the execution time of both the I-Loop and S-Loop.

Performance modeling can also provide a theoretical way to guide the algorithmic adaptation strategy. With algorithmic adaptation, an optimized execution time for the

serial simulation can be achieved compared to the I-Loop or the S-Loop. Moreover, the performance models for Multi-core architectures can be obtained by extending the serial performance models. The optimized swapping strategy for algorithmic adaptation in a multi-threaded parallel processing environment can also be expressed using mathematical models. The simulation performance using parallel processing on a Multi-core architecture is compared with the serial execution environment.

1.3.2 Application to General Models of Information Propagation

Inspired by the runtime adaptation between the I-Loop and the S-Loop in infectious disease simulation, the optimized algorithmic adaptation strategy can be applied in more general applications in the field of information propagation. Two basic models of information propagation are first introduced: the Independent Cascade Model and the Linear Threshold Model. Furthermore, these two basic models can be modified and extended to the General Cascade Model and the General Threshold Model, respectively. More importantly, these two general models can be shown to be equivalent and interchangeable. Such model equivalence provides the possibility to combine the two general models together in order to set up optimization strategies which enable algorithmic adaptation at runtime in order to choose the most efficient algorithm at each simulation step.

How to design an efficient data structure to store the network is a difficult research issue. With a proper data structure, we may achieve simulation benefits from temporal or spatial domains. One approach is to use a bidirectional graph, named the vertex-oriented structure. Vertex-oriented processing with runtime algorithmic adaptation (A-Loop) can then be conducted on this data structure. With the vertex-oriented structure, each node has a contact list for its connected neighbors, and it also appears in its neighbors' contact list. However, such a bidirectional network layout occupies a large amount of memory

resources. In order to reduce memory usage, an alternative approach is to use an undirected graph to store the network structure, named the edge-oriented structure. Since the network can be considered as an undirected graph, we only need one copy of each edge instead of two to represent the network structure. Thus, another simulation algorithm, edge-oriented processing (E-Loop) needs to be carefully designed in order to incorporate the undirected network structure.

In addition, using the performance models of different simulation algorithms, optimized strategies for simulating information propagation can be derived. These optimized strategies can be based on practical execution rules during simulation.

1.3.3 Performance Studies on Multi-core CPU and GPGPU

To demonstrate the actual simulation performance, we implement the proposed simulation strategies on both Multi-core CPU and Many-core GPGPU. The experimental results of simulation performance are comprehensively compared in multiple execution environments.

Since the algorithmic adaptation strategies may be influenced by the structure of the given network, it is necessary to explore the optimized adaptive vertex-oriented processing strategies for different network structures such as random networks and scale-free networks. The correctness and accuracy of algorithmic adaptation needs to be shown and evaluated in experiments.

Moreover, as a single GPU's memory and computation resources are limited, it is necessary to introduce multiple GPU devices to enhance the system scalability. To obtain high performance simulation on multiple GPU devices, an efficient network partitioning and replication method is very important. It is important not only to balance the workload in each GPU device but also to reduce the data communication and synchronization cost.

To show the efficiency and scalability of the simulation on a Multi-GPU system, the performance needs to be comprehensively evaluated on large scalable networks. From a careful analysis of the experimental results on multiple GPU devices, factors that influence the simulation performance can be discussed in detail.

1.3.4 A Case Study of Advertisement Diffusion on a Real World Network

To illustrate how to apply our information propagation simulation strategies to a real world complex social network, another case study on viral advertisement diffusion over the large-size LiveJournal online social network is introduced. Both the E-Loop and the A-Loop are used for the advertisement diffusion simulations.

Simulations are conducted on CPU and multiple GPUs with varying value of threshold. The simulation performance is investigated in detail. According to the performance results, we can choose the optimal simulation strategy to simulate the diffusion behaviors influenced by different initial nodes selection policies.

According to the simulation results, we can further discuss improved viral advertising strategies that maximize the propagation effect. To validate the simulation studies on viral advertisement diffusion, the conclusions given by the simulation results can be compared with existing marketing cases in the real world.

1.4 Structure of the Thesis

The thesis is organized into eight chapters.

Chapter 1 provides the background, motivation, objectives and methodology of this research.

Chapter 2 first provides an overview of HPC architectures and programming models. It then discusses performance estimation and prediction, and performance tuning and runtime adaptation. Some applications of information propagation are presented, followed by a discussion of HPC techniques in simulations of information propagation. Existing approaches and related tools are described and compared.

Chapter 3 introduces the infectious disease simulation as a motivational example. An introduction to the design of the agent-based model and runtime algorithmic adaptation framework is given. The performance modeling methodology and optimized kernel adaptation strategy for a serial and multi-threaded execution environment are proposed. The optimized kernel adaptation strategy is derived to guide the framework to obtain the most efficient execution time. Furthermore, the bottleneck restricting parallel speedup on the Multi-core architecture is explored and the major factor is identified to be the overhead of OpenMP (OMP), in particular creating and destroying threads.

Chapter 4 introduces two basic models of information propagation and their generalizations. The model equivalence provides us with an opportunity to set up optimization strategies which enable algorithmic adaptation at runtime in order to choose the most efficient algorithm at each simulation step. Two different data structures are proposed to store the network layout: the vertex-oriented structure and the edge-oriented structure. With a specific data structure, we have optimized adaptive vertex-oriented processing and edge-oriented processing schemes which can achieve performance optimization in the temporal and spatial domains respectively.

Chapter 5 shows the simulation performance of the optimized adaptive vertex-oriented processing on the vertex-oriented structure and edge-oriented processing on the edge-oriented structure on a single GPU and Multi-core CPU. Furthermore, we also present the detailed algorithmic adaptation optimizations for Random networks and Scale-free networks. The proposed optimization strategies are shown to have very good performance in practice.

Chapter 6 introduces the detailed system design of Multi-GPU simulation, including the network partitioning and replication strategy, and the data synchronization scheme. Furthermore, the simulation performance of scalable networks is carefully evaluated. The experimental results show that the number of simulation steps, computation time, synchronization time, and data transfer time affect the overall simulation performance.

To show the advantages of our simulation strategies, Chapter 7 describes a case study of viral advertisement diffusion over a realistic online social network. Improved viral advertising strategies are discussed based on the simulation studies.

Finally, Chapter 8 concludes the whole thesis by presenting a summary of achievements and a discussion of future work.

2

Literature Review

In this chapter, we discuss related work, including HPC architectures and programming models, performance estimation and prediction, performance tuning and runtime adaptation techniques, and applications of information propagation, as well as the use of HPC architectures in simulations of information propagation.

2.1 High Performance Computing Architectures and Programming Models

HPC architectures such as Multi-core CPU and Many-core GPGPU have been widely used to accelerate various applications such as simulation. Compared to traditional CPU, GPGPU has more computing processors and bandwidth. In this section, we introduce Multi-core architectures such as the Intel X5450 and GPGPU architectures such as the Nvidia Fermi. The programming models on Multi-core CPU and Many-core GPGPU are also presented.

2.1.1 Multi-core Architecture and OpenMP

A Multi-core processor is a single computing component with two or more independent cores. Multiple cores can read and execute multiple instructions at the same time in order to increase overall speed for programs amenable to parallel computing. The cores can be

integrated onto a single IC die or on multiple dies in a single chip package. The design of Multi-core architectures can vary significantly. In particular, the cache sharing between multiple cores may be different. For example, each core in one die may have its own private L1 cache but share the L2 cache with multiple cores in one die. The interconnect approaches between different cores can be bus, ring, two-dimensional mesh, or crossbar. Homogeneous multi-core systems include only identical cores. Heterogeneous multi-core systems have cores that are not identical.

The layout of the Intel Xeon X5450 Quad-core processor is shown in Figure 2.1. The Intel Xeon X5450 is designed with '*Harpertown*' 45 nm micro architecture.

Figure 2.1. Layout of the Intel X5450 Quad-core Processors

The CPU chip is composed of dual-dies with 2 cores in each. Each core owns one private 32 KB L1 cache, and each die employs a 6 MB L2 cache shared by 2 cores. Furthermore, the dual-dies share a single Front Side Bus (FSB) with speed of 1333MHz for data transferring. The FSB links to the Memory Control Hub (MCH) which is outside of the CPU chip. Two MCHs belonging to dual CPU chips can connect to the main memory in

one cluster node [66].

OpenMP [132] is a multi-threading programming model. The OpenMP model is comprised of compiler directives, runtime library routines and environment variables that are specified for C/C++ and FORTRAN programming languages. It enables programmers to implement parallel programs through pre-defined directives. Furthermore, OpenMP is portable across a variety of shared memory architectures and platforms. In particular, the basic idea behind OpenMP is to schedule multiple threads using the shared memory programming paradigm. Typically, a program with OpenMP begins as an ordinary program which executes sequentially until encountering a parallel region specified by the programmer using an OpenMP directive. The main program (master thread) creates a group of parallel threads to execute the specified statements in parallel. At the end of the region, the parallel threads are synchronized and control returns to the main program.

2.1.2 GPGPU Architecture and CUDA

Compared with CPU, a GPU can be considered as a Many-core processor. A GPU has much higher computational power and bandwidth. Because of the massive parallel processing capacity, GPUs have been applied in many applications for performance acceleration. A GPU consists of a number of Streaming Multiprocessors (SM) and a dedicated graphics memory hierarchy. All the SMs share the memory bandwidth through an interconnection network.

Figure 2.2 shows the architecture of the Nvidia Fermi SM [105]. The SM has 32 Scalar Processors (SP), which are assembled into two groups. Correspondingly, the SM has two sets of instruction issue and dispatch units. The SM is designed to execute hundreds of lightweight threads concurrently. A group of 32 concurrent threads forms a scheduling unit called a warp. Threads in the same warp are executed synchronously. The warp is the basic scheduling unit of GPU threads. When threads in the same warp execute different code paths, divergence occurs and different paths are executed sequentially. Such a thread

divergence may significantly degrade GPU performance. Besides the scalar processors, there are 16 Load/Store Units (LD/ST), which are responsible for executing memory load and store instructions. To increase the speed of executing transcendental instructions such as sin, cosine, reciprocal, and square root instructions, Fermi SM includes four Special Function Units (SFU). Each of the SP, LD/ST, or SFU units can execute one instruction per thread, per clock cycle. The two dispatch units can issue instructions from two different warps concurrently, if there is no structural or data dependence.

Figure 2.2. Nvidia Fermi GPU Architecture (1 SM)

Furthermore, each SM has a L1 cache and a scratchpad memory called shared memory. All the SMs share a L2 cache. One Nvidia Fermi C2050 GPU device has 14 SMs and each SM consists of 32 SPs. The 448 cores can share 3 GBs device memory with bandwidth of 144 GBs/sec. However, the high bandwidth utilization is achieved by coalesced memory access. A memory request issued by a warp can be coalesced into one

memory transaction if all the memory addresses fall into the same cache line of GPU memory with size of 128 bytes. Otherwise, the number of memory transactions is equal to the number of cache lines accessed.

CUDA (Compute Unified Device Architecture) is a GPU programming model developed by Nvidia. The CUDA programming model enables users to write GPU based parallel programs with familiar languages such as C/C++ or FORTRAN. For example, CUDA C is the standard C programming language with a set of language extensions. CUDA programs are called kernels, which are multithreaded and usually launched with thousands of threads. The massive number of threads is organized in a hierarchy. First, threads are organized into thread blocks. A thread block is executed on exactly one of the SMs during its life time. The size of the thread block can be one, two or three dimensional. Second, thread blocks are further divided into one or two dimensional grids. The block and grid configuration can be explicitly specified by users.

2.2 Performance Estimation and Prediction

Performance estimation and prediction of scientific applications is very challenging given the background of emerging hardware architectures such as Multi-core CPU, General Purpose Graphics Processing Units (GPGPU), etc. Currently, there are three main approaches to performance estimation and prediction.

2.2.1 Model-based Approach

The *Model-based Approach* can be applied to describe both application and hardware behaviors. In particular, workload characterization is the most important part of performance modeling. It can be used to determine the performance bottleneck of a specific application based on algorithmic analysis [68], probabilistic calculation [84], Markov Chain simulation [123], etc. The advantage of such an approach is that the performance models can be considered as a well defined bridge combining application

and platform characteristics together [55] [52] [122] [104].

2.2.2 Data Processing Approach

The *Data Processing Approach* is based on the idea of a dedicated data training tool. In general, data training tools such as various neural networks are introduced to predict performance behaviors such as network bandwidth [45], and are also used to build predictive models for architecture performance [69]. As neural networks offer black box data processing, such a performance prediction approach is able to handle data uncertainty and is not constrained by applications and platforms [80].

2.2.3 Heuristic Execution Approach

The *Heuristic Execution Approach* is based on a partial execution method. It assumes that short partial executions of an application are sufficient to predict behavior since most applications have a regular behavior after an initial startup period. Such initial execution performance provides the decision for the next stage [142]. Some existing projects have introduced such a heuristic execution method to optimize execution cost, such as ATLAS [9] for basic linear algebra subprograms, FFTW [51] for discrete Fast Fourier Transform, and SPIRAL [124] for digital signal processing.

2.3 Performance Tuning and Runtime Adaptation

In general, performance tuning techniques aim to choose the most appropriate data structures, optimized configurations, and code variants for the target platform. The performance tuning approach can be further classified according to two criteria for a more detailed analysis: Tuning Level and Offline/Online Tuning.

2.3.1 Tuning Level

In general, the process of performance tuning can be conducted at different architectural levels. The levels include *Application Level*, *Library Level*, *Compiler Level*, and *Operating System Level*. Figure 2.3 illustrates the tuning level of some existing tools.

Figure 2.3. Tuning Levels of Existing Projects

Application Level. The description of tunable parameters and the range of tunable values are accessible by APIs. The architectural characteristics of the underlying system are determined by the tunable parameters. Some application level tuning tools are integrated with library level tuning tools and compiler level tuning tools in order to provide multi-level performance optimization [129] [12].

Library Level. Highly optimized algorithmic kernels are manually developed by experienced programmers and are encapsulated into a scientific library or software package. A general library/package based approach is *Automated Empirical Optimizations of Software* (AEOS) [139]. Using the techniques of partial execution and empirical timing discussed in section 2.2.3, it is possible to choose the optimal or near optimal solution on a given execution platform.

Compiler Level. In this approach, the performance tuning is focused on both the

application and architecture, and tightly combines code structure to architecture, especially the hierarchical memory usage [23] [140] [90]. The motivation of compiler level optimization is to improve the code execution efficiency at a low level. However, due to limited knowledge of code execution behaviors on various platform, there is still a significant performance gap between hand-tuned and compiler-generated codes [32].

Operating System Level. The performance tuning tools can be integrated into an operating system. The values of tunable parameters can be adjusted within the binary code. A position paper [71] briefly describes this approach. The OS-integrated auto-tuner is implemented based on a modified Linux kernel and it is evaluated on multicore platforms for different applications [70].

2.3.2 Offline/Online Tuning

The method of *Offline Tuning* is applied to detect the architecture characteristics of the underlying platform and to search for an optimal implementation. *Online Tuning* is used to search and choose the optimal solution based on the results from runtime profiling. The work flow of *Offline/Online Tuning* phases is shown in Figure 2.4.

Figure 2.4. Work Flow of Offline/Online Tuning

The following tuning tools are all capable of performing *Offline Tuning*: Active Harmony [1], *Automatically Tuned Linear Algebra Software* (ATLAS) [9], *Optimized Sparse Kernel*

Interface (OSKI) [108], *Portable High Performance ANSI C Page for BLAS3 Compatible Fast Matrix Matrix Multiply* (PHiPAC) [133], *Fastest Fourier Transform in the West* (FFTW) [51], *SPIRAL Software/Hardware Generation for DSP Algorithms* (SPIRAL) [124], and *Composable High-Level Loop Transformation and Parallelization Framework* (CHiLL) [34]. Meanwhile, some of them also provide *Online Tuning* techniques: Active Harmony, OSKI, FFTW, and SPIRAL.

2.3.3 Runtime Adaptation

Runtime adaptation is an advanced technology to enable appropriate configurations to be substituted dynamically in order to satisfy both the application and platform preferences and constraints. Currently, there are two popular approaches to achieving such runtime adaptation: *Parameter Adaptation* and *Compositional Adaptation* [91]. In these approaches, an appropriate parameter configuration or resource composition is chosen at runtime for optimal execution. The selection can be updated dynamically according to the execution process. A generic runtime adaptation framework is shown in Figure 2.5.

Figure 2.5. Runtime Resource/Configuration Adaptation Framework

The feature of runtime adaptation is realized by interactions between three modules. They are the *Resource/Configuration Monitoring Module*, the *Resource/Configuration Scheduling Module*, and the *Resource/Configuration Steering Module* [28]. The monitoring module is designed to monitor the resource characteristics of interest to the application, as well as the progress of execution. The scheduling module collects

performance results from the monitoring module and updates the execution strategy based on performance models. The steering module performs the actual re-configuration.

The runtime adaptation mechanism has also been used in simulations, including continuous simulations [103] and parallel discrete event simulations [40]. Uhrmacher's group has proposed a more general concept of adaptive simulator to allow the adaptive mechanism to be applied across simulation domains [62]. "Plug-in Type" interfaces provide a description of simulation parameters and simulation components (i.e. simulation algorithms) to carry out simulation tasks [50]. Such an approach combines parameter adaptation and compositional adaptation. Furthermore, based on reinforcement learning techniques, the general adaptive simulator is able to choose the best configuration of existing simulation algorithms to achieve the best performance.

2.4 Applications of Information Propagation

Information propagation has recently become an active research area. In general, a rumor, a disease, or an innovation can be considered as a message propagated in a social network. Thus, modeling and simulation of information propagation over networks has been extensively studied in sociology, epidemiology, as well as the field of economics. In this section, we discuss two widely studied applications in information propagation, including the spread of infectious disease and viral advertising.

2.4.1 Spread of Infectious Disease

Two approaches have been widely used in infectious disease simulation: the *Differential Equation Approach* and the *Agent-based Network Approach*. The *Differential Equation Approach* has a longer history in modeling epidemics, with early papers such as the *Reed-Frost Epidemic Model* in 1952 [1]. Systematic studies of the dynamics and control of infectious diseases using the differential equation approach were introduced in 1991 [7]. Meanwhile, with growing computational power, the *Agent-based Network Approach*

has been increasingly applied into this research area.

2.4.1.1 Differential Equation Approach

The differential equation approach is based on mathematical methods. It can easily capture a wide range of feedbacks by the transformation of mathematical expressions. Using this approach, the population is grouped into a relatively small number of states with transitions that model the spread of infectious disease. For example, the *SEIR* model which is widely used in current research work includes the states of *Susceptible*, *Exposed*, *Infectious*, and *Removed* (or *Recovered*) [21]. However, due to the constraint of differential equations, individuals must be associated with the same attributes in each state (attributes are constrained to be homogeneous in each state) [112]. Moreover, the differential equation approach has additional assumptions such as a fixed population size and constrained dynamics of disease spread.

The differential equation infectious disease model can be classified into two categories, the *Deterministic Differential Equation Model* and the *Stochastic Differential Equation Model*. The deterministic differential equation model can only generate a single path of transition from the susceptible state to the infectious state with a mathematical approximation. This simple method is used in infectious disease studies in early papers [61] [75]. In contrast, with the advances in research work on stochastic modeling, the stochastic differential equation model is becoming increasingly exploited. Since the stochastic differential equation model produces distribution outcomes rather than specific approximations, the results may be influenced by stochastic events, uncertainty of parameters, etc. With its capability of realistic representation of the real world, such stochastic models are widely applied in recent infectious disease studies [11] [7] [63].

2.4.1.2 Agent-based Network Approach

Compared with the differential equation approach, the agent-based network approach is

able to capture easily the different attributes of people within a specific state (attributes are able to be heterogeneous) [44]. Besides, the agent-based approach is not constrained by a fixed population size defined at the beginning of the simulation. However, as each agent is associated with heterogeneous attributes, the cost of information processing of the agents increases significantly. Furthermore, the agent interaction costs depend very much on the topological complexity of the agent-based contact network [112].

Similar to the differential equation approach, infectious disease simulation based on the agent-based network approach can be classified into two categories, also known as the *Simple Random Network Model* and the *Realistic Social Network Model*. The model with a simple random network describes the infectious disease propagation behaviors with a specific transmission rate between two different states [101] [93] [94]. In contrast to the simple random network model, the realistic social network model is more realistic. Chance events occurring during the simulation procedure may change the behaviors of the disease propagation, represented by changes of network dynamics. Therefore, simulating the network dynamics and control of infectious disease becomes much more complicated. Currently, such realistic agent-based infectious disease studies have been used in social behaviors, vaccination activities as well as government control under the background of epidemic outbreaks [8] [46].

2.4.2 Information Diffusion and Viral Advertising

Most of the previous work on information propagation or information diffusion focuses on exploring the diffusion behavior through networks over time. Rogers in his book *Diffusion of Innovations* [118] proposed four main elements that influence the spread of a new idea: the innovation, communication channels, time, and a social system. That is, “diffusion is the process by which an innovation is communicated through certain channels over time among the members of a social system”. Rogers also described an S-Curve behavior in diffusion of innovations: early adopters select the innovation first, followed by the majority, until the innovation becomes common. Bass proposed a more

sophisticated probabilistic model to describe the S-Curve: the adoption progress is slow at the beginning, and then grows exponentially, the growth rate slows down again and finally reaches the whole network [18].

The Independent Cascade Model [57] and the Linear Threshold Model [59] are two basic diffusion models widely used in studying diffusion behaviors through networks. In the independent cascade model, if a neighbor v_i of node u adopts the information then node u may also adopt it with an independent probability $p_{(v_i,u)}$. Every time a neighbor of u adopts the information, there is a chance that u will decide to adopt it as well. In the linear threshold model, each node u in the network has a threshold $T_u \in [0, 1]$. The connection weight $w_{(v_i,u)}$ is assigned on every edge of the network. A node adopts the information if the sum of the connection weights from active neighbors is greater than the threshold T_u .

Furthermore, Kempe et al. [73] have shown that these two basic models can be generalized. The two general models can be shown to have equivalence. In the cascade model, authors have also shown that the diffusion behavior is affected by both the strong ties within a spatial proximate group and weak ties between groups [57] [58]. According to the detailed discussion of the diffusion behaviors over regular, small-world, and random network structures, the results show the weak ties between different groups play a more important role in propagating information [36].

As a specific application of information diffusion, viral advertising (or viral marketing) over social networks is analogous to the spread of viruses through a self-replicating viral process [136]. With the emergence of online social network services, viral advertising is enhanced by the network effects of the internet and mobile networks [65]. It can take forms of video clips, images, email messages or webpages. Viral advertising exploits existing social networks by encouraging customers to share product information with their friends. The social influence is the major force in the continuous spread of the

advertisement [67].

The term “Viral Marketing” is popularized by Rayport in the 1996 article “The Virus of Marketing” [114]. The goal of viral marketers is to create viral advertisements that appeal to individuals with great influence in the given social network. Those influential seed-individuals can spread the advertisement through their individual networks in a short period of time.

The term Social Networking Potential (SNP) is proposed to represent both the size of an individual’s social network and their ability to influence that network [136]. However, the variables of SNP can be varied according to different conditions. It is an open research question how to select the initial group under a limited budget to maximize the influence of advertisement diffusion. In the Bass Model [18], a probabilistic model is used to predict the number of people who adopt the product over time. Probabilistic modeling is also applied to develop an algorithm to maximize the viral marketing efficiently [41] [115]. Kempe et al. has shown the optimization of selecting the most influential nodes is an NP-hard problem [73]. They further proposed a greedy strategy that shows a performance guarantee with 63% near optimality in several cases [41] [73]. Leskovec et al. developed an efficient approximation algorithm which is close to the optimal solution of outbreak detection (e.g. large-size of information cascades initiated from a single or a few nodes) by exploring the sub-modularity property [82]. A maximization algorithm of selecting the most influential target nodes in dynamic networks is described in [60].

Compared to the design of efficient algorithms to maximize the diffusion influence, the heuristic and empirical approaches can be also applied to investigate the diffusion performance. Without much knowledge of the structure of a given network, heuristic selection of the nodes with large degree is an alternative and efficient way to maximize the spread of advertisement or innovation over complex social networks [35] [109] [49]. In the empirical approach, real data can be tracked during the diffusion process and used to quantify the important factors [81]. The empirical approach is also used to compare

different seeding strategies in different network structures [13] [64].

2.5 High Performance Computing in Simulations of Information Propagation

Studies of information propagation over large-scale networks usually require large amounts of computing resources. HPC simulation platforms can meet the computational requirements. In general, there are two approaches to model information propagation over complex social networks: *Person-Location Network* and *Person-Person Network*. In these approaches, HPC techniques can be used to simulate information propagation over networks in order to gain performance improvement. In this section, some existing work in HPC simulations of information propagation is presented.

2.5.1 Person-Location Network Approach

In the person-location network approach, a bipartite graph is used to represent a person-location network that indicates the detailed individual activities. EpiSims is an early parallel discrete event simulation tool to study the spread of infectious diseases in an urban environment developed at the Los Alamos National Laboratory [48]. EpiSims includes an efficient network generation method and network structure analysis algorithms. These algorithms can be mapped to large distributed processor architectures [46] [47] using MPI, a standardized and portable message-passing system [100]. Based on the EpiSims prototype, a more sophisticated HPC based epidemiology simulation tool EpiSimdemics [16] was developed. In EpiSimdemics, the parallel efficiency is improved by decoupling spatial and temporal interdependency. In the most recent implementation, the compute intensive part of EpiSimdemics is offloaded to GPU devices for further performance acceleration [19]. Both EpiSims and EpiSimdemics are based on the parallel discrete event simulation (PDES) approach. Such a person-location network approach is also applied to study the spread of malware in wireless networks [29] [30].

2.5.2 Person-Person Network Approach

Compared to the person-location network approach, the person-person network approach is also widely used in modeling and simulation of interaction-based information propagation. The person-person network is a more general approach that can be used not only in simulations of location based applications such as propagation of epidemiology but also in simulations of some other applications without location property such as information propagation on online social networks. In the person-person approach, a network is usually represented by a general graph. InterSim [77] is a general purpose graph dynamic system. It is implemented in a large-scale distributed simulation environment and it enables users to quickly produce agent-based simulations in many network applications domains. Medusa [143] is a GPU-based high performance general graph processing framework which can be modified and applied to network applications such as simulations of information propagation over complex networks.

EpiFast [20] is implemented in MPI/C++ to simulate the stochastic disease propagation in a contact network in parallel and distributed architectures. Incorporating a master-slave synchronization scheme, EpiFast shows very good scalability over distributed computing nodes. SEECN [112] is another HPC based framework for parallel contact network based information propagation simulation. SEECN reduces computational complexity by exploiting the hierarchically modular community structure. SEECN also improves cache efficiency by choosing an optimized data structure.

