5-2004

Accelerating Exact Stochastic Simulation

James Michael McCollum
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by James Michael McCollum entitled "Accelerating Exact Stochastic Simulation." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Electrical Engineering.

Gregory D. Peterson, Major Professor

We have read this thesis and recommend its acceptance:

Donald W. Bouldin, Chirs D. Cox

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by James Michael McCollum entitled "Accelerating Exact Stochastic Simulation." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree Master of Science, with a major in Electrical Engineering.

Gregory D. Peterson
_____
Major Professor

We have read this thesis
and recommend its acceptance:

Donald W. Bouldin
_____

Chris D. Cox
_____

Accepted for the Council:

Anne Mayhew
_____
Vice Chancellor and Dean of Graduate Studies

(Original Signatures are on file with official student records.)

Accelerating Exact Stochastic Simulation

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

James Michael McCollum
May 2004

DEDICATION

This thesis is dedicated to my wife, Sarah, for her love;

my parents, Greg and Michelle, and my brother, Nathan,

for their support, guidance, and encouragement;

and to my grandfather, James,

the first electrical engineer in my family.

ACKNOWLEDGEMENTS

I wish to thank all of the people who helped me complete my Master of Science degree in Electrical Engineering, primarily my advisor, mentor, and friend, Dr. Peterson. I also wish to thank Dr. Cox and Dr. Simpson for introducing me to the field of microbiology and Dr. Bouldin for guiding my work in VLSI design.

I would also like to thank all of the organizations that supported this work, including the National Science Foundation via grants #0075792, #0130843, #0311500, and #9972889, the Defense Advanced Research Projects Agency BioComputation program, the University of Tennessee Center for Environmental Biotechnology, and the University of Tennessee Center for Information Technology Research.

ABSTRACT

Currently, the applicability of computer modeling to whole-cell and multi-cell biochemical models is limited by the accuracy and efficiency of the simulation tools used to model gene regulatory networks. It is widely accepted that exact stochastic simulation algorithms, originally developed by Gillespie and improved by Gibson and Bruck, accurately depict the time-evolution of a spatially homogeneous biochemical model, but these algorithms are often abandoned by modelers because their execution time can be on the order of days to months. Other modeling techniques exist that simulate models much more quickly, such as approximate stochastic simulation and differential equations modeling, but these techniques can be inaccurate for biochemical models with small populations of chemical species. This work analyzes the performance of exact stochastic simulation algorithms by developing software implementations of exact stochastic simulation algorithms and measuring their performance for a wide variety of models. Through this study, several techniques are developed and tested that improve the performance of certain algorithms for specific models. A new algorithm called the *Adaptive Method* is then presented which attempts to select the optimal simulation algorithm for the particular model based on periodic measurements of simulator performance during execution. Other algorithmic changes are proposed to aid in the development of hardware accelerators for exact stochastic simulation. The work serves as another step in the process of making exact stochastic simulation a practical modeling solution for molecular biologists.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Computer modeling has forever changed the field of integrated circuit design. Every analog and digital circuit designed today is first built on a virtual workbench, where a schematic diagram or textual description of the circuit is captured in a software tool and the time evolution of the circuit's output based on varying stimuli is estimated using a program such as SPICE (Simulation Program with Integrated Circuit Emphasis) [1]. Modeling allows the engineer to sweep parameters quickly, optimize the design, analyze tradeoffs, and verify that the design meets specifications. Without these tools, it would be impossible to design circuits that meet today's size, complexity, and performance requirements.

A similar revolution may soon take place in the field of molecular biology. New biochemical analysis techniques like the use of micro-arrays have led to an explosion in the amount of data biologists can gather about the internal state of a cell [2]. Using this information, biologists can begin to analyze and understand gene regulatory networks, the complex interactions between proteins and genes that control cell functions. If accurate computer models of these gene regulatory networks could be developed, biologists could begin working in a virtual laboratory, and thus save time and money.

Unfortunately, several roadblocks prevent the development of gene regulatory network models. One of the most fundamental problems is the tradeoff between efficiency and accuracy in gene regulatory network simulation algorithms. Some modelers have used boolean networks, Bayesian statistics, and differential equations to

model gene regulatory networks [3]. These techniques have fast simulation times (less than a day for biologically relevant problems), but can produce inadequate or inaccurate results because they fail to accurately model the noise of a biochemical process and they are not accurate for models with small species populations. Others modelers have used molecular dynamics models and exact stochastic simulation, which are believed to accurately predict the time-evolution of a biochemical system and properly model noise, but can have execution times on the order of months and years for single whole-cell models [4].

In an attempt to improve the performance of gene regulatory network simulation, this work focuses on analyzing and improving the performance of exact stochastic simulation algorithms, specifically the *First Reaction Method* and *Direct Method* developed by Gillespie [5] and the *Next Reaction Method* developed by Gibson and Bruck [6]. Through this analysis, techniques are discovered that improve the performance of these various algorithms for a specific set of models. An algorithm is then developed called the *Adaptive Method* that combines these enhancements and attempts to select the optimal algorithm for simulating the particular model by monitoring the performance of the simulator as the simulation progresses.

Additionally, a technique called *Fast Propensity Calculation* is suggested for reducing several of the multiplication steps involved in exact stochastic simulation to addition steps. After analyzing the performance of this technique in a software implementation of the algorithm, no significant performance gain is shown, but this work does have implications for creating efficient hardware accelerators for exact stochastic simulation.

The second chapter of this work is intended to provide the reader with a basic introduction to molecular biology and gene regulatory networks. Chapter three introduces existing techniques for modeling gene regulatory networks, focusing in detail on the exact stochastic simulation algorithms developed by Gillespie. The fourth chapter discusses the results of performing rigorous analysis on exact stochastic simulation algorithms by running a large set of input models. The fifth chapter introduces and analyzes the *Adaptive Method,* a new algorithm that matches or outperforms existing stochastic simulation algorithms. The final chapter introduces and analyses the performance of *Fast Propensity Calculation.*

As a whole, this work provides several new insights into exact stochastic simulation of gene regulatory networks and will hopefully serve as another step in the development of a virtual laboratory for molecular biologists.

Chapter 2

Gene Regulatory Networks

The goal of this chapter is to provide the reader with a basic understanding of gene regulatory networks and their importance to the life of a cell. To sufficiently introduce this topic, we must first discuss several concepts fundamental to molecular biology and genetics, such as:

- What are cells?
- What are proteins?
- How do cells use proteins?
- How do cells produce proteins?

After these topics are introduced, a detailed definition of gene regulatory networks is provided along with several illustrative examples. By gaining an appreciation for the importance of gene regulatory networks, we can then begin to discuss the importance of modeling them accurately. For a more detailed introduction to gene regulatory networks and molecular biology, readers are encouraged to consult [7,8,9].

2.1 What are cells?

Cells are the basic building blocks of life. Microorganisms like bacteria, fungi, protozoa, and algae are comprised of a single cell functioning as an individual entity. Multicellular organisms like plants and animals are comprised of millions of cells working together for the common good of the organism.

The simplest organisms, called prokaryotes, consist of a single cell containing one strand of deoxyribonucleic acid (DNA) and a cell wall or membrane that separates the organism from its environment. Through this membrane, the cell constantly interacts with its environment, absorbing nutrients and discarding waste products. DNA is a chain of nucleic acids that contains instructions that control the behavior and reproduction of the cell. DNA will be discussed in more detail later in this chapter.

Organisms with more complex cellular structure are called eukaryotes. Eukaryotes have a cell wall or membrane, one or multiple strands of DNA, and individual structures called organelles that perform specific cell functions. Examples of cell organelles include nuclei which isolate the DNA from the rest of the cell, mitochondria which are used for energy generation, and chloroplasts which are used in photosynthesis (the conversion of light energy into chemical energy).

Yeast and algae are examples of single-celled eukaryotes, while bacteria are single-celled prokaryotes. Multi-cellular organisms like plants and animals are comprised of the more complex eukaryotic cells.

2.2 What are proteins?

To define the term protein, we must first discuss amino acids, the molecules that chain together to form proteins. Amino acids are a group of molecules that are characterized by a central carbon atom (C) connected to a hydrogen atom (H), an amino group (NH_2), a carboxyl group (COOH), and a side chain which differs for each amino acid. Figure 2-1 shows the chemical structure of an amino acid, where R represents the

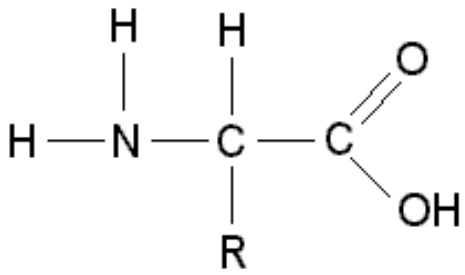

Figure 2-1. Chemical Structure of an Amino Acid.

side chain. The side chain of an amino acid can range in complexity. Alanine, shown in figure 2-2, has a simple side chain consisting of a methyl group (CH_3). Tryosine, shown in figure 2-3, has a more complex side chain, including a phenolic ring.

Proteins are chains of amino acids bonded together by peptide bonds. In a peptide bond, one hydrogen (H) from the amino group (NH_2) of an amino acid "A" bonds with the hydroxyl group (OH) from the carboxyl group (COOH) of another amino acid "B" to form water (H_20). The NH^- ion from amino acid A and the CO^+ ion from amino acid B bond to join the amino acids together. A chain of these amino acids are shown in Figure 2-4.

Typically, when amino acids have bonded together to form proteins, they are referred to as residues. Proteins can consist of as few as 100 residues or as many as 5,000.

2.3 How do cells use proteins?

Many critical biological processes, such as food digestion, movement, and cell reproduction, depend on chemical reactions that would naturally occur too slowly to be useful for sustaining life. For life to be possible, cells must accelerate these reactions by providing catalysts, chemicals that increase the speed of reactions. In cells, proteins called enzymes serve as the catalysts for many of these important biochemical reactions. In addition to catalytic function, proteins provide regulatory function and are building blocks in large multi-protein machines such as flagella (used in cell movement), ribosomes, and ribonucleic acid polymerase (RNAP), which are used in the production of proteins.

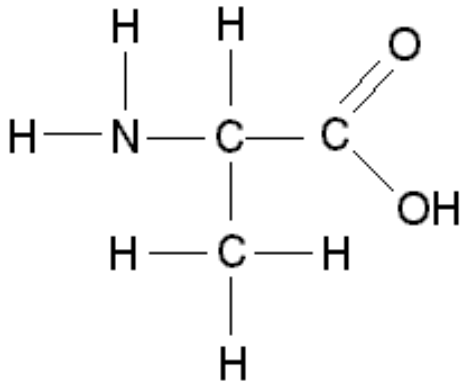

Figure 2-2. Chemical Structure of Alanine [7].

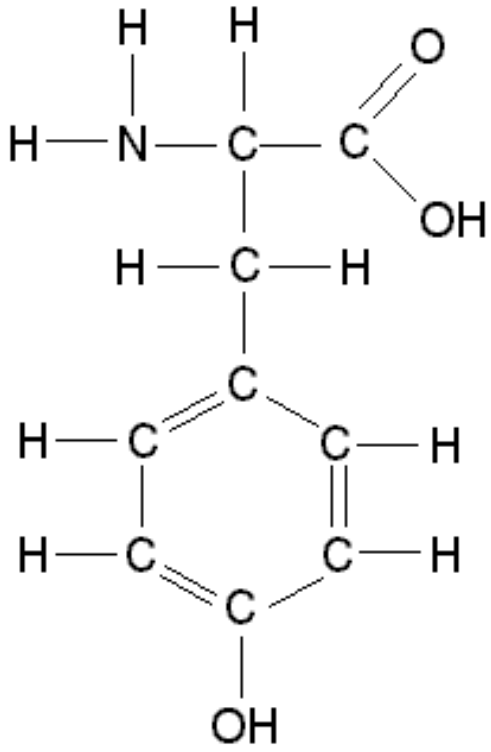

Figure 2-3. Chemical Structure of Tyrosine [7].

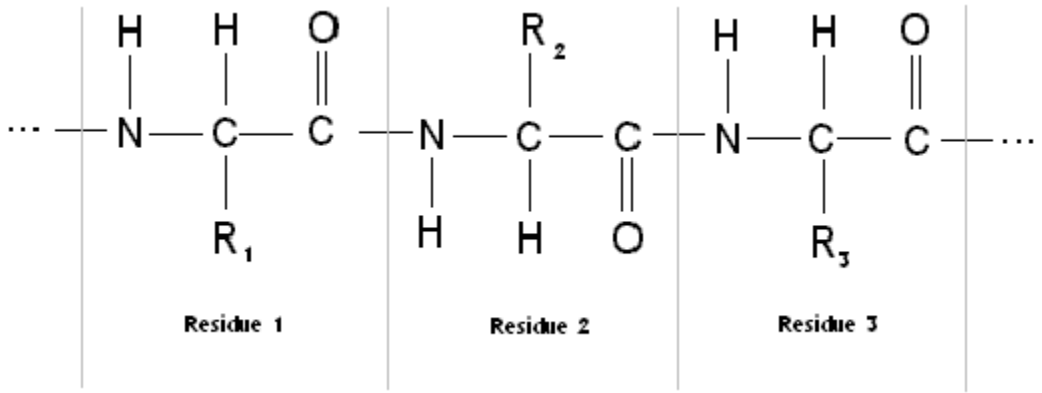

Figure 2-4. Chemical Structure of a Protein [7].

Because proteins are such large molecules, sections of the protein can bond to other sections of the same protein creating a more complex shape. This attribute individualizes proteins limiting the protein to serve as a catalyst for only specific biochemical reactions. Cells take advantage of this attribute and use it to trigger the specific biochemical reactions that the cell needs to execute for a specific situation.

2.4 How do cells produce proteins?

To begin a discussion of protein synthesis, the biochemical process by which proteins are produced, we return to a term introduced earlier, DNA. DNA is a chain of smaller molecules called nucleic acids. There are four nucleic acids in DNA: Adenine (A), Thymine (T), Guanine (G), and Cytosine (C). Nucleic acids are connected together by a "backbone" consisting of deoxyribose (a sugar) and phosphate. A simplified picture of the chemical structure of DNA is shown in figure 2-5, where P represents the phosphate group, D represents deoxyribose, and A, T, G, and C represent nucleic acids. In cells, DNA is double stranded, where complimentary nucleic acids are bonded together; adenine always bonds to thymine and guanine always bonds to cytosine.

During protein synthesis, DNA serves as a recipe or program for creating a specific protein. A series of three consecutive nucleic acids (i.e. GCC, CAT, TTA) on a DNA strand is called a codon. One codon, called the transcription start site, represents the location on the DNA where instructions for producing the protein start and another codon, called the stop codon, represents the location on the DNA where the instructions end. The codons between the transcription start site and the stop codon contain a set of

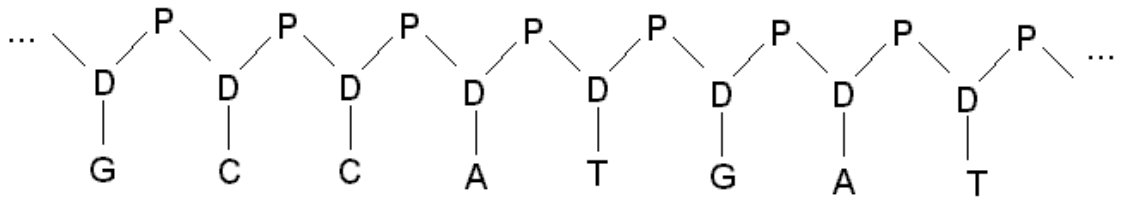

Figure 2-5. Chemical Structure of DNA

codons which code for each amino acid that makes up the protein, for instance AAG or Adenine-Adenine-Guanine code for the amino acid Valine. The entire section of DNA that codes for a complete protein is called a gene. On a single strand of DNA, there can exist hundreds or even thousands of genes.