2.6 Summary

In this chapter, an overview of HPC architectures and programming models is first presented. Related work is then reviewed in the area of performance estimation and prediction. Three approaches, performance modeling, data processing, and heuristic execution are introduced. Performance tuning and runtime adaptation techniques are then discussed, including different tuning levels, offline/online tuning, and runtime adaptation

approaches. As this project aims to explore information propagation behaviors over complex networks, two different related applications are presented: the spread of infectious diseases and viral advertising. Finally, high performance computing approaches in simulations of information propagation over complex networks are also discussed. We notice there is a substantial amount of existing work in the area of parallel network simulation and also in runtime adaptation. However, little work has been carried out using runtime adaptation and HPC techniques together to address the performance bottleneck in large-scale network simulations. This gap provides us with an opportunity to explore this interesting area and drives the research of this thesis.

Motivational Example: Infectious Disease Simulation

In this chapter, we introduce an infectious disease simulation application as a motivational example. Two algorithms for infectious disease simulation are discussed. The design of a component-based runtime kernel adaptation scheme for agent-based infectious disease simulation is illustrated. A performance modeling approach is applied to analyze the performance behaviors during runtime.

3.1 Agent-based Discrete-time Simulation Model

The agent-based discrete-time infectious disease simulation can be viewed as a state machine model, which is designed to describe the behavior of disease spread with five epidemic states. The agent-based network represents a large-scale population in a region, where the discrete-time simulation model describes the behavior of disease spread in the human community. Each agent goes through the following states, *Susceptible* (S), *Exposed* (E), *Infectious-asymptomatic* (I_a), *Infectious-symptomatic* (I_s), and *Recovery* (R). This ‘ SEI_aI_sR ’ state machine model is similar to the ‘ $SEIR$ ’ model [21].

According to the nature of the infectious disease, there are two aspects of disease propagation that need to be modelled. One is the disease transmission between *Infectious* people and *Susceptible* people (State transition: $S \rightarrow E$). The other is the disease

progression within the *Exposed* individuals (State transition: $E \rightarrow Ia \rightarrow Is \rightarrow R$). Furthermore, in the simulation of the disease transmission between *Infectious* people and *Susceptible* people, the algorithm needs to browse the contact network. In addition, in this project, we assume that people in the *Recovery* state cannot be infected again.

Figure 3.1 shows the state transition diagram during the infectious disease spread. At the very beginning, everyone is considered as a *Susceptible* individual, and a few *Infectious-asymptomatic* individuals are introduced to the whole population. In natural society, a single infected individual may be the cause of the disease starting to spread widely. However, since the simulation uses a specific transmissibility rate by infected individuals, only introducing one infectious individual may not ensure that the simulation starts correctly. Therefore, a few infectious individuals are initially introduced. After that, other *Susceptible* people can be infected by those *Infectious-asymptomatic* individuals. Then, the infected *Susceptible* people go through the *E*, *Ia*, *Is*, and *R* states according to specific probabilities. Each member of the whole population may transit its current state to the next state at each simulation step, until the end of the simulation. In this infectious disease simulation, we specify that the simulation ends when no more *Infectious-symptomatic* individuals are left in the population.

Figure 3.1. Infectious Disease Spread Flow based on *SEIaIsR* Model

3.1.1 Two Types of Contact Network Browsing Algorithms

At each step, the most important part of the spread of infectious disease is when the *Susceptible* individuals are infected by other *Infectious* individuals (either *Infectious-asymptomatic* or *Infectious-symptomatic*). According to the infecting process of the simulation, we develop two types of algorithms to address this state updating and random number generation (RNG). Figure 3.2 shows examples of connections between different agents, where a Red Circle denotes *Infectious*, a Black Square denotes *Susceptible*, and other states are denoted by a Green Triangle.

1) *I-Loop*: The contact network browsing process starts from the infectious individuals. Each infectious individual will go through his contact list and test whether he can infect people in this contact list who are still *Susceptible* with a probability given by the transmissibility rate. If the generated random number is less than the transmissibility rate of disease spread, the simulator confirms this *Susceptible* contact is infected and the next state is changed to the *Exposed* state. In Figure 3.2 (a), the person (ID=3) with infectious state is shown as an example.

2) *S-Loop*: In contrast to the I-Loop, the S-Loop browses the contact network of *Susceptible* individuals. Each *Susceptible* individual will go through his contact list and tests whether he can be infected by *Infectious* contacts with a probability given by the transmissibility rate. Once there is a contact person where the generated random number is less than the transmissibility rate, the simulator confirms this *Susceptible* individual to be infected and the next state is changed to the *Exposed* state. Then, the loop breaks and he does not continue to browse the remaining contacts. In Figure 3.2 (b), the person (ID=6) with susceptible state is shown as an example.

Figure 3.2. Description of I-Loop and S-Loop

3.1.2 Data Structures

Since the basic behavior of agent-based infectious disease simulation is browsing the contact network and updating individuals' states with specific state transition probabilities, the data structures of both the contact network and state arrays are important factors in the application design.

As shown in Figure 3.3, the population information is stored in an array, named as *Contact Network Array*. Each individual with a specific ID is represented by an element in that array, which links to a *MAP* structure to store its contact partners' ID information. As shown in Figure 3.4, the state arrays are stored separately from the contact network. There are two state arrays to store the *Current State* and the *Next State* for all individuals, respectively. Each state array has an identical data structure and the element IDs are the same as those in the contact network array.

Such a design minimizes the cost of locking overhead of critical code sections when multiple threads are processing concurrently. That is, the state browsing operations are conducted on the contact network and the current state array, but state updating operations are conducted on the next state array. We assume that writing a single value to the state array is an atomic operation supported by the hardware. The current state will migrate to

the next state at the end of each simulation step.

Figure 3.3. Data Structure of Contact Network

Figure 3.4. Data Structure of State Arrays

3.1.3 Interchangeability of Algorithms

The varying computational complexity of the two algorithms provides an opportunity to choose the best algorithm at runtime. In order for the runtime adaptation to be valid, we need to assume that at each step, the I-Loop and S-Loop generate the same state results. However, the I-Loop and S-Loop start from different types of individuals and the contact

network browsing is different. We cannot therefore expect exactly the same behavior from both the I-Loop and S-Loop. Even if the same initial seed is used to generate random numbers by the GNU Scientific Library [56], a minor difference arises due to the different order of browsing under the I-Loop and S-Loop, which may consume different amounts of random numbers. This phenomenon results in some state differences at each simulation step.

We define the following state notations and metrics to analyze the difference between the I-Loop (I) and S-Loop (S).

- N : *Total Individuals*
- N_S : *Susceptible Individuals*
- N_E : *Exposed Individuals*
- N_{Ia} : *Infectious asymptomatic Individuals*
- N_{Is} : *Infectious symptomatic Individuals*
- N_R : *Recovered Individuals*

Using $N_{m,j}(I)$ and $N_{m,j}(S)$ to denote the number of individuals of state m at step j for the I-Loop and S-Loop respectively, the ratio of state m difference to total population size at step j can be denoted as $R_D(m, j)$.

$$R_D(m, j) = \frac{|N_{m,j}(I) - N_{m,j}(S)|}{N}$$

Figure 3.5 shows the results of the state difference ratio $R_D(m, j)$ for each simulation step. The data shown in Figure 3.5 are the average values of the results over three simulation runs. In each run, we choose different random seeds to generate a random number sequence. The plots demonstrate that, for each state, the difference per step between the I-Loop and S-Loop is very small during the whole simulation. The worst case is less than 0.5% of the total population size.

Figure 3.5. State Differences between I-Loop and S-Loop

Thus, at each step, the I-Loop and S-Loop generate the same state results. This can be expressed by

$$N_{m,j}(I) = N_{m,j}(S) \quad (3.1)$$

This supports the assumption that the I-Loop and S-Loop can be swapped at runtime for optimal performance gain.

3.1.4 Component-based Design

In order to develop a component-based infectious disease simulation, we use the *Ccaffeine Framework* [4], a Common Component Architecture (CCA) extension. The most important benefit of the CCA based approach is that it allows multiple component implementations to reside in one program. In order to achieve better performance, it supports runtime component substitution and composition to assemble the most suitable components. Since the infectious disease simulator uses contact network browsing and state updating at each step, the most compute intensive part of the program is in these two phases. The state updating process changes little during the simulation but the contact browsing routine can vary significantly. Thus, we can implement multiple contact network

browsing components as different kernels for runtime adaptation.

Figure 3.6 illustrates the structure of the component-based infectious disease simulator. There are two types of contact network browsing kernel components residing in this program. During the simulation, the CCA framework enables ‘*I-Loop*’ and ‘*S-Loop*’ components to be dynamically swapped based on a specific strategy.

Figure 3.6. Layout of Component-based Infectious Disease Simulator

In this component-based kernel adaptation design, the contact network browsing kernels do not encapsulate the state information of the contact network as this may be updated at each step of the simulation by different kernel executions. Otherwise, there would be additional cost for transferring state information when kernel swapping occurs. Furthermore, an identical interface is needed for the kernels to ensure seamless kernel substitution during execution.

In practice, a decision component is developed to decide how to swap kernels at an appropriate condition according to the performance monitoring. The kernel swapping strategy helps the driver choose the most efficient kernel. Finally, the component builder recomposes the component-based code units to be an executable program at runtime.

3.2 Performance Modeling in a Serial Execution Environment

In each simulation step, the simulator first uses a specific browsing kernel to infect *Susceptible* individuals. We denote this process as the Infect Browsing Phase (IBP). The simulator then updates the current states of all individuals not in the Susceptible or Recovery state to their next states. We denote this process as the State Transition Phase (STP). The total time cost of each simulation step should be the sum of the time taken in these two phases.

3.2.1 Performance Monitoring

In this sub-section, we discuss the measured performance of the infectious disease simulation under the I-Loop and S-Loop respectively. The simulation parameters are listed in Table 3.1 and the hardware specifications are shown in Table 3.2.

Based on the performance results generated by TAU [121], the performance monitoring tool built into the simulator, we find that the function called ‘*run_next_itr()*’, which is invoked to execute the IBP and STP phases, takes almost 100% of the total execution time. STP takes around 1% of the total computation workload during the simulation, while IBP consumes 99% of the time. Thus, this performance monitoring proves that the bottleneck is IBP, and justifies its design as multiple ‘*Kernel*’ components.

Table 3.1. Simulation Parameters

Number of Individuals (<i>Population Size</i>)	100,000
Number of Initial Infected Individuals	10
Average Contacts per Individual (<i>Contact Degree</i>)	200
Transmissibility Rate (<i>Susceptible</i> $\rightarrow$ <i>Exposed</i>)	1/100
Infect Rate (<i>Exposed</i> $\rightarrow$ <i>Infectious-Asymptomatic</i>)	1/4.5
Symptomatic Rate (<i>Asymptomatic</i> $\rightarrow$ <i>Symptomatic</i>)	1/1.0
Recover Rate (<i>Infectious-Symptomatic</i> $\rightarrow$ <i>Recovery</i>)	1/6.0

Table 3.2. Hardware Specifications (One Computing Node)

CPU	2 x Intel Xeon E5450 3 GHz Quad-core
Size of L2 Cache	2 x 12 MB (6 MB Shared by 2 Cores)
Speed of FSB	1333 MHz
Size of Main Memory	16 GB (Shared by 2 CPUs)
Scale of Parallel Processing	Max. 8 Threads by 8 Physical Cores

The performance of the two kernels, expressed as the execution time per execution step, is shown in Figure 3.7.

Figure 3.7. Serial Execution Time of I-Loop and S-Loop

In Figure 3.7 (a), the total serial execution time of the I-Loop is around $2.7 \times 10^7 \mu sec$. In Figure 3.7 (b), the total serial execution time of the S-Loop is around $8.5 \times 10^7 \mu sec$, which corresponds to roughly 3 times more than the I-Loop.

3.2.2 Algorithm Analysis

As the previous sub-section showed that the performance bottleneck is due to IBP, we would like to analyze the IBP algorithms in more detail. However, since STP is the part that updates the state at each step and allows the simulation to continue, we first describe the behavior of STP briefly.

First, we introduce additional notations:

- $N_I = N_{Ia} + N_{Is}$: *Infectious Individuals*
- D : *Contact Degree (size of contact list)*

STP has two parts to the computation, it browses the whole population and for each individual that is not in the *Susceptible* or *Recovery* state, it updates the next state according to specific probabilities. *Susceptible* individuals will be passed to and handled by IBP afterwards. In terms of computation, the first part of STP is memory accessing with a total count equal to the population size N , $\bar{E}_{STP}^{MA} = N$. The second part of STP is random number generation, with a count of $\bar{E}_{STP}^{RNG}$.

$$\bar{E}_{STP}^{RNG}(i) = N_E(i) + N_I(i) \quad (3.2)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

Since the state results of the two loops are assumed to be the same by Equation (3.1), the $\bar{E}_{STP}^{RNG}$ for the I-Loop and S-Loop can also be considered the same, expressed as

$$\bar{E}_{STP}^{RNG}(I) = \bar{E}_{STP}^{RNG}(S) \quad (3.3)$$

In contrast to STP, IBP plays the role of the infecting operation. As shown in Figure 3.8, both the I-Loop and S-Loop also have two parts to the computation. First, the loop not only browses the population but also searches target individuals' contact lists, to find the *Susceptible* or *Infectious* partners in the contact list. Then, in the second part of IBP, infecting operations will be conducted. The successfully infected *Susceptible* individuals will change their states to *Exposed*. The fundamental behavior of the first part of IBP is still memory accessing, but the total cost of this browsing is more complex than STP. It depends on the mechanism of looking for *Susceptible* or *Infectious* partners.

	Algorithm 1 I-Loop IBP	Algorithm 2 S-Loop IBP
1	for(i=0; i<N; i++) {	for(i=0; i<N; i++) {
2	/*Check if current state	/*Check if current state
3	is Infectious*/	is Susceptible*/
4	if(state==	if (state==Susceptible)
5	Infectious_Asymptomatic	for(j=0; j<contact_deg; j++)
6	Infectious_Symptomatic)	/*Check if Infectious*/
7	for(j=0; j<contact_deg; j++)	if(contact==
8	/*Check if Susceptible*/	Infectious_Asymptomatic
9	if(contact==Susceptible)	Infectious_Symptomatic)
10	infect; /*Call RNG*/	If(Infect); /*Call RNG*/
11		break;
12	}	}

Figure 3.8. Algorithms for I-Loop and S-Loop

Furthermore, the average memory accessing counts including searching the target contact lists for both the I-Loop and S-Loop can be calculated as $\bar{E}_{IBP}^{MA}(I)$ and $\bar{E}_{IBP}^{MA}(S)$.

$$\bar{E}_{IBP}^{MA}(i) = N + N_i(i) \cdot D \quad (3.4)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

The fundamental behavior of the second part of IBP is random number generation. Similarly, the average counts of random numbers generated for each target contact list can be expressed as $\bar{E}_{IBP}^{RNG}(I)$ and $\bar{E}_{IBP}^{RNG}(S)$ for the I-Loop and S-Loop respectively. For the I-Loop, the count is equal to the average number of *Susceptible* contacts per individual:

$$\bar{E}_{IBP}^{RNG}(I) = N_S(I) \cdot D/N \quad (3.5)$$

However, for random number generation in the S-Loop, there is a possibility of a loop break determined by the transmissibility rate. With this uncertainty, we can only provide a statistical estimation. The following notations are defined:

- P_T : *Transmissibility Rate*
- $\bar{N}_I(S)$: *Average Infected Contacts per Individual*

- $P_{No\ Break} : \text{Probability No Break} = (1 - P_T)^{\overline{N_I}(S)}$
- $P_{Break} : \text{Probability Break} = 1 - (1 - P_T)^{\overline{N_I}(S)}$

Here, $\overline{N_I}(S) = N_I(S) \cdot D/N$. If there is no break when going through the inner loop, the estimated count of random numbers generated can be denoted as

$$\bar{E}_{No\ Break} = (1 - P_T)^{\overline{N_I}(S)} \cdot \overline{N_I}(S)$$

In contrast, if there is break when going through the inner loop, the expectation for the count of random numbers generated before the break can be denoted as

$$\bar{E}_{Break} = \sum_{k=1}^{\overline{N_I}(S)} k \cdot (1 - P_T)^{k-1} \cdot P_T$$

Thus, the total estimated count of random numbers generated for each target contact list, $\bar{E}_{IBP}^{RNG}(S)$ is equal to the sum of $\bar{E}_{No\ Break}$ and $\bar{E}_{Break}$.

$$\bar{E}_{IBP}^{RNG}(S) = (1 - P_T)^{\overline{N_I}(S)} \cdot \overline{N_I}(S) + \sum_{k=1}^{\overline{N_I}(S)} k \cdot (1 - P_T)^{k-1} \cdot P_T \quad (3.6)$$

3.2.3 Serial Performance Models

Although STP takes 1% of the total execution time compared with 99% consumed by IBP, we take both phases into consideration to improve the accuracy of performance modeling.

Based on the analysis discussed above, the performance of the contact network browsing is affected by the data structure used to store the contact network in the memory. As shown in Figure 3.3, there is an array with N elements, which represents the total population. Each element has a map structure to store its contacts. The array structure resides within a consecutive space in main memory, while the map structure is similar to a linked list structure, which is stored in different parts of main memory. Therefore, the cost of browsing the array and map are very different.

According to the algorithms, we introduce additional notations as follows:

- $C^{Serial}(i, j)$: *Cost of i Loop Serial Execution Time at Step j*
- $T^{Serial}(i)$: *Total Cost of i Loop Serial Execution Time*

Besides these application notations, some architecture parameters are also denoted as follows:

- C_{Ba}^{MA} : *Cost of Browsing One Element in the Array*
- C_{Bm}^{MA} : *Cost of Browsing One Element in the Map*
- C_{RNG} : *Cost of Generating One Random Number .*

The calibration of the architecture parameters is performed in a dedicated computing environment using the following methods:

- By browsing the array structure, $C_{Ba}^{MA} \approx 0.016 \mu sec$.
- By browsing the contact network under the S-Loop, without random number generation, $C_{Bm}^{MA} \approx 0.184 \mu sec$.
- By generating 100 million random numbers, the cost of one random number generation can be calculated, $C_{RNG} \approx 0.021 \mu sec$.

Both application and architecture parameters have been expressed to formulate the performance characteristics. We combine the two together to define the performance models. The cost of the I-Loop and S-Loop at each step j can be expressed as follows:

$$C^{Serial}(i, j) = [\bar{E}_{STP}^{RNG}(i) \cdot C_{RNG} + 2N \cdot C_{Ba}^{MA} + N_i(i) \cdot \bar{E}_{IBP}^{RNG}(i) \cdot C_{RNG} + N_i(i) \cdot D \cdot C_{Bm}^{MA}]_j \quad (3.7)$$

In addition, the total serial execution cost of both the I-loop and S-Loop can be expressed as follows:

$$T^{Serial}(i) = \sum_{j=1}^{Steps} C^{Serial}(i, j) \quad (3.8)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

3.2.4 Validation of Performance Models

In the previous sub-section, we presented a set of performance models based on the algorithm analysis. In this sub-section, we present the validation of these performance models by comparing the estimated cost with the actual measured execution cost of simulation. Figure 3.9 shows the measured and estimated performance of both the I-Loop kernel implementation and S-Loop kernel implementation.

Figure 3.9. Estimated and Measured Performance of Serial Execution

In both Figure 3.9 (a) and Figure 3.9 (b), the two curves lie almost on top of one another, which demonstrate that the estimated performance very closely fits the measured performance cost.

In order to evaluate the accuracy of the estimated performance, criteria of the error rate of total execution cost, $E_R(i)$, and the average error rate of the execution cost, $\overline{E_R}(i)$, are defined as follows:

$$E_R(i) = \frac{|T^{Predicted}(i) - T^{Measured}(i)|}{T^{Measured}(i)} \quad (3.9)$$

$$\overline{E}_R(i) = \frac{\sum_{j=1}^{Steps} \left(|C^{Predicted}(i,j) - C^{Measured}(i,j)| / C^{Measured}(i,j) \right)}{Steps} \quad (3.10)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

Table 3.3 lists the $E_R(i)$ and $\overline{E}_R(i)$ of both the I-Loop and S-Loop. The more detailed error rate analysis can be found in Appendix A.

Table 3.3. Details of Error Rate

	I-Loop	S-Loop
$E_R(i)$	3.2%	1.7%
$\overline{E}_R(i)$	9.2%	3.8%

3.3 Optimized Kernel Adaptation Strategy

Figure 3.7 illustrates that the optimized kernel adaptation strategy is to choose the I-Loop first and then swap to the S-Loop. As we discussed in Section 3.1, the *Ccaffeine* framework allows runtime kernel substitution. In Section 3.2, we have also constructed and validated the performance models. Thus, we can compare theoretically the execution cost of $C^{Serial}(I, j)$ and $C^{Serial}(S, j)$, and then formulate the optimized kernel swapping strategy. The problem definition is expressed by

$$C^{Serial}(I, j) = C^{Serial}(S, j) \quad (3.11)$$

The condition to switch between the I-Loop and S-Loop can be either $C^{Serial}(I, j) > C^{Serial}(S, j)$ or $C^{Serial}(I, j) < C^{Serial}(S, j)$. Taking kernel swapping from I-Loop to S-Loop as an example, we can have the condition $C^{Serial}(I, j) > C^{Serial}(S, j)$. This can be simplified to the following inequality according to Equations (3.3) and (3.7).

$$N_I(I) \cdot \bar{E}_{IBP}^{RNG}(I) \cdot C_{RNG} + N_I(I) \cdot D \cdot C_{Bm}^{MA} > N_S(S) \cdot \bar{E}_{IBP}^{RNG}(S) \cdot C_{RNG} + N_S(S) \cdot D \cdot C_{Bm}^{MA}$$

Based on Equation (3.5), we have

$$N_I(I) \cdot [N_S(I) \cdot D/N] \cdot C_{RNG} + N_I(I) \cdot D \cdot C_{Bm}^{MA} > N_S(S) \cdot \bar{E}_{IBP}^{RNG}(S) \cdot C_{RNG} + N_S(S) \cdot D \cdot C_{Bm}^{MA}$$

Based on Equation (3.1), we have

$$N_I(S) \cdot [N_S(S) \cdot D/N] \cdot C_{RNG} + N_I(S) \cdot D \cdot C_{Bm}^{MA} > N_S(S) \cdot \bar{E}_{IBP}^{RNG}(S) \cdot C_{RNG} + N_S(S) \cdot D \cdot C_{Bm}^{MA} \quad (3.12)$$

A mathematical regression method is introduced to represent the relation between $\bar{E}_{IBP}^{RNG}(S)$ and $\bar{N}_I(S) = N_I(S) \cdot D/N$. Details are given in Appendix B. Here, $\bar{E}_{IBP}^{RNG}(S)$ can be expressed as $\bar{E}_{IBP}^{RNG}(S) = 0.65 \cdot N_I(S) \cdot D/N + 4.98$. Therefore, Equation (3.12) can be simplified to

$$\begin{aligned} N_I(S) \cdot [N_S(S) \cdot D/N] \cdot C_{RNG} + N_I(S) \cdot D \cdot C_{Bm}^{MA} \\ > N_S(S) \cdot [0.65 \cdot N_I(S) \cdot D/N + 4.98] \cdot C_{RNG} + N_S(S) \cdot D \cdot C_{Bm}^{MA} \end{aligned}$$

As the I-Loop and the S-Loop can be swapped at runtime, the inequality above can be further simplified to

$$N_I(i) > \frac{(C_{Bm}^{MA} \cdot D + 4.98 \cdot C_{RNG}) \cdot N_S(i) \cdot N}{C_{Bm}^{MA} \cdot N \cdot D + 0.35 \cdot N_S(i) \cdot C_{RNG} \cdot D} \quad (3.13)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

This is the critical performance point for kernel adaptation. The simulator runs the I-Loop first and swaps to the S-Loop once the runtime states result satisfies the kernel swapping condition. We refer to this adaptive execution procedure as the A-Loop.

Figure 3.10 demonstrates the procedure of the A-Loop simulation with the proposed model. For the given simulation parameters, from step 1 to step 27, the simulator chooses

the I-Loop kernel for IBP. At step 28, the simulator dynamically swaps to the S-Loop kernel for the remaining work.

Figure 3.10. Optimized Kernel Adaptation Strategy

The performance of the three execution schemes are shown together in Figure 3.11 (a), and the comparison of total execution time is displayed in Figure 3.11 (b). Figure 3.11 (a) demonstrates that the A-Loop executes the optimal kernel at each step of the simulation. Figure 3.11 (b) illustrates that, with the A-Loop implementation, the performance achieves a 14.7 speedup in total execution time compared to the S-Loop as well as a 4.7 speedup compared to the I-Loop.

(a) Comparison of the I, S, A loops execution schemes

(b) Comparison of total execution time

Figure 3.11. Performance of Serial Executions

3.4 Performance Modeling on Multi-core Architectures

We have noted that the behavior of the contact network browsing significantly affects the overall performance. By enabling multi-threaded parallel processing with OpenMP [132], we can obtain further performance gains on a shared memory multi-core architecture. In this section, we present the performance models for a multi-core parallel processing environment. Before constructing the performance models on multi-core architectures, we improve the existing serial performance models by introducing the CCA overhead (O_{CCA}). Details are given in Appendix C.

3.4.1 Multi-threaded Parallel Processing Models

Compared with serial execution, there are more factors which may affect the parallel performance. Therefore, more performance metrics are needed to represent the execution behaviors, including OMP overhead, fractions of serial and parallel execution time, etc. First, we introduce additional notations to describe the parallel performance metrics:

- f : *Estimated Time Fraction of Inherently Serial Part*
- $1 - f$: *Estimated Time Fraction of Parallelizable Part*
- O_P : *Ratio of Measured OMP Overhead by Single Thread*
- P : *Number of Threads*

The overhead includes the time to create, start, and stop multiple threads, collectively called parallel overhead [31]. Since the parallel overhead of multiple threads is difficult to calibrate at runtime, we assume O_P as a constant for managing one thread. The parallel overhead can then be denoted as $O_P \cdot P$ [31].

With knowledge of Amdahl's Law [5], we break up the performance models into an inherently serial part and a parallelizable part in order to estimate the fractions f and $1 - f$, respectively. As we have demonstrated that the serial performance models can represent the performance of the I and S Loops with good accuracy, we can estimate f and $1 - f$ based on those serial performance models.

The GNU Scientific Library (GSL) [56] is used for serial random number generation at runtime. GSL is a thread safe library that is freely available [53]. This feature ensures that the same random number sequence is generated under the same seed, even in a multi-threaded environment. However, it does not implement shared memory parallel random number generation. Thus, when multiple threads are executing concurrently, the locking mechanism in the GSL to guarantee thread safety may affect parallel performance.

Since the process of random number generation and the CCA overhead cannot be parallelized, we express the serial and parallelizable time fractions at each simulation step j by the following formulas:

$$f(i, j) = \frac{[\bar{E}_{STP}^{RNG}(i) \cdot C_{RNG} + N_i(i) \cdot \bar{E}_{IBP}^{RNG}(i) \cdot C_{RNG} + O_{CCA}]_j}{C^{Serial}(i, j)} \quad (3.14)$$

$$1 - f(i, j) = \frac{[2N \cdot C_{Ba}^{MA} + N_i(i) \cdot D \cdot C_{Bm}^{MA}]_j}{C^{Serial}(i, j)} \quad (3.15)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

Furthermore, the OpenMP Library has an overhead for creating and destroying threads. This overhead depends on the number of threads P . The overhead can be assumed to be identical at each simulation step and linear with the number of threads P for simplicity, and is expressed as $O_P \cdot P$ [31].

Thus, the parallel elapsed time $C^{Parallel}(i, j)$ for P threads at step j of kernel i is as follows, including the effect of OMP overhead.

$$C^{Parallel}(i, j) = [f(i, j) + \frac{1-f(i, j)}{P} + O_P(i) \cdot P] \cdot C^{Serial}(i, j) \quad (3.16)$$

According to Equations (3.14) and (3.15), Equation (3.16) can be expressed as

$$\begin{aligned}
 C^{Parallel}(i, j) = & [\bar{E}_{STP}^{RNG}(i) \cdot C_{RNG} + N_i(i) \cdot \bar{E}_{IBP}^{RNG}(i) \cdot C_{RNG} + O_{CCA}]_j \\
 & + [2N \cdot C_{Ba}^{MA} + N_i(i) \cdot D \cdot C_{Bm}^{MA}]_j / P \\
 & + O_P(i) \cdot P \cdot C^{Serial}(i, j)
 \end{aligned} \tag{3.17}$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

Since the RNG sequence may be used differently by the multi-threaded program, the total number of simulation steps is similar but may not be exactly identical with the serial or a single threaded execution. However, it is necessary to assume that the number of simulation steps is identical under multiple execution conditions when we derive the theoretical performance models.

The total execution cost can then be expressed as

$$T^{Parallel}(i) = \sum_{j=1}^{Steps} C^{Parallel}(i, j) \tag{3.18}$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

3.4.2 Validation of Multi-threaded Parallel Processing Models

In the last sub-section, we presented multi-threaded parallel performance models by extending the serial performance models. In this sub-section, we present the validation of these proposed performance models by comparing the estimated cost with the actual measured execution cost of simulation.

We firstly calibrate the OMP overhead $O_P(i)$ of both the I-Loop and S-Loop. We calculate the $O_P(i)$ ratios by comparing the serial execution cost and single threaded OMP execution cost. The experimental results are shown in Table 3.4. Although the $O_P(i)$ is small, it may become a significant factor reducing the overall performance when

many threads are processing concurrently.

Table 3.4. Ratio of $O_P(i)$

	I-Loop	S-Loop
$O_P(i)$	1.37%	1.24%

According to the metrics of error rate defined in Formulas (3.9) and (3.10), the error rate of the parallel performance models can be evaluated, as shown in Figure 3.12. In Figure 3.12 (a), the average error rate of the total execution cost of I-Loop is 8.05%, compared with 4.91% for the S-Loop. In Figure 3.12 (b), average error rate value of the step-based execution cost of the I-Loop is 16.54%, compared with 13.76% for the S-Loop.

(a) Total Cost Error Rate

(b) Step-based Error Rate

Figure 3.12. Evaluation of Multi-threaded Parallel Performance Models

Factors such as the accuracy of the serial models, model assumptions, low level compilation, thread scheduling efficiency of OMP, etc., may affect the parallel performance estimation. Although the error rates as shown in Figure 3.12 are not that accurate due these reasons, we can still try to derive the kernel adaptation strategy in a parallel execution environment.

3.5 Optimized Kernel Adaptation Strategy on Multi-core Architectures

The previous section proposed performance models for component-based parallel infectious disease simulation enabling kernel adaptation at runtime. Thus, we can compare theoretically the execution cost of $C^{Parallel}(I, j)$ and $C^{Parallel}(S, j)$, and then formulate the optimized kernel swapping strategy. The problem is expressed by

$$C^{Parallel}(I, j) = C^{Parallel}(S, j) \quad (3.19)$$

The comparison of $C^{Parallel}(I, j)$ and $C^{Parallel}(S, j)$ is necessary with the same baseline of state results. Equation (3.1) demonstrated that the state results of the I-Loop and S-Loop can be replaced by each other. Thus, we use a similar technique to that described in Section 3.3.

As shown in Table 3.4, the experiments show there is not a significant difference in the OpenMP overhead between the I-Loop and the S-Loop. Thus, we can assume that both the I-Loop and S-Loop have an identical OpenMP overhead during parallel processing, expressed by

$$O_P(I) = O_P(S) = O_P \quad (3.20)$$

The critical point for kernel adaptation is performed when the following expression is satisfied:

$$N_I(i) > \frac{N_S(i) \cdot [4.98 \cdot C_{RNG} \cdot P \cdot (1 + P \cdot O_P) + D \cdot C_{Bm}^{MA} \cdot (1 + P^2 \cdot O_P)]}{0.35 \cdot P \cdot N_S(i) \cdot D / N \cdot C_{RNG} \cdot (1 + P \cdot O_P) + D \cdot C_{Bm}^{MA} \cdot (1 + P^2 \cdot O_P)} \quad (3.21)$$

where $\begin{cases} i = I, I \text{ Loop State} \\ i = S, S \text{ Loop State} \end{cases}$

The details can be found in Appendix D.

As shown in Figure 3.7, the I-Loop is more efficient than the S-Loop at the beginning of the simulation. Thus, we should choose the I-Loop as the initial kernel for the simulation and then possibly swap to the S-Loop. In practice, we can initially set $i = I$ for kernel swapping defined in Inequality (3.21).

The pseudo-code of the proposed algorithm for kernel swapping is illustrated in Figure 3.13, where $Nu(i)$ and $De(i)$ are the numerator and denominator of Equation (3.21) respectively. In practice, the adaptive execution procedure is still referred to as the A-Loop.

	Algorithm 3 Kernel Swapping Strategy
1	Next_Kernel_Type = I; /* I=I-Loop; S=S-Loop. */
2	
3	while (Simulation Not Completed) {
4	if (Next_Kernel_Type == I)
5	if ($N_I(I) > Nu(I) / De(I)$)
6	Next_Kernel_Type = S;
7	else
8	if ($N_I(S) < Nu(S) / De(S)$)
9	Next_Kernel_Type = I;
10	run_next_itr();
11	}

Figure 3.13. Algorithm for Kernel Swapping

3.6 Results and Analysis

In the last two sections, we presented the performance models for parallel processing. Furthermore, we discussed a generic optimized kernel adaptation strategy for multiple

execution conditions. In this section, we evaluate the performance of runtime kernel adaptation and discuss the performance behaviors on a multi-core architecture.

3.6.1 Runtime Kernel Adaptation for Multi-threaded Parallel Processing

As the estimated execution cost of multi-threaded parallel processing for both the I-Loop and S-Loop are not that accurate, it is necessary to demonstrate whether the strategy proposed in Figure 3.13 is valid in practice.

Note that as the RNG sequence may be used differently during parallel processing; the state results may not be exactly the same in different runs of the simulation. Thus, the kernel adaptation step may be slightly different. Furthermore, the experimental results for the kernel adaptation step show that the kernel adaptation occurs within a small range of step-space, from 26 to 29 of the simulation. Figure 3.14 (a) shows a specific example of the A-Loop execution scheme when OMP=8. Figure 3.14 (b) indicates the adaptation steps under multiple execution conditions (serial and multi-threaded with 1 to 8 threads).

Furthermore, Figure 3.14 (b) also shows that there is only a single kernel swapping point that satisfies the kernel adaptation strategy proposed in Figure 3.13. That is, only one kernel swapping operation is conducted to adapt from the I-Loop to the S-Loop during the simulation.

(a) Example of A-Loop execution scheme under OMP =8

(b) Kernel adaptation steps under multiple execution schemes

Figure 3.14. Runtime Kernel Adaptation

3.6.2 Parallel Speedup and Parallel Efficiency

In parallel processing, the parallel speedup and parallel efficiency are two important performance metrics. The overall parallel speedup with P threads Sp^P and overall parallel efficiency with P threads E^P can be expressed as follows:

$$Sp^P(i) = \frac{T^{Serial}(i)}{T^{Parallel}(i)} \quad (3.22)$$

$$E^P(i) = \frac{Sp^P(i)}{P} \quad (3.23)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \\ i = A, A \text{ Loop} \end{cases}$

Figure 3.15 gives an overview of the performance of the I-Loop, S-Loop, as well as the A-Loop. Due to OMP overhead, the total time cost of single threaded execution is slightly larger than the original serial execution time. For multi-threaded execution, the time cost for each type of loop reduces as the number of threads increases. However, the time reduction slows down, especially from 5 threads to 8 threads. In particular, with up to 8 threads executing concurrently, the I-Loop, S-Loop, and A-Loop achieve at most 4.04, 4.54, and 2.78 times speedup compared to the original serial execution.

We have presented the serial and parallel performance models to estimate the execution cost of both the I-Loop and S-Loop, which have been validated with a detailed error rate analysis. In addition, these performance models are also valid for the A-Loop. Thus, we are able to compare the results of the measured and estimated parallel speedup, as shown in Figure 3.16.

Figure 3.15. Performance of Multi-threaded Parallel Processing

Figure 3.16. Measured and Estimated Parallel Speedup

Figure 3.17. Measured and Estimated Parallel Efficiency

Furthermore, the error rate metric of the overall parallel speedup is defined as follows:

$$\overline{E}_{Sp}(i) = \frac{\sum_{P=1}^{MaxP} \left(\left| Sp^{P, Predicted}(i) - Sp^{P, Measured}(i) \right| / Sp^{P, Measured}(i) \right)}{MaxP} \quad (3.24)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$, and P denotes the number of threads.

The error rates of the overall speedup averaged over 8 different multi-threaded execution conditions is shown in Table 3.5.

Table 3.5. Error Rate

	I-Loop	S-Loop	A-Loop
$\overline{E_{Sp}}(i)$	7.66%	9.79%	14.32%

The error rate of the A-Loop is a little larger than that of the I-Loop and S-Loop. That is, the much smaller execution cost of the A-Loop under both serial and parallel processing results in an increased error rate for the performance models. Therefore, the estimated parallel speedup is less accurate than with the I-Loop and S-Loop.

As shown in Figure 3.16, parallel processing achieves speedup as the number of threads increases. However, the parallel speedup is far from ideal. According to the parallel performance models we have proposed, it is therefore necessary to explore the potential reasons. The time fractions of the non-parallelizable processing are shown in Table 3.6, including the serial execution fraction and OMP overhead. Clearly, the bottleneck of parallel speedup performance is due to the OMP overhead when many threads are processing concurrently. In particular, the non-parallelizable fraction of the A-Loop is as much as 22.39% under 8 threads, compared with 12.23% for the I-Loop and 10.25% for the S-Loop.

Table 3.6. Fraction of Non-Parallel Processing (f and O_P)

	I-Loop	S-Loop	A-Loop
$f(i)$	1.27%	0.33%	5.27%
$O_P(i)$	1.37%	1.24%	2.14%
$O_P \cdot P$ ($P = 8$)	10.96%	9.92%	17.12%

By comparing the non-parallelizable processing fractions of each loop, we demonstrate the reason why the S-Loop achieves the most speedup and the A-Loop the least. It also explains the reason why the practical speedup is far from ideal.

In addition, the parallel efficiency is shown in Figure 3.17. From single thread to 4 threads, the efficiency of the I-Loop and S-Loop are quite similar. However, from 5 threads to 8 threads, the S-Loop achieves better performance than the I-Loop. Moreover, both the I-Loop and S-Loop gain much better parallel efficiency than the A-Loop. These results are comparable with the parallel speedup of each loop.

3.6.3 Discussion on OpenMP Overhead

Note that the OpenMP overhead is the bottleneck when many threads are executing in parallel. As shown in Figure 3.6, the runtime adaptation framework currently used allows algorithmic variations to be interchangeable at runtime but requires the program to be suspended at each simulation step and to wait for re-composition by the kernel adaptation modules. Thus, the OpenMP threads have to be created and destroyed at each simulation step. Such behavior eventually results in a high OpenMP overhead. Furthermore, we illustrate the performance modeling and optimized kernel adaptation methodology by using a relative small size contact network in our case study. Since simulation on small size programs is fast, it could be another reason why the OpenMP overhead takes up a relatively large portion of the total execution time. However, this overhead would be less significant when simulating networks representing large population cities.

3.7 Summary

In this chapter, we illustrate the runtime kernel adaptation design by using an infectious disease simulation as an example. The agent-based *SEIaIsR* model of infectious disease spread is presented first where an agent-based state machine is used to simulate each individual's state transition under the spread of infectious disease. Two contact network browsing kernels are described, together with an adaptation strategy to choose the best kernel at runtime. The specific design of the data structure of the contact network and state arrays is also discussed.

The performance modeling approach allows us to explore the bottleneck of a specific application as well as the characteristics of the underlying platform. Using such an analysis, the performance models can estimate accurately the execution time of both the I-Loop and S-Loop. The error rates of the estimated execution remain at a very low level. Furthermore, we demonstrate that the proposed method of performance modeling can provide a theoretical way to guide the kernel adaptation strategy, and that it works well in practice. With kernel adaptation, the total execution time of the A-Loop achieves a 14.7 times speedup compared to the S-Loop and a 4.7 times speedup compared to the I-Loop.

We also present the performance models for multi-core architectures by extending the serial performance models after including the CCA overhead. Moreover, the optimized kernel adaptation strategy in a multi-threaded parallel processing environment is expressed using mathematical models. With parallel processing on a multi-core architecture, the I, S, and A-Loops further obtain up to 4.04, 4.54, and 2.78 times speedup with up to 8 threads compared to the serial execution. Furthermore, the bottleneck of parallel execution is explored analytically using the performance models. It is shown that the OpenMP overhead is a major factor that restricts the parallel performance.

General Information Propagation Models and Optimized Simulation Strategies

In this chapter, we show how the techniques developed in the previous example can be applied to more general models of information propagation. There are two basic models widely used in studies of information propagation: the independent cascade model and the linear threshold model. Based on these two models, we show a method to generalize them and we also show their equivalence. In addition, we relate these general models to the motivational example of infectious disease simulation which has been analyzed in detail. Furthermore, we introduce optimized strategies for modeling and simulation of information propagation over complex networks.

4.1 Two Basic Information Propagation Models

In modeling and simulation of information propagation over complex networks, the given network can be represented by a directed or undirected graph G . Each node in the network is considered as either *Active* (v) or *Inactive* (u). Usually, the process of information propagation or information diffusion follows a one way process. That is, the nodes can transit state only from *Inactive* to *Active*, but cannot change from *Active* to *Inactive*. At the beginning of the simulation, a set of nodes are initialized as *Active* while the others are considered as *Inactive*. During the process of information propagation, the

inactive nodes are influenced by their active neighbors and they may change their states from *Inactive* to *Active*. The independent cascade model and the linear threshold model are the most basic and most widely used models in studies of information propagation behaviors over networks.

4.1.1 Independent Cascade Model

In the Independent Cascade Model [57], we define an initial set of active nodes A_0 at step 0. The information propagation unfolds in discrete time steps: at step t , the active node v_i has a chance to activate its inactive neighbor u with an independent probability $p_{(v_i,u)} \in [0, 1]$. If v_i succeeds in activating its neighbor u , u will transit state from inactive to active at step $t + 1$. Such a process continues until no more possible activations are available. As shown in Figure 4.1, a directed graph is used to represent the cascade process on a network. The cascade process starts from the active nodes and pushes the information by the directed edges to the linked inactive nodes. For example, as shown in Figure 4.1, suppose n_1 denotes an active node. It can cascade information to its inactive neighbors n_2 , n_6 , and n_7 with probability p_{12} , p_{16} , and p_{17} respectively.

Figure 4.1. Network Layout with the Cascade Model

In [73], the cascade model is defined with an additional assumption that whether or not v_i succeeds in activating u at step t , v_i cannot make further attempts to activate u in

subsequent steps. Such an assumption indicates that the random number generated to compare with the probability on the corresponding edge is fixed. Thus, subsequent attempts cannot be made if it failed the first time. However, this assumption is not mentioned in the original paper [57] and some other literature [58] [81].

4.1.2 Linear Threshold Model

In the Linear Threshold Model [59], each node in the network is randomly assigned a threshold $T_u \in [0, 1]$ with a specific probability distribution (usually with uniform distribution). At step t , each inactive node is influenced by its active neighbors (a set A_t , where A_t is $\emptyset$ if no active neighbor exists). The influence weight between the active node v_i and the inactive node u can be expressed as $w_{(v_i, u)}$. Thus, node u 's influence weight from its active neighbors can be calculated and represented by $\sum_{v_i \in A_t} w_{(v_i, u)}$ (note that $\sum_{v_i \in N} w_{(v_i, u)} \in [0, 1]$, where N is the set of all neighbors). At step t , u transits its state from inactive to active if $\sum_{v_i \in A_t} w_{(v_i, u)} > T_u$. As shown in Figure 4.2, a directed graph is used to represent the threshold model where the directed edge represents the influence. Each inactive node may adopt the information by receiving an influence from its active neighbors.

Figure 4.2. Network Layout with the Threshold Model

For example, suppose n_4 denotes an inactive node in Figure 4.2. It can receive an influence weight from its active neighbors n_2 , n_3 , and n_5 . Compared to the cascade behavior in the independent cascade model, the threshold model represents the nature of collecting information from others.

Both the two models represent the interactive behaviors between active and inactive nodes, but with different perspectives. The independent cascade model is based on the active nodes, while the linear threshold model is based on the inactive nodes. Furthermore, the cascade model represents the influence cascade behavior from the active node to its inactive neighbors independently but the threshold model represents the collective behavior of the inactive node which receives influence from all its active neighbors.

4.2 General Models and Equivalence

The independent cascade model and linear threshold model have been introduced above. In [73], the authors show the model generalizations and model equivalence. In their definition of the independent cascade model, there is a constraint that only a newly active node can trigger the activation of its inactive neighbors with a specific probability. This means that the random numbers used to compare with the probabilities $p_{(v_i,u)}$ are static during simulation. Thus, the active node has only one chance to activate its neighbors. However, it is possible in many real applications such as the spread of infectious disease and some viral advertising that every active node on the network may make frequent activation attempts on its neighbors. That is, each pair of connected nodes may have interactions at different time steps and there are possibly subsequent activations after the first attempt. For example, in infectious disease studies with SIR/SEIR models, any susceptible (inactive) individual has the possibility to catch the disease from any infected (active) neighbor at any time step [63] [101] [102]. More importantly, the constraint no longer allows the study of issues such as heterogeneity in dynamics of information propagation on networks [112] [113]. Thus, such an assumption can be removed. For

example, the contact network structure and the system transmission rate can be changed due to policy controls such as school closure during an epidemic. These interventions during simulation lead to the heterogeneity in information propagation. In our generalization, the active node can propagate the information to its inactive neighbors with or without subsequent activations according to the needs of different applications.

A detailed study of model equivalence between the cascade model and the threshold model can be found in [73]. A similar method can be followed for our model generalizations to show their equivalence. Our generalized models are named the General Cascade Model and the General Threshold Model respectively. According to these two general models, we can show their equivalence in order to unify the two information propagation models together.

4.2.1 General Cascade Model

In the general cascade model, we remove the assumption that exists in [73] that only a newly active node has a single chance to activate its inactive neighbors. According to the needs of different applications, the active nodes can propagate the information with or without subsequent attempts. Consider the following assumption in the process of information propagation: at step t , the inactive node can be activated by its active neighbors in an arbitrary order with the same result. It indicates that if there is a set of active neighbors (denoted as $A_t = \{v_1, v_2, \dots, v_l\}$) around u , u can be activated at the end of these l attempts no matter in which order the attempts are made [73]. In the general cascade model, the probability that an active node v_i ($v_i \in A_t$) succeeds in activating u can be represented by a specific probability $p_{(v_i, u)}(A'_t) \in [0, 1]$, where A'_t is the set of active neighbors that have already tried to activate u but failed. However, the overall probability that u is activated by A_t can be expressed as an incremental probability $1 - \prod_{v_i \in A_t} (1 - p_{(v_i, u)})$, where $p_{(v_i, u)}$ is an independent probability. The total probability increases as the size of A_t increases.

4.2.2 General Threshold Model

The general threshold model is similar to the original linear threshold model. In the linear threshold model, each inactive node needs to calculate the sum of influence weights $\sum_{v_i \in A_t} w_{(v_i, u)}$ from u 's active neighbors. Here, A_t denotes the set of all active neighbors of the inactive node. The inactive node u can then transit its state to become active if $\sum_{v_i \in A_t} w_{(v_i, u)} > T_u$, where T_u is the threshold. According to the definition of the linear threshold model, we see that the total influence weight on u increases with an increased number of its active neighbors. Therefore we can define a monotonic incremental function $f_u(A_t) \in [0, 1]$ to represent the sum of influence weight around u . Each inactive node u is initially assigned a threshold $T_u \in [0, 1]$. The inactive node u transits its state to active if $f_u(A_t) > T_u$. In particular, the linear threshold model is a special case that $f_u(A_t) = \sum_{v_i \in A_t} w_{(v_i, u)}$ [73].

4.2.3 Equivalence of General Models

We have introduced two general models in the previous sub-section. We can show that these two models are indeed equivalent. As shown in the general cascade model, the inactive node u can be activated by its active neighbors $A_t = \{v_1, v_2, \dots, v_l\}$ in an arbitrary order. We have assumed that the order of attempts does not affect the final result. Thus, the overall probability that u is activated by active neighbors in A_t can be expressed as an incremental probability $1 - \prod_{v_i \in A_t} (1 - p_{(v_i, u)})$. Therefore we can derive the monotonic incremental function in the general threshold model from the incremental probability in the general cascade model. The incremental function can be defined as $f_u(A_t) = 1 - \prod_{v_i \in A_t} (1 - p_{(v_i, u)})$. The threshold T_u is assumed to be uniformly distributed. In addition, the definition of active set A_t may vary since applications may or may not allow subsequent attempts in the cascade model. In those applications where the active nodes can conduct subsequent propagation attempts, A_t denotes the active

nodes at step t . However, in applications where the active nodes cannot conduct subsequent propagation attempts A_t denotes the newly activated nodes that appear at step t .

Furthermore, given an instance of the general threshold model with a threshold function of $f_u(A_t)$, we can also derive an equivalent general cascade model. Consider the case in the general cascade model that an additional active node v_i activates u when nodes in A'_t have already tried but failed. This behavior has an analogy to the general threshold model that $f_u(A'_t) < T_u < f_u(A'_t \cup \{v_i\})$, where $v_i \in A_t$, $v_i \notin A'_t$. As nodes in A'_t have tried but failed, we can conclude that the threshold of u must be within the range of $T_u \in (f_u(A'_t), 1]$. As shown in Figure 4.3, the additional active node v_i adds an influence weight to u and activates u if $f_u(A'_t \cup \{v_i\}) > T_u$. Such a successful activation influence weight can be used to derive a probability in the general cascade model that v_i succeeds in activating u . Assuming T_u is uniformly distributed, this probability can be expressed as $p_{(v_i, u)}(A'_t) = \frac{f_u(A'_t \cup \{v_i\}) - f_u(A'_t)}{1 - f_u(A'_t)}$.

Figure 4.3. Threshold Model Used to Derive Cascade Model

We have shown that the overall probability of activating an inactive node in the general cascade model can be used to derive an incremental function in the general threshold model. Moreover, the monotonic incremental function in the general threshold model can be converted to a specific probability that one active node succeeds in activating the given

inactive node.

4.3 Simulation Models in Practice

In the previous section, we have shown a method of generalizing the two basic models. In addition to the model generalizations, we have also demonstrated the equivalence between these two general models. The model equivalence in information propagation provides an opportunity that we can combine the two general models together and to choose the optimal one at runtime in simulation of the information propagation behaviors over complex networks. In the agent-based modeling approach, each node in the network is considered as an agent. Each agent can push the information to others and it can also receive information from others. The interactive behaviors between agents are mutual and may or may not allow subsequent attempts according to the needs of different applications.

We can use the general cascade model to derive the general threshold model. In particular, we can start the simulation process from the inactive node and trace the independent probabilities from the active neighbors of the given inactive node. In practice, we introduce a system transmission probability P to each node in the network. As a specific case of the general cascade model, this means each available attempt can make a successful activation with a fixed probability $p_{(v_i, u)} = P$. Thus, the overall probability to activate an inactive node u can be expressed as $1 - (1 - P)^{|A_t|}$, where $|A_t|$ denotes the number of active neighbors of u at step t . Such an incremental probability can be used to derive a monotonic incremental function in the general threshold model that $f_u(A_t) = 1 - (1 - P)^{|A_t|}$.

We can also use the general threshold model to derive the general cascade model. According to the model equivalence which has been discussed in the previous section, the probability of the activation attempt by node v_i can be expressed as

$$p_{(v_i, u)} = \frac{f_u(A'_t \cup \{v_i\}) - f_u(A'_t)}{1 - f_u(A'_t)} = \frac{1 - (1-P)^{|A_t|} - [1 - (1-P)^{|A_t|-1}]}{1 - [1 - (1-P)^{|A_t|-1}]} = \frac{(1-P)^{|A_t|-1} - (1-P)^{|A_t|}}{(1-P)^{|A_t|-1}} = P.$$

Therefore, we show the model equivalence in this special case.

In practice, we can assign a random number r to each edge and compare the random number to the pre-defined P in order to decide whether the activation attempt succeeds. The inactive node u can be activated if the value of the random number is less than the pre-defined transmission rate ($r < P$). According to whether the active nodes can propagate information subsequently, the assignment of uniform random numbers on the edges can be varied. In applications with subsequent attempts, the random numbers are re-assigned to every edge element at each simulation step in order to allow subsequent attempts to be made. However, in those applications without subsequent attempts, the random numbers are assigned only at the initialization step.

We have an important conclusion that the overall probability for an inactive node to accept the information increases with the increased number of active neighbors. It also indicates the phenomenon in collective behavior that persons may not feel pressure from their social connections when only a few people adopt the information, but they may suddenly change their mind to accept the information if the persons around them who have adopted the information reach a certain number [59].

4.4 Relation to Infectious Disease Simulation

The spread of infectious disease is a specific case of information propagation. With the widely used SIR model or SEIR model, the susceptible (S) individuals may be infected by their infectious (I) neighbors with a specific transmissibility and then transit state from susceptible to infectious (or exposed (E) then infectious in the SEIR model) and finally removed (R) from the epidemic. Although there are more types of state transitions during the spread of an infectious disease, the interaction between different individuals only happens between the susceptible individuals (S) and the infectious individuals (I), which

is similar to the interaction between the inactive nodes and the active nodes in the general information propagation models.

The infection behavior can be considered as an independent operation without history or impact from other infections. In addition, it is possible to have subsequent infection behaviors after the first infection attempt during the spread of epidemics. Both the general cascade model and the general threshold model can be applied to simulate the interactive behaviors between susceptible individuals and infectious individuals. In particular, a specific transmissibility can be considered as the system transmission rate applied to every susceptible individual. Although the two general models are equivalent, the number of contact neighbors to visit varies. It means that the general cascade model starts from infectious individuals to trace the susceptible neighbors, compared to the general threshold model where it traces susceptible individuals' infectious neighbors. Such a difference may result in different execution cost during simulation. The model equivalence provides an opportunity for performance optimization if a crossover of simulation performance exists. In these cases, we can swap different models at runtime in order to obtain performance gain.