Protein synthesis is a multi-step process. First, in a process called transcription, a complimentary copy of the genetic code from the DNA is created and stored in a molecule called messenger ribonucleic acid (mRNA). mRNA is composed of the same nucleic acids as DNA, except that Uracil substitutes for Thymine. Transcription is carried out by a multi-protein complex called ribonucleic acid polymerase (RNAP) which sequentially attaches the nucleic acids to the growing mRNA chain. Then, in a process called translation, molecules called "ribosomes" bond to the promoter site on the mRNA, move down the mRNA strand decoding the genetic instructions, and build the protein molecules with the help of another molecule called transfer ribonucleic acid (tRNA). When the ribosome reaches the stop codon, it detaches from the mRNA and protein synthesis is complete.

2.5 What are gene regulatory networks?

Now that we understand that process by which proteins are produced, we can discuss gene regulatory networks, the complex interdependence between genes that allows cells to regulate the production of proteins.

Not all genes on the DNA strand can be transcribed into mRNA at all times. Some genes are inducible, meaning they require the presence of an inducer protein that increases the rate of gene transcription. Other genes are repressible, meaning that a

repressor protein can bond to the DNA at the promoter site and prevent RNAP from binding and beginning the transcription process. By limiting the amount of mRNA that is produced, the amount of protein that can be produced is also limited. The cell can therefore use inducers or repressors to control what and when biochemical reactions are executed. When combinations of these complex interdependencies between inducible and repressible genes perform a specific cellular function, the system is called a gene regulatory network.

2.6 Example

To more easily understand the concept of gene regulatory networks, we will examine a naturally occurring example, bacterial quorum sensing in *Vibrio fischeri.* *Vibrio fischeri* is a species of bacteria that can live as free-living organisms, but are often found in a symbiotic relationship with some marine fish and squid [10]. When *Vibrio fischeri* cell density is high and the bacteria is contained within the light organ of a host organism, the bacteria produces enzymes which trigger reactions that cause the bacteria to luminesce (emit light). When in low concentration or in a free-living state, the bacteria are non-luminescent [10].

The ability for cells to change their behavior based on changes in cell density (number of cells per unit volume) is called “quorum sensing” [11]. Because *Vibrio fischeri* use quorum sensing to regulate luminescence, their quorum sensing can be easily measured by light detectors, thus making them an ideal candidate for studying quorum sensing.

The study of quorum sensing is important because many other bacteria which cause infections also take advantage of this cellular process. For example, *Pseudomonas aeruginosa* is a bacteria that can contaminate surgical wounds, abscesses, burns, ear infections, and the lungs [8]. Patients who have received prolonged treatment with immunosuppressive agents, antibiotics, and radiation are particularly susceptible to infection by *Pseudomonas aeruginosa* [8]. In *Pseudomonas aeruginosa*, quorum sensing is used to regulate the production of toxins, allowing colonies of the bacteria to grow to sufficient size undetected by the immune system of the host [11]. Because several species of bacteria use quorum sensing to regulate toxin production, studying this cell behavior is highly applicable to pharmaceutical research.

A diagram of the quorum sensing gene regulatory network in *Vibrio fischeri* is given in figure 2-6.

The DNA strand that controls quorum sensing contains two genes, luxR and luxI. The set of genes luxCDABEG produce enzymes that cause bioluminescence to occur. Each gene is transcribed to produce mRNA at a basal rate (a nominal rate of production) and each mRNA then produces corresponding proteins LuxR, LuxI, and a set of proteins, which are labeled LuxCDABEG. At the basal rate of transcription, insufficient LuxCDABEG proteins are produced to generate light. The LuxI protein binds with substrates, additional molecules that bind to enzymes, to form N-3-oxo-hexanoyl-L-homoserine lactone (OHHL). OHHL then bonds with the LuxR protein to form a complex, which when bound to the lux box on the DNA, induces production of more luxR, luxI, and luxCDABEG mRNA. By inducing this protein production, a large

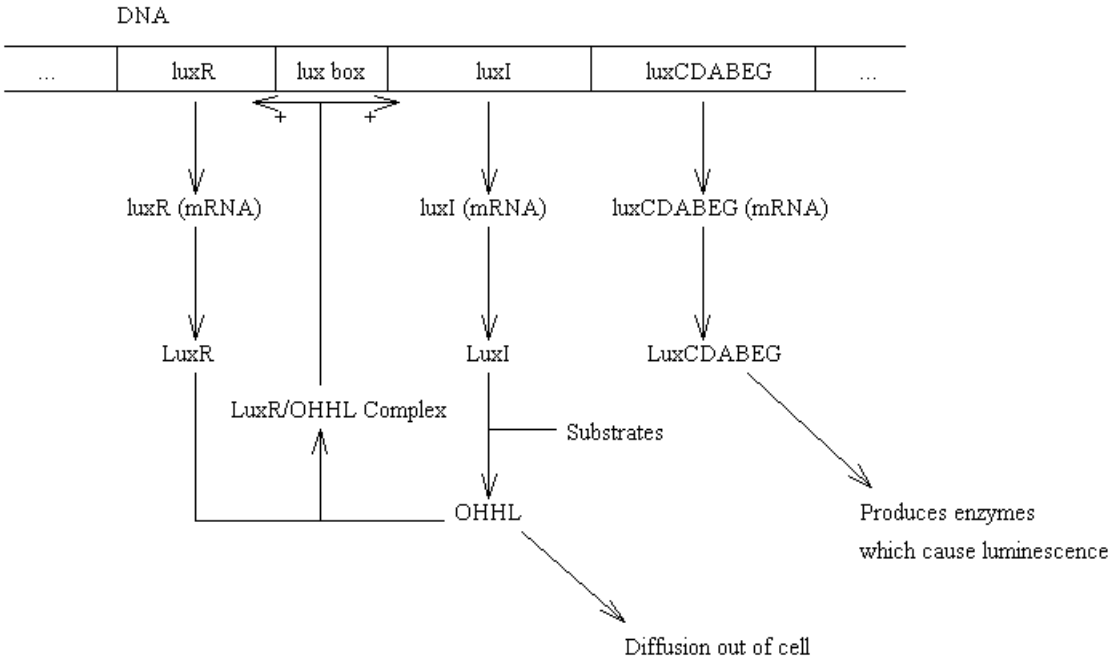

Figure 2-6. Quorum Sensing in *Vibrio fischeri*.

enough concentration of LuxCDABEG is created to trigger other reactions that cause bio-luminescence to occur.

The OHHL protein is referred to as an auto-inducer. This molecule passes freely in and out of the cell membrane. When cell density is low, the cell produces these proteins at a low rate and the likelihood of the OHHL/LuxR complex bonding to the DNA to induce production of the various genes is very low, therefore bio-luminescence does not occur. When cell density is high, all of the cells produce enough auto-inducer within the environment to cause a greater concentration of the auto-inducer to be present in the cell. This greatly increases the likelihood of the LuxR/OHHL complex binding to the DNA and inducing protein production, which then triggers bio-luminescence.

This example illustrates a very simple gene regulatory network that allows the cell to detect high cell density. The gene regulatory network acts as a switch, creating two steady states: one which produces light and another which does not. Almost all cell functions are controlled by similar gene regulatory networks; however, many cell functions can be much more complex and contain multiple interactions between genes, inducing and repressing multiple genes.

2.7 Why model gene regulatory networks?

By understanding how cells operate and communicate, we can begin to develop methods to interrupt and control cellular processes. Understanding cells at a level of detail that is exposed by modeling could lead to a revolution in the development of new medicines, allowing biologists to develop treatments for illnesses ranging from HIV to cancer.

Already, biologists and modelers have had some success using gene regulatory network modeling to understand cellular behavior in new ways. In 1998, John Tyson and his coworkers developed mathematical models for cell mitosis (the process that controls cell-division) in frog eggs [12,13]. The model predicted details of the protein interaction involved in this process that had not yet been discovered. In 2003, these predictions were confirmed in a laboratory [14].

Adam Arkin and his coworkers have used exact stochastic simulation, the same type of simulation analyzed throughout this work, to help show that stochastic variation (or noise) can participate in pathway selection in gene regulatory networks by analyzing and developing models of phage λ-infected *Escherichia coli* cells [15]. Cox et al. later used similar techniques to gain insight on the importance of noise in the quorum sensing gene regulatory network in *Vibrio fischeri* [16].

Drew Endy has also developed models of the phage T7 virus in *Escherichia Coli* [17]. His model "accounts for entry of T7 genome into its host, expression of T7 genes, replication of T7 DNA, assembly of T7 procapsids, and packaging of T7 DNA to finally produce intact T7 progeny" and matches experimental observations made on this virus for the past 30 years [17]. He argues this model is a "useful tool for exploring and understanding the dynamics of cell-growth" and claims that biological simulation can be used to [17]:

1. Find mismatches between the published mechanisms and data.
2. Test multiple treatment strategies before attempting them in a lab.
3. Provide insights from nature for the design of nanoscale biological tools.

2.8 Conclusion

Gene regulatory networks are complex interactions between different biochemicals that control cellular functions. By modeling these gene regulatory networks, we hope to develop innovative techniques for suppressing or eliminating some of the world's most deadly illnesses. The long term goal of this research is to give biologists a software platform where they can conduct experiments quickly and inexpensively in a virtual laboratory.

Chapter 3

Modeling Gene Regulatory Networks

Now that we have introduced the concept of a gene regulatory network and explained the importance of developing gene regulatory network models, we can begin to discuss the algorithms used to simulate gene regulatory network models. This chapter will cover several techniques including molecular dynamics simulation, ordinary differential equations modeling and exact stochastic simulation. This discussion will create a foundation for our analysis of the performance of these techniques provided in the next chapter.

3.1 Molecular Dynamics Simulation

To simulate gene regulatory network models, we need an algorithm that can determine the time evolution of reacting chemical species within a volume. One possible approach is to use a classical molecular dynamics simulation. For more information on this topic, consult [18,19]. In its simplest form, each molecule is treated as a sphere and the simulator tracks each molecule's three-dimensional position and velocity. During each iteration of the algorithm, the kinetic and potential energies of each molecule are determined and the molecule's velocity and position are adjusted. When two molecules collide, the algorithm assesses if a reaction has occurred and changes the structure of the system accordingly. More complex versions of these simulation algorithms account for the structure of each molecule, the rotation of the molecule, the rotation of each of the

atoms that compose a molecule, and other details regarding the quantum mechanics of the system.

Although this technique is very accurate, this solution is similar to the classic N-Body problem in computer science and is computationally complex. Even for just a few molecules, this solution requires too much simulation time to be an effective modeling technique for a biochemical system. This technique also captures unnecessary detail about the chemical system. Although some biologists have attempted to model individual proteins using molecular dynamics, see "protein folding" in [9], most biologists modeling gene regulatory networks tend to only be concerned with the genes that are being expressed/regulated and the overall concentrations of chemical species, not the exact position of each molecule.

3.2 Ordinary Differential Equations Modeling

Another approach to modeling gene regulatory networks is to treat the system as spatially homogeneous, where the position of each individual molecule is irrelevant. For this technique to be accurate, the chemical system must be well-stirred, meaning that the molecules in the mixture are constantly colliding with each other and that the probability of two molecules colliding can be determined by the concentration of the various species in the mixture.

Using a differential equations modeling approach, we convert the concentration of each chemical species to a single-valued continuous function with respect to time [5]. To illustrate this idea, suppose we wished to model the biochemical network given in figure 3-1.

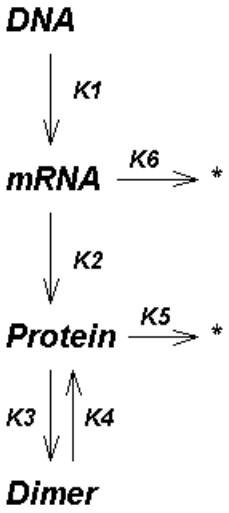

Figure 3-1. Sample Biochemical Network

Here we see a system that includes DNA, which produces mRNA at a rate *K1*. mRNA can decay at a rate *K6* and can produce Protein at a rate *K2*. Protein can decay at a rate of *K5* and can dimerize at a rate *K3*. The reverse dimerization reaction can occur at a rate *K4*. To use a differential equations model, these reactions would be converted into the following set of ordinary differential equations.

$$\frac{dDNA}{dt} = 0$$

$$\frac{dMRNA}{dt} = k_1 DNA - k_6 MRNA$$

$$\frac{dP}{dt} = k_2 MRNA - k_5 P - k_3 P^2 + k_4 Dimer$$

$$\frac{dDimer}{dt} = k_3 P^2 - k_4 Dimer$$

Using standard stiff differential equation solvers, like those provided in Matlab, we can simulate biologically relevant problems in less than a day, but not necessarily accurately. Ordinary differential equations models ignore the inherent random fluctuations of the systems, which is their major limitation when applied to systems with small numbers of molecules. Special care must be taken when selecting the time-step to ensure that the differential equations model does not allow the concentrations of the chemical species to fall below zero, which makes no physical sense in a chemical system. Models with low concentration can yield results where species concentrations can contain real values, i.e. 1.5 molecules of mRNA, when the actual system can only contain integer

values of molecules. This inaccuracy is often present when modeling gene regulatory networks because the population of certain chemical species can be very small, but can have a large effect on the time-evolution of the system

3.3 Gillespie's *First Reaction Method*

Another approach that resolves this accuracy problem was proposed by Daniel T. Gillespie in the mid 1970s. His algorithm, called the *First Reaction Method*, treats the systems as a stochastic process with discrete variables that represent the populations, not concentrations, of chemical species [5,6]. This algorithm is given in figure 3-2.

To understand this algorithm fully, we must first define the term propensity used in step 5. Consider the following chemical reaction:

$$A + B \rightarrow C \qquad (1)$$

The probability that the reaction given in equation 1 occurs, or the probability that a given molecule A reacts with a given molecule B, in a small time dt is

$$P_1 = a_1 dt + o(dt) \qquad (2)$$

As dt approaches zero, the propensity term a_1 dominates equation 2. The propensity may be a function of volume, temperature, concentration, etc. In step 4 of the *First Reaction Method* we calculate the propensity of the reaction based on the stochastic rate

1. Initialize a list of n chemical species and their initial numbers of molecules $X_1, X_2, \ldots, X_n$.
2. Initialize a list of m chemical reactions and their associated stochastic rate constants $k_1, k_2, \ldots, k_m$.
3. Initialize the current time $t \leftarrow 0$.
4. Calculate the propensity, $a_1, a_2, \ldots, a_m$, for each of the m chemical reactions.
5. For each reaction i, generate a putative time τ_i, according to an exponential distribution with parameter a_i.
6. Let μ be the reaction whose putative time, τ_μ is least.
7. Change the number of molecules $X_1, X_2, \ldots, X_n$, to reflect the execution of reaction μ.
8. Set $t \leftarrow t + \tau_\mu$.
9. Go to Step 4.

Figure 3-2: Gillespie's *First Reaction Method.*

constant associated with the reaction k_1 and the current populations of the reactants X_A and X_B, using the following equation:

$$a_1 = X_A \cdot X_B \cdot k_1 \qquad (3)$$

Multiplying X_A and X_B together in equation 3 reflects the number of combinations by which the reaction could occur, thus making the propensity depend on the concentrations of the chemical reactants. The input rate constant k_1 is used to account for all other factors (volume, temperature, etc.) that may determine the propensity of the reaction.

Step 5 of the algorithm uses the propensity generated in step 4 to generate a putative time or the amount of time it will take for this reaction to occur based on the current state of the system. This is accomplished by generating a uniformly distributed random number between 0 and 1 (URN), scaling it to fit the exponential distribution, and multiplying that number by the propensity, as shown in the equation below.

$$\tau_i = -\frac{1}{a_i}\log(URN) \qquad (4)$$

Step 6 selects the reaction with the smallest putative time using a linear search. Step 7 updates the number of molecules based on the stoichiometry of the reaction. For example, if we were executing the reaction given in equation 1, we would decrement the

values X_A and X_B and increment the value of X_C. Step 8 updates the current time based on the putative time selected in Step 6.