In the agent-based infectious disease simulation described in chapter 3, we apply two types of simulation algorithms named I-Loop and S-Loop. In the I-Loop, the simulation starts from the infectious individuals (active nodes) and propagates the disease to their susceptible contacts (inactive neighbors) with the system transmission rate (transmissibility). The infection succeeds if the random number assigned to the edge is less than the transmissibility. Compared to the I-Loop, the S-Loop starts from the susceptible individuals in the contact network and each susceptible individual's infectious contacts conduct independent infection attempts in an arbitrary order. The susceptible individual transits its state if any infection (out of the attempts by all active neighbors) succeeds. It is therefore not difficult to conclude that our proposed I-Loop and S-Loop in the agent-based infectious disease simulation represent the specific cases of the general cascade model and the general threshold model.

Furthermore, the state differences between the I-Loop and S-Loop in infectious disease simulation are shown in Figure 3.5. Due to the randomness of stochastic number generation, the I-Loop and S-Loop may not produce exactly the same epidemic states during simulation but the state differences are very small. As the worst case is less than 0.5% of the total population in experiments, we can consider the I-Loop and S-Loop (as specific cases of the general cascade model and the general threshold model) to be equivalent in practice. The experimental results provide a practical demonstration of the model equivalence.

4.5 Optimized Strategies in Modeling and Simulation for Information Propagation

In this section, we present the optimized simulation algorithm in information propagation derived from the two general models we have introduced in the previous sections. In order to ensure the interaction behaviors can propagate information among different nodes in the network, it is necessary to design suitable simulation algorithms on the corresponding data structures. In this thesis, we assume that the relationship between nodes is mutual.

As mentioned previously, each node can receive information from others when it is inactive and it can also push the information to others when it is active. Therefore, we can build a vertex-oriented structure where each node can trace its connected neighbors' states during simulation. In the vertex-oriented approach, we can trace the connected nodes' states from an active node via the directed edges and make activation attempts to propagate information (cascade model). We can also trace the connected nodes' states from an inactive node via the directed edges to receive the information from other active nodes (threshold model). This requires that each node can reach its connected nodes via the edges and can also be reached by its connected nodes in a reverse way. Thus, the vertex-oriented approach is designed with bidirectional network structure and it enables runtime model adaptation.

Furthermore, we introduce another algorithm to process the simulation using an edge-oriented approach. In the edge-oriented approach, an edge is used to represent the contact relation and we can trace a pair of connected nodes' states via the edge between them. As the information flow is decided by states of the connected nodes, the edge-oriented structure can be designed with an undirected network structure.

According to the underlying execution scheme of the different simulation algorithms, we introduce different data structures to store the network layout. As discussed in the motivational example on infectious disease simulation, the I-Loop and S-Loop show very different simulation cost during simulation. Although we have applied a detailed performance modeling approach to explain the factors which affect the simulation performance, it is more attractive to give a general approach to demonstrate the behavior of simulation performance by the different algorithms. Thus, we show the way to model the computational complexity for the different simulation algorithms which can be used for algorithmic adaptation at runtime to achieve the performance gain.

4.5.1 Vertex-Oriented and Edge-Oriented Approaches

We can consider the information propagation with subsequent activation attempts as a case study to illustrate our optimized simulation strategies. In addition, we can further assume that each node can be activated by a specific system probability P , such as the transmissibility in SIR infectious disease studies on SARS [95]. According to the model definition, we first introduce two types of simulation algorithms named as C-Loop, and T-Loop. The C-Loop and T-Loop can be considered as the general simulation approaches to represent the cascade model and the threshold model respectively.

- C-Loop: Starting from the active nodes in the network, each active node will go through its contact neighbors at each simulation step and test whether it can propagate the information to the inactive neighbors with a specific probability. If

the inactive nodes receive the information, they will change state to active at the next step.

- T-Loop: In contrast to the C-Loop, the T-Loop starts from the inactive nodes and traces the contact neighbors' state at each step. The inactive node can receive information from any active neighbor with a specific probability. If successful, the node will change state to active at the next step.

Besides the vertex-oriented approach such as the C-Loop or the T-Loop, we can also process the simulation using an edge-oriented approach. It is therefore necessary to introduce another network browsing algorithm based on edge-oriented, named as the E-Loop.

- E-Loop: Different from the vertex-oriented approach, the E-Loop starts from each edge element and checks the state of the connected pair of nodes. If the two connected nodes have heterogeneous state such as *Active - Inactive*, the information can be propagated from active to inactive with the given probability.

Furthermore, we define the following notations: V : vertex set, E : edge set, V^{dst} : corresponding destination vertex set of the source vertex, src_i : state of the source vertex i , dst_j : state of the destination vertex j (for the E-Loop, src_i and dst_i are the states of the two vertices associated with edge i , where it is arbitrary which is considered the source vertex and which is considered the destination vertex), p : transmission probability. The pseudo code of the C-Loop, T-Loop, and the E-Loop can be found in Figure 4.4.

Algorithm 1 C-Loop

```

for  $i = 1 \rightarrow |V|$  do
  if  $src_i == Active$  then
    for  $j = 1 \rightarrow |V^{dst}|$  do
      if  $dst_j == Inactive$  then
         $src_i \rightarrow dst_j$  with  $p$ 
      end if
    end for
  end if
end for
    
```

Algorithm 2 T-Loop

```

for  $i = 1 \rightarrow |V|$  do
  if  $src_i == Inactive$  then
    for  $j = 1 \rightarrow |V^{dst}|$  do
      if  $dst_j == Active$  then
         $dst_j \rightarrow src_i$  with  $p$ 
      end if
    end for
  end if
end for
    
```

Algorithm 3 E-Loop

```

for  $i = 1 \rightarrow |E|$  do
  if  $src_i == Active$  then
    if  $dst_i == Inactive$  then
       $src_i \rightarrow dst_i$  with  $p$ 
    end if
  else
    if  $dst_i == Active$  then
       $dst_i \rightarrow src_i$  with  $p$ 
    end if
  end if
end for
    
```

Figure 4.4. Simulation Algorithms

4.5.2 Data Structures for Simulations of Networks

In this subsection, we introduce the data structures for the network storage in hardware. As illustrated in the last sub-section, we have three different simulation algorithms based on two types of network processing approaches. It is therefore necessary to design different data structures for the corresponding simulation algorithms, as shown in Figure 4.5.

Figure 4.5. Two Types of Data Structures for Network Simulation

4.5.2.1 Vertex-Oriented Structure

The C-Loop and the T-Loop start simulation from active nodes and inactive nodes respectively. Each node in the network can interact with its connected nodes (propagate or receive information according to whether it is active or not). It is necessary to ensure each node has a contact list for its connected neighbors, and it also appears in its neighbors' contact list. That is, starting from each node we can trace its connected nodes via corresponding edges. Thus, we need to build a proper data structure based on the bidirectional graph for network storage for efficient processing.

Figure 4.5 (a) illustrates the vertex-oriented structure. Each vertex is considered as a source vertex and aligned in a vertex array with a *Source_Verex_ID*. Each vertex maintains an edge list which stores the *Destination_Verex_ID* on a corresponding *Edge_ID*. These edges are grouped together into an edge array according to the *Source_Verex_ID*. Note that the edge directions are used to represent the contact information. We use the state of the vertex attribute (*Active or Inactive*) to determine the direction of information flow between two connected nodes.

In practice, suppose node *A* is active and node *B* is inactive at step *t*. In the C-Loop, we start simulation from node *A*. According to edges of node *A* (*E*[0], *E*[1], and *E*[2]), we can trace the state results of its connected nodes (*B*, *C*, and *D*). As node *B* is inactive, node *A* can make an activation attempt to node *B*. In the T-Loop, we start simulation from node *B* and trace its connected nodes (*A*, *C*, and *D*) via edges (*E*[3], *E*[4], and *E*[5]). As node *A* is active, node *B* receive activation attempt by node *A*.

4.5.2.2 Edge-Oriented Structure

In the E-Loop, simulation starts from the edge and checks the connected pair of nodes' states. Compared to the vertex-oriented structure, where there are duplicated edge elements in the edge array which represent the same contact relation (for example,

both $E[0]$ and $E[3]$ indicate the same contact relation in Figure 4.5 (a)), such duplication can be removed according to the E-Loop's algorithmic behavior. That is, the network can be considered as an undirected graph, where we only need one copy of each edge instead of two to represent the network structure. It is therefore possible to reduce the spatial complexity in the network storage. In practice, the memory space of the edge-oriented structure is approximately one half of the vertex-oriented structure.

The edge-oriented structure is shown in Figure 4.5 (b). According to an *Edge_ID*, we can trace the corresponding *Source_Verex_ID* and *Destination_Verex_ID*. Depending on the state results between the *Source_Verex_ID* and the *Destination_Verex_ID*, the information can be propagated with a specific probability.

In practice, suppose node A is active and node B is inactive at step t . In the E-Loop, we start simulation from each edge. According to the first edge element in the edge array, we can trace node A and node B . Thus, node A can make an activation attempt on node B with a specific probability.

4.5.3 Optimized Simulation Strategies

In the previous sub-section, we introduced two different simulation algorithms using vertex-oriented processing and another algorithm using edge-oriented processing. The varying computational complexity of the three algorithms provides an opportunity to choose the best simulation algorithm using performance modeling. In this subsection, we discuss the optimized simulation strategies for information propagation.

4.5.3.1 Vertex-Oriented Adaptive Processing

According to the vertex-oriented approach, we can calculate the computational complexity for the C-Loop and T-Loop at each simulation step. First, we introduce the following notations: V_A^{dst} : corresponding destination vertex set of the active vertices at step t ,

V_I^{dst} : corresponding destination vertex set of the inactive vertices at step t , E_A^{dst} : corresponding out-going edge set of the active vertices at step t , E_I^{dst} : corresponding out-going edge set of the inactive vertices at step t . Thus, we can present the algorithmic complexity of the C-Loop at each step. As shown in Figure 4.5 (a), the C-Loop needs to browse the vertex array to obtain the state results of each vertex. This complexity can be expressed as $O(|V|)$. If the vertices' states are active, the C-Loop traces the corresponding connected vertices via edges. The complexity of this behavior can be expressed as $O(|E_A^{dst}|)$. Finally, the C-Loop needs to check the destination vertices' states and make activation attempts with a specific probability. It can be expressed as $O(|V_A^{dst}|)$. Thus, the total algorithmic complexity of the C-Loop can be expressed as

$$O(|V|) + O(|E_A^{dst}|) + O(|V_A^{dst}|) \quad (4.1)$$

Compared to the C-Loop, the T-Loop browses the vertex array and obtains those inactive vertices. After that, the T-Loop traces the corresponding edges in order to check the connected vertices' states. Thus, the algorithmic complexity of the T-Loop can be expressed similarly as

$$O(|V|) + O(|E_I^{dst}|) + O(|V_I^{dst}|) \quad (4.2)$$

According to the model equivalence between the C-Loop and the T-Loop, we can carry out the algorithmic adaptation at the proper time to obtain the best simulation performance gain. Since only one or a few active nodes are introduced into the network in the beginning, we can choose the C-Loop first and swap to the T-Loop at the simulation step where the workload of the T-Loop is less than that of the C-Loop. According to Formula (4.1) and Formula (4.2), we can have the following algorithmic adaptation condition:

$$|E_A^{dst}| + |V_A^{dst}| = |E_I^{dst}| + |V_I^{dst}| \quad (4.3)$$

Since V^{dst} denotes the destination vertex set of the corresponding edge E^{dst} , $|V^{dst}| =$

$|E^{dst}|$. Thus, we can further conclude that the simulation performance at each simulation step depends on $|E_A^{dst}|$ in the C-Loop and $|E_I^{dst}|$ in the T-Loop. In practice, we can calculate $|E_A^{dst}|$ and $|E_I^{dst}|$ at each simulation step. We can then make a decision whether to adapt the simulation algorithm in order to achieve performance gain. The simulation algorithm of the vertex-oriented adaptive A-Loop is shown in Figure 4.6.

Initialization:

Set *C-Loop* as the Initial Configuration
Set *AdaptFlag* = *false*

Iteration:

```

1: if AdaptFlag == false then
2:   Process Network with C-Loop
3:   for  $i = 1 \rightarrow |V|$  do
4:     if  $src_i == Active$  then
5:       for  $j = 1 \rightarrow |V^{dst}|$  do
6:         if  $dst_j == Inactive$  then
7:            $src_i \rightarrow dst_j$  with  $p$ 
8:         end if
9:       end for
10:    end if
11:  end for
12:  if the Adaptation Condition is Satisfied then
13:    Set AdaptFlag = true
14:  end if
15: else
16:   Process Network with T-Loop
17:   for  $i = 1 \rightarrow |V|$  do
18:     if  $src_i == Inactive$  then
19:       for  $j = 1 \rightarrow |V^{dst}|$  do
20:         if  $dst_j == Active$  then
21:            $dst_j \rightarrow src_i$  with  $p$ 
22:         end if
23:       end for
24:     end if
25:   end for
26: end if

```

Figure 4.6. Simulation Algorithm of Adaptive A-Loop

4.5.3.2 Edge-Oriented Processing

For the edge-oriented approach, we introduce additional notations: $\overline{V^{src}}$: vertex set of all the source vertices associated with the edge array, $\overline{V^{dst}}$: vertex set of the destination vertices corresponding to set $\overline{V^{src}}$. As shown in Figure 4.5 (b), the E-Loop starts the simulation from the edge array. This complexity can be expressed as $O(|E|)$. According to the edge, the E-Loop checks the states results of the associated pair of nodes with complexity, expressed as $O(|\overline{V^{src}}|) + O(|\overline{V^{dst}}|)$. Thus, the algorithmic complexity of the E-Loop at each simulation step can be expressed by

$$O(|E|) + O(|\overline{V^{src}}|) + O(|\overline{V^{dst}}|) \quad (4.4)$$

Note that, the size of E is equal to that of $\overline{V^{src}}$ and $\overline{V^{dst}}$ ($|E| = |\overline{V^{src}}| = |\overline{V^{dst}}|$). In contrast to the varying computational complexity at each simulation step with the C-Loop and T-Loop, the E-Loop is an algorithm with constant computational complexity in network browsing at each simulation step.

4.6 Summary

In this chapter, we first describe two basic models in modeling information propagation, the Independent Cascade Model and the Linear Threshold Model. These two models represent the cascade behavior and the collective behavior, respectively. With suitable model modifications, we are able to extend the basic models to be more general. Furthermore, we can show the model equivalence between these two general models. Since the two models are equivalent, it is possible to combine them together in order to obtain performance optimization during simulation.

Next, we relate the general models to the motivational example in infectious disease

simulation which has been introduced in chapter 3 and analyzed in detail. We show that the I-Loop and the S-Loop in the infectious disease simulation are special cases of the cascade model and the threshold model, respectively. In addition, the different simulation performance between the I-Loop and the S-Loop provides an opportunity for performance optimization using the detailed performance modeling and analysis of the infectious disease simulation.

Finally, we show the optimized strategies in modeling and simulation for general information propagation over complex networks. According to the two approaches, Vertex-oriented and Edge-oriented, we propose two different types of data structures to store the network layout: the Vertex-Oriented Structure and the Edge-Oriented Structure. The vertex-oriented structure can obtain benefits from optimized performance by enabling runtime model adaptation. Compared to the vertex-oriented structure, the edge-oriented structure achieves optimized memory space in hardware.

Simulation of Information Propagation on GPGPU and Multi-core CPU

In the previous chapter, we introduced three different simulation algorithms based on two types of processing approaches: the vertex-oriented and edge-oriented approaches. We further proposed an optimized simulation strategy with algorithmic adaptation at runtime. In this chapter, we describe the detailed system designs on GPGPU and Multi-core CPU.

5.1 Simulation of Information Propagation with Medusa

5.1.1 System Design with Medusa

We develop an HPC simulator in cooperation with Medusa [143] for the information propagation simulation. Medusa is a GPU based framework for graph processing which leverages the GPU power in common graph computation tasks. Medusa provides APIs for developers to implement their own applications. It executes the user-defined APIs in parallel on all the processing cores within the GPU and hides the programming complexity from users.

In particular, Medusa provides three kinds of APIs for developers to implement GPU graph processing algorithms. They are collectively named Edge-Message-Vertex (EMV) operators and are used for edge processing (EDGE), message processing (MESSAGE), and vertex processing (VERTEX). Using these APIs, users can define their operations on vertices, edges and messages, respectively. Medusa can then automatically transform these implemented APIs into CUDA kernels. Furthermore, the storage of the graph structure is also handled by Medusa. After the definition and initialization of the graph structure in the main memory on the host, Medusa transforms the user defined graph structure into suitable data structures to meet GPU memory characteristics. At runtime, each GPU thread executes one instance of the user-defined APIs. The thread block and grid configuration is tuned by Medusa in order to maximize the GPU utilization.

To implement an application based on Medusa, developers first need to define the basic data structures such as edge, message and vertex in a C/C++ struct. Secondly, according to the specific application logics, the EMV APIs need to be implemented. Finally, the developers need to compose these APIs and data structures to a complete executable program, including initializing the graph structure, configuring the framework parameters and invoking the customized EMV APIs. As Medusa can ease CUDA threads management and also because of the efficient programmability of Medusa, we choose Medusa to develop our own information propagation simulator on GPU.

In practice, we adopt the EDGE and VERTEX APIs in our information propagation simulation. As we need to check specific nodes and trace their neighbors' state results at each simulation step, the VERTEX API (to check a vertex v 's state) and the EDGE API (to check the destination vertex of v via edge e) need to be implemented. Moreover, as we use independent probabilities in the information propagation model, we can avoid implementing the MESSAGE API in our simulation approach. After the initialization of the graph structure in the main memory, we invoke the Medusa function API *Init_Device_DS()* to build the corresponding graph structure in GPU global memory. The Medusa APIs used in this thesis are listed in Table 5.1.

Table 5.1. System Provided APIs and Parameters in Medusa

<i>API/Parameter</i>	<i>Description</i>
<i>AddEdge (void* e), AddVertex (void* v)</i>	Add an edge or a vertex into the graph
<i>maxIteration</i>	The maximum number of iterations that Medusa executes (iterations = simulation steps)
<i>halt</i>	A flag indicating whether Medusa stops the iteration
<i>Medusa::Run (Func f)</i>	Execute <i>f</i> iteratively according to the iteration control
<i>EMV<type>::Run (Func f')</i>	Execute EMV API <i>f'</i> on GPU
<i>Init_Device_DS()</i>	Initialize the data structure on GPU

As introduced in chapter 2, the coalesced memory access is usually the most important programming issue to consider in obtaining high performance on a GPU. If a memory request issued by a warp can be coalesced into one memory transaction (when memory addresses fall into the same cache line), the bandwidth utilization of the GPU can be significantly improved. To exploit this architecture feature of the GPU, the data structures in GPU memory and the corresponding processing mechanism need to be carefully designed. Note that as the number of edges is usually much larger than the number of vertices in a graph, the graph processing workload mainly depends on edge processing. As shown in Figure 4.5 (a), in the vertex-oriented approach, each access to an edge is from the corresponding vertex. When we execute the vertex-oriented approach in parallel, it is unlikely that edge addresses from consecutive vertices are in the same cache line. This leads to uncoalesced memory access, especially for those graphs where vertices have a large out-degree.

In contrast, the edge-oriented approach exploits the coalesced memory access feature well. When we process each edge element in parallel, the edge accesses are coalesced.

Furthermore, as shown in Figure 4.5 (b), the memory accesses to the vertex array from edges are usually associated with the same vertex or consecutive vertices, which also leads to coalesced memory access. Therefore, we can conclude that the edge-oriented approach can take full advantage of the GPU architecture by coalesced memory access. However, this must be offset against the optimized computational complexity that can be achieved by runtime algorithmic adaptation using the vertex-oriented approach.

In order to compare the execution performance between the GPGPU and Multi-core CPU, we also implement a CPU-based simulation program using the same functionalities of the EMV APIs and the same data structures as the Medusa implementation. Figure 5.1 illustrates the overview of the simulation design using either CPU or GPU. Note that a random number generator is essential to cooperate with the network browsing algorithms for testing if the inactive nodes successfully receive the information from their active neighbors. In order to compare the simulation performance on CPU and GPU with the same baseline, we should ensure that the simulation on CPU and GPU generates the same network state results at each step. After initializing the graph structure on CPU and GPU respectively, we execute the simulation on the GPU first and then on the CPU. Curand [38] is a high performance random number generation library on the GPU, and it is also available for CPU based random number generation. Therefore we adopt Curand to generate random number sequences on the CPU and GPU. As Curand is designed to meet the feature of parallel processing on the GPU, generating random numbers in a batch is much more efficient than generating them one by one.

At each simulation step, a new random number sequence (by a new random seed) is generated in parallel and the numbers are attached to each edge in order to compare to the pre-defined activation probability. At the same time, the Monitoring Module collects state results from either the CPU or GPU. If the on-going simulation is using vertex-oriented processing, at each step the Steering Module selects the optimal simulation algorithm according to the network state results analysis from the Runtime Scheduler. The same operations are conducted whether the simulation is executed on the CPU or GPU. Thus,

by implementing the same network browsing algorithm on the CPU and GPU, we can have identical state results at each step from both architectures. Since we are focusing on network browsing algorithms in this chapter, we do not consider the cost of random number generation. For a proof of concept, we conduct an initial set of experiments on two samples with 1 million vertices and 8 million edges (1M8M). Here, “M” denotes million.

Figure 5.1. Overview of Simulation

5.1.2 Experiments and Results

Using the system design introduced in the previous sub-section, we show preliminary experimental results on a random network and a scale-free network. As shown in Figure 5.2, different from the random networks, in the scale-free networks with power-law degree distribution, a few nodes are attached to most of the edges and these hubs have an important influence on the whole network. Furthermore, we note that the scale-free networks may not be fully connected, and it is therefore necessary to calculate the number of isolated nodes in the network before simulation.

Figure 5.2. Network Structures (adopted from [119])

In the beginning of the simulation, a single active node (the first node, $v[0]$) is introduced to the network. After that, according to the selected simulation algorithm, the active node can propagate the information to those inactive nodes in that network with a specific probability. The detailed simulation parameters can be found in Table 5.2.

Table 5.2. Simulation Parameters

Simulation Algorithm	C-Loop, T-Loop, E-Loop
Probability to be Activated	0.01
Network Type	Random, Scale-free
Hardware Architecture	Nvidia Tesla C2050 (1x)
	AMD Quad-core Opteron 2378 (1x)
Processing Units	14 x 32 (448) streaming processors on GPU
	4 CPU cores (1 core used in this sub-section)
Memory	3 GB on GPU
	8 GB on CPU
Programming Models	CUDA and Medusa on GPU
	OpenMP on CPU (serial processing in this sub-section)

Figure 5.3 illustrates the information propagation simulation on a 1M8M random network.

Figure 5.3. Random Network with 1M8M

In particular, Figure 5.3 (a) and Figure 5.3 (b) show the execution time per simulation step on Fermi GPU and AMD CPU respectively. Compared to the CPU serial simulation performance, the Fermi GPU based simulation shows 12.49x, 13.10x, 15.60x parallel speedup with the C-Loop, T-Loop, and the E-Loop respectively. Figure 5.3 (c) illustrates the state results of the network during simulation (which are the same on the CPU and the GPU). After 500 steps, the number of active nodes changes from 1 to 999459, 999457, and 999494 with the C-Loop, T-Loop, and the E-Loop, respectively. In Figure 5.3 (b), there is a peak in the E-Loop. This is because many nodes are changing their states during this period, involving extra computation. Compared with the CPU execution time, the GPU shows its powerful parallel execution performance can offset the extra computation in state changing.

Furthermore, as shown in Figure 5.3 (c), the state results using the three different algorithms are very similar at each simulation step. It indicates that our three algorithms are equivalent in practice. In particular, according to Figure 5.3 (c), the inactive nodes are mainly changed to active state in the period of steps [150, 250]. It is the extra computing cost of state updating that causes the obvious curve protuberance (steps [150, 250]) in the CPU serial simulation performance, compared to that of the GPU. That is, the massively parallel processing cores on GPU share the computing cost of state updating.

Figure 5.4 shows the simulation on a scale-free network with 1 million nodes and 8 million edges. However, there are 480551 isolated nodes without any connection to others. In addition, according to the feature of the power-law degree distribution, a few nodes have very large contact lists compared to others. That is, the information will spread widely from these hubs if they receive the information. Figure 5.4 (c) shows that the simulation converges very quickly among the connected nodes, and the number of active nodes finally reaches 519290, 519290, 519287 after 500 steps by the C-Loop, T-Loop, and the E-Loop respectively. Figure 5.4 (a) and Figure 5.4 (b) illustrate that the Fermi GPU based simulation performance shows 6.79x, 12.01x, and 14.43x parallel speedup compared to the CPU serial simulation performance with the C-Loop, T-Loop, and the

E-Loop respectively. Due to the extra computing cost of state updating during steps [1, 50], the curve protuberance of the C, T, E Loops appears in the CPU serial simulation performance during these steps.

Figure 5.4. Power-Law Network with 1M8M

5.2 Optimized Algorithmic Adaptation Strategies for Different Networks

In the previous section, we have shown the simulation performance with three different algorithms on the random network with uniform degree distribution and also on the scale-free network with power-law degree distribution. In this section, we describe the optimized algorithmic adaptation strategies for the vertex-oriented processing on different network structures. According to the experimental results shown in Figure 5.3 and Figure 5.4, the optimized algorithmic adaptation chooses the C-Loop first and then swaps to the T-Loop at a proper time step. In chapter 4, we have also formulated the computational complexity for each algorithm. That is, the optimal algorithm swapping should be conducted at the step where the C-Loop's workload equals that of the T-Loop's. Such an optimized simulation algorithm with runtime adaptation is named as the A-Loop.

5.2.1 Random Networks

As shown in Figure 5.2 (a), edges in the random networks are randomly assigned to nodes with uniform degree distribution. It means that the contact degree of each node can be considered as the same. It is therefore not difficult to derive an optimization strategy which satisfies Equation (4.3). That is, the algorithmic adaptation can be conducted when the number of active nodes is more than the number of inactive nodes in the network, expressed as $|V_{Act}| > |V| - |V_{Act}|$. Figure 5.5 (a) and Figure 5.5 (c) illustrate that the A-Loop nearly executes the most efficient algorithm at each step of the simulation based on the model prediction, both on the GPU and CPU. However, as shown in Figure 5.5 (b) and Figure 5.5 (d), the adaptation may not be completely accurate due to the limits of model accuracy. Furthermore, we observe that the vertex-oriented processing with the A-Loop obtains better simulation performance than edge-oriented processing with the E-Loop. The experimental results also demonstrate that the A-Loop achieves 8.18x, 5.92x, and 4.98x speedup compared to that of the C-Loop, T-Loop, and the E-Loop on Fermi GPGPU. In addition, the A-Loop also achieves 5.74x, 4.36x, and 4.37x compared to that

of the C, T, and E Loops in the CPU serial execution environment.