3.4 Gillespie's *Direct Method*

Gillespie's *First Reaction Method* accurately simulates chemicals reacting in a spatially homogeneous mixture, but is inefficient. Gillespie realized that for each iteration of the algorithm *M* exponentially distributed random numbers must be generated. This calculation is time-consuming because it involves calculating the natural log function which is computationally intense. To reduce this costly computation, Gillespie developed the *Direct Method*, which sums the individual reaction propensities to create a system propensity. A single scaled exponential random number is then generated from the system propensity which represents the time elapsed between reaction events for the entire system. A scaled uniform random number is then generated to determine which reaction occurred during the time period. By collapsing the reaction propensities, this technique requires only two random numbers to be generated per iteration of the algorithm regardless of problem size. This algorithm is given in figure 3-3.

Steps 5 and 6 of the *First Reaction Method* determine how much time elapses in the system and which reaction to execute by generating putative times for each reaction and selecting the reaction whose putative time is least. Gillespie's *Direct Method* replaces steps 5 and 6 of the *First Reaction Method* with steps 5, 6, and 7, by summing the propensities and determining when the next reaction event should occur for the entire

1. Initialize a list of n chemical species and their initial numbers of molecules $X_1, X_2, \ldots, X_n$.
2. Initialize a list of m chemical reactions and their associated stochastic rate constants $k_1, k_2, \ldots, k_m$.
3. Initialize the current time $t \leftarrow 0$.
4. Calculate the propensity, $a_1, a_2, \ldots, a_m$, for each of the m chemical reactions.
5. Sum the propensity values: $a_{total} = \sum_{i=1}^{M} a_i$.
6. Generate a putative time for the chemical system τ_μ according to an exponential distribution with parameter a_{total}.
7. Choose a reaction μ using a uniformly distributed random number and a distribution of the form $P(\text{Reaction}=\mu) = \frac{a_\mu}{a_{total}}$.
8. Change the number of molecules $X_1, X_2, \ldots, X_n$, to reflect the execution of reaction μ.
9. Set $t \leftarrow t + \tau_\mu$.
10. Go to Step 4.

Figure 3-3: Gillepie's *Direct Method*.

system. The reaction selection is then handled by dividing each individual propensity by the total propensity, giving a normalized uniform distribution which can be used to select a reaction based on the generation of a uniform random number. Using the *Direct Method*, we only need to generate one exponentially distributed random number and one uniformly distributed random number per iteration of the algorithm, regardless of chemical system size, thus improving performance over the *First Reaction Method*.

3.5 The Dependency Graph

In the late 1990's, Gibson and Bruck developed several key enhancements to Gillespie's methods that improve performance by an order of magnitude. These enhancements were presented in [6].

The first enhancement was the creation of a dependency graph to determine which propensity values need to be updated when a particular reaction is executed. During initialization, each reaction is examined to see what effect its execution has on the species population values of the system. If a reaction A is executed that modifies the species populations of another reaction B, reaction B's propensity must be recalculated upon the execution of reaction A. This is more easily understood with an example. Suppose we convert the example system given in section 3.2 to the following set of reactions:

```
A:    DNA -> DNA + mRNA
B:    mRNA -> *
C:    mRNA -> mRNA + Protein
```

```
D:   Protein -> *

E:   2 Protein -> Dimer

F:   Dimer -> 2 Protein
```

Notice the execution of Reaction A only modifies the population of the species `mRNA`. Since the calculation of the propensity values for Reaction B and C depend on the population of `mRNA`, Reaction A affects Reaction B and C. Reactions A, D, E, and F are unaffected by the execution of Reaction A and will therefore have the same propensity before and after the execution of Reaction A. We can therefore build a dependency graph which stores which propensity values must be recalculated for the execution of each reaction and use this dependency graph to prevent the unnecessary calculation of reaction propensities thus improving performance. The adjacency matrix for the dependency graph is given in figure 3-4, where the rows represent the reactions executed and the columns represent the reaction affected.

Because the adjacency matrix for most gene regulatory network models is very sparse, this enhancement greatly improves the performance of Gillespie's methods.

3.6 Gibson and Bruck's *Next Reaction Method*

Using a dependency graph and several other enhancements to Gillespie's methods, Gibson and Bruck proposed a new method called the *Next Reaction Method*, which today remains the fastest known algorithm for exactly stochastically simulating a spatially homogeneous system. This algorithm is stated in figure 3-5 [6].

	A	*B*	*C*	*D*	*E*	*F*
A		X	X			
B		X	X			
C				X	X	
D				X	X	
E				X	X	X
F				X	X	X

Figure 3-4: Example Adjacency Matrix.

1. Initialize:
   (a) Initialize $X_1, X_2, \ldots, X_n$, set $t \leftarrow 0$;
   (b) Generate a dependency graph G based on the stoichiometry of the m chemical reactions;
   (c) Calculate the propensity function, $a_1, a_2, \ldots, a_m$, for each of the m chemical reactions;
   (d) For each reaction i, generate a putative time, τ_i, according to an exponential distribution with parameter a_i;
   (e) Store the τ_i values in an indexed priority queue P.
2. Let μ be the reaction whose putative time, τ_μ stored in P is least.
3. Let τ be τ_μ.
4. Change the number of molecules to reflect the execution of reaction μ. Set $t \leftarrow \tau$.
5. For each edge (μ, α) in the dependency graph G,
   (a) Update a_α;
   (b) If $\alpha \neq \mu$, set $\tau_\alpha \leftarrow (a_{\alpha,old} / a_{\alpha,new})(\tau_\alpha - t) + t$;
   (c) If $\alpha = \mu$, generate a random number, ρ, according to an exponential distribution with parameter a_μ, and set $\tau_\alpha \leftarrow \rho + t$;
   (d) Replace the old in τ_α value in P with the new value.
6. Go to step 2.

Figure 3-5: Gibson and Bruck's *Next Reaction Method*.

Besides the dependency graph mentioned in the previous section, which is initialized in step 1b and is used in step 5 to reduce the number of calculations required for each step of the algorithm, Gibson and Bruck make two key enhancements to the algorithm to improve performance.

The first enhancement is to change from relative time to absolute time and reuse random numbers that have not yet been used to effect the system time. This is accomplished by generating putative times for each reaction on initialization (see step 1d), selecting the reaction to execute (see step 2), and generating a new putative time for the reaction executed (see step 5c). For reactions that were not executed, but had their propensity values affected by the execution of the selected reaction, their putative times are scaled in step 5b using the following equation.

$$\tau_\alpha \leftarrow (a_{\alpha,old} / a_{\alpha,new})(\tau_\alpha - t) + t \qquad (5)$$

This reduces the number of random numbers that must be generated for each iteration of the algorithm to only one, which is an improvement on the *M* random numbers required for the *First Reaction Method* and the two random numbers required for the *Direct Method*.

The other enhancement made to Gillespie's methods is the use of an indexed priority queue, initialized in Step 1e of this method and used in Step 2. Recall that in the *First Reaction Method*, after the putative times for each of the reactions had been determined, a linear search of the putative times is required to determine which reaction to execute. This search has a time complexity which is proportional to *M*, the number of

reactions in the system. In the *Next Reaction Method*, the indexed priority queue is a data structure which stores putative times in a binary tree, where the root node contains the node with the minimum putative time. This data structure can be maintained by operations which are proportional to the logarithm of *M*.

3.7 Gibson and Bruck's *Efficient Direct Method*

Although Gibson and Bruck propose that the *Next Reaction Method* is the most efficient technique for stochastically simulating coupled chemical reactions, they also suggest ways to improve the *Direct Method*. The first enhancement is to use a dependency graph to reduce the number of propensity calculations required for each iteration of the algorithm. The next enhancement is used to accelerate the calculation of the total propensity and reaction selection stages by storing the propensity values in a data structure which we will refer to as a "sum tree." A sum tree consists of a binary tree where each of the leaf nodes contains a propensity value and each parent node contains the sum of its ancestors. Using this data structure, the total propensity will always be stored at the root of the binary tree and this data structure can be used as a search tree when selecting the reaction. This reduces the reaction selection step from time complexity O(n) to O(log n) and reduces the number of calculations that must be performed when determining the total propensity of the system. Because Gibson and Bruck did not name this algorithm, we will refer to it as the *Efficient Direct Method*. This algorithm is stated in figure 3-6.

1. Initialize:

   (a) Initialize $X_1, X_2, \ldots, X_n$, set $t \leftarrow 0$;
   (b) Generate a dependency graph G based on the stoichiometry of the m chemical reactions;
   (c) Calculate the propensity function, $a_1, a_2, \ldots, a_m$, for each of the m chemical reactions;
   (d) Store the propensity values $a_1, a_2, \ldots, a_m$ in a sum tree T;

2. Retrieve the value $a_{total} = \sum_{i=1}^{M} a_i$ by reading the root node of the sum tree T.
3. Generate a putative time for the chemical system τ_μ according to an exponential distribution with parameter a_{total}.
4. Choose a reaction μ using a uniformly distributed random number and a distribution of the form P(Reaction=μ)=$\frac{a_\mu}{a_{total}}$ by searching T.
5. Change the number of molecules $X_1, X_2, \ldots, X_n$, to reflect the execution of reaction μ.
6. Set $t \leftarrow t + \tau_\mu$.
7. For each edge (μ, α) in the dependency graph G,
   (a) Update a_α;
   (b) Replace the old a_α value in T with the new value.
   (c) Update a_α's parent nodes' values in T.
10. Go to Step 4.

Figure 3-6: Gibson and Bruck's *Efficient Direct Method*.

3.8 Other Techniques

Several other techniques exist for modeling gene regulatory networks. The simplest way is to represent the regulatory network as a directed graph where the vertices are genes and the edges represent interactions between genes. Often the basic definition of a directed graph is expanded to store whether the interaction between the genes is positive or negative regulation. Another modeling approach is to use Bayesian networks, where vertices in an acyclic graph represent genes or other elements that effect the system and correspond to random variables. Boolean networks are another technique used where the expression of a gene is considered on or off and the interaction between genes can be described by a digital circuit composed of AND and OR gates. A good review of these techniques along with more information on the various differential equations models used to represent gene regulatory networks is given in [3].

One of the most recent developments in gene regulatory network modeling has been Gillespie's work on "tau-leaping" methods, which is a form of approximate stochastic simulation where multiple reactions are executed in a single step. These techniques greatly improve the performance of stochastic simulation at the potential cost of accuracy. For more information on this technique, consult [20,21].

3.9 Conclusion

This chapter has introduced the reader to several techniques for modeling gene regulatory networks. Because molecular dynamics modeling is inefficient and other modeling techniques can be inaccurate, this work will focus on exact stochastic

simulation. A review of the existing techniques for exact stochastic simulation were presented, highlighting the works of Gillespie and Gibson and Bruck. In the following chapters we can use this discussion to analyze the performance of these techniques and discuss further enhancements to these methods.

Chapter 4

Analyzing Exact Stochastic Simulation

The previous chapter introduced several different exact stochastic simulation algorithms, primarily the *First Reaction Method*, the *Direct Method*, the *Next Reaction Method*, and the *Efficient Direct Method.* In this chapter, we will analyze the performance of each these algorithms by coding the algorithms in C++ and examining their performance on a set of generated input files ranging in model size. We also will suggest modifications to these algorithms and demonstrate that these modifications can improve simulator performance.

4.1 Creating a Model Set

To model the performance of the simulation algorithms accurately we must first create a set of models that produce consistent results. To do this, we create a program that generates models based on two parameters, a reaction count and an "update factor". The reaction count is the number of reactions represented in the model and indicates model size. The update factor dictates the average number of propensity updates that must be performed when a reaction is executed. For instance, with the following model,

```
1: A -> B
2: B -> C
3: C -> D
4: D -> E
```

```
5: E -> F
6: F -> A
```

the reaction count is 6 and when any reaction is executed, 2 reactions must be updated (i.e. if reaction 3 is executed, the propensity values for reaction 3 and 4 must be updated). The update factor is defined in this work as the number of reactions that must be updated for each reaction. Therefore the update factor for the model given above is 2. The following model has 6 reactions and an update factor 4.

```
1: A -> B
2: A -> B
3: B -> C
4: B -> C
5: C -> A
6: C -> A
```

In this case, the execution of reaction 3 impacts the propensity value of reactions 3, 4, 5, and 6, therefore the update factor is 4.

The maximum possible update factor for a given model is the model size (the number of reactions). In this scenario, every reaction would affect the propensity value of every other reaction. In real chemical systems, the update factor tends to be much less than the model's reaction count (less than 10%), and this underlying assumption is what makes Gibson and Bruck's performance enhancements possible. With a high update

factor, the dependency graph's adjacency matrix would be very dense making the number of propensity updates approach the number of reactions in the model for each step of the algorithm.

The two models presented previously have uniform structure in that each reaction affects the same number of reactions. Models could be built that affect different numbers of reactions, like the following model.

```
1: A -> B
2: B -> C
3: C -> A
4: B -> A
```

In this case, reaction 1 affects the propensity value of 3 reactions and reaction 3 only affects the propensity value of 2 reactions. Depending on the rate constants of the reaction 1 and reaction 3, along with the time evolution of the population species, the update factor of the model can vary as the simulator executes. Using a model with varying update factor would make characterizing performance of each simulator difficult, so we generate models with uniform structure. Our model set sequentially sweeps update factors 2 through 8 by increments of 2 and sweeps reaction counts from 12 to 600 at 12 reaction increments. This creates 200 models to run for each simulator. The code used to generate these models can be found in the appendices.

Although the simulation algorithms allow reactions to be entered that contain multiple reactants with varying coefficients, to compare the performance of our

simulators we limit our models to have uniform structure to simplify our analysis. Each reaction consists of one reactant and one product. We assume that using more complex reactions will not effect our performance comparison, because the cost of calculating the propensity functions will be the same for each simulator.

Each simulation is executed on a Sun Ultra-60 processor running Sun OS v5.8. The simulators are compiled using the Sun Workshop 6 C++ compiler with the highest performance optimization settings. The performance results printed in this work are gathered by averaging four separate runs of the simulators running each model for 5,000,000 reactions.

4.2 The *First Reaction Method*

We start our performance analysis by coding the Gillespie's First Reaction Method (*First*), the oldest, simplest and most inefficient algorithm for exact stochastic simulation. Because this algorithm must generate *m* (the number of reactions) random numbers for each iteration of the algorithm and does not take advantage of a dependency graph, we would expect the execution time of this algorithm to grow linearly with reaction count and to be unaffected by the update factor. The figure 4-1 shows the performance of the *First* simulator and verifies our expectation.

We then add the dependency graph proposed by Gibson and Bruck to see if this enhancement positively impacts the performance of the simulator. We will call this simulator *Gillespie's First Reaction Method with a Dependency Graph* and abbreviate it *FirstDG*. Using a dependency graph we can significantly reduce the number of propensity recalculations that must be performed in each step of the algorithm, but we

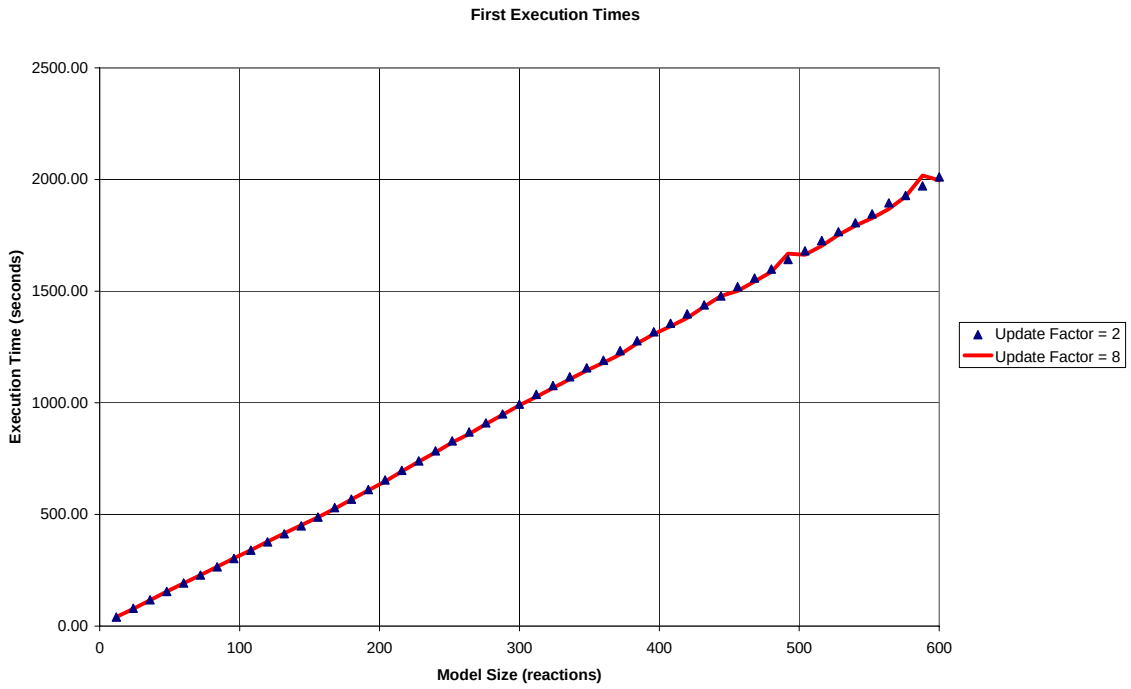

Figure 4-1: *First* Execution Times.

still must generate *m* exponentially distributed random numbers for each iteration. It would be expected that *FirstDG* would outperform *First* for small update factors, because *FirstDG* would update only a small number of propensities per iteration. Figure 4-2 shows that for update factors as high as 8, *FirstDG* does outperform *First*.