Figure 5.5. Algorithmic Adaptation

5.2.2 Scale-free Networks

With scale-free networks, the number of links connecting to a node follows the power-law degree distribution. In practice, such networks can be generated by the recursive matrix (R-MAT) model [27]. For the scale-free networks, we could also use the same condition for algorithmic adaptation as that on random networks. However, we discuss a more accurate algorithmic adaptation strategy in this section. As shown in Figure 5.6 (a), with the R-MAT model, the adjacency network matrix A ($N \times N$) denotes the connection relations between the N nodes. The element $a(i, j) = 1$ if the edge $e(i, j)$ exists, otherwise $a(i, j) = 0$. The matrix is recursively divided into four equal sized sub-matrices. Moreover, the edge assignment on the sub-matrices is under biased probabilities: starting from an empty adjacency matrix, R-MAT assigns edges into the matrix one by one. Each edge chooses a matrix cell with probabilities a, b, c, d (here $a + b + c + d = 1$) respectively. The chosen partition is sub-divided into four smaller partitions again. Such a procedure is repeated until the edge is assigned to a single matrix cell in that adjacency matrix. The R-MAT model has three important properties that exist in real world networks [27].

- The partitions a and d denote two separate groups (communities) in the network.
- The partitions b and c denote the cross-links between group a and group d .
- The recursive procedure of edge assignment indicates the links between sub-groups within the existing group.

Typically, the cross-links between two groups are less than the internal links within a group. Therefore, we have $a \geq b$, $a \geq c$, $d \geq b$, $d \geq c$. Furthermore, in order to match the power-law degree distribution, we have $a \geq d$ [27]. It indicates that vertices in group a have more connections than the vertices in group b . As the undirected graph should have a symmetric adjacency matrix, R-MAT can achieve it by using $b = c$ [27]. Thus, our bidirectional network modification can still follow the power-law distribution if $b = c$. In practice, we use the values of 0.45, 0.15, 0.15, 0.25 for the parameters a, b, c, d to

generate networks. That is the default setting in GTgraph and it is approximate in many real world graphs [10].

Figure 5.6. R-MAT Graph Generation Model

In the adjacency matrix, as shown in Figure 5.6 (b), the *Left* $\rightarrow$ *Right* arrows show the propagation of information from a given node to other nodes. The *Top* $\rightarrow$ *Down* arrows show the receiving of information by a given node from other nodes. Thus, the *Left* $\rightarrow$ *Right* arrows show the C-Loop, and the *Top* $\rightarrow$ *Down* arrows show the T-Loop.

In practice, we choose the C-Loop first to conduct information propagation simulation starting from the first node $v[0]$. According to the given edge distribution in the network matrix, the information will be propagated to partition a with a very high probability. This is because large number of edges exist in partition a when the network is generated. We can assume that the left-top area (denoted as the $S1$ zone) has a large number of edges and it leads to a very high probability that the node in the $S1$ zone will first receive the information from node $v[0]$. We can further assume that there exists a simulation step t at which there are k percent of nodes with active state in the network.

As discussed in chapter 4, the performance at each simulation step depends on the number of edges to visit. Thus, the execution cost of the C-Loop can be simply

represented as the area of *Left* $\rightarrow$ *Right* arrows at the simulation step t , expressed as $O(S1 + S2)$. The execution cost of the T-Loop at step t can then be represented as $O(S2 + S4)$. The problem of finding the algorithmic adaptation point can be considered as one of finding the simulation step t at which there are k percent of nodes active and associated with half of the total edges. In addition, as the scale-free network contains isolated nodes (denoted as V^{Iso}), we can ignore these. We can therefore carry out the algorithmic adaptation on the scale-free networks when $|V_{Act}| > k \cdot (|V| - |V^{Iso}|)$. In particular, with the default parameters in the edge distribution matrix ($a = 0.45$, $b = c = 0.15$, $d = 0.25$), we can obtain the value of k by calculating the edge assignment probability of those sub-partitions. As shown in Figure 5.7, we can visit approximately 50% of the edges within the network with $k = 0.375$. Thus, we can have an approximation that there are $k \approx 0.37$ percent of nodes active and associated with half of the total edges.

Figure 5.7. Calculation of value of k

Figure 5.5 (e) and Figure 5.5 (g) indicate that the optimized algorithmic adaptation strategy works very well on the scale-free network, both on the GPGPU and CPU. However, some inaccuracy of runtime algorithmic adaptation still exists, as shown in Figure 5.5 (f) and Figure 5.5 (h). The experimental results also demonstrate that the optimized vertex-oriented processing with the A-Loop is better than the edge-oriented processing with the E-Loop with regard to execution time. Furthermore, according to the experimental results, on Fermi GPGPU with the A-Loop, we can achieve 24.78x, 2.12x,

6.78x speedup compared to that of the C-Loop, T-Loop, and E-Loop. In the CPU serial simulation, the A-Loop obtains 10.59x, 1.60x, 6.16x speedup compared to that of the C-Loop, T-Loop, and E-Loop.

5.3 Experiments and Results with Scalable Networks

In the last section, we have shown that the proposed optimized algorithmic adaptation strategies perform well in practice. In this section, we demonstrate more experimental results on scalable networks in order to further evaluate our proposed method. As shown in Figure 5.8, we conduct a set of simulations on random networks and scale-free networks with size of 1M4M, 1M16M, 1M64M, and 1M256M. Due to the GPU memory limits (maximum 3GBs in Fermi C2050 GPGPU), the vertex-oriented approach is only feasible on 1M4M, 1M16M, 1M64M networks but the edge-oriented approach is feasible on all the test cases. According to the performance results, it is obvious that we can achieve the best simulation performance with the optimized algorithmic adaptation strategy both on random and scale-free networks. Compared to the CPU serial processing environment, we can further obtain significant performance acceleration on the Fermi GPGPU.

The simulations over networks with large contact degree converge faster than those networks with relatively small contact degree. We also note that the larger the contact degree of the network, the better the performance of the T-Loop. According to the algorithmic analysis, the T-Loop has excellent performance when most of the nodes have changed to active state. We therefore observe that the T-Loop achieves better simulation performance with the increased contact degree, both on random and scale-free networks. Furthermore, the T-Loop shows better performance on scale-free networks than that on random networks since the information spreads much faster on scale-free networks. As a summary, we should choose the optimized A-Loop for performance improvement if the vertex-oriented structure is feasible in practice given the memory constraints. Otherwise, we should choose the simulation with the E-Loop for better spatial scalability.

Random Networks

Scale-free Networks

Figure 5.8. Performance Summary of Different Algorithms

Given the excellent data parallelism, we can also use the OpenMP [107] library for multi-threaded parallel processing to accelerate the simulation performance on the multi-core CPU. At each simulation step, we parallelize the “*for*” loop in the network browsing algorithm (A-Loop or E-Loop). The OpenMP thread number is set to 4, which divides the workload equally into 4 cores on the AMD Quad-core CPU. Using the baseline of CPU serial processing performance, Figure 5.9 illustrates the parallel speedup of the E-Loop and A-Loop on Fermi GPGPU and Quad-core CPU (OMP = 4).

With the E-Loop, the Fermi GPGPU achieves an average 14.89x speedup as opposed to 3.61x speedup in the CPU multi-threaded parallel processing environment. With the A-Loop, the Fermi GPGPU achieves an average 15.08x speedup, and CPU multi-threading obtains an average 2.74x speedup (in Figure 5.9, the ‘R’ denotes Random networks, and the ‘S’ denotes Scale-free networks).

Figure 5.9. Parallel Speedup of E-Loop and A-Loop

In addition, we note that the parallel speedup of the E-Loop on the Fermi GPGPU keeps stable in the experiments. However, the A-Loop's performance on the Fermi GPGPU indicates that the parallel speedup decreases with an increased contact degree. This is due to the thread scheduling mechanism and memory access pattern on Nvidia GPUs we have discussed in section 5.1. As shown in Figure 4.5, we assign CUDA cores to the vertex array in the vertex-oriented approach. If the given network has a small contact degree, the uncoalesced accesses can be reduced when CUDA cores in a warp are concurrently tracing edge information. In this case, we can achieve excellent parallel speedup on Fermi GPU. Otherwise, the parallel performance on GPU decreases.

5.4 Summary

In this chapter, we present the performance of our three algorithms for simulation of information propagation. With an optimized algorithmic adaptation strategy (A-Loop) on the vertex-oriented structure, we can achieve the best performance in execution time. With the edge-oriented processing (E-Loop), we obtain better spatial scalability than that of vertex-oriented processing. Furthermore, we implement our proposed method on many-core GPGPU and multi-core CPU respectively. Compared to the CPU serial execution, we can obtain average 2.74x and 3.61x speedup in the A-Loop and E-Loop respectively in the Quad-core multi-threading environment. Furthermore, an average 15.08x and 14.89x parallel speedup in the A-Loop and E-Loop can be gained on the Fermi GPGPU compared to CPU serial performance.

Simulation of Information Propagation: Performance Studies on Multi-GPU

With the increased computing resources required for large-scale network simulation, one GPU device may not have enough memory and computation capacities. It is therefore necessary to enhance the system scalability by introducing multiple GPU devices. For example, one Nvidia Fermi C2050 device has only 3 GB memory space. By introducing multiple GPU devices, we can have more memory space to contain larger complex networks in memory. Improving the spatial scalability of simulation is the most important motivation of our Multi-GPU design. In addition, it is also attractive to investigate the execution performance scalability of Multi-GPU simulation. This chapter describes the simulation of information propagation over complex networks on multiple GPUs, including the network partitioning and replication strategy, and the data synchronization scheme. The design is based on the optimized network simulation algorithms described in Chapter 4.

6.1 System Design of Multi-GPU Simulation

In this section, we introduce the method to partition one logical network to multiple physical partitions on GPU devices. Furthermore, we describe the synchronization algorithm between different GPU devices that is executed at the end of each simulation step.

6.1.1 Network Partitioning

In order to perform efficient network partitioning, we apply Metis [92] to generate the partition file. The algorithms implemented in Metis are based on the multilevel recursive-bisection, multilevel k-way scheme [92]. Metis balances the number of vertices in each partition and also minimizes the number of edges cut in order to minimize the data replication and cost of synchronization. In Metis, every network structure is considered to be a directed graph, for network partitioning purposes only. A node in the undirected graph still needs a link and its head and tail elements to store node information. As the result of graph partitioning, each cross-partition edge is cut and the connected *head-vertex* and *tail-vertex* are placed into two different partitions. It is therefore necessary to store the cross-partition edge and associated nodes' information in the proper partition. We use the convention that in the diagram we define “A” as the head-vertex and “B” as the tail-vertex.

For both vertex-oriented processing and edge-oriented processing, we maintain replicas for the head vertex of each cross-partition edge in the partition where the tail vertex resides. Also, each cross-partition edge is stored in its tail partition. Thus, we can process our network simulation on different GPU devices independently.

6.1.1.1 Partitioning with a Vertex-Oriented Structure

With a vertex-oriented structure, the complex network is represented by a bidirectional graph. For each edge that is cut, the nodes connected by that edge are both head vertices and tail vertices. As shown in Figure 6.1 (where the shaded circles denote replicas), the partitioning and replication scheme can ensure the correct interaction between the active and the inactive nodes.

For example, suppose A is an active node in Partition 1 and node I is an inactive node in Partition 2. As the simulation executes, the C-Loop processes $A \rightarrow I$ in Partition 2 and

the T-Loop processes $I \rightarrow A$ in Partition 1.

Figure 6.1. Partitioning with Vertex-Oriented Structure

6.1.1.2 Partitioning with an Edge-Oriented Structure

As shown in Figure 6.2, we have only one copy of the edge between a pair of vertices in the edge-oriented structure. However, for the purpose of partitioning, we treat the graph as unidirectional, and for each edge that is cut, we store a replica of the head vertex (e.g. node I) in the tail vertex (e.g. node A) partition. During simulation execution, the direction of the edge is not important as the arrows in Figure 6.2 are for partitioning purposes only. With the edge-oriented structure, the number of replicas is less than that in the vertex-oriented structure. However, in some very large-scale complex networks, there are many cross-partition edges between vertices. It is possible that nearly every vertex is replicated in each of the other partitions. In this case, the number of replicas may be almost the same as that in the vertex-oriented structure.

In Multi-GPU simulation, each GPU device maintains a *Boundary Table* and a *Replica Table*. The *Boundary Table* stores the boundary vertices (nodes) in its partition which have replicas in other partitions that point to tail vertices. In contrast, the *Replica Table* is used to replicate the head vertices which reside in the other partition but point to vertices in its own partition.

Figure 6.2. Partitioning with Edge-Oriented Structure

As shown in Figure 6.3 (a), GPU 1 maintains a boundary table which consists of 3 segments for the boundary nodes with replicas that reside in GPUs 2, 3, and 4 respectively. The replica table in GPU 1 stores the replica nodes from GPUs 2, 3, and 4 respectively. Figure 6.3 (b) shows the boundary and replica tables of Partition 1 in the example given in Figure 6.1.

Figure 6.3. Boundary and Replica Tables

6.1.2 Distributed GPU Simulation and Synchronization

Since the boundary node and its replicas in other partitions represent the same logical node in the original network, it is necessary to synchronize the state of the boundary and replica nodes at the end of each simulation step.

Either the boundary or replica node may contain the updated state result. With vertex-oriented processing, as shown in Figure 6.1, the original node can be updated by the C-Loop, e.g. node I in partition 2. In contrast, the replica node can be updated by the T-Loop, e.g. node I in partition 1. With edge-oriented processing, as shown in Figure 6.2, either the boundary or replica node may be updated, e.g. replica node I in partition 1 can be updated by node A and original node I in partition 2 can be updated by node J . Thus the replica node may contain the latest state result which needs to be copied to the original node.

To perform the synchronization, we first need to copy the boundary nodes' state to the boundary table and copy the replica nodes' state to the replica table. We can then transfer the boundary and replica tables from the GPU devices to the *Boundary Buffer* and the *Replica Buffer* on the host CPU to process the synchronization. The boundary/replica buffer consists of boundary/replica tables from different GPU devices. The synchronization between the boundary table and replica table for 3 GPU devices is illustrated in Figure 6.4. The boundary buffer updates and records the correct nodes' state after the *Boundary-Replica* synchronization.

We note that there may be multiple copies of boundary nodes that reside in the boundary table. For example, in Figure 6.3 (b), the node C has two copies in the boundary table of Partition 1. It is therefore necessary to synchronize the multiple copies of the boundary node after the *Boundary-Replica* synchronization. As shown in Figure 6.5 (a), we construct a *Redundancy Table* for each GPU device which resides in the CPU. With the same size as the boundary table, the redundancy table maintains the redundancy lists for each boundary node. The redundancy list stores the boundary buffer index of multiple copies of a specific boundary node. In particular, the size of the redundancy list depends on the number of partitions. Given a network which is partitioned into P partitions, we can have *Redundancy List Size* $\in [0, P - 1]$.

Figure 6.4. Boundary / Replica Synchronization for 3 GPUs

As shown in Figure 6.3 (b), there are two copies of boundary node *C* which map to Partition 3 and Partition 4 respectively. Thus, each copy of *C* keeps the other copy's array index in the redundancy table, as shown in Figure 6.5 (b). According to the redundancy table of each partition, we are able to further update the boundary buffer in order to synchronize those boundary nodes with multiple copies.

Figure 6.5. Redundancy Table for Partition 1

After the synchronization process, we can obtain the updated node state in the boundary buffer on the CPU memory. We can transfer the synchronized data from the boundary buffer to each GPU's boundary table and replica table. Each GPU device can further update the boundary and replica nodes to complete the entire synchronization procedure. The synchronization is illustrated in Figure 6.6. The detailed synchronization algorithm on CPU is shown in Figure 6.7.

Algorithm 3 Synchronization Algorithm

```

1: // Data Copy on Devices (GPUs)
2: for  $i = 1 \rightarrow NumGPUs$  do
3:   Copy boundary nodes' status and replica nodes' status to
   Boundary and Replica Tables respectively
4: end for
5: // Data transfer from Devices to Host (GPUs  $\rightarrow$  CPU)
6: for  $i = 1 \rightarrow NumGPUs$  do
7:   Transfer GPU's Boundary and Replica Tables to CPU
8: end for
9: // Synchronization on Host (CPU) with OpenMP
10: for  $i = 1 \rightarrow NumGPUs$  do
11:   for  $j = 1 \rightarrow NumSegments$  do
12:     Synchronization between Boundary and Replica nodes in
     different partitions
13:     Update the Boundary Buffer
14:   end for
15: end for
16: for  $i = 1 \rightarrow NumGPUs$  do
17:   for  $k = 1 \rightarrow RedundancyListSize$  do
18:     Check Redundancy Table and Synchronize multiple bound-
     ary nodes
19:   end for
20: end for
21: // Data transfer from Host to Devices (CPU  $\rightarrow$  GPUs)
22: for  $i = 1 \rightarrow NumGPUs$  do
23:   Transfer corresponding Boundary Buffer segments to other
   GPUs's Replica Tables
24:   Transfer corresponding Boundary Buffer segments to its GPU
   Boundary Table
25: end for
26: // Data Copy on Devices (GPUs)
27: for  $i = 1 \rightarrow NumGPUs$  do
28:   Copy Boundary and Replica Tables to boundary nodes' status
   and replica nodes' status respectively
29: end for

```

Figure 6.6. Synchronization Algorithm

Algorithm 4 Detailed Synchronization on CPU

```

1: // 1 for Active; 0 for Inactive //
2: // Boundary/Replica Synchronization //
3: for  $i = 1 \rightarrow NumGPUs$  do
4:   for  $j = 1 \rightarrow NumSegments$  do
5:     for  $m = 1 \rightarrow sizeof(NumSegment[j])$  do
6:        $Boundary[i][j][m] = Boundary[i][j][m] || Replica[i][j][m]$ 
7:     end for
8:   end for
9: end for
10: // Redundancy Synchronization //
11: for  $i = 1 \rightarrow NumGPUs$  do
12:   for  $l = 1 \rightarrow sizeof(Boundary[i])$  do
13:     for  $k = 1 \rightarrow RedundancyListSize$  do
14:       if  $Boundary[i][l][k] == 1$  then
15:         for  $r = 1 \rightarrow RedundancyListSize$  do
16:            $Boundary[i][l][r] = 1$ 
17:         end for
18:       end if
19:     end for
20:   end for
21: end for

```

Figure 6.7. Detailed Synchronization on CPU

6.2 Performance Evaluation with Random Networks

In this section, we evaluate the Multi-GPU simulation performance. The transmission probability for node activating is set to be 0.01 (as the same as in Chapter 5). The experiments are conducted on a server with two Intel Xeon E5645 CPUs and four Fermi C2050 GPU devices. First, we show a case study on a random network with 1 million nodes and 256 million edges (1m256m). This is followed by a performance investigation on scalable networks.

6.2.1 Case Study with a 1m256m Random Network

The propagation behaviors on a 1m256m random network with both vertex-oriented processing and edge-oriented processing are shown in Figure 6.8. As each GPU now performs its own random number generation, the state results may not be exactly the same for different numbers of GPU devices. However, the state results are very similar at each simulation step. We note that all simulations converge within 20 steps. Compared to the edge-oriented structure, the vertex-oriented structure of the 1m256m network is too large to be executed on one GPU device. The simulation performance with vertex-oriented processing and edge-oriented processing is shown in Figure 6.9 and Figure 6.10 respectively. The execution times shown are for network browsing only and do not include the random number generation. In these figures, the overall execution time is broken down into separate times for computation, synchronization on CPU, and data transfer between CPU and GPUs.

Figure 6.8. State Results during Simulation

Figure 6.9. Vertex Processing on 1m256m Random Network

Figure 6.10. Edge Processing on 1m256m Random Network

As shown in Figure 6.9 (a) and (b), and Figure 6.10 (a) and (b), the computation performance still dominates the overall performance both in vertex-oriented processing and edge-oriented processing. We also note that the multiple GPUs can share the original computation workload.

However, by increasing the number of GPU devices, the cost of data synchronization and the cost of data transfer between CPU and GPUs grow significantly. Figure 6.9 (c) and Figure 6.10 (c) indicate that the cost of synchronization increases with an increasing number of newly active nodes in the network and decreases as the propagation on the network eventually converges. This is comparable with the network propagation behavior shown in Figure 6.7. That is, most of the nodes are activated at step 11 and step 12, and the large number of newly active nodes may result in a large number of state differences between boundary and replica nodes. Thus, the maximum synchronization cost appears

between step 11 and step 12. Furthermore, according to the experimental results, we note that the more GPU devices we use, the higher the synchronization cost.

In Figure 6.9 (d) and Figure 6.10 (d), we show the data transfer performance between CPU and GPUs. Since the boundary and the replica tables are pre-built before simulation, the data transfer performance is quite stable at each simulation step. Interestingly, for this 1m256m random network, we note that the synchronization performance and the data transfer performance with vertex-oriented processing are comparable with edge-oriented processing. It indicates that in the networks where the number of edges per node is very large, the number of boundary and replica nodes in the edge-oriented structure and the vertex-oriented structure are very similar since nearly every node is replicated in each of the other partitions.

As shown in Figure 6.11, according to the simulation performance at each step, we can calculate the total execution time for the entire simulation. With vertex-oriented processing, the simulation takes 397.9ms on 2 GPUs, 418.8ms on 3 GPUs, and 488.4ms on 4 GPUs. In contrast, with edge-oriented processing, the simulation takes 2234.9ms on 1 GPU, 1120.0ms on 2 GPUs, 846.9ms on 3 GPUs, and 816.4ms on 4 GPUs. The experimental results indicate that vertex-oriented processing can obtain the best performance gain on 2 GPUs. We also observe that edge-oriented processing shows excellent speedup on multiple GPUs compared to that of vertex-oriented processing.

Figure 6.11. Total Execution Time of 1m256m Random Network

6.2.2 Performance Studies with Scalable Networks

In this subsection, we investigate the Multi-GPU simulation performance on scalable networks. In order to show the performance scalability, we conduct two sets of experiments. They are: (1) *Node-Scaling* experiments: we fix the number of edges but scale up the number of nodes, and (2) *Edge-Scaling* experiments: we fix the number of nodes but scale up the number of edges (“m” denotes “million”).

- In *Node-Scaling* experiments, we have 1m128m, 2m128m, 4m128m, 8m128m random networks.
- In *Edge-Scaling* experiments, we have 4m32m, 4m64m, 4m128m, 4m256m random networks.

We show the total time of edge-oriented processing in Figure 6.12 (a) and (b), and the total time of vertex-oriented processing in Figure 6.12 (c) and (d), excluding time for random number generation. Since the number of simulation steps may differ for different networks, this affects the total time (sum of the execution times for each simulation step). We therefore show the number of simulation steps on each network in the figure.

In the *Node-Scaling* experiments, we observe that the number of simulation steps becomes larger as the number of nodes increases, from 35 steps (1m128m network) to 200 steps (8m128m network). Note that on networks with a smaller number of nodes, we can obtain a performance gain by multiple GPU devices for edge-oriented processing. In particular, we can achieve better performance with 2 GPUs for networks with 1m, 2m, and 4m nodes than that with a single GPU device. Compared to the constant network processing complexity in edge-oriented processing, the vertex-oriented processing can achieve optimized computational complexity by using the adaptive A-Loop algorithm. Thus, the cost of synchronization and data transfer may dominate the overall performance with multiple GPU devices. Furthermore, with an increased number of GPU devices, the cost of synchronization and data transfer increase significantly and may become performance bottlenecks and degrade the Multi-GPU simulation performance.

Figure 6.12. Total Execution Time

In the *Edge-Scaling* experiments, we fix the number of nodes but increase the number of edges. We note that with the very large 4m256m network, a single GPU device is not

enough to contain the network structure. We also note that the number of simulation steps decreases as the number of nodes increases, from 1000 steps (4m32m network) to 60 steps (4m256m network). Due to the increasing number of edges between the same number of nodes, the propagation can converge in fewer simulation steps. As shown in Figure 6.12 (b) and (d), multiple GPUs with edge-oriented processing can share the computational workload. However, there is a tradeoff between the computation performance improvement and the synchronization and data transfer cost. Compared to edge-oriented processing, the cost of synchronization and the cost of data transfer in vertex-oriented processing are the dominant factors that influence the simulation performance.

According to the performance comparison between Single GPU and Multi-GPU simulation using edge-oriented processing, we can obtain better performance by Multi-GPU on 1m128m (3 GPUs), 2m128m (3 GPUs), 4m128m (2 GPUs), and 4m256m (3 GPUs) networks respectively. In vertex-oriented processing, we can obtain a performance gain by Multi-GPU only on the 1m128m (2 GPUs) network.

Essentially, there are four factors which affect the simulation performance, *Stimulation Steps*, *Computation Time*, *Synchronization Time*, and *Data Transfer Time*. It is therefore necessary to discuss each factor in detail.

The number of simulation steps measures how long the information propagation takes to converge. Such a convergence behavior is highly influenced by the average number of edges per node in random networks. However, in scale-free networks [119], the convergence behavior is also influenced by a few nodes which have a very large number of edges compared to others. In the experiments with random networks, we note that the number of simulation steps varies according to the average number of edges per node. In practice, all the networks can converge within the number of simulation steps shown in the figures except the 4m32m network. In particular, the simulation of the 4m32m network can propagate to 99.7% nodes within 400 steps and eventually reach to 99.9%

nodes at 1000 steps. Therefore, we can conclude that if we double the average number of edges per node, the number of convergence steps is roughly halved.

The total computation time (sum of the computation times for each simulation step) is shown in Figure 6.13. We can observe that the workload of network processing can be shared by multiple GPU devices, both in edge-oriented processing and in vertex-oriented processing. However, compared to the total execution time shown in Figure 6.12, the cost of synchronization and data transfer may become a bottleneck in Multi-GPU processing. In Figure 6.13 (a), the number of simulation steps is the main factor which affects the edge-oriented execution time in *Node-Scaling* experiments. However, as shown in Figure 6.13 (b), the computation time in *Edge-Scaling* experiments is influenced not only by the number of simulation steps but also by the number of edges. In vertex-oriented processing, the computation time is obviously influenced by the number of nodes. More importantly, we can observe that the Multi-GPU computation performance is highly optimized but the total performance is dominated by the cost of synchronization and data transfer.

We also identify the maximum synchronization time during simulation, which is shown in Figure 6.14. As we discussed in the case study of the 1m256m network, the maximum synchronization time appears in the middle of the simulation when many nodes are newly activated. In the *Node-Scaling* experiments, the maximum synchronization cost grows significantly when increasing the number of nodes. In the *Edge-Scaling* experiments, with an increasing number of edges, the number of cross-partition edges also increases. Thus, the number of boundary nodes and replica nodes increases as well. This results in a high synchronization cost since more boundary-replica nodes need to be synchronized when we scale up the number of edges. The experimental results also indicate that the synchronization cost highly depends on the number of partitions.