4.3 The *Direct Method*

We now look at the Direct Method, Gillespie's improvement to the First Reaction Method. Because the Direct Method (*Direct*) does not utilize a dependency graph, we once again expect that its performance will not be affected by the update factor. Figure 4-3 demonstrates this. The outlier at model size 175 is due to machine load imbalance.

We can also show that *Direct* outperforms *First* and *FirstDG* in all cases by looking at figures 4-4 and 4-5. This occurs because *Direct* generates only two random numbers for each iterations of its algorithm, where *First* and *FirstDG* must generate *m* (the number of reactions in the model) random numbers.

Now we add a dependency graph to the Direct Method, to create simulator *DirectDG*, and see if this improves the performance of *Direct*. Figure 4-6 shows that *DirectDG* outperforms *Direct* for update factors as high as 8.

In the *DirectDG* simulator, we noticed that for each iteration of the algorithm, the propensity values must be summed to generate the total propensity, which becomes a significant performance bottleneck as reaction count increases. To reduce this update time, we implemented another simulator called *DirectDG2*, which when the individual reaction propensity values are being updated, the total propensity value is also updated by

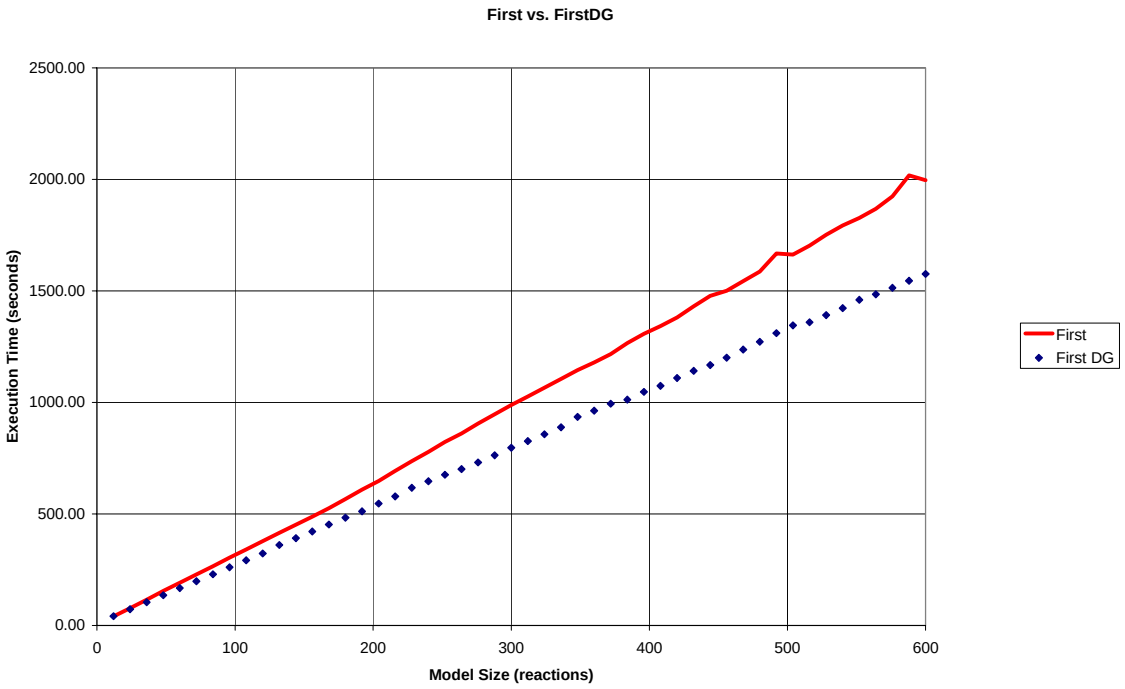

Figure 4-2: *First* vs. *FirstDG* (Update Factor = 8)

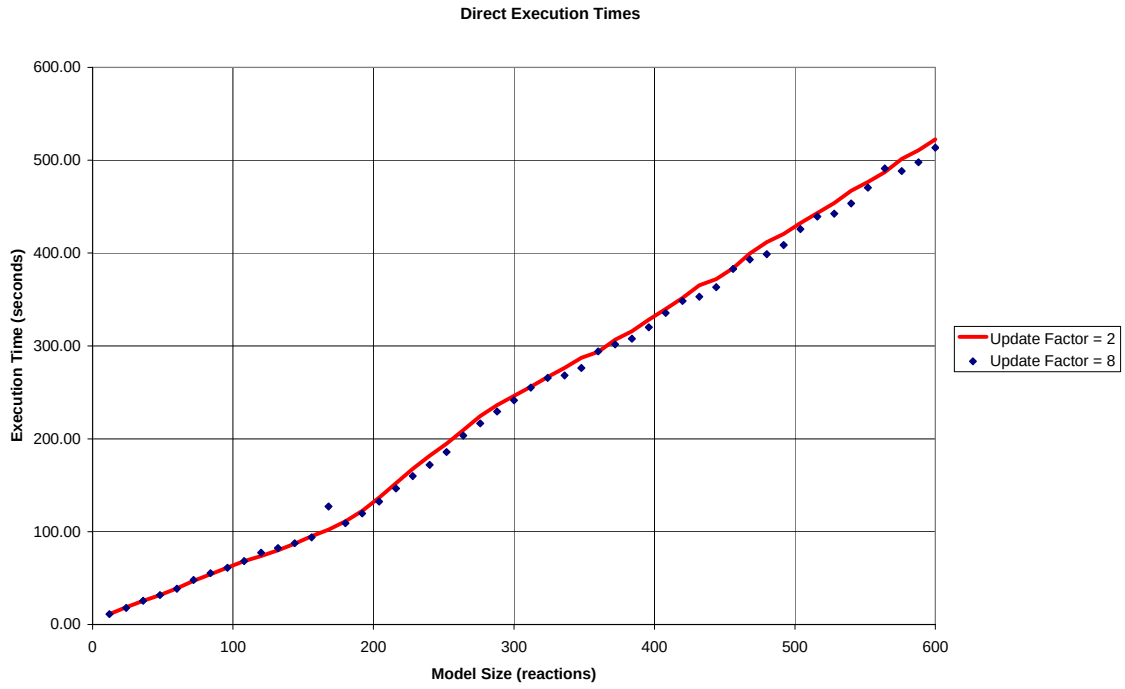

Figure 4-3: *Direct* Execution Times

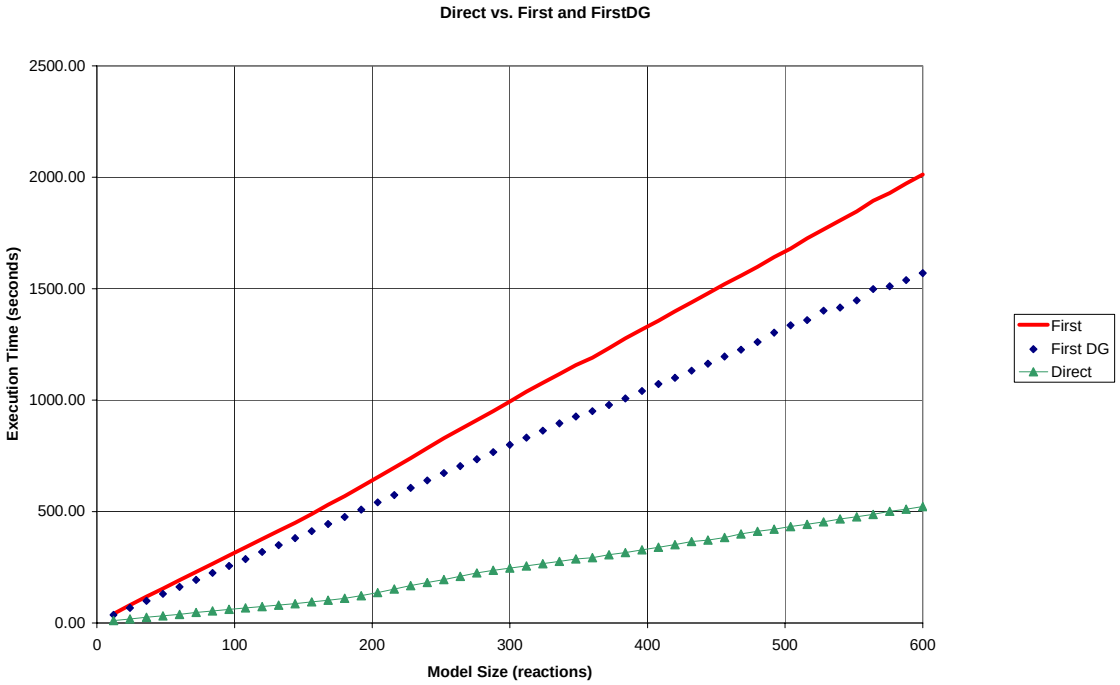

Figure 4-4: *Direct* vs. *First* and *FirstDG* (Update Factor = 2)

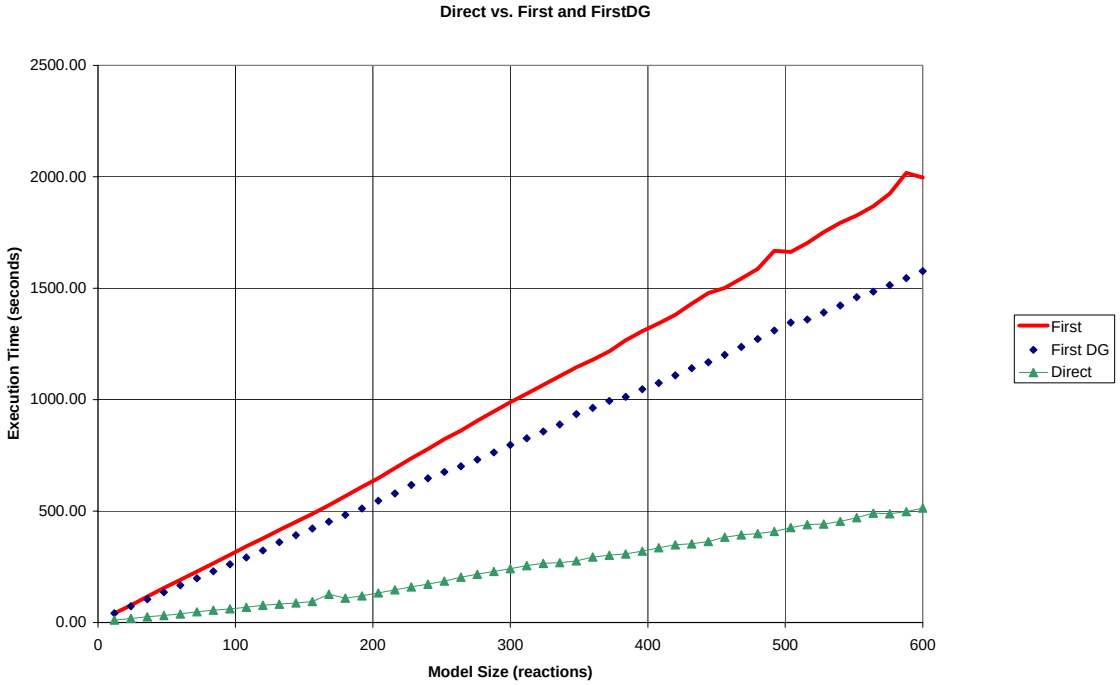

Figure 4-5: *Direct* vs. *First* and *FirstDG* (Update Factor = 8)

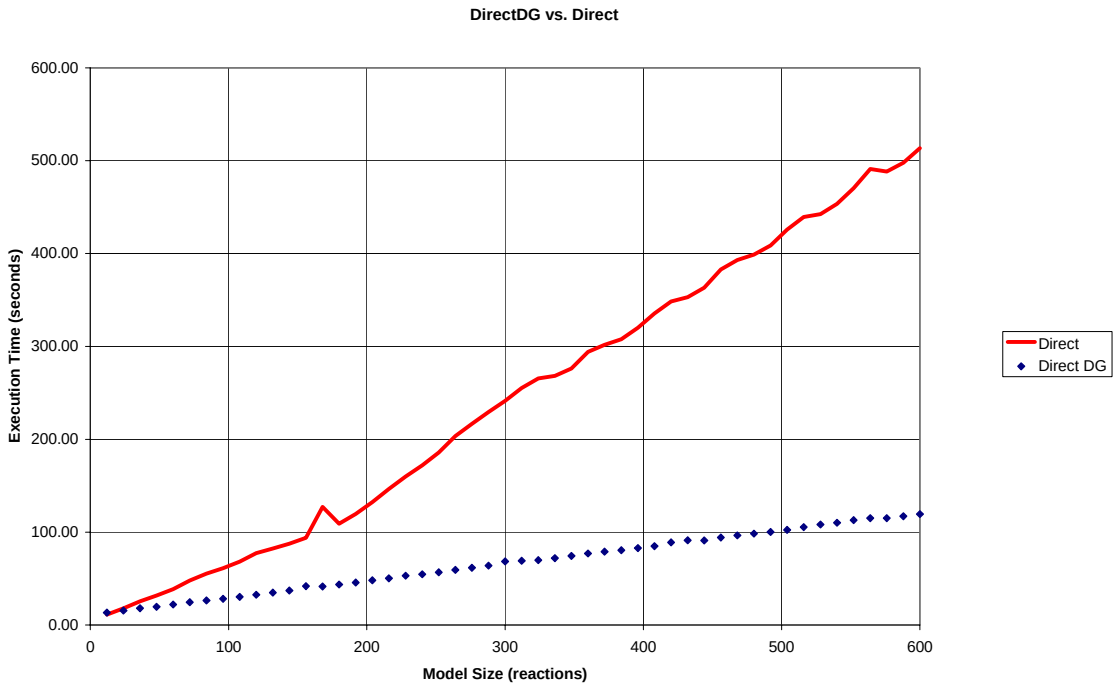

Figure 4-6: *DirectDG* vs. *Direct* (Update Factor = 8)

subtracting the old propensity value and adding the newly calculated propensity value. This enhancement makes the total propensity calculation time depend on the update factor instead of the reaction count, improving performance. A comparison of the *DirectDG* and *DirectDG2* simulators shows that *DirectDG2* outperforms *DirectDG* for update factors as high as 8. A plot of this is shown in figure 4-7.

4.4 The *Efficient Direct Method*

The Efficient Direct Method (*EfficientDirect*) was first outlined in Gibson and Bruck's paper on improving the performance of Gillespie's simulation algorithms [5,6]. Their enhancements included the addition of a sum tree. This algorithm is stated in detail in chapter 3. Figure 4-8 shows that this algorithm's performance is greatly affected by the update factor and grows slowly with respect to reaction count. Once again the outliers are caused by machine load imbalance. Figures 4-9 and 4-10 compare the performance of *EfficientDirect* with *DirectDG2*. Notice that for small models, *DirectDG2* outperforms *EfficientDirect*, because the sum tree adds significant overhead and almost no performance gain for small models and a large performance improvement for models with high reaction count.

4.5 The *Next Reaction Method*

The Next Reaction Method (*Next*) developed by Gibson and Bruck is widely accepted as the fastest known simulator for performing exact stochastic simulation [6]. Figure 4-11 shows that because of the indexed priority queue and other performance

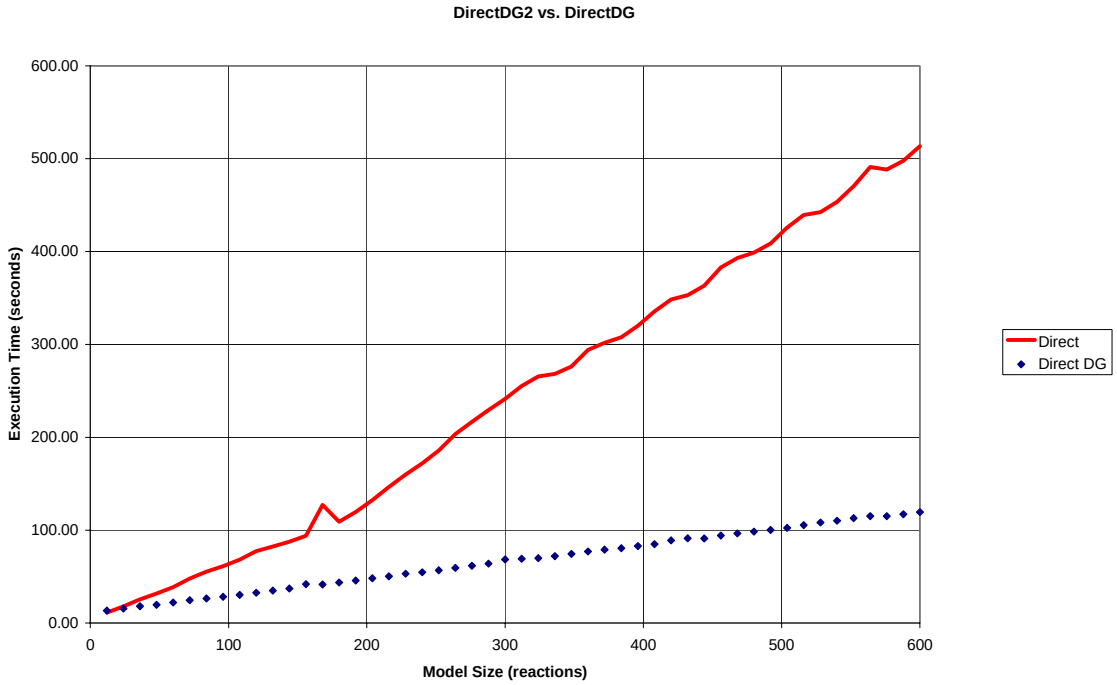

Figure 4-7: *DirectDG2* vs. *DirectDG* (Update Factor = 8)

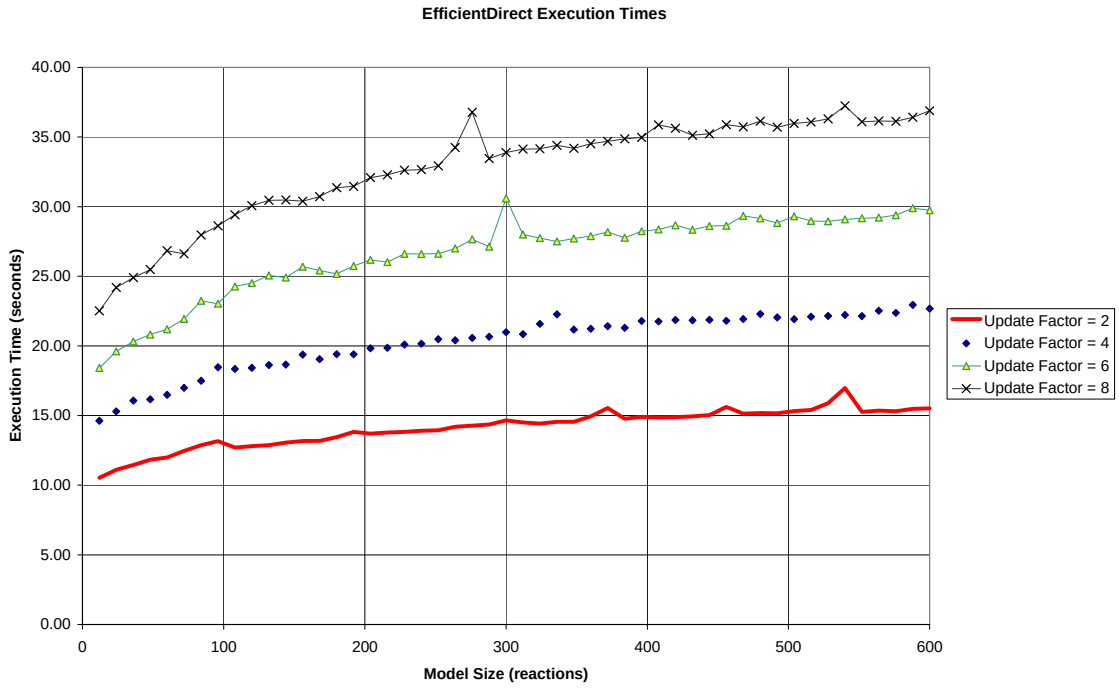

Figure 4-8: *EfficientDirect* Execution Times

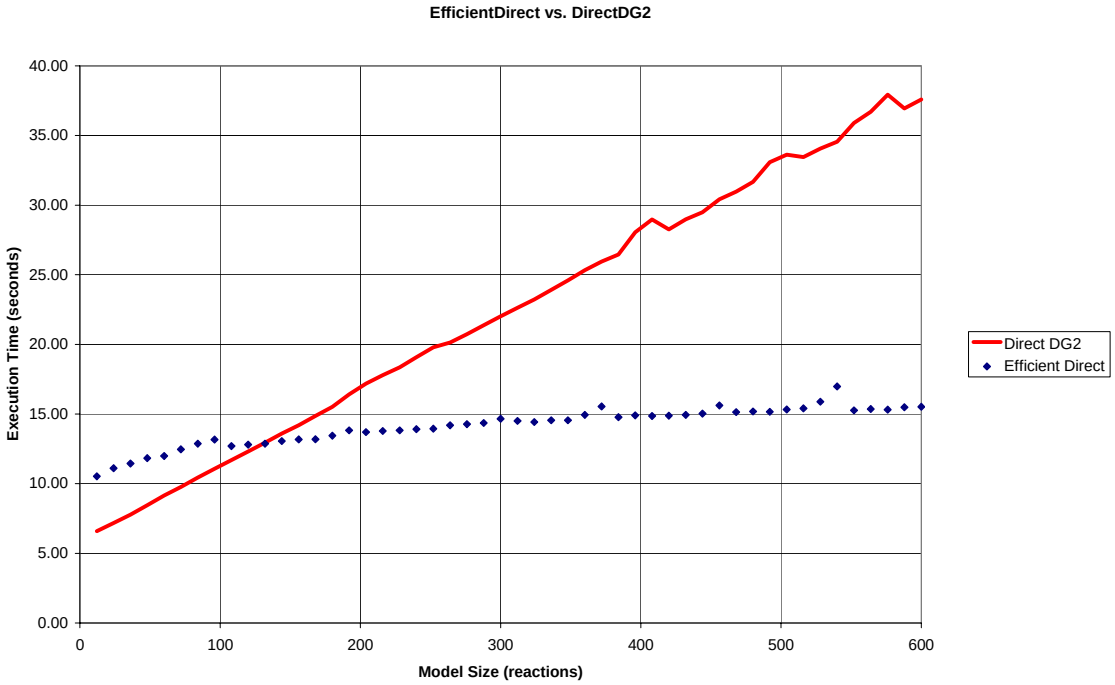

Figure 4-9: *EfficientDirect* vs. *DirectDG2* (Update Factor = 2)

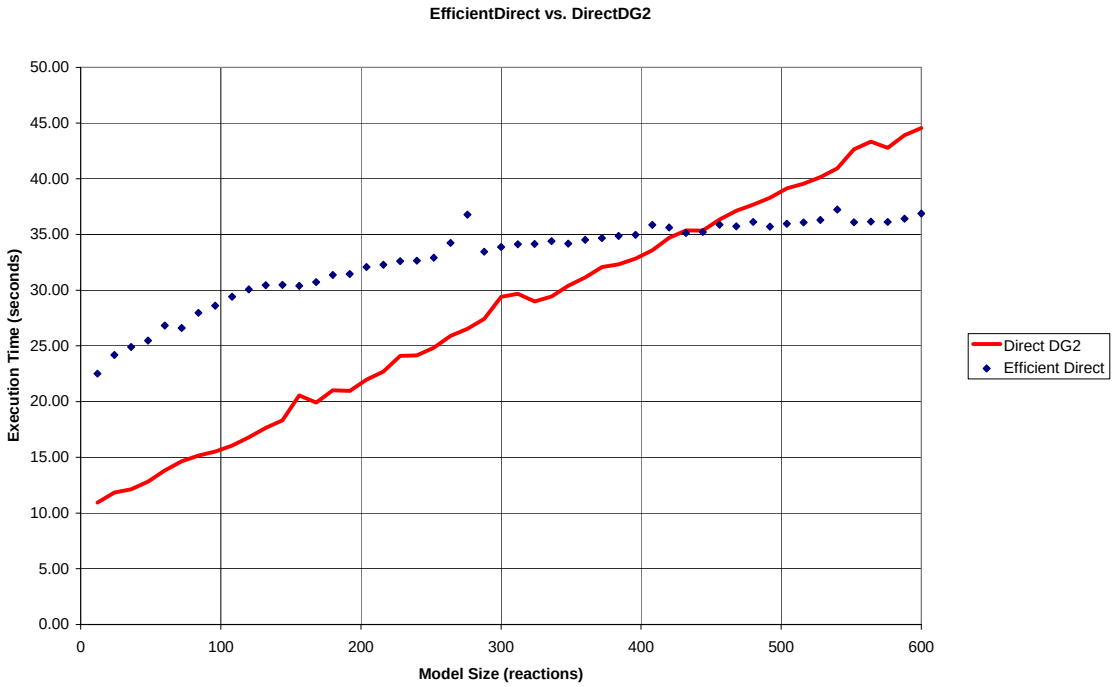

Figure 4-10: *EfficientDirect* vs. *DirectDG2* (Update Factor = 8)

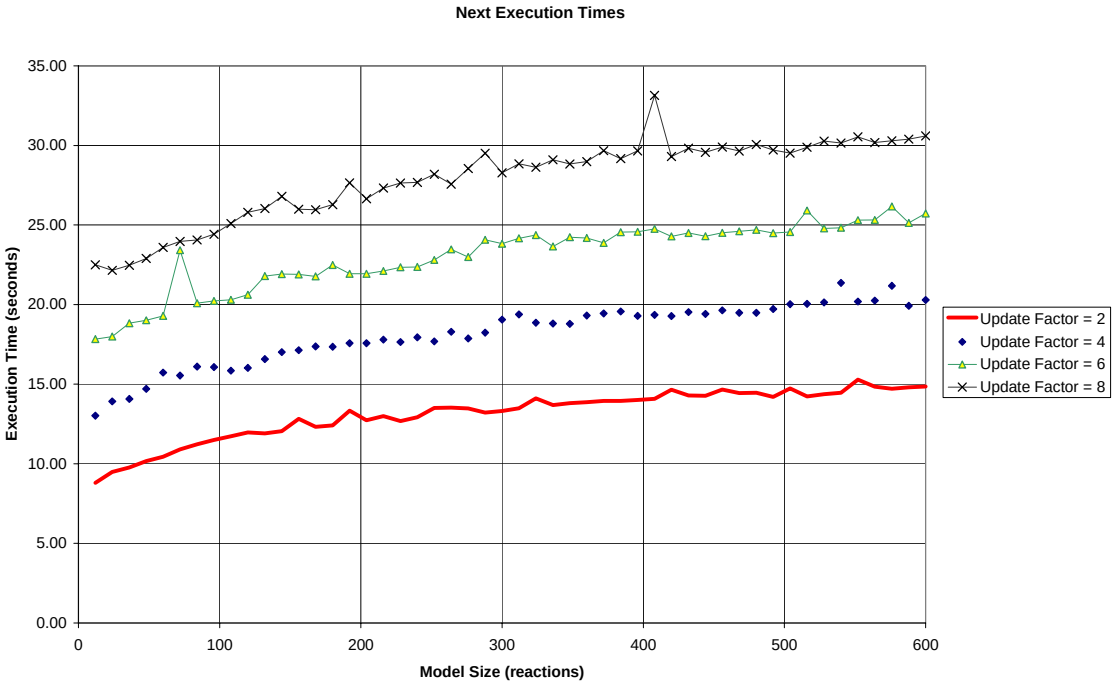

Figure 4-11: *Next* Execution Times

enhancements implemented in the Next Reaction Method, the execution time for the Next Reaction Method does not grow in relation to reaction count, but is greatly affected by update factor. Figures 4-12 and 4-13 show that *Next* outperforms *EfficientDirect* for all models. Figures 4-14 and 4-15 show that *Next* only outperforms the *DirectDG2* algorithm for larger models. Outliers are caused by machine load imbalance.

4.6 Conclusion

The performance data demonstrates that the new *DirectDG2* algorithm is an efficient algorithm for simulating models with small reaction counts and larger update factors. The *Next* simulator is shown to be the ideal solution for simulating large models. The other approaches are not as competitive in the scenarios considered.