Figure 6.13. Total Computation Time

Figure 6.14. Maximum Synchronization Time

The data transfer time measures the time spent to collect data from GPUs and the time spent to transfer the updated data from CPU back to GPUs. As the performance of data transfer only depends on the size of the boundary table and the size of replica table in the GPU devices, the *Data Transfer Time* is quite stable at each step. We can therefore measure the data transfer performance according to the average value at each simulation step, which is shown in Figure 6.15. According to the experimental results, we note that scaling the network size and changing the number of partitions change the size of boundary and replica tables. Furthermore, the boundary table size and the replica table size directly affect the data transfer performance.

Figure 6.15. Average Data Transfer Time

According to the synchronization performance we have discussed above, we note the maximum synchronization cost and the average data transfer cost are directly influenced by the same factors, including the size of the boundary and replica tables, and the number of partitions. Therefore, we can define a metric to investigate the data replication performance in the Multi-GPU system. The *Replication Ratio R* denotes the average node

replication, where $R = \frac{\text{Replica Buffer Size}}{\text{Number of Nodes}}$. Given a network which is partitioned into P partitions, we can have $R \in [0, P - 1]$. Here, $R = 0$ if there are no edges between different partitions. However, $R = P - 1$ if each node is replicated in each of the other $P - 1$ partitions. To compare the replication ratio between the vertex-oriented structure and the edge-oriented structure in the Multi-GPU system, we show the replication ratio of scalable networks in Figure 6.16. We can observe that the replication ratio increases with an increased number of partitions. In addition, the vertex-oriented structure shows a higher replication ratio than the edge-oriented structure in the cases where the average number of edges per node is relatively small, both in the node-scaling experiments and in the edge-scaling experiments. However, in the cases with a large number of edges per node, the vertex-oriented structure and the edge-oriented structure show a similar performance in data replication. That is, the replication ratio R approximates to $P - 1$.

Figure 6.16. Replication Ratio of Scalable Networks

6.2.3 Discussion of Simulation Performance of Random Networks on Multi-GPU

According to the detailed performance analysis of scalable random networks, we can observe that the replication ratio of random networks is high. The high replication ratio results in an expensive cost of data transfer and synchronization at the end of each simulation step. It offsets the massive processing capability of multiple GPU devices.

Furthermore, in the simulations with vertex-oriented processing, we note that the cost of data transfer and synchronization becomes a bottleneck and dominates the total execution time. As the adaptive simulation algorithm is optimized according to its algorithmic complexity, the pure computation time on GPU is very small. It approximates to zero in the beginning and the end of the simulation (e.g. Figure 6.9 (b)). The Multi-GPU system can only show its computing advantages in the middle of the simulation where a large network browsing workload is required for processing. In the beginning and the end of the simulation, the cost of data transfer and synchronization can be considered as a constant overhead (e.g. Figure 6.9 (c) and (d)). Such an overhead is accumulated with the number of simulation steps. Therefore, those vertex-oriented processing experiments with a larger number of simulation steps are significantly influenced by the cumulative cost of data transfer and synchronization at the end of each simulation step. However, simulations with edge-oriented processing are not significantly influenced by the number of simulation steps since all performance factors are stable at each simulation step except for the synchronization (e.g. Figure 6.10).

6.3 Performance Evaluation of Scale-free Networks

We have noted that the high replication ratio of random networks on Multi-GPU results in an expensive cost of data transfer and synchronization. In this section, we show how the simulation performance is influenced by the network structures. We use the RMAT method to generate a few scale-free networks. The simulation performance between random networks and scale-free networks are compared on multiple GPUs.

As shown in Figure 5.2, the nodes in a scale-free network show an unbalanced number of connected edges. With the power-law degree distribution, a few nodes have very large contact lists compared to others and form node clusters within the network. The number of connections between these clusters may be very small compared to the random networks with uniform degree distribution. Thus, we may benefit from the reduced cost of data transfer and synchronization. As shown in the previous section, the replication

ratio is an important factor that influences the cost of data transfer and synchronization. Such a cost of data transfer and synchronization at each simulation step further influences the performance at each step. The replication ratio of certain scale-free networks is shown in Figure 6.17. It shows that the replication ratio of the scale-free networks is smaller than that of the random networks.

Figure 6.17. Replication Ratio of RMAT Scale-free Networks vs. Random Networks

In addition, it is very interesting that the replication ratio of 3 GPUs is close to that of 2 GPUs. With the detailed analysis of the replica buffer, we can observe there are only a few boundary nodes in one of the three partitions so that the total size of the replica buffer of 3 GPUs is comparable with that of 2 GPUs. Thus, the cost of data transfer and the cost of maximum synchronization on 3 GPUs are comparable with that on 2 GPUs. The performance analysis of data transfer and synchronization of a 1m128m RMAT scale-free network is shown in Figure 6.18.

Figure 6.18. Cost of Data Transfer and Data Synchronization (1m128m)

As the cost of data transfer and synchronization is reduced significantly in simulations of scale-free networks, we may achieve performance improvement using multiple GPU devices. The total execution time is shown in Figure 6.19, again excluding time for random number generation. Simulation terminates at 20 steps, 37 steps and 85 steps for 1m128m, 2m128m, 4m128m networks respectively. We can observe that vertex-oriented processing is still the best algorithm in simulations of scale-free networks. Furthermore, compared to the Multi-GPU vertex-oriented processing performance on random networks, we note that a performance gain can be obtained in the Multi-GPU vertex-oriented processing on scale-free networks.

Figure 6.19. Total Execution Time of Scale-free Networks

Using a baseline of single GPU vertex-oriented processing performance, the Multi-GPU system can achieve 11.3%, 30.7%, 29.1% performance improvement for simulation of the 1m128m scale-free network on 2, 3 and 4 GPUs respectively. For the simulation of the 2m128m scale-free network, we can achieve 7.4% performance improvement on 3 GPUs compared to the single GPU vertex-oriented processing. For the simulation of 4m128m scale-free network, we cannot obtain a performance gain in vertex-oriented processing on multiple GPUs. This is because the number of simulation steps for the 4m128m network is larger than that for the 1m128m and 2m128m scale-free networks. Its cumulative cost of data transfer and synchronization during simulation increases significantly compared to the other two cases. Compared to the optimized computation of vertex-oriented processing, the cost of data transfer and synchronization dominates the total execution time. Thus, the Multi-GPU system cannot show its computing advantage in

vertex-oriented processing of the 4m128m network.

However, we can obtain performance improvement in edge-oriented processing in scale-free networks. This is because the algorithmic complexity of edge-oriented processing is approximately constant at each simulation step. Thus, we can observe that the Multi-GPU system can show its massive parallel processing advantages in simulation of information propagation. Using a baseline of single GPU processing performance, we can obtain an average 10.0%, 34.4%, and 36.1% performance improvement on 2, 3, and 4 GPUs.

In addition, as shown in Figure 5.4 (c), we note that in simulations of scale-free networks, information can propagate to most of the nodes quickly but it takes a longer time to converge compared to simulations of random networks. Furthermore, in the beginning and the end of the simulation, the performance of vertex-oriented processing is significantly influenced by the cumulative cost of data transfer and synchronization at the end of each simulation step. Thus, the performance of vertex-oriented processing of scale-free networks is inefficient near the end of simulation since it takes a long time to converge. In practice, many studies of information propagation over complex networks are concerned with the time it takes to propagate information to the majority of nodes in the network, rather than all nodes [6][43][85]. We therefore show the execution time with 95% convergence, as shown in Figure 6.20. The experiments terminate at 10 steps, 16 steps, and 27 steps to reach 95% of the nodes of 1m128m, 2m128m, and 4m128m scale-free networks respectively.

Figure 6.20. Total Execution Time of Scale-free Networks (95% nodes convergence)

The performance improvement compared to a single GPU for vertex-oriented processing using the Multi-GPU system is shown in Table 6.1. We can observe that the simulation performance is significantly improved in those experiments with 95% convergence compared to those with 100% convergence since the number of simulation steps is reduced. The accumulated cost of data transfer and synchronization at each simulation step is therefore also reduced.

For edge-oriented processing, comparing the experiments of 100% convergence with those of 95% convergence, the performance improvement is similar, as shown in Table 6.2. This is because edge-oriented processing is not influenced by the number of simulation steps and can be considered as a simulation algorithm with a constant complexity.

Table 6.1. Performance Improvement over Single GPU (Vertex-Oriented Processing)

	1m128m		2m128m		4m128m	
Convergence	(100%)	(95%)	(100%)	(95%)	(100%)	(95%)
2 GPU _s	11.3%	16.9%	-10.4%	8.0%	-73.4%	-32.7%
3 GPU _s	30.7%	39.9%	7.4%	31.1%	-58.5%	11.4%
4 GPU _s	29.1%	45.8%	-38.1%	11.6%	-151.1%	-13.2%

Table 6.2. Performance Improvement over Single GPU (Edge-Oriented Processing)

	1m128m		2m128m		4m128m	
Convergence	(100%)	(95%)	(100%)	(95%)	(100%)	(95%)
2 GPU _s	15.5%	15.3%	10.4%	10.5%	4.0%	2.4%
3 GPU _s	35.4%	35.1%	40.4%	39.9%	27.5%	26.2%
4 GPU _s	44.0%	42.6%	35.3%	33.9%	28.9%	26.5%

6.4 Summary

In this chapter, with the motivation of extending the space scalability of GPU simulation, we present a Multi-GPU simulation design for information propagation over complex networks. To obtain high performance simulation on multiple GPU devices, an efficient network partitioning and replication method is important. It should not only balance the workload in each GPU device but also reduce the data communication and synchronization cost. Metis is adopted for partitioning the input network and generating a partition file as the output. According to the original network file and the partition file, we propose a replication scheme to store the information of boundary nodes and the corresponding cross-partition edges. The specified network structure (vertex-oriented or edge-oriented) can then be built in different GPUs. The boundary and replica nodes in different devices need to be synchronized in order to maintain the network state consistency at each simulation step.

With the proposed data replication and synchronization scheme, we can map the optimized A-Loop and E-Loop simulation algorithms onto multiple GPU devices. The simulation performance is comprehensively evaluated in large scalable networks. According to the simulation performance of large scalable networks on multiple GPU devices, we observe that there are four factors that influence the performance, including (1) the number of simulation steps, (2) computation time, (3) synchronization time, and (4) data transfer time. Although the Multi-GPU system introduces an overhead, this is acceptable if it is more important to enhance the system scalability for very large network simulations than to obtain a performance gain.

Furthermore, we generate certain scale-free networks using the RMat method in GTGraph. It shows that the replication ratio of scale-free networks is much lower compared to random networks. The cost of data transfer and synchronization at the end of each simulation step is also reduced significantly. Therefore, we can obtain performance improvement using multiple GPU devices.

Simulation Studies of Viral Advertisement Diffusion over a Real World Social Network

In this chapter, we apply our simulation strategies to real world networks as they may have more complex network structures than the synthetic networks we used in previous chapters. We explore how the complex network structure affects the simulation performance on single GPU and Multi-GPU. Furthermore, we choose an interesting example of viral advertisement diffusion as a case study to show how simulation can help us to understand the nature of advertisement diffusion over social networks.

7.1 Background

In traditional social communities, individuals interact with others in order to share ideas, events, and interests within their individual networks. With the emergence of Social Network Services (SNS), such interactions are not only delivered by word-of-mouth communication but also propagated on the websites or by online communities. The online social network service or platform facilitates faster and wider reach of information or innovation. In traditional mass marketing, the aim is to provide products and services that will appeal to the whole market [88]. It usually focuses on the media that can reach a large number of people, such as radio, television, as well as newspapers. Different from mass marketing, viral marketing or viral advertising can propagate a product or an advertisement through self-replicating viral processes which are analogous to the spread

of viruses or computer viruses [136].

The behavior of viral advertising can be different according to the different marketing strategies used. One important difference is whether the informed nodes can propagate the advertisement in subsequent rounds. Consider the case of Hotmail, one of the earliest firms to utilize viral marketing techniques in 1996/1997 by inserting the tagline “Get Your Private, Free e-mail at www.hotmail.com” at the bottom of every e-mail sent out by its users [72] [136]. In this case, the viral advertising is delivered by the active users in many rounds. Consider another case of online contents of multimedia firms such as New York Times, which offers a “Share This Post on Facebook” option on their sites. This feature causes the contents to be added on the recommender’s Facebook page. If one costumer shares it, such a post can then be circulated to the recommender’s connected friends. In such a case, the advertisement is usually propagated only once by the same recommender.

In addition, the effect of advertisement diffusion through a social network depends on the initial selected nodes. It is an open research question how to select the most influential nodes to propagate the information. Domingos and Richardson investigated the problem of how to estimate the overall spreading effect of new products though the network by probabilistic modeling [115]. Kempe, Kleinberg and Tardos have shown the optimization of selecting the most influential nodes is an NP-hard problem [73]. They proposed a greedy strategy which shows a performance guarantee with 63% near optimality in several cases [41] [73]. However, without much knowledge of the structure of a given network, heuristically selecting the nodes with large degree is an alternative and efficient way to maximize the spread of advertisement or innovation over complex social networks [35] [109].

7.2 Agent-based Advertisement Diffusion Model

In the model definition, each agent (or individual) is represented by a node in a social network. The friendship between agents is represented by edges and is assumed to be

mutual. Before we start execution, a set of initial selected (IS) nodes are initialized with the informed state. Others are initialized with the uninformed state. During simulation, the transmission probability for informing equals P . For simplicity, in this chapter, we define a threshold $T \in [0, 1]$ on each node, where $T = 1 - P$. This represents the threshold for a propagation attempt by a single neighbor. If a node u has a set of active neighbors A_t , the overall threshold can be expressed as $T_u = (1 - P)^{|A_t|}$, as discussed in Chapter 4. According to the pre-defined threshold value, the advertisement can be propagated from the IS set to the rest of the individuals among the network. The threshold value is an important factor representing the tolerance between a pair of connected agents and it can affect the diffusion behaviors significantly. In the real world, the threshold value can be reduced by mass marketing. That is, an investment on mass marketing can increase the willingness of individuals to accept a product or an advertisement [76].

In the simulations of viral advertising with subsequent propagation attempts, we need to re-assign random numbers to the edges connecting informed and uninformed nodes. However, the GPU is very efficient at generating a large amount of random numbers in a batch. Thus, we re-assign random numbers (with uniform distribution) to every edge element at the end of each simulation step in order to avoid the extra cost of specifying the edges connecting nodes with different states. In the simulations without subsequent propagation attempts, we can assign the random numbers only once in the initialization stage. As we have limited knowledge about the input network for the simulation, we can apply a heuristic approach to investigate the diffusion behaviors influenced by different initial nodes selection policies. In practice, we choose the nodes with largest degree or randomly selected. Before we start the simulation, the initial selected nodes are initialized to informed state. Once the simulation begins, informed nodes can trigger the cascade influence by advertising or recommending the products to their friends in connected groups.

7.3 Simulations Methods and Performance

As discussed in section 4.5, the network can be stored as a vertex-oriented structure (bidirectional graph) or an edge-oriented structure (undirected graph). Two different algorithms, runtime adaptive vertex-oriented processing or edge-oriented processing can be used to simulate the advertisement diffusion behaviors over social networks. LiveJournal [86] is a free online community with almost 10 million members which allows members to maintain and share journals, individual and group blogs, and it allows people to connect to other members as their friends. Such a large real world online social network is sampled with a directed graph dataset with 4,847,571 nodes and 68,993,773 edges and it can be accessed and downloaded via the internet [87]. The original graph is represented by directed edges and there may be multiple edges between two specific nodes to represent different types of relationship. As the friendship between agents is assumed to be mutual in our approach, we need to remove multiple edges and change the structure to an undirected or bidirectional network for edge-oriented processing and vertex-oriented processing respectively. According to the graph modification, we obtain an undirected graph with 4,847,571 nodes and 42,851,237 edges for edge-oriented processing and a bidirectional graph with 4,847,571 nodes and 85,702,474 edges for vertex-oriented processing. In addition, we note that there are 962 completely isolated nodes among the network. Other simulation parameters are: shown in Table 7.1

Table 7.1. Simulation Parameters for Varying Thresholds

Simulation Algorithm	Vertex-Oriented Processing, Edge-Oriented Processing
Diffusion Threshold	$T = 0.1, 0.3, 0.5, 0.7, 0.9$
Hardware Platform	Nvidia Tesla C2050, Intel Xeon E5645
Initial Informed Node	1 (Random Selection)

7.3.1 Simulations with Subsequent Propagation Attempts

As shown in Figure 7.1, the simulation results indicate that the value of the threshold affects the diffusion behaviors significantly. That is, the convergence time (number of

simulation steps) increases with an increased threshold. In particular, the simulations converge within 15, 22, 30, 50, and 150 steps with the threshold of 0.1, 0.3, 0.5, 0.7, and 0.9 respectively. The microscope view of the first 30 steps is also shown in the lower right corner of Figure 7.1.

In addition, we also note that the spread of the advertisement can reach only 4,843,953 nodes compared to the total 4,847,571 nodes in the network (with 962 isolated nodes). It indicates that there are 2656 nodes belonging to small clusters which are not connected to the main part of the network. Using our proposed simulation strategies, we can adopt either the E-Loop or the A-Loop to simulate the advertisement diffusion behaviors on multiple GPUs. Furthermore, as we do not have detailed knowledge of the network structure, the algorithmic adaptation strategy in the A-Loop (switching from C-Loop to T-Loop) can be conducted when the number of informed nodes is greater than the number of uninformed nodes.

Figure 7.1. Simulation with Varying Threshold (with subsequent attempts)

The detailed network browsing performance is shown in Figure 7.2. The execution times shown are for network browsing only and do not include random number generation. The parallel speedup of the network browsing algorithms (A-Loop and E-Loop) compared to CPU serial execution are shown in Figure 7.3.

According to the experimental results, we can observe that the E-Loop on 3 GPUs can

obtain the best performance with edge-oriented processing. The A-Loop on 1 GPU can achieve the best performance with vertex-oriented processing except for the experiment with threshold $T = 0.3$ where the best performance is obtained on 2 GPUs.

As a summary, the E-Loop on 3 GPUs achieves an average 16.3x speedup compared to the CPU serial execution. However, the A-Loop on 1 GPU only achieves an average 5.1x speedup compared to the A-Loop execution on the CPU. This is due to the uncoalesced memory access in vertex-oriented processing in which we assign CUDA cores to the vertex array. As shown in Figure 5.9, such uncoalesced memory accesses may decrease the parallel performance on GPU. Compared to vertex-oriented processing, the edge-oriented processing with the coalesced memory access can obtain significant parallel speedup on GPU.

Figure 7.2. Simulation Performance on Multi-GPU System (with subsequent attempts)

Figure 7.3. Parallel Speedup of Network Browsing Algorithms (with subsequent attempts)

7.3.2 Simulations without Subsequent Propagation Attempts

As shown in Figure 7.4, the simulation results indicate that the advertisement may not reach the entire network without multiple rounds of propagation. Furthermore, the final number of informed nodes we can reach is greatly influenced by the threshold value. According to the experimental results shown in Figure 7.4, we can reach 97.9%, 92.7%, 86.0%, 74.5%, and 41.6% of the total number of nodes with the threshold of 0.1, 0.3, 0.5, 0.7, and 0.9 respectively.

Figure 7.4. Simulation with Varying Threshold (without subsequent attempts)

The simulation performance of the E-Loop and the A-Loop on Multi-GPU is shown in Figure 7.5, again excluding the time for random number generation. The parallel speedup of the network browsing algorithms (A-Loop and E-Loop) compared to CPU serial

execution are shown in Figure 7.6.

Figure 7.5. Simulation Performance on Multi-GPU System (without subsequent attempts)

Figure 7.6. Parallel Speedup of Network Browsing Algorithms (without subsequent attempts)

In general, the E-Loop on 3 GPUs can obtain the best performance with edge-oriented processing. The A-Loop on 1 GPU can achieve the best performance with vertex-oriented processing except for the experiment with threshold $T = 0.3$ and $T = 0.9$. As a summary, the E-Loop on 3 GPUs achieves an average 22.1x speedup compared to the CPU serial execution. However, the A-Loop on 1 GPU only achieves an average 6.0x speedup compared to the A-Loop execution on the CPU.

Furthermore, we can observe that the A-Loop can obtain a better performance gain than the E-Loop with the threshold of 0.1, 0.3, 0.5, and 0.7. In particular, the A-Loop on 1 GPU achieves the best overall simulation performance with the threshold of 0.1, 0.5, and 0.7. The A-Loop on 2 GPUs achieves the best performance with the threshold of 0.3. However, it is interesting that the E-Loop performs better than the A-Loop with the threshold of 0.9. The best E-Loop configuration (E-Loop on 3 GPUs) obtains 3.1x speedup compared to the best A-Loop configuration (A-Loop on 3 GPUs). In the simulation with threshold of 0.9, the number of informed nodes reaches only 42% of the total number of nodes in the network. The algorithmic adaptation condition is not satisfied. Thus, the A-Loop follows the C-Loop without swapping to the T-Loop.

The detailed simulation performance with the threshold of 0.9 on Multi-GPU is shown in Figure 7.7. Although a significant performance gain can be achieved by Multi-GPU, the A-Loop is not an optimized network browsing algorithm as it only follows the C-Loop without runtime adaptation to the T-Loop.

Figure 7.7. Detailed Simulation Performance of Threshold = 0.9 on Multi-GPU

In order to compare the A-Loop performance with or without algorithmic adaptation, the simulation performance with the threshold of 0.9 and 0.7 is shown in Figure 7.8.

Figure 7.8. Detailed Simulation Performance ($T = 0.9$, $T = 0.7$)

7.3.3 Replication Ratio on Multi-GPU

We observed in Section 6.2.2 that the cost of synchronization and data transfer may become performance bottlenecks and degrade the Multi-GPU simulation performance. The Replication Ratio was introduced to denote the average node replication. To compare the replication ratio for this social network between the vertex-oriented structure and the edge-oriented structure in the Multi-GPU system, we show the replication ratio in Figure 7.9. The replication ratio R denotes the average node replication. Given a network which is partitioned into P partitions, we can have $R \in [0, P - 1]$.

Figure 7.9. Replication Ratio

We can observe that the replication ratio is not high on Multi-GPU. An average replication ratio with the edge-oriented structure is around 60% of that with the vertex-oriented structure. The relatively low replication ratio indicates that without the high cost of synchronization and data transfer it is possible in some cases to obtain a performance gain on a Multi-GPU system, as shown in Figures 7.2 and Figure 7.5.

7.3.4 Performance of Random Number Generation

In addition to the simulation performance for network browsing, it is also necessary to discuss the performance of random number generation (RNG) since we have to generate a large set of random numbers in order to assign one number to each edge element. Thus, the size of the required set of random numbers is equal to the number of edges. The average time cost of RNG for one step on different platforms for the given social network is shown in Table 7.2.

Table 7.2. Time Cost of Random Number Generation (millisec)

	CPU	1 GPU	2 GPUs	3 GPUs	4 GPUs
Edge-Processing (43m RNG)	3482.3	20.5	37.4	55.1	72.6
Vertex-Processing (86m RNG)	6912.3	22.8	39.8	57.4	74.8

We note that the GPU-based RNG performs much better than that on the CPU. This is because the Curand library is highly optimized to fully utilize GPU streaming processors. For example, 1 GPU obtains a 169.9x speedup compared to CPU with edge-oriented processing and a 303.2x speedup compared to CPU with vertex-oriented processing. Interestingly, we have also observed that a single GPU performs better than multiple GPUs in the experiments of GPU-based RNG. This is due the serialized process on the CPU host for the construction and de-construction of the Curand library instance at each simulation step. This is denoted by O_{cu} . According the performance calibration, the overhead is approximately 18.2 milliseconds. The RNG performance on Multi-GPU can

be expressed as

$$T_{num_GPU}^{RNG} = O_{cu} \times num_GPU + \frac{Workload}{num_GPU}$$

where *Workload* denotes the actual processing cost of RNG and *num_GPU* denotes the number of GPU devices. Such an approximation of time cost is very similar to the experimental results shown in Table 7.2. According to the detailed RNG performance on 1 GPU and CPU which is shown in Figure 7.10, we can observe that the GPU can achieve an excellent performance gain in random number generation compared to the CPU serial counterpart, especially when a very large set of random number is required.

Figure 7.10. Performance of Random Number Generation

The results of average total parallel speedup (an average value over different threshold values) including the cost of network browsing and the cost of RNG compared to CPU serial execution are shown in Figure 7.11.

Figure 7.11. Total Parallel Speedup

As the GPU performs much better in random number generation compared to the CPU, we can obtain significant speedup in the GPU system if we include the cost of network browsing and the cost of random number generation, especially in the simulations with subsequent attempts. This is because the cost of RNG becomes a bottleneck and constrains the performance of the CPU.

In the simulations with subsequent attempts, we need to re-assign random numbers at the end of each simulation step. As shown in Table 7.2, the cost of RNG at each step for vertex-oriented processing is more than twice as much as for edge-oriented processing. The speedup of the A-Loop is approximately twice as much as the E-Loop. In the simulations without subsequent attempts, we include the cost RNG only at the initialization step (no re-assignment). According to the results, we can observe that the E-Loop can obtain a better parallel speedup (on average) compared to the A-Loop.

As a conclusion, the A-Loop on 1 GPU can obtain the best overall simulation performance gain in the simulations with subsequent propagation attempts. However, in the simulations without subsequent propagation attempts, we should choose the A-Loop on 1 GPU with $\{T = 0.1, 0.5, 0.7\}$, the A-Loop on 2 GPUs with $\{T = 0.3\}$ and the E-Loop on 3 GPUs with $\{T = 0.9\}$. We can apply these optimal system configurations to conduct the following experiments on the initial nodes selection policies.

7.4 Diffusion Behaviors using Different Initial Nodes Selection Policies

In this section, we investigate the advertisement diffusion behaviors using different policies for the set of initial selected nodes. The greedy algorithm can be used to find the most influential nodes to spread the information or products [73] [82]. Various measurements of centrality of a node determine the relative importance of such a node in the given graph. In the definition of the “betweenness centrality”, nodes that have a high probability to appear on a shortest path between two randomly chosen nodes have a high betweenness [25]. The “degree centrality” is defined as the number of links connected to a given node [25], i.e. the node degree.

As we have limited knowledge about the input network for the simulation, we can apply a heuristic approach to investigate how the diffusion behaviors are influenced by different selection policies. In practice, we choose the nodes with largest degree or use a random selection. Before we start the simulation, the initial selected nodes are initialized to informed state. Once the simulation begins, informed nodes can trigger the cascade influence by advertising or recommending the products to their friends. In the following experiments, the number of initial selected nodes is scaled from 1 to 1000 either with the largest degree or the random node selection policies. The simulation parameters are shown in Table 7.3.