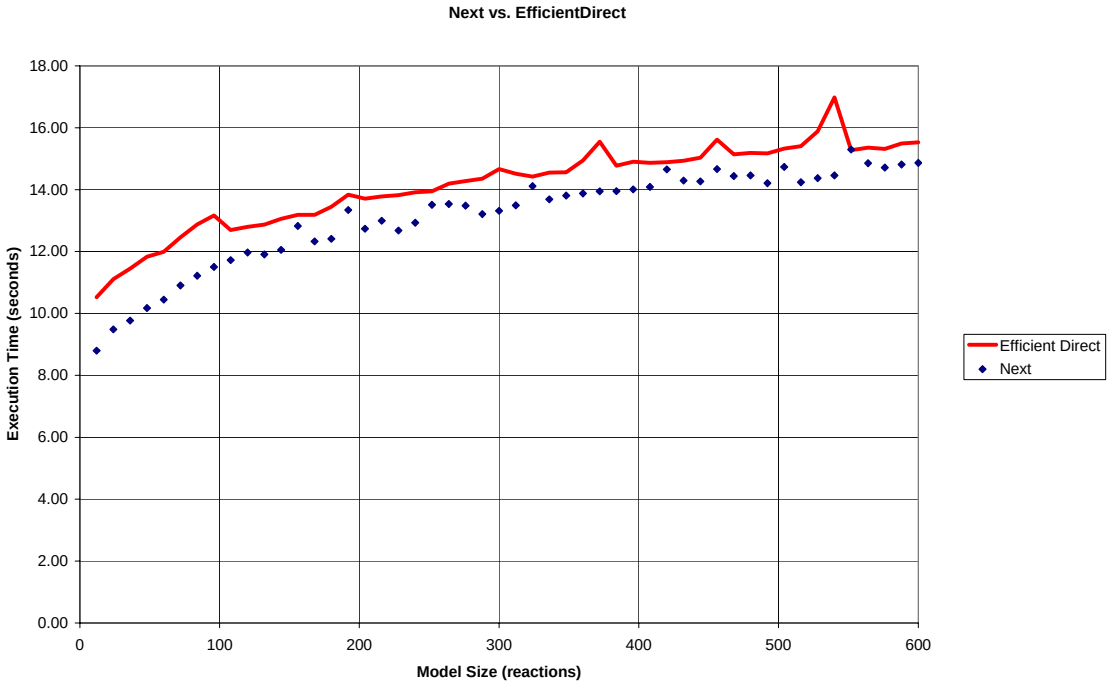

Figure 4-12: *Next* vs. *EfficientDirect* (Update Factor = 2)

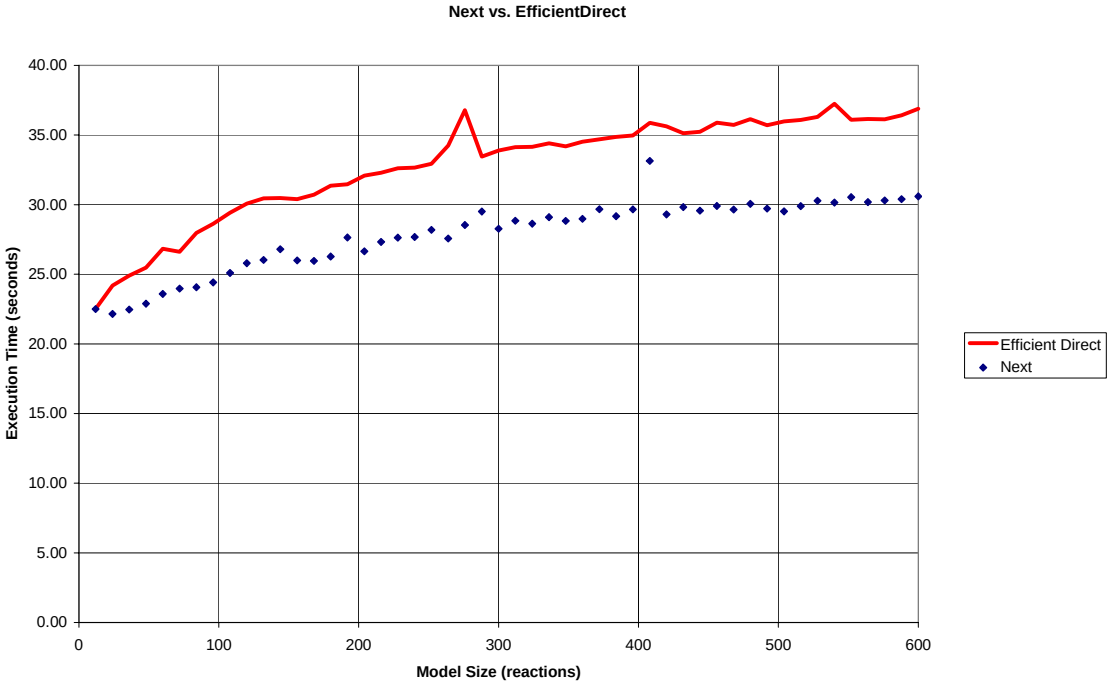

Figure 4-13: *Next* vs. *EfficientDirect* (Update Factor = 8)

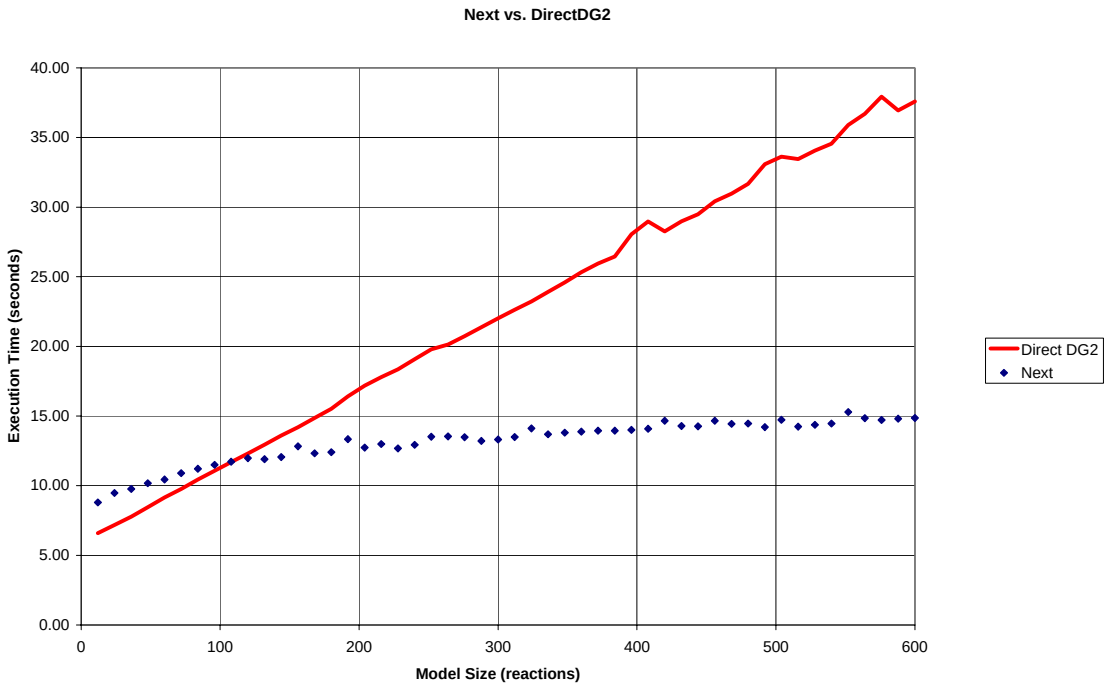

Figure 4-14: *Next* vs. *DirectDG2* (Update Factor = 2)

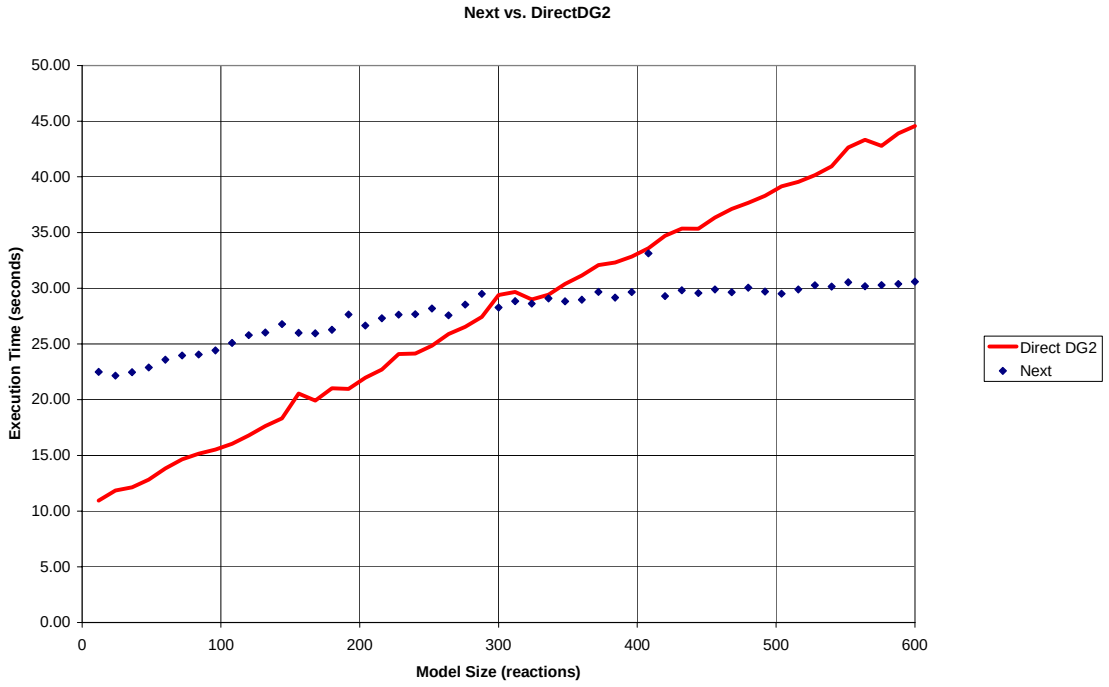

Figure 4-15: *Next* vs. *DirectDG2* (Update Factor = 8)

Chapter 5

The Adaptive Method

Now that we have measured the performance of several exact stochastic simulation algorithms and discussed how their performance is impacted by model size and update factor, we now focus on creating a system for selecting the algorithm that optimizes performance when simulating a particular model. To accomplish this, we derive an estimate for how to determine which algorithm to select based on update factor and reaction count. Unfortunately, it is impossible to calculate the update factor before simulating the model. To overcome this problem, we propose a new algorithm for stochastic simulation called the *Adaptive Method*, which monitors the update factor as the simulator progresses and adaptively controls which simulator to use for the model. We then build this simulator and compare it to the performance of the other simulators. The analysis shows that this technique is effective in improving the overall performance of stochastic simulation.

5.1 *Next* vs. *DirectDG2*

The two best performing simulators from chapter four were the *Next* and the *DirectDG2* simulators. A comparison of their execution times is given in figures 5-1 to 5-4.

The results show that *DirectDG2* outperforms *Next* for models with a small reaction count. The point at which *Next* begins to outperform *DirectDG2* depends on the update factor. Figure 5-5 shows a plot of the approximate points where the execution

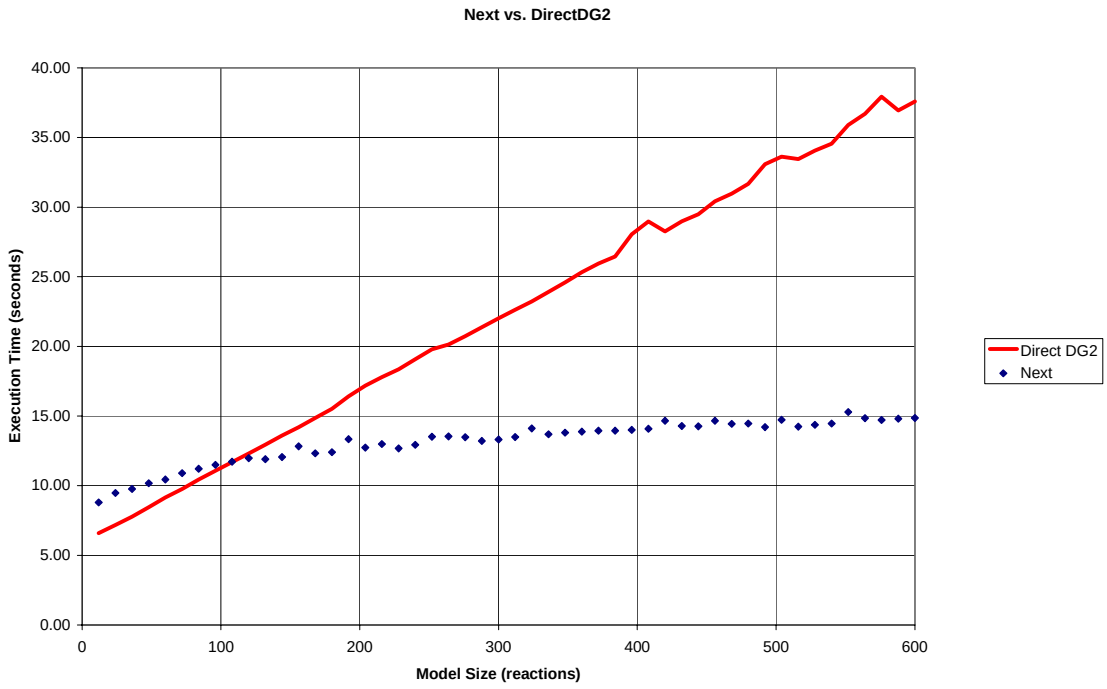

Figure 5-1: *Next* vs. *DirectDG2* (Update Factor = 2)

Figure 5-2: *Next* vs. *DirectDG2* (Update Factor = 4)

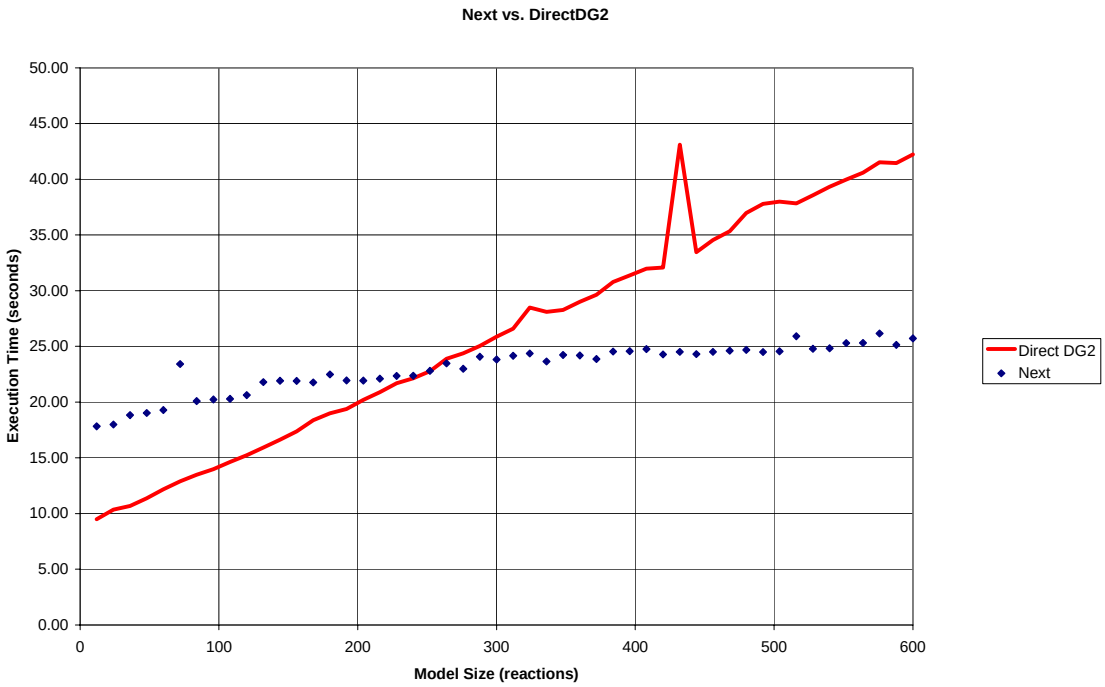

Figure 5-3: *Next* vs. *DirectDG2* (Update Factor = 6)

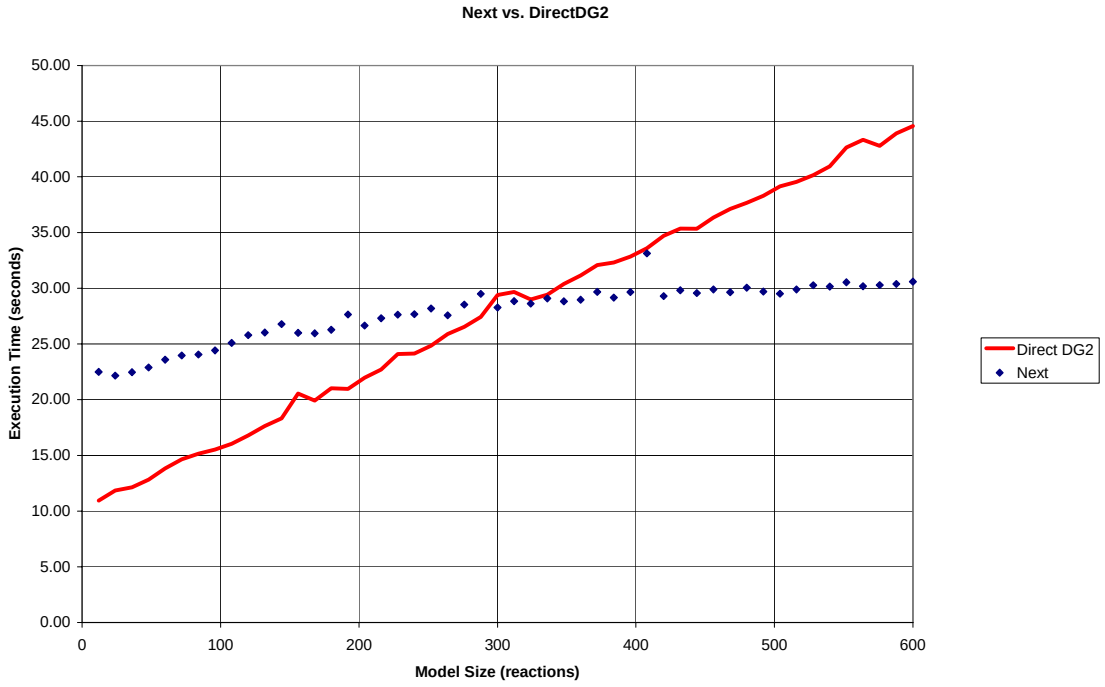

Figure 5-4: *Next* vs. *DirectDG2* (Update Factor = 8)

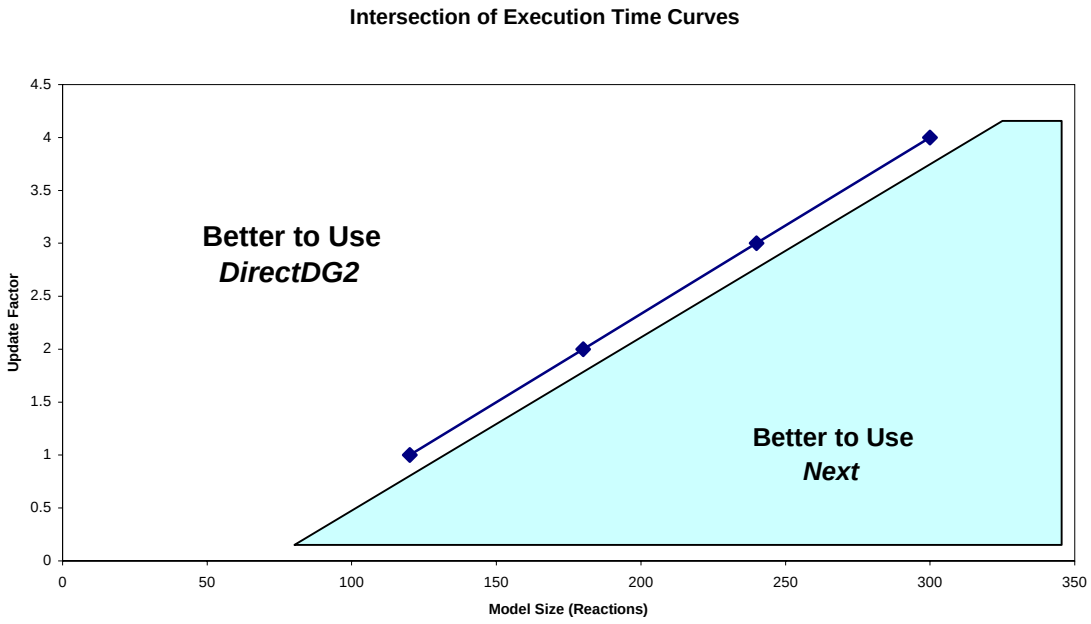

Figure 5-5: Intersection of Execution Time Curves for *DirectDG2* vs. *Next*

time curves meet and shows which simulator is better to use based on the properties of the model.

Using figure 5-5 we should be able to design a simulation algorithm that examines the model at startup and determines which algorithm is better to use. To do this, we need to know the model size and the update factor. Measuring the model size at startup is trivial, but the update factor is dependent on which reactions are executed and can vary as the simulation progresses. Therefore to correctly handle the update factor, we must measure the update factor as the simulation progresses and adapt our simulator to select the simulation algorithm that optimizes performance. To do this, we have designed a simulation algorithm called the *Adaptive Method*.

5.2 The *Adaptive Method*

A summarized version of the *Adaptive Method* algorithm is stated in figure 5-6. The complete *Adaptive Method* algorithm is given in figure 5-7.

The first step of the algorithm is similar to the *Next* simulator, initializing a dependency graph and an indexed priority queue. The first step also initializes a variable called `Mode` which stores the current simulation algorithm that is being used, `ReactionsExecuted` which stores the number of reactions executed since the last update factor check, `UpdateCount` which stores the number of updates of propensity values performed since the last update factor check, and `UpdateFactorThreshold` which stores the estimated value of where the *DirectDG2* simulator will begin to outperform the *Next* simulator.

```
1. Initialize
     (a) Initialize data structures for performing NEXT;
     (b) Set Mode = NEXT;
     (c) Reset ReactionsExecuted and UpdateCount to 0;
     (d) Estimate the UpdateFactorThreshold;
2. If (Mode == NEXT)
     (a) Run a single step of NEXT;
     (b) update UpdateCount;
3. If (Mode == DIRECTDG2)
     (a) Run a single step of DIRECTDG2;
     (b) update UpdateCount;
4. Increment ReactionsExecuted;
5. If (ReactionsExecuted == PERIOD)
     (a) Compute the current UpdateFactor;
     (b) If we should switch to DIRECTDG2
          i. Initialize DIRECTDG2 data structures;
          ii. Set Mode = DIRECTDG2;
     (c) If we should switch to NEXT
          i. Initialize NEXT data structures;
          ii. Set Mode = DIRECTDG2;
     (d) Reset ReactionsExecuted and UpdateCount to 0;
5. Goto 2.
```

Figure 5-6: Summary of the *Adaptive Method*

1. Initialize:
   (a) Initialize $X_1, X_2, \ldots, X_n$
   (b) Set t = 0;
   (b) Generate a dependency graph G based on the stoichiometry of the m chemical reactions;
   (c) Calculate the propensity function, $a_1, a_2, \ldots, a_m$, for each of the m chemical reactions;
   (d) For each reaction i, generate a putative time, τ_i, according to an exponential distribution with parameter a_i;
   (e) Store the τ_i values in an indexed priority queue P.
   (f) Set Mode = NEXT;
   (g) Set ReactionsExecuted = 0;
   (h) Set UpdateCount = 0;
   (i) Set UpdateFactorThreshold = (m – 60) / 30 / 2;
2. if (MODE == NEXT) then
   (a) Let μ be the reaction whose putative time, τ_μ stored in P is least.
   (b) Let τ be τ_μ.
   (c) Change the number of molecules to reflect the execution of reaction μ. Set $t \leftarrow \tau$.
   (d) For each edge (μ, α) in the dependency graph G,
      i. Update a_α;
      ii. If $\alpha \neq \mu$, set $\tau_\alpha \leftarrow (a_{\alpha,old} / a_{\alpha,new})(\tau_\alpha - t) + t$;
      iii. If $\alpha = \mu$, generate a random number, ρ, according to an exponential distribution with parameter a_μ, and set $\tau_\alpha \leftarrow \rho + t$;
      iv. Replace the old in τ_α value in P with the new value.
      v. Increment UpdateCount;
3. if (Mode == DIRECTDG2) then
   (a) Generate a putative time for the chemical system τ_μ according to an exponential distribution with parameter a_{total};
   (b) Set $t \leftarrow t + \tau_\mu$;

Figure 5-7: The *Adaptive Method*

(c) Choose a reaction μ using a uniformly distributed random number and a distribution of the form $\Pr(\text{Re}\,action = \mu) = \frac{a_\mu}{a_{total}}$;

(d) Change the number of molecules $X_1, X_2, ..., X_n$, to reflect the execution of reaction μ;

(e) For each edge (μ, α) in the dependency graph G,

i. Let temp = a_α;

ii. Update a_α;

iii. Let $a_{total} = a_\alpha$ - temp;

iv. Increment UpdateCount;

4. Increment ReactionsExecuted;
5. If (ReactionsExecuted == PERIOD) then
   (a) Set UpdateFactor = UpdateCount / PERIOD / 2;
   (b) if ((UpdateFactor > UpdateFactorThreshold) and (Mode == NEXT)) then
      i. Set Mode = DIRECTDG2;
      ii. Sum the propensity values: $a_{total} = \sum_{i=1}^{M} a_i$;
   (c) else if ((UpdateFactor < UpdateFactorThreshold) and (Mode == DIRECTDG2)) then
      i. Set Mode = NEXT;
      ii. For each reaction i, generate a putative time, τ_i, according to an exponential distribution with parameter a_i added to the current time t;
      iii. Store the τ_i values in an indexed priority queue P.
   (d) set UpdateCount = 0;
   (e) set ReactionsExecuted = 0;
6. Goto step 2.

Figure 5-7: continued.

If the current mode is set to `NEXT`, the second step of the algorithm performs a *Next* simulation step and increments `UpdateCount` when a propensity value is updated. If the current mode is set to `DIRECTDG2`, The third step of the algorithm performs a *DirectDG2* simulation step and increments `UpdateCount` when a propensity value is updated. Step 4 then increments the `ReactionsExecuted` variable.

For every `PERIOD` reactions executed, Step 5 of the algorithm performs an update factor check by computing the current update factor and comparing it to the update factor threshold to see if the optimal simulation algorithm is currently being executed. If not, Steps 5b and 5c set the algorithm to the correct algorithm and reinitialize the necessary data structures for performing that algorithm. The value of `PERIOD` used in this performance analysis was 5,000. Increasing `PERIOD` will reduce the rate at which the algorithm checks the update factor, reducing algorithm switching overhead, but causing the simulator to possibly use a non-ideal algorithm for a longer period of time. Decreasing `PERIOD` will have the opposite affect, increasing the overhead associated with switching algorithms, but reducing the amount of time spent executing the incorrect algorithm. Further research is required to determine the best value for `PERIOD` or whether its value could be adaptively controlled. In this analysis the value 5,000 generated results that closely matched the performance of the best performing simulation algorithm.

5.3 Performance Analysis

To analyze the performance of the adaptive method, the same setup that was described in chapter 4 was used to test a C++ implementation of the *Adaptive Method*. The results are given in figures 5-8 through 5-11. Outliers are the result of machine load imbalance.

Figures 5-8 through 5-11 show that performance of the *Adaptive Method* closely matches the best performing simulator. The overhead associated with monitoring the update factor throughout the execution of the simulator appears to be minimal.

All of these tests were run on models where the update factor was fixed to a particular value. When running on a real model, the update factor can change throughout the execution of the model. Because the *Adaptive Method* checks the simulation's update factor every `PERIOD` steps, the *Adaptive Method* can actually outperform the *Next* and *DirectDG2* algorithms by choosing an optimal simulation algorithm to run as the simulation progresses.

To verify that the Adaptive Method is an effective solution for other computers, the *Adaptive*, *Next*, and *DirectDG2* simulators are ported to the Windows platform and the test models are run on a 1 GHz Intel Pentium III with 512 MB of RAM running Windows XP. The simulators are compiled using Microsoft Visual C++ using the highest optimization settings. Figures 5-12 through 5-15 show the performance on this machines. The reported values are collected by averaging the execution times for four separate runs of each simulator for 1,000,000 reactions.

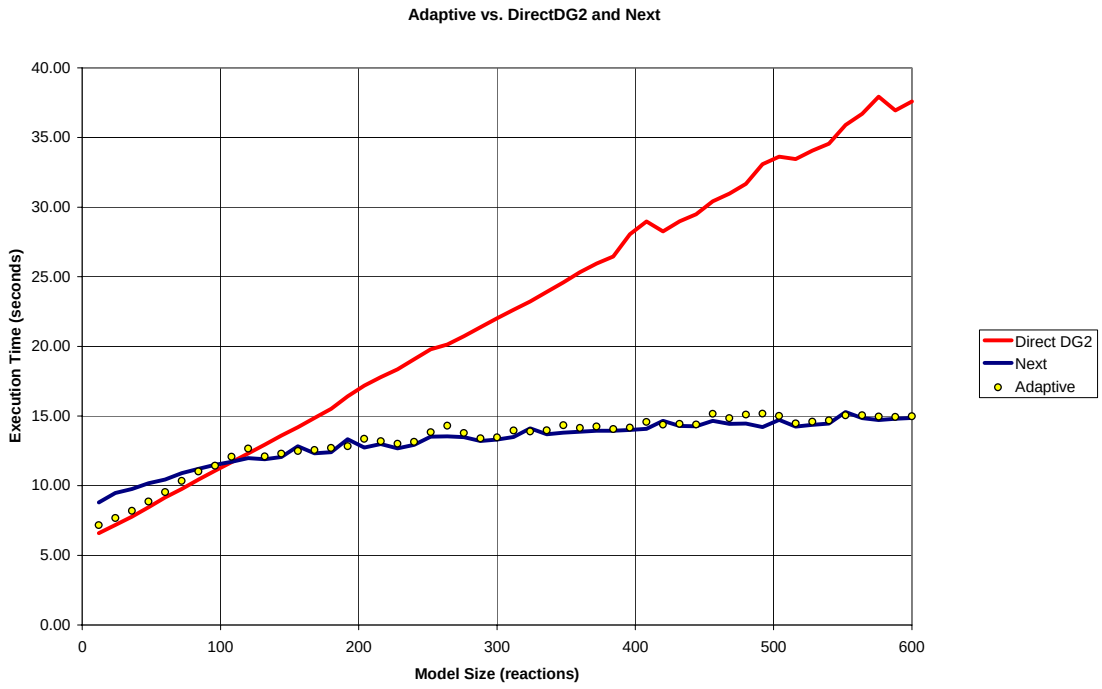

Figure 5-8: *Adaptive Method* vs. *DirectDG2* and *Next* (Update Factor = 2)

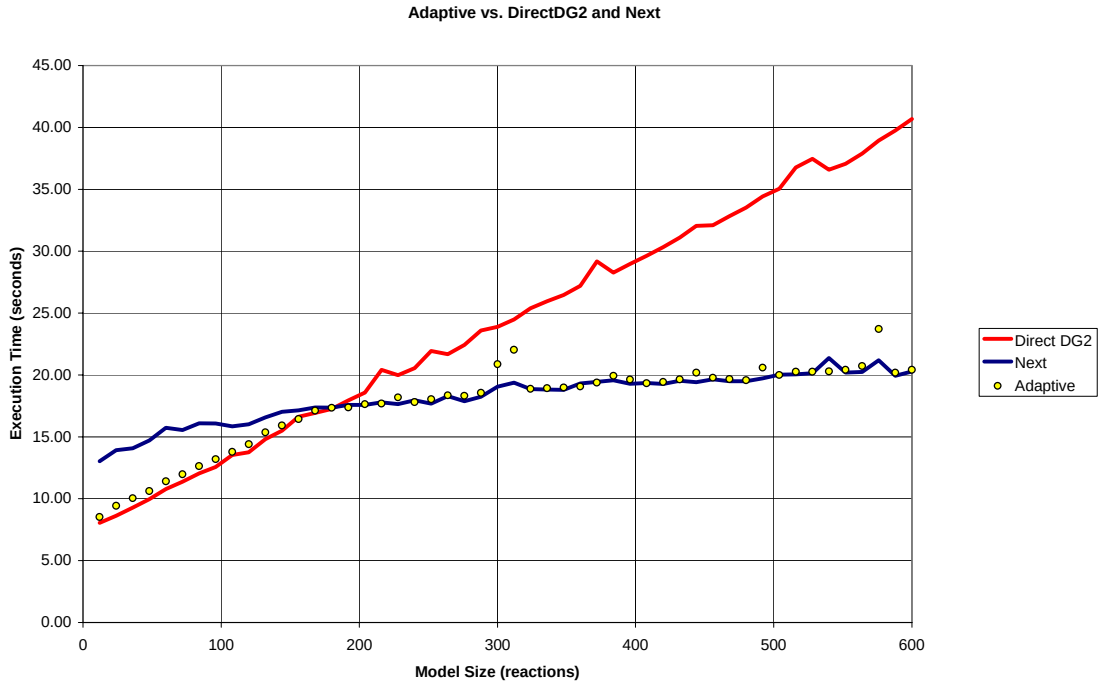

Figure 5-9: *Adaptive Method* vs. *DirectDG2* and *Next* (Update Factor = 4)