Table 7.3. Simulations of Different Initial Nodes

Simulation Algorithm	A-Loop, E-Loop
Diffusion Threshold	$T = 0.1, 0.3, 0.5, 0.7, 0.9$
Node Selection Policy	Largest (L), Random (R)
Number of Initial Selected Nodes	IS = 1, 10, 100, 1000

7.4.1 Simulation with Subsequent Propagation Attempts

Both the overview and the microscope view (the small figure on the right) behaviors during simulation are shown in Figure 7.9. With a specific threshold, the largest degree nodes selection policy can achieve better diffusion performance compared to that of the random nodes selection policy. The diffusion influence by targeting selected nodes (L (size of IS) or R (size of IS)) has a descending order as shown below. Interestingly, such an order applies to all simulation experiments with different threshold values.

$$L(1000) > L(100) > L(10) > R(1000) > R(100) > R(10) \approx L(1) > R(1)$$

We can observe that the number of initial selected nodes is important to the diffusion behaviors. However, we also note that the initial selection policy plays a limited role in the final result of advertisement diffusion. That is, the largest degree selection and a larger number of initial selected nodes can only help in maximizing the spread in the beginning and early stage of simulation.

Figure 7.12. Simulation with Different Initial Selected Nodes (with subsequent attempts)

The initial node selection policy is unlikely to improve advertisement diffusion continuously. For example, in the experiment with threshold equal to 0.1, the advertisement can reach nearly the whole network within 7 steps even if we randomly select 1 initial informed node. In another example, in the experiment with threshold equal to 0.9, no matter which initial selection policy we choose, the diffusion behaviors during the entire simulation process are very similar. It is therefore not necessary to select a large number of initial informed nodes or to choose the nodes with the very largest degree.

We also note that the propagation behaviors influenced by different node selection policies are not crucial compared to the impact of varying the threshold value. The advertisement can be propagated to nearly the whole network quickly by decreasing the threshold value.

7.4.2 Simulation without Subsequent Propagation Attempts

In Figure 7.13, we show how the diffusion behaviors are influenced by different initial nodes selection policies in simulations without subsequent propagation attempts.

Figure 7.13. Simulation with Different Initial Selected Nodes (without subsequent attempts)

We note that the largest degree selection policy can obtain better performance in maximizing the diffusion influence than the random node selection policy in the beginning and early stages. Furthermore, the performance of different initial nodes selection policies follows approximately the same descending order as that in the simulations with subsequent attempts.

According to the experimental results shown in Figure 7.13, we note that the final number of informed nodes is determined mainly by the value of the threshold. It does not depend on which initial nodes selection policy is chosen. That is, the optimal initial nodes

selection policy is not a crucial factor in maximizing the final number of informed nodes compared to reducing the threshold value. In Figure 7.14, we show the detailed diffusion performance that is obtained by varying threshold values and varying initial node selection policies. According to the experimental results, the advertisement can eventually reach an average 41.8%, 78.4%, 88.7%, 94.1%, and 98.2% nodes of the given network with threshold equal to 0.9, 0.7, 0.5, 0.3, and 0.1 respectively.

Figure 7.14. Ratio of Final Informed Nodes with Different Initial Selected Nodes

7.5 Discussion of Viral Advertising Strategies

We have illustrated the simulation results of viral advertising over a large-scale social network in the previous sections. We can observe that a good initial nodes selection policy performs an important role in the early stage of advertisement diffusion but it has a limited influence in the later stage. Such a behavior is shown both in the simulations with subsequent propagation attempts and in the simulations without subsequent propagation attempts. The threshold value plays a more important role in maximizing the diffusion

velocity through the network (with subsequent attempts) or in maximizing the diffusion range over the network (without subsequent attempts). Furthermore, with a relatively small threshold value, the viral advertising without subsequent attempts can achieve a similar diffusion performance to that of the viral advertising with subsequent attempts and is able to propagate to almost the entire network. For example, the viral advertising can propagate to 94.1% and 98.2% of the nodes in the network with threshold equal to 0.3 and 0.1 respectively.

In fact, the diffusion threshold in the social network represents the tolerance between a pair of connected persons. If the attraction of the advertisement from his/her friends is greater than the tolerance, he/she may accept such information and is willing to spread it within his/her personal social network. Therefore, we can combine mass marketing and viral marketing together to achieve the best strategy to spread the advertisement [141]. In practice, we can first use mass marketing to reduce the tolerance of all the members within the social network. After that, we can apply a viral marketing strategy with an initial selection of a limited number of efficient nodes to facilitate the maximization of the rate of spread of the advertisement. Furthermore, once the threshold value is reduced, we can apply a viral advertising strategy without subsequent attempts since frequent viral advertising by many attempts cannot further enhance the diffusion significantly. Otherwise, frequent viral advertising may be considered as a spam which is toxic in marketing [72].

According to our studies of advertisement diffusion using this simulation approach, we obtain results that correspond to existing viral marketing reviews on real cases [138]. That is, viral marketing is attractive in the era of active social network services (web 2.0 or online social community) because it seems like a free lunch strategy: select a small number of people to seed the idea or product and let it go viral. Then it can spread to millions of people automatically in a short period. However, it shows the success of purely viral marketing is quite limited. The idea of “Big Seed Marketing” is proposed to combine traditional mass marketing and viral marketing together [138]. Traditional mass

marketing supplemented by viral marketing shows an advantage in achieving better return on the advertising and marketing budget.

7.6 Summary

This chapter investigates a case study on advertisement diffusion over a real world complex social network by advanced simulation of information propagation strategies that have been presented in chapters 4 to 6. An agent-based model is introduced to simulate the large-size LiveJournal online social network. Both the E-Loop and the A-Loop are used for the advertisement diffusion simulations with and without subsequent attempts.

In the first part of the experiments, simulations are conducted on CPU and multiple GPUs with varying value of threshold. The simulation performance is discussed in detail, especially the network browsing cost. We also discuss the cost of random number generation. In the second part, we use these experimental results to choose the optimal simulation algorithm to investigate how the diffusion behaviors are influenced by different initial nodes selection policies. We observe that the largest degree selection and the large number of initial selected nodes can help in maximizing the spread only in the beginning and early stage of advertisement diffusion. The influence of a good node selection policy in advertisement diffusion is not that crucial compared to reducing the threshold.

In this chapter, we also discuss improved viral advertising strategies based on the simulation results. To propagate an advertisement or a product efficiently, a company could first invest on mass marketing to reduce the threshold (increase the willingness) and then deploy a proper viral marketing campaign in order to facilitate the spread of the advertisement. Such a conclusion by simulation based studies represents the phenomenon that exists in real world cases in marketing.

Conclusions and Future Work

8.1 Conclusions

Simulations of information propagation over complex social networks have become an important method of obtaining a better understanding of social behaviors. The simulation results can provide valuable implications for policy making, such as government control policies during epidemic outbreaks. However, simulations of large-scale networks are computationally expensive since a large size memory is required to store the network layout and a large amount processing capacity is needed to conduct the simulation. In order to improve the computational performance, HPC techniques are increasingly applied to large-scale scientific applications such as simulations. Although the technological advancements are exciting, it is difficult to reap the full benefits of these advanced computer systems without developing an optimized processing strategy. It is an open research question how to map complex computing problems efficiently onto new architectures. It is therefore necessary to develop new performance optimization techniques that can be used for simulations of information propagation in an HPC environment.

This project aims to design optimized simulation strategies of information propagation over complex networks. Furthermore, the project explores how to map these optimized

simulation strategies onto the new generations of HPC platforms to improve simulation performance. A component-based infectious disease simulator with two different network browsing kernels (the I-Loop and S-Loop) is presented as a motivational example. Using algorithmic analysis, we construct accurate performance models to estimate the program behaviors such as the execution time of both the I-Loop and S-Loop. Furthermore, we demonstrate that the proposed method of performance modeling can provide a theoretical way to guide the kernel adaptation at runtime (A-Loop), and that it works well in practice. With the kernel adaptation, the optimized serial execution time of the simulation achieves a 14.7x speedup compared to the original S-Loop and a 4.7x speedup compared to the original I-Loop.

Performance models for Multi-core architectures are also presented by extending the serial performance models. Moreover, the optimized kernel swapping strategy in a multi-threaded parallel processing environment can be also expressed using mathematical models. With parallel processing on a Multi-core architecture, the I, S, and A-Loops further obtain up to 4.04x, 4.54x, and 2.78x speedup with up to 8 threads compared to their serial execution. Furthermore, the bottleneck of parallel execution is explored analytically using the performance models. It is shown that the OpenMP overhead is a major factor in the bottleneck of parallel execution.

Inspired by the runtime kernel swapping between the I-Loop and the S-Loop in infectious disease simulation, the optimized algorithmic adaptation approach is applied to other applications in the general field of information propagation. Two basic models of information propagation are introduced: the Independent Cascade Model and the Linear Threshold Model. Furthermore, these two basic models can be modified to give the General Cascade Model and the General Threshold Model, respectively. More importantly, these two general models can be shown to be equivalent and interchangeable. Such model equivalence provides the possibility to combine the two general models together to derive optimization strategies which enable algorithmic adaptation at runtime in order to choose the most efficient algorithm at each simulation step.

To show the optimized simulation strategies, we propose two different types of data structures to store the network layout: the vertex-oriented structure and the edge-oriented structure. The vertex-oriented network structure can obtain performance benefits from optimized computational complexity by using runtime adaptation. Compared to the vertex-oriented structure, the edge-oriented structure is used to achieve optimized memory space in hardware.

With an optimized adaptive vertex-oriented processing algorithm (A-Loop) on the vertex-oriented structure, we can achieve better execution performance. With an edge-oriented processing algorithm (E-Loop), we obtain better spatial scalability than that with vertex-oriented processing. The simulation performance is shown on both Multi-core CPU and many-core GPGPUs respectively. We can obtain an average 2.74x speedup with the A-Loop and an average 3.61x speedup with the E-Loop in a Quad-core multi-threading environment, compared to the CPU serial execution. Moreover, an average 15.08x and 14.89x parallel speedup with the A-Loop and E-Loop can be gained on one Nvidia Fermi C2050 GPGPU device, compared to the CPU serial execution.

Furthermore, with the motivation of extending the space scalability of GPU simulation, we present a Multi-GPU simulation design for information propagation over complex networks. To obtain high performance simulation on multiple GPU devices, an efficient network partitioning and replication method is important. An efficient data synchronization scheme is proposed to allow the vertex-oriented processing and edge-oriented processing algorithms to execute on multiple GPU devices concurrently. The design aims to balance the workload in each GPU device and to reduce the data communication and synchronization cost. The simulation performance is comprehensively evaluated in large scalable networks. According to the simulation performance results on multiple GPU devices, we observe that there are four factors that influence the performance: (1) the number of simulation steps, (2) computation time, (3) synchronization time, and (4) data transfer time. These factors depend on the network structure and the replication ratio, a measure of the average node replication. Although

the Multi-GPU system introduces an overhead, this is acceptable if it is more important to enhance the system scalability for very large network simulations than to obtain a performance gain.

Finally, a case study on viral advertisement diffusion over the LiveJournal online social network is presented to illustrate how to apply our HPC information propagation simulation strategies over a real world complex social network. Both the E-Loop and the A-Loop are used for the advertisement diffusion simulations with or without subsequent attempts of activation. Simulations are conducted on CPU and multiple GPU devices with varying threshold values. The simulation performance is investigated in detail. According to the experimental results, we can choose the optimal simulation strategy to simulate the diffusion behaviors influenced by different initial nodes selection policies. We observe that the largest degree selection policy and a large number of initial selected nodes can help in maximizing the spread only in the beginning and early stages of advertisement diffusion. The influence of a good node selection policy in advertisement diffusion is not that crucial compared to reducing the threshold.

According to the simulation results, we further discuss improved viral advertising strategies. To propagate an advertisement or a product efficiently, a company could first invest on mass marketing to reduce the threshold (increase the willingness) and then deploy a proper viral marketing campaign in order to facilitate the spread of the advertisement. Such a conclusion by simulation based studies presents the phenomenon that exists in real world cases in marketing.

8.2 Future Work

8.2.1 Hybrid Multi-GPU Processing Scheme

In future work, we can further optimize the execution performance scheme on a Multi-GPU system. As discussed in Chapter 6, in the beginning and end of

vertex-oriented processing on multiple GPU devices, the cost of data transfer and synchronization may become a bottleneck that restricts the overall execution performance. Assuming the complete network can be stored on one GPU device, we can design a hybrid Multi-GPU processing scheme, as shown in Figure 8.1.

Figure 8.1. Hybrid Multi-GPU Processing Scheme

In the initialization stage, we can store two copies of a network structure in the Multi-GPU system. One copy (Copy 1) is stored in GPU 0 as a complete network structure. The other copy (Copy 2) is partitioned into three partitions and stored in GPU 1, 2, and 3 respectively.

In the beginning of the simulation, we first process the network simulation on GPU 0, described as Phase 1. When the computation requirement increases significantly due to many state transitions, we can change to Phase 2. In Phase 2, we need to copy the state results of the nodes from Copy 1 to different partitions in Copy 2. The Multi-GPU simulation scheme in Phase 2 uses vertex-oriented processing on multiple GPU devices as described in Chapter 6. In the end of the simulation, we change the Multi-GPU scheme back to the single GPU scheme, described as Phase 3. To do this, we copy the state results

from Copy 2 back to Copy 1 and process the simulation on GPU 0.

Thus, we can avoid the expensive overhead of data transfer and synchronization in the beginning and end of the simulation, thus improving the execution performance.

8.2.2 Dynamic Network Structures on Multi-GPU

In future work, we can also consider how to handle a dynamic network structure during simulation. This is required as some complex social networks have a dynamic contact network, where new relationships between individuals can be created and existing relationships can be destroyed.

Directly adding or removing graph elements is not feasible on the current graph layouts since the underlying data structures are all arrays in GPU devices. This is because the GPU only support basic arrays as the data structure for efficient coalesced memory access. It is difficult to change an array-based network structure during runtime. We therefore need to design a practical and efficient mechanism to address the simulation of dynamic complex networks.

For small-size networks, we can allocate a relationship matrix with N^2 elements (where N denotes the number of nodes) to indicate the connection relationships between different nodes. However, for those networks with a large number of nodes, such a relationship matrix is too large to be built. Alternatively, to provide a dynamic network structure for large-scale networks, we can pre-allocate *Reserved Edge Array* elements according to a specific pre-allocate policy (e.g. we can reserve 20% extra edge elements of the existing number of edges per node).

As shown in Figure 8.2, we can define a Boolean Index to indicate whether an edge is enabled. We can then change the network structure by enabling/disabling the Boolean Index during simulation.

Figure 8.2. Network Dynamics by Boolean Index

In order to reduce the complexity of adding/deleting edges, the edge-oriented structure is a better layout to store and process the network rather than the vertex-oriented structure. It is because only one edge between connected nodes exists in the edge-oriented structure. If a network is stored with the vertex-oriented structure, extra cost is needed to maintain a consistency check for the state of bidirectional edges.

In practice, we can sort the vertices by descending number of out-going edges before copying the network structure from CPU main memory to GPU memory. According to the sorted vertices, we can pre-allocate a number of *Reserved Edge* elements for each vertex in a descending order, as shown in Figure 8.2. The *Edge Array* which includes *Reserved* and the *Existing Edge* elements can then be transferred to GPU memory for simulation.

According to the dynamically changing network, we can adjust the Boolean Index by marking the edge elements to be zero (disabled) or one (enabled) in order to obtain the relationship dynamics between vertices. Such a changing operation can be performed at the end of each simulation step or after a number of simulation steps. Adding elements is more difficult than deleting since we have to provide more information to build the connection between the specified vertices.

8.3 Summary

In this chapter, we present the conclusions of this research and discuss future work. First, we summarize the research outcomes we have achieved by using HPC techniques in simulations of information propagation over complex networks. Secondly, we present two interesting research issues that we can further explore in the future: (1) a hybrid Multi-GPU processing scheme, and (2) a dynamic network changing scheme on Multi-GPU.

Appendix A. Detailed Error Rate Analysis

Appendix A provides a detailed error rate analysis of the serial performance models described in Section 3.2.4. The error rate at each step j is defined as

$$E_R(i, j) = \frac{|C^{Predicted}(i, j) - C^{Measured}(i, j)|}{C^{Measured}(i, j)}$$

The average error rate is defined as

$$\overline{E}_R(i) = \frac{\sum_{j=1}^{Steps} \left(|C^{Predicted}(i, j) - C^{Measured}(i, j)| / C^{Measured}(i, j) \right)}{Steps}$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

The details of $\overline{E}_R(i)$ are illustrated in Figure A.1. Both Figure A.1 (a) and Figure A.1 (b) show a similar phenomenon that the accuracy varies with the change of contact network accessing cost, which is determined by $\bar{E}_{IBP}^{MA}(I)$ and $\bar{E}_{IBP}^{MA}(S)$.

The algorithmic analysis has shown that the fundamental difference of overall execution cost between the I-Loop and S-Loop is the contact list browsing routine. In particular, the I-Loop looks for *Infectious* individuals and then checks their contacts, while the S-Loop does a similar check for *Susceptible* ones. The ratio of contact list browsing workload can be defined as

$$R_W(i) = N_i(i) / N, \quad \begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$$

In order to evaluate the accuracy in detail, we further divide the entire simulation into Intensive Computation (IC), Medium Computation (MC), Low Computation (LC), and No Computation (NC), respectively. The range of each phase is defined as follows:

- $IC: R_W(i) \in [0.5, 1)$
- $MC: R_W(i) \in [0.05, 0.5)$
- $LC: R_W(i) \in (0, 0.05)$
- $NC: R_W(i) = 0$.

NC illustrates the situation that the IBP computations have finished but the simulation is still processing STP computations.

Figure A.1 demonstrates that the proposed performance models have better accuracy at IC and MC, compared with the LC phase.

(a). I-Loop error rate at each step

(b). S-Loop error rate at each step

Figure A.1. Error Rate Analysis of I-Loop and S-Loop

The accuracy depends mainly on the workload of contact list browsing, which accesses the map data structure in main memory. Also, performance prediction works well in the NC phase. This is due to the fact that the NC phase only refers to the STP part of the entire simulation procedure, which has a known cost in array browsing and the amount of random number generation.

By comparing the estimated and measured performance, we demonstrate that the models are accurate enough to represent the performance of both the I-Loop and S-Loop.

Appendix B. Regression Method

Appendix B shows the mathematical relation between $\bar{E}_{IBP}^{RNG}(S)$ as defined by Equation (3.6) and $\bar{N}_I(S) = N_I(S) \cdot D/N$ that is used in Section 3.3. We use a regression method to express this mathematical relation for simplification of the expression for kernel swapping.

Figure B.1 illustrates the ratio of $N_I(S)$ to the total population size, which ranges from 0 to 0.45. The expected maximum number of *Infectious* individuals in any contact list is $0.45 \times D = 90$ (for $D = 200$). Thus, the mathematical relation between $\bar{E}_{IBP}^{RNG}(S)$ and $\bar{N}_I(S)$ could be further explored with the constraint of $\bar{N}_I(S) \in [0, 90]$. $\bar{E}_{IBP}^{RNG}(S)$ and $\bar{N}_I(S) = N_I(S) \cdot D/N$

Figure B.1. Ratio of $N_I(S)$ to Total Population Size

Figure B.1 shows $\bar{E}_{IBP}^{RNG}(S)$ within the domain of $\bar{N}_I(S) \in [0, 90]$, compared with two possible fitting curves which represent mathematical regression functions.

Figure B.2. Comparison of Regression Functions

These two curve fitting functions are suggested by an online mathematical regression tool [131]. The details of the regression function are illustrated in Table B.1, where X is the sample size of infectious individuals in the contact list and Y is the amount of RNGs of the S-Loop.

Table B.1. Comparison of Regression Methods

	Linear Regression	Power Regression
Regression Function	$Y = 0.65 \cdot X + 4.98$	$Y = 1.23X^{0.88}$
Residual Sum of Squares	347.99	321.19

Basically, both linear and power functions can satisfy the curve fitting and data regression requirements with low Residual Sum of Squares (RSS). Furthermore, as shown in Table B.1, the RSS of the linear and power regressions indicates that the efficiency of these two regression methods can be considered roughly the same. In order to simplify the following expression, the linear function is chosen to represent the mathematical relation between $\bar{E}_{IBP}^{RNG}(S)$ and $\bar{N}_I(S) = N_I(S) \cdot D/N$.

Appendix C. Serial Performance Models including CCA Overhead

Appendix C shows the serial performance models including CCA overhead are used in Section 3.4.

We presented the performance models for a serial execution environment in section 3.2. In this appendix, we discuss the validation of the new performance models for estimation of the I-Loop, S-Loop, as well as the A-Loop.

Overhead of CCA Framework

Using runtime composition, the executable program is constructed by dynamic composition of components, based on *Ccaffeine* in our design. As shown in Figure 3.6, the runtime compositional framework coordinates the adaptation for optimal execution resources at each simulation step. There is an overhead of coordination in the framework. We denote it as CCA overhead, expressed by O_{CCA} . Thus, the total time of each simulation step should be the sum of the time taken in IBP, STP, and O_{CCA} .

Moreover, in order to calculate the cost of O_{CCA} , we set up another timer to calculate the time consumed by the entire simulation (from simulation start to end). By subtracting the sum of time cost of ‘*run_next_itr ()*’ at each simulation step from the entire simulation time, we can calculate the overall cost of CCA overhead, which is around 0.1% of the overall serial execution time. Therefore, we can calculate the average O_{CCA} at each step. That is, $O_{CCA} \approx 466 \mu sec$.

Performance Models with CCA Overhead in a Serial Execution

Environment

According to the discussion on the CCA overhead, the serial performance models of the I-Loop and S-Loop at each step j are expressed as follows:

$$C^{Serial}(i, j) = [\bar{E}_{STP}^{RNG}(i) \cdot C_{RNG} + 2N \cdot C_{Ba}^{MA} + N_i(i) \cdot \bar{E}_{IBP}^{RNG}(i) \cdot C_{RNG} + N_i(i) \cdot D \cdot C_{Bm}^{MA} + O_{CCA}]_j \quad (C.1)$$

The total cost of serial execution of both the I-loop and S-Loop can be expressed as follows:

$$T^{Serial}(i) = \sum_{j=1}^{Steps} C^{Serial}(i, j) \quad (C.2)$$

where $\begin{cases} i = I, I \text{ Loop} \\ i = S, S \text{ Loop} \end{cases}$

Kernel Adaptation Model in a Serial Execution Environment with CCA Overhead

As discussed in Section 4.3, the optimized kernel adaptation strategy is obtained by solving the following equation:

$$C^{Serial}(I, j) = C^{Serial}(S, j) \quad (C.3)$$

Since both $C^{Serial}(I, j)$ and $C^{Serial}(S, j)$ in Equation (C.3) have the same O_{CCA} term, it can be canceled from both sides. Thus, in a serial execution environment, the kernel adaptation strategy can be expressed in the same way as the expression with no CCA overhead,. As discussed in Section 3.3, the A-Loop follows the I-Loop from the beginning to step 27, and then adapts to the S-Loop from step 28 until the end.

Validation of Performance Models

In order to evaluate the accuracy of the performance models, criteria of the error rate of

total execution cost, $E_R(i)$, and the average step error rate of the execution cost, $\overline{E_R}(i)$, are used as defined in Formula (3.9) and Formula (3.10).

As shown by the error rate in Table C.1, the serial performance models of the I-Loop, S-Loop, as well as the A-Loop estimate the runtime execution cost at each simulation step with very low error rate, thus showing their validity.

Table C.1. Error Rate of Serial Performance Models with CCA Overhead

	I-Loop	S-Loop	A-Loop
$E_R(i)$	1.24%	2.90%	5.76%
$\overline{E_R}(i)$	6.54%	6.10%	8.28%

Appendix D. Optimized Kernel Adaptation Strategy on Multi-core Architectures

On multi-core architectures, we can also take kernel swapping from the I-Loop to S-Loop as an example. We can have the condition $C^{Parallel}(I, j) > C^{Parallel}(S, j)$. According to Equations (3.2), (3.5), (3.17) and (3.20), $C^{Parallel}(I, j)$ at each step j can be expressed as follows:

$$\begin{aligned} C^{Parallel}(I, j) = & [N_E(I) + N_I(I)] \cdot C_{RNG} + N_I(I) \cdot N_S(I) \cdot \frac{D}{N} \cdot C_{RNG} + O_{CCA} \\ & + \frac{2N \cdot C_{Ba}^{MA} + N_I(I) \cdot D \cdot C_{Bm}^{MA}}{P} + O_P \cdot P \cdot [N_E(I) + N_I(I)] \cdot C_{RNG} \\ & + O_P \cdot P \cdot [N_I(I) \cdot N_S(I) \cdot \frac{D}{N} \cdot C_{RNG} + O_{CCA}] \\ & + O_P \cdot P \cdot [2N \cdot C_{Ba}^{MA} + N_I(I) \cdot D \cdot C_{Bm}^{MA}] \end{aligned}$$

According to Equations (3.2), (3.17), and (3.20) and the mathematical regression shown in Appendix B, $C^{Parallel}(S, j)$ at each step j can be expressed as follows

$$\begin{aligned} C^{Parallel}(S, j) = & [N_E(S) + N_I(S)] \cdot C_{RNG} + N_S(S) \cdot [0.65 \cdot N_I(S) \cdot \frac{D}{N} + 4.98] \cdot C_{RNG} + O_{CCA} \\ & + \frac{2N \cdot C_{Ba}^{MA} + N_S(S) \cdot D \cdot C_{Bm}^{MA}}{P} + O_P \cdot P \cdot [N_E(S) + N_I(S)] \cdot C_{RNG} \\ & + O_P \cdot P \cdot \left\{ N_S(S) \cdot \left[0.65 \cdot N_I(S) \cdot \frac{D}{N} + 4.98 \right] \cdot C_{RNG} + O_{CCA} \right\} \\ & + O_P \cdot P \cdot [2N \cdot C_{Ba}^{MA} + N_S(S) \cdot D \cdot C_{Bm}^{MA}] \end{aligned}$$

where $\begin{cases} i = I, I \text{ Loop State} \\ i = S, S \text{ Loop State} \end{cases}$

As the state results of the I-Loop and the S-Loop are assumed to be the same, the inequality $C^{Parallel}(I, j) > C^{Parallel}(S, j)$ can be further simplified to

$$N_I(I) \cdot N_S(I) \cdot \frac{D}{N} \cdot C_{RNG} + \frac{N_I(I) \cdot D \cdot C_{Bm}^{MA}}{P} + O_P \cdot P \cdot [N_I(I) \cdot N_S(I) \cdot \frac{D}{N} \cdot C_{RNG}] + O_P \cdot P \cdot [N_I(I) \cdot D \cdot C_{Bm}^{MA}]$$

>

$$N_S(S) \cdot [0.65 \cdot N_I(S) \cdot \frac{D}{N} + 4.98] \cdot C_{RNG} + \frac{N_S(S) \cdot D \cdot C_{Bm}^{MA}}{P} + O_P \cdot P \cdot \left\{ N_S(S) \cdot \left[0.65 \cdot N_I(S) \cdot \frac{D}{N} + 4.98 \right] \cdot C_{RNG} \right\} + O_P \cdot P \cdot [N_S(S) \cdot D \cdot C_{Bm}^{MA}]$$

Thus, we can have

$$N_I(i) > \frac{N_S(i) \cdot [4.98 \cdot C_{RNG} \cdot P \cdot (1 + P \cdot O_P) + D \cdot C_{Bm}^{MA} \cdot (1 + P^2 \cdot O_P)]}{0.35 \cdot P \cdot N_S(i) \cdot \frac{D}{N} \cdot C_{RNG} \cdot (1 + P \cdot O_P) + D \cdot C_{Bm}^{MA} \cdot (1 + P^2 \cdot O_P)}$$

where $\begin{cases} i = I, I \text{ Loop State} \\ i = S, S \text{ Loop State} \end{cases}$

Author's Publications

- [1] **J. Jin**, S. J. Turner, B. Lee, S. Kuo, R. S. M. Goh, T. Hung, “Performance Modeling for Runtime Kernel Adaptation: A Case Study on Infectious Disease Simulation”. In *Proceedings of the 11st International Conference on Grid Computing*, pp. 349-358 October 2010.
- [2] **J. Jin**, S. J. Turner, B. Lee, J. Zhong, B. He, “HPC Simulations of Information Propagation over Social Networks”, In *Procedia Computer Science* 9, pp. 292-301, 2012.
- [3] **J. Jin**, S. J. Turner, B. Lee, J. Zhong, B. He, “Simulation Studies of Viral Advertisement Diffusion on Multi-GPU”, In *Proceedings of 2013 Winter Simulation Conference*, pp. 1592-1603, 2013.
- [4] **J. Jin**, S. J. Turner, B. Lee, J. Zhong, B. He, “Simulation of Information Propagation over Complex Networks: Performance Studies on Multi-GPU”, In *Proceedings of 2013 IEEE DS-RT Conference*, pp. 179-188, 2013.