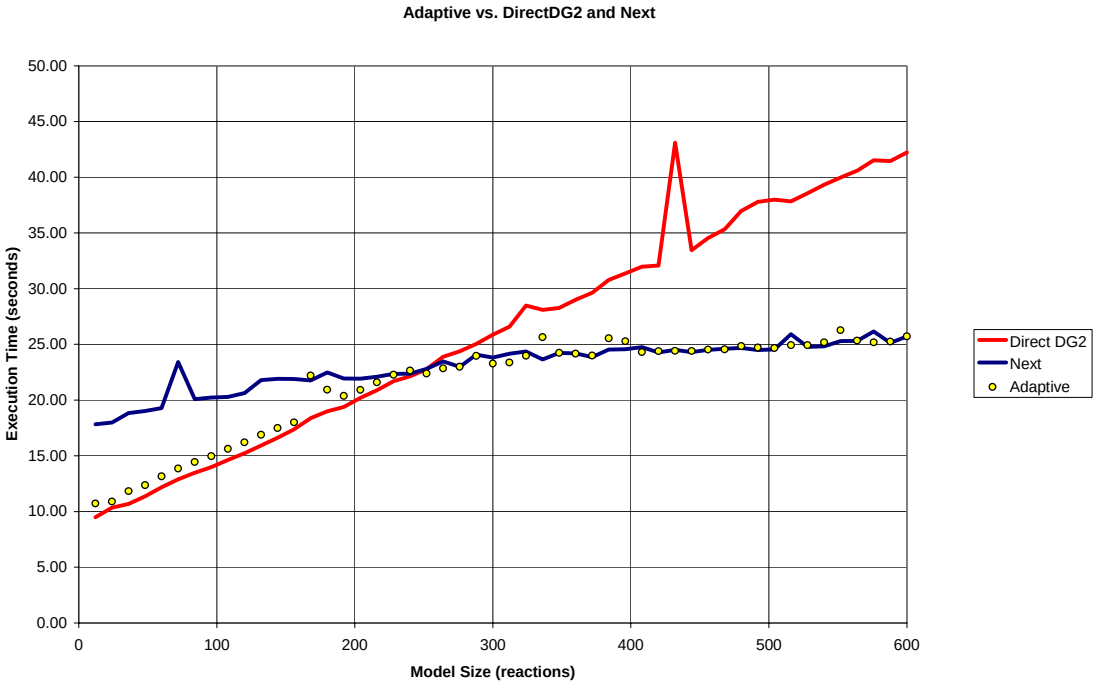

Figure 5-10: *Adaptive Method* vs. *DirectDG2* and *Next* (Update Factor = 6)

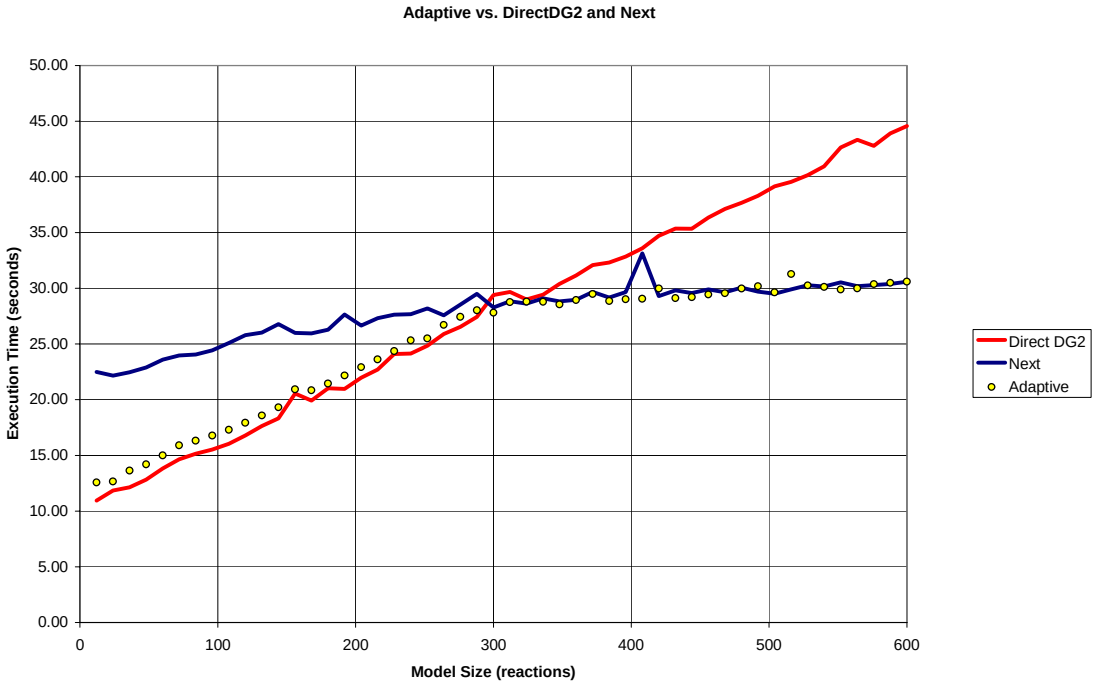

Figure 5-11: *Adaptive Method* vs. *DirectDG2* and *Next* (Update Factor = 8)

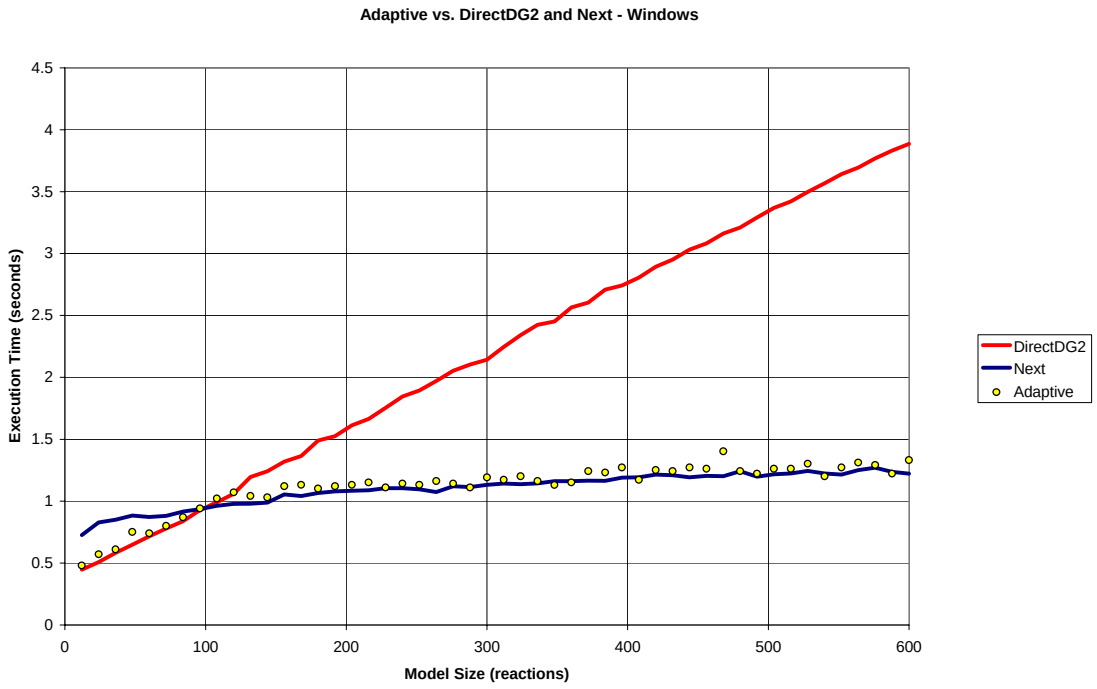

Figure 5-12: Windows - *Adaptive Method* vs. *DirectDG2* and *Next* (UF = 2)

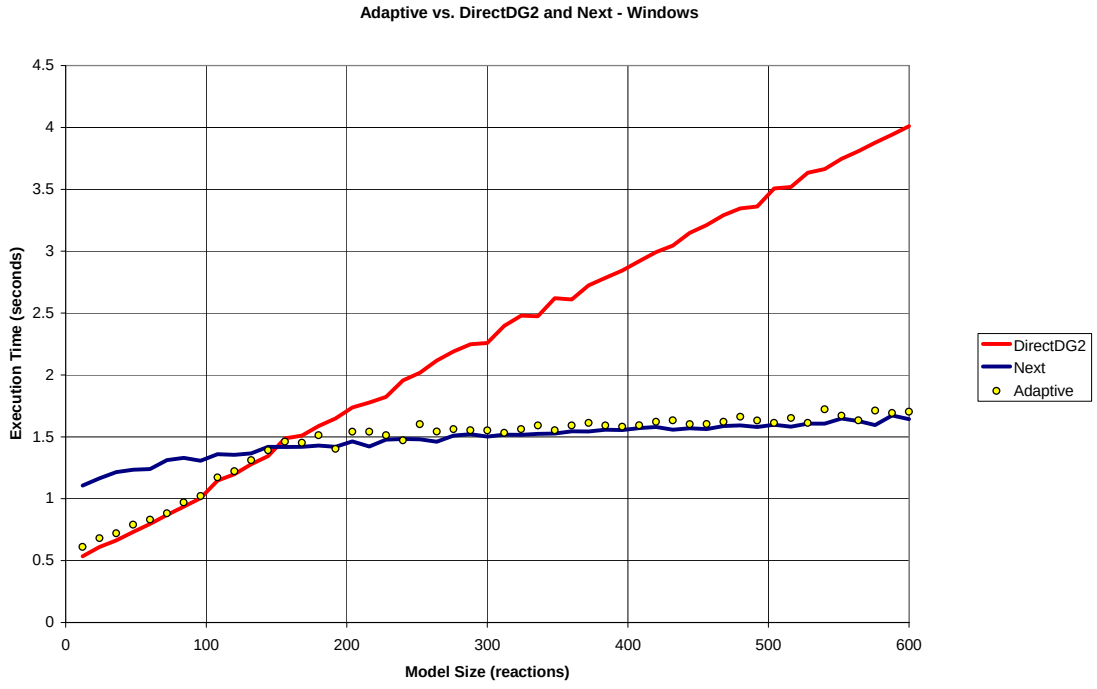

Figure 5-13: Windows - *Adaptive Method* vs. *DirectDG2* and *Next* (UF = 4)

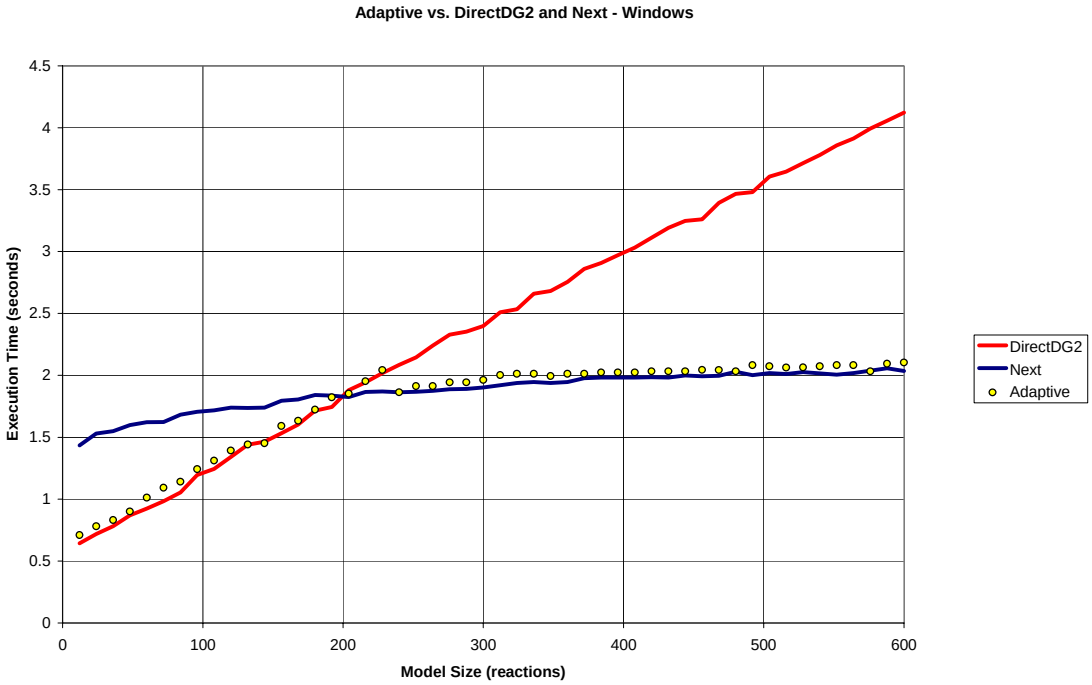

Figure 5-14: Windows - *Adaptive Method* vs. *DirectDG2* and *Next* (UF = 6)

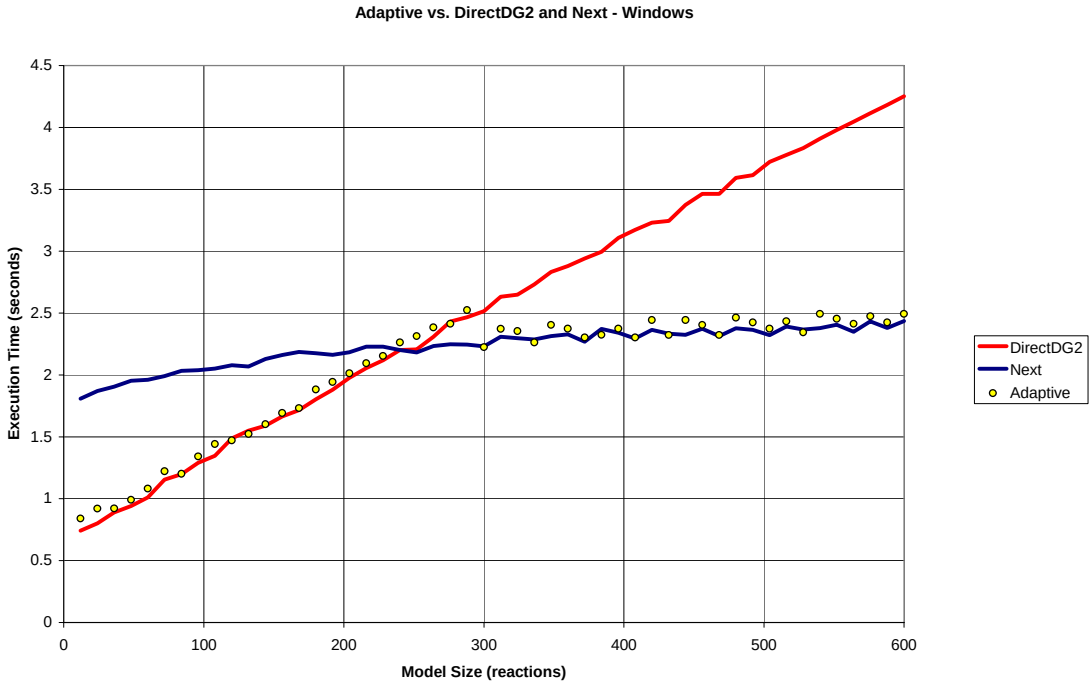

Figure 5-15: Windows - *Adaptive Method* vs. *DirectDG2* and *Next* (UF = 8)

From these figures, we observe that our prediction of which algorithm to use for a particular model is inaccurate when using a different computer. This is most apparent in Figure 5-15, where the ideal simulation algorithm for models of size greater than 250 would be the *Next* method, but the *Adaptive Method* chooses to continue to use the *DirectDG2* method up to 300 reactions. To fix this, we recommend creating a program that runs on the computer during installation of the *Adaptive Method* simulator that characterizes the performance of the machine and provides a more accurate estimate of the best update factor threshold.

5.4 Implications

At this point, some may argue that this work is insignificant because in the future, modelers will want to run models with such high reaction counts that the *Adaptive Method* will always choose to use the *Next* simulation algorithm. Although this point is valid, the key to modeling in the future may be to run smaller models exhaustively and build macromodels that characterize the performance of a specific gene regulatory networks, then combine these macromodels to form larger models. For this reason, it may be very useful to have a simulator that runs very quickly for models with a small reaction count.

5.5 Applying the *Adaptive Method* to Real Models

To validate that the *Adaptive Method