Bibliography

- [1] H. Abbey, “An Examination of the Reed Frost theory of Epidemics”, *Human Biology*, 24, pp. 201-233, 1952.
- [2] Active Harmony, <http://www.dyninst.org/harmony>, 2010.
- [3] T. T. Ahonen, R. Kasper, S. Melkko, “3G Marketing: Communities and Strategic Partnerships”, John Wiley & Sons, 2004.
- [4] B. A. Allan, R. Armstrong, “The Ccaffeine Framework: Composing and Debugging Applications Interactively and Running Them Statically”, *Compframe* 2005, 2005.
- [5] Amdahl's Law, http://en.wikipedia.org/wiki/Amdahl's_law, 2010.
- [6] A. Anagnostopoulos, G. Brova, E. Terzi, “Peer and Authority Pressure in Information-Propagation Models”, *Machine Learning Discovery in Databases*, LNCS 2011.
- [7] R. M. Anderson, R. M. May “Infectious Diseases of Humans Dynamics and Control”, Oxford University Press, Oxford, 1991.
- [8] A. Apolloni, K. Channakeshava, L. Durbeck, M. Khan, C. Kuhlman, B. Lewis, and S. Swarup, “Proceedings of The IEEE International Symposium on Social Computing Applications”, pp. 675-682, August, 2009.
- [9] Automatically Tuned Linear Algebra Software, <http://www.netlib.org/atlas/>, 2010.
- [10] D. A. Bader, K. Madduri, “GTgraph: A Synthetic Graph Generator Suite”, 2006.
- [11] N. T. J. Bailey, “The Mathematical Theory of Infectious Diseases and Its Applications”, Hafner Press, New York, 1975.
- [12] D. Bailey, J. Chame, C. Chen, J. Dongarra, M. Hall, J. K. Hollingsworth, P. Hovland, S. Moore, K. Seymour, J. Shin, A. Tiwari, S. Williams, H. You. “PERI

- Auto-Tuning”, *Journal of Physics: Conference Series* 125 (2008), November 2008.
- [13] M. Bampo, M. T. Ewing, D. R. Mather, D. Stewart, M. Wallace, “The Effects of the Social Structure of Digital Networks on Viral Marketing Performance”, *Information System Research*, Vol. 19 (3), pp. 273-290, 2008.
 - [14] A. L. Barabasi, R. Albert, “Emergence of Scaling in Random Networks”, *Science* 286 (5439), 509-512, October 1999.
 - [15] A. Barabasi, E. Bonabeau, “Scale-Free Network”, *Scientific American* (05, 2003), pp. 50-59, 2003.
 - [16] C. L. Barrett, K. R. Bisset, S. Eubank, X. Feng, M. V. Marathe, “EpiSimdemics: An Efficient Algorithm for Simulating the Spread of Infectious Disease over Large Realistic Social Networks”, *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC 08)*, pp. 37-48, 2008.
 - [17] C. L. Barrett, S. Eubank, M. V. Marathe, “An Interaction-Based Approach to Computational Epidemiology”, in *Proceedings of the Annual Conference of AAAI*, 2008.
 - [18] F. Bass, “A New Product Growth for Model Consumer Durables”, *Management Science*, Vol. 15, pp. 215-227, 1969.
 - [19] K. R. Bisset, A. M. Aji, M. V. Marathe, W. Feng, “High Performance Biocomputing for Simulating the Spread of Contagion over Large Contact Networks”, *BMC Genomics*, 2012.
 - [20] K. R. Bisset, J. Chen, X. Feng, V. S. A. Kumar, M. V. Marathe, “EpiFast: A Fast Algorithm for Large Scale Realistic Epidemic Simulations on Distributed Memory Systems”, *Proceedings of the International Conference on Supercomputing (ICS 09)*, pp. 430-439, 2009.
 - [21] O. N. Bjørnstad, B. F. Finkenstädt, Bryan T. Grenfell. “Dynamics of Measles Epidemics: Estimating Scaling of Transmission Rates Using a Time Series SIR Model”, *Ecological Monographs*: 72 (2), pp. 169-184, 2002.

- [22] A. T. Campbell, G. Coulson, and M. E. Kounavis, “Managing complexity: Middleware explained,” *IT Professional*, IEEE Computer Society, pp. 22–28, September/October 1999.
- [23] S. Carr, K. Kennedy. “Improving the Ratio of Memory Operations to Floating Point Operations in Loops”, *ACM Transactions on Programming Languages and System*, 16 (6), pp. 1768-1810, November 1994.
- [24] CCA specifications, <http://www.cca-forum.org/specification/>, 2010.
- [25] Centrality, <http://en.wikipedia.org/wiki/Centrality>, 2012.
- [26] Complex Network, http://en.wikipedia.org/wiki/Complex_network, 2012.
- [27] D. Chakrabarti, Y. Zhan, C. Faloutsos, “R-MAT: A Recursive Model for Graph Mining”, In *Proceedings of the fourth SIAM International Conference on Data Mining*, 2004.
- [28] F. Chang, V. Karamcheti, “Automatic Configuration and Run-time Adaptation of Distributed Applications”, *Proceedings of the 9th International Symposium on High-Performance Distributed Computing*, pp. 11-20, 2000.
- [29] K. Channakeshava, K. Bisset, A. Vullikanti, M. V. Marathe, S. Yardi, “High Performance Scalable and Expressive Modeling Environment to Study Mobile Malware in Large Dynamic Networks”, *Proceedings of the IEEE International Conference on Parallel & Distributed Processing Symposium (IPDPS 11)*, pp. 770-781, 2011.
- [30] K. Channakeshava, D. Chafekar, K. Bisset, V. S. A. Kumar, M. V. Marathe, “EpiNet: A Simulation Framework to Study the Spread of Malware in Wireless Networks”, *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (SimuTools 09)*, 2009.
- [31] B. Chapman, G. Jost, R. van der Pas, “Using OpenMP: Portable Shared Memory Parallel Programming”, MIT Press (2007), ISBN 0-262-53302-2.

- [32] C. Chen, J. Chame and M. Hall, "CHiLL: A Framework for Composing High-Level Loop Transformations", University of Southern California Computer Science Technical Report, 08-897, <http://www.cs.usc.edu/research/08-897.pdf>.
- [33] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing Adaptive Software in Distributed Systems," Proceedings of the 21st International Conference on Distributed Computing Systems, pp. 635–643, April 2001.
- [34] CHiLL: Composible High-Level Loop Transformation and Parallelization Framework, <http://www.cs.utah.edu/~chunchen/chill/>, 2010.
- [35] R. Cohen, S. Havlin, D. ben-Avraham, "Efficient Immunization Strategies for Computer Networks and Populations", Physical Review Letters, 91, 2003.
- [36] R. Cowan, N. Jonard, "Network Structure and the Diffusion of Knowledge", Journal of Economic Dynamics and Control, Vol. 28 (8), pp. 1557-1575, 2004.
- [37] CUDA, Nvidia "http://www.nvidia.com/object/cuda_home_new.html", 2011.
- [38] CUDA CURAND Library,"<http://developer.nvidia.com/cuRAND>", 2011.
- [39] T. L. Dahlgren, P. T. Devanbu, "Improving Scientific Software Component Quality Through Assertions", Proceedings of the second international workshop on Software engineering for high performance computing system applications, pp. 73-77, 2005.
- [40] S. R. Das. "Adaptive Protocols for Parallel Discrete-Event Simulation", Proceedings of Winter Simulation Conference, pp. 186-193, 1996.
- [41] P. Domingos, M. Richardson, "Mining the Network Value of Customers", Proceedings of 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 57-66, 2001.
- [42] J. Dongarra, R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. McMahon, A. Snavely, J. Vetter, K. Yelick, S. Alam, R. Campbell, L. Carrington, T. Chen, O. Khalili, J. Meredith and M. Tikir, "DARPA's HPCS Program: History, Models, Tools, Languages", Advances in COMPUTERS - High Performance Computing, Volume 72, pp. 1-100, 2008.

- [43] L. Dyagilev, S. Mannor, E. Yom-Tov, “Generative Models for Rapid Information Propagation”, Workshop on Social Media Analytics (SOMA), 2010.
- [44] J. M. Epstein, D. Cummings, S. Chakravarty, R. M. Singa and D. S. Burke, “Toward a Containment Strategy for Smallpox Bioterror An individual-Based Computational Approach”, Brookings Institution Press, 2002.
- [45] A. Eswaradass, X.-H. Sun, M. Wu, “A Neural Network based Predictive Mechanism for Available Bandwidth”, Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium, pp. 33.1, April 2005.
- [46] S. Eubank, H. Guclu, V. S. A. Kumar, M. V. Marathe, A. Srinivasan, Z. Toroczkai, and N. Wang, “Modelling Disease Outbreaks in Realistic Urban Social Networks”, Nature 429, pp. 180-184, 2004.
- [47] S. Eubank, V. S. A. Kumar, M. V. Marathe, A. Srinivasan, N. Wang, “Structural and Algorithmic Aspects of Massive Social Networks”, in Proceedings of 15th ACM-SIAM Symposium on Discrete Algorithms (SODA 2004), pp. 711-720, 2004.
- [48] S. Eubank, J. Smith, “Scalable, Efficient Epidemiological Simulation”, in Proceedings of the 2002 ACM Symposium on Applied Computing (SAC 2002), pp. 139-145, 2002.
- [49] E. Even-Dar, A. Shapira, “A Note on Maximizing the Spread of Influence in Social Networks”, LNCS 4845, pp. 281-286, 2007.
- [50] R. Eward. “Automatic Algorithm Selection for Complex Simulation Problems”, PhD thesis, University of Rostock, 2010.
- [51] Fastest Fourier Transform in the West, <http://www.fftw.org/>, 2010.
- [52] D. G. Feitelson, “Workload Modeling for Performance Evaluation”, in M. C. Calzarossa, and S. Tucci (Eds.), Performance 2002, LNCS 2459, pp. 114-141, 2002.

- [53] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, “GNU Scientific Library Reference Manual - Third Edition”, ISBN 0954612078.
- [54] R. Gani, S. Leach, “Transmission Potential of Smallpox in Contemporary Populations”, *Nature* 414 (6865), pp. 748-751, December 2001.
- [55] A. van Gemund, “Symbolic Performance Modeling of Parallel System”, *IEEE Transactions on Parallel and Distributed Systems*, 14 (2), 2003.
- [56] GNU Scientific Library, <http://www.gnu.org/software/gsl/>, 2010.
- [57] J. Goldenberg, B. Libai, E. Muller, “Talk of the Network: A Complex Systems Look at the Underlying Process of Word-of-Mouth”, *Marketing Letters* 12(3), pp. 211-223, 2001.
- [58] J. Goldenberg, B. Libai, E. Muller, “Using Complex Systems Analysis to Advance Marketing Theory Development: Modeling Heterogeneous Effects on New Product Growth through Stochastic Cellular Automata”, *Academy of Marketing Science Review*, 2001.
- [59] M. Granovetter, “Threshold Models of Collective Behavior”, *American Journal of Sociology* 83(6):1420-1443, 1978.
- [60] Habiba, T. Y. Berger-Wolf, “Maximizing the Extent of Spread in a Dynamic Network”, *DIMACS Technical Report*, 2007.
- [61] W. H. Hamer, “Epidemic Disease in England--the Evidence of Variability and Persistency of Type”, *The Lancet* i, pp. 733-739, 1906.
- [62] T. Helms, R. Ewald, S. Rybacki, A. M. Uhrmacher, “A Generic Adaptive Simulation Algorithm for Component-based Simulation Systems”, *Proceedings of Principles of Advanced Discrete Simulation*, pp. 11-22, 2013.
- [63] H. W. Hethcote, “The Mathematics of Infectious Diseases”, *SIAM Review* 42 (4) , pp. 599-653, 2000.

- [64] O. Hinz, B. Skiera, C. Barrot, J. U. Becker, “Seeding Strategies for Viral Marketing: An Empirical Comparison”, *Journal of Marketing*, Vol. 1 (1), pp. 1-50, 2012.
- [65] T. Howard, “USAToday: Viral Advertising Spreads through Marketing Plans”, *USA Today*, 2005.
- [66] Intel Product Online Reference, <http://ark.intel.com/Product.aspx?id=34446>, 2010.
- [67] R. Iyengar, C. Bulte, T. W. Valente, “Further Reflections on Studying Social Influence in New Product Diffusion”, *Marketing Science*, Vol. 30 (2), pp. 230-232, 2011.
- [68] J. Jin, S. J. Turner, B.-S. Lee, S.-H. Kuo, R. S. M. Goh, T. Hung, “Performance Modeling for Runtime Kernel Adaptation: A Case Study on Infectious Disease Simulation”, *Proceedings of ACM/IEEE Conference on Grid Computing Workshop on Component-based High Performance Computing*, (to appear) 2010.
- [69] P. J. Joseph, K. Vaswani, M. J. Thazhuthaveetil, “A Predictive Performance Model for Superscalar Processors”, *International Symposium on Micro-architecture (MICRO)*, pp. 161-170, 2006.
- [70] T. Karcher, V. Pankratiyus, “Auto-Tuning Multicore Applications at Run-Time with a Cooperative Tuner”, *Proceedings of the 17th International Conference on Parallel Processing (Euro-Par 11)*, pp. 03-14, 2011.
- [71] T. Karcher, C. Schaefer, V. Pankratiyus, “Auto-Tuning Support for Manycore Applications: Perspectives for Operating Systems and Compilers”, *ACM SIGOPS Operating Systems Review*, 43 (2), pp. 96-97, April 2009.
- [72] E. Kelly, “This is One Viral That You Want to Spread”, *Fortune* (2000), pp. 297-300, 2000.
- [73] D. Kempe, J. Kleinberg, É. Tardos, “Maximizing the Spread of Influence through a Social Network”, In *Proceedings of the ninth ACM SIGKDD*, pp. 137-146, 2003.

- [74] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," IEEE Computer, vol. 36, pp. 41–50, January 2003.
- [75] W. O. Kermack and A. G. McKendrick, "A Contribution to the Mathematical Theory of Epidemics", Proceedings of the Royal Society (London) A 115, pp. 700-721, 1927.
- [76] P. Kolter, "From Mass Marketing to Mass Customization", Strategy & Leadership, 17 (5), pp. 10-47, 1989.
- [77] C. J. Kuhlman, V. S. A. Kumar, M. V. Marathe, H. S. Mortveit, S. Swarup, G. Tuli, "A General-Purpose Graph Dynamical System Modeling Framework", Proceedings of the 2011 Winter Simulation Conference (WSC 11), pp. 296-308, 2011.
- [78] G. Kumfert, Understanding the CCA specification using Decaf, Technical Report UCRLMA-145991, Lawrence Livermore National Laboratory, 2003.
<http://www.llnl.gov/CASC/components/docs.html>.
- [79] J. F. Kurose, K. W. Ross, "Computer Networking: A Top-Down Approach Featuring the Internet", Boston, Massachusetts: AddisonWesley, 2001.
- [80] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, S. A. McKee, "Methods of Inference and Learning for Performance Parallel Applications", Symposium on Principles & Practice of Parallel Programming, pp. 249-258, 2007.
- [81] J. Leskovec, L. A. Adamic, B. A. Huberman, "The Dynamics of Viral Marketing", ACM Transactions on the Web, Volume 1 Issue 1, 2007.
- [82] J. Leskovec, A. Krause, C. Guestrin, C. Faloutsos, J. VanBriesen, N. Glance, "Cost-effect Outbreak Detection in Networks", Proceedings of the 13th ACM International Conference on Knowledge Discovery and Data Mining, pp. 420-429, 2007.
- [83] P. E. Lekone, B. F. Finkenstädt, "Statistical Inference in a Stochastic Epidemic SEIR Model with Control Intervention: Ebola as a case study", Biometrics 62, pp. 1170-1177, 2006.

- [84] H. Li, D. Groep, L. Wolters, “Workload Characteristics of a Multi-cluster Supercomputer”, in D. Feitelson, L. Rudolph, and U. Schwiegelshohn (Eds.), JSSSP 2004, LNCS 3277, pp. 176-193, Berlin Heidelberg: Springer-verlag, 2004.
- [85] J. Liu, H. Nishiyama, N. Kato, “A Framework for Information Propagation in Mobile Sensor Networks”, IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS), 2013.
- [86] LiveJournal, “<http://www.livejournal.com>”, 2012.
- [87] LiveJournal Social Network, “snap.stanford.edu/data/soc-livejournal1.html”, 2012.
- [88] Mass Marketing, “http://en.wikipedia.org/wiki/Mass_marketing”, 2012.
- [89] M. Matsumoto and T. Nishimura, Dynamic Creation of Pseudorandom Number Generators, Monte Carlo and Quasi-Monte Carlo Methods 1998, pp 56-69, Springer, 2000.
- [90] K. S. McKinley, S. Carr, CW, Tseng, “Improving Data Locality with Loop Transformations”, ACM Transactions on Programming Languages and System, 18 (4), pp. 424-453, July, 1996.
- [91] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, B. H.C. Cheng, “Composing Adaptive Software”, Computer, 37 (7), pp. 56-64, July 2004.
- [92] Metis, “<http://glaros.dtc.umn.edu/gkhome/views/metis/>”, 2012.
- [93] L. A. Meyers, M. E. J. Newman, M. Martin, and S. Schrag, “Applying Network Theory to Epidemics: Control Measures for Mycoplasma Pneumoniae Outbreaks”, Emerging Infectious Diseases 9, no. 2 (204), 2003.
- [94] L. A. Meyers, M. E. J. Newman, and B. Pourbohloul, “Predicting Epidemics on Directed Contact Networks”, Journal of Theoretical Biology 240, pp. 400-418, 2006.
- [95] L. A. Meyers, B. Pourbohloul, M. E. J. Newman, D. M. Skowronski, R. C. Brunham, “Network theory and SARS: Predicting Outbreak Diversity”, Journal of Theoretical Biology 232, pp. 71-81, 2004.

- [96] Microsoft Corporation, “COM: Delivering on the Promises of Component Technology”, <http://www.microsoft.com/com/default.asp>, 2010.
- [97] Microsoft Corporation, Microsoft COM Technologies - DCOM, 2010.
- [98] Microsoft Corporation, .NET, <http://www.microsoft.com/net/>, 2010.
- [99] Moore’s Law, http://en.wikipedia.org/wiki/Moore's_law/, 2010.
- [100] MPI, http://en.wikipedia.org/wiki/Message_Passing_Interface, 2013.
- [101] M. E. J. Newman, “Spread of Epidemic Disease on Networks”, *Physical Review E* 66, pp. 016128, 2002.
- [102] M. E. J. Newman, “The Structure and Function of Complex Networks”, *SIAM Review* 45, pp. 167-256, 2003.
- [103] D. M. Nicol, P. F. Reynolds, “Optimal Dynamic Remapping of Data Parallel Computation”, *IEEE Transactions on Computers* 39 (2), pp. 206-219, 1990.
- [104] G. R. Nudd, D. J. Kerbyson, E. Papaefstathiou, S. C. Perry, J. S. Harper, D. V. Wilcox, “PACE—A Toolset for the Performance Prediction of Parallel and Distributed Systems”, *The International Journal of High Performance Computing Applications*, 14 (3), pp. 228-251, 2000.
- [105] Nvidia Fermi, “NVIDIA’s Next CUDA Compute Architecture: Fermi”, Nvidia Whiterpaper, 2009.
- [106] Object Management Group, Framingham, Massachusetts, The Common Object Request Broker: Architecture and Specification Version 3.0, July 2003.
- [107] OpenMP Library, “<http://openmp.org/wp/>”, 2011.
- [108] Optimized Sparse Kernel Interface, <http://bebop.cs.berkeley.edu/oski/>, 2010.
- [109] R. Pastor-Satorras, A. Vespignani, “Immunization of Complex Networks”, *Physical Review E*, 65, 2002.
- [110] Performance Application Programming Interface, <http://icl.cs.utk.edu/papi/>, 2010.
- [111] V. Podlozhnyuk, Parallel Mersenne Twister,

- http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf, 2010.
- [112] R. Quax, D. A. Bader, P. M. A. Slood, “SEECN: Simulating Complex System using Dynamic Complex Networks”, *International Journal for Multiscale Computational Engineering*, Vol. 9(2), 2011.
 - [113] H. Rahmandad, J. Sterman, “Heterogeneity and Network Structure in the Dynamics of Diffusion: Comparing Agent-Based and Differential Equation Models”, *Management Science* vol. 54, No.5, pp. 998-1014, May 2008.
 - [114] J. Rayport, “The Virus of Marketing”, Fast Company, 1996.
 - [115] M. Richardson, P. Domingos, “Mining Knowledge-Sharing Sites for Viral Marketing”, *Proceedings of 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 61-70, 2002.
 - [116] T. M. Riechers, S. Kuo, R. S. M. Goh, T. Hung, “Component-based Design for Adaptive Large-scale Infectious Disease Simulation”, *International Conference on Computational Science, Procedia Computer Science*, Volume 1, Issue 1, May 2010, pp. 535-544, 2010.
 - [117] T. M. Riechers, S. Kuo, R. S. M. Goh, H. Soh, T. Hung, A. M. Malik, “A Case Study on Dynamic Kernel Adaptation in a Component-based Infectious Disease Simulator”, *Workshop on Component-Based High Performance Computing, Supercomputing Conference*, pp. 1-4, 2009.
 - [118] E. M. Rogers, “Diffusion of Innovations”, 5th Edition Free Press, New York, 2003. (1st Edition in 1962)
 - [119] Scale-free network, “http://en.wikipedia.org/wiki/Scale-free_network”, 2011.
 - [120] J. P. Scott, “Social Network Analysis: A Handbook (2nd edition)”, Thousand Oaks, CA: Sage Publications, 2000.
 - [121] S. Shende and A. D. Malony, “The TAU Parallel Performance System, *International Journal of High Performance Computing Applications*”, Volume 20, Number 2, pp. 287-311 Summer 2006.

- [122] A. Snavey, L. Carrington, N. Wolter, J. Labarta, R. Badia, A. Purkayastha, “A Framework for Performance Modeling and Prediction”, Proceedings of ACM/IEEE Conference in Supercomputing, pp. 21-37, 2002.
- [123] B. Song, C. Ernemann, R. Yahyapour, “Parallel Computer Workload Modeling with Markov Chains”, in D. Feitelson, L. Rudolph and U. Schwiegelshohn (Eds.), JSSPP 2004, LNCS 3277, 47-62, Berlin Heidelberg: Springer-Verlag, 2004.
- [124] SPIRAL Software/Hardware Generation for DSP Algorithms, <http://www.spiral.net/>, 2010.
- [125] S. H. Strogatz, “Exploring Complex Networks”, Nature 410, pp. 268-276, 2001.
- [126] S. H. Strogatz, D. J. Watts, “Collective Dynamics of ‘Small-World’ Networks”, Nature 393, pp. 440-442, 1998.
- [127] Sun Microsystems, <http://java.sun.com/products/ejb/>, Enterprise JavaBeans Technology, 2010.
- [128] C. Szyperski, “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley, 1999.
- [129] C. Țăpuș, I.-H. Chung, J. K. Hollingsworth. “Active Harmony: Towards Automated Performance Tuning”, Proceedings of SuperComputing, pp. 1-11, November 2002.
- [130] The Common Component Architecture, <http://www.cca-forum.org/>, 2010.
- [131] The Online Regression Tools, <http://www.xuru.org/rt/TOC.asp>, 2010.
- [132] The OpenMP API Specification for Parallel Programming, <http://openmp.org/wp/>, 2010.
- [133] The PHiPAC (Portable High Performance ANSI C) Page for BLAS3 Compatible Fast Matrix Matrix Multiply, <http://www.icsi.berkeley.edu/~bilmes/hipac/>, 2010.
- [134] N. Trebon, A. Morris, J. Ray, S. Shende, A. D. Malony, “Performance Modeling using Component Assemblies”, Concurrency and Computation: Practice and Experience 19(5), pp. 685-696, 2007.

- [135] N. Venkatasubramanian, "Safe Composability of Middleware Services", *Communications of the ACM*, vol. 45, pp. 49-52, June 2002.
- [136] Viral Marketing, "http://en.wikipedia.org/wiki/Viral_marketing", 2012.
- [137] S. Wasserman, K. Faust, "Social Network Analysis: Methods and Applications", Cambridge University Press, pp. 1-27, 1994.
- [138] D. J. Watts, J. Peretti, "Viral Marketing for the Real World", *Harvard Business Review*, 2007.
- [139] R. C. Whaley, A. Petitet, J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project", *Parallel Computing*, 27 (1), pp. 3-25, 2001.
- [140] M. E. Wolf, M. S. Lam, "A Data Locality Optimizing Algorithm", *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pp. 30-44, 1991.
- [141] J. Wu, B. Hu, Y. Zhang, "Maximizing the Performance of Advertisements Diffusion: A Simulation Study of the Dynamics of Viral Advertising in Social Networks", *Simulation: Transactions of the Society for Modeling and Simulation International*, Submitted.
- [142] L. T. Yang, X. Ma, F. Mueller, "Cross-Platform Performance Prediction of Parallel Applications using Partial Execution", *Proceedings of ACM/IEEE Conference on Supercomputing*, pp. 40-50, 2005.
- [143] J. Zhong, B. He, "An Overview of Medusa: Simplified Graph Processing on GPUs", In *Proceedings of ACM SIGPLAN symposium on Principles and Practice of Parallel Programming 2012*, pp. 283-284, 2012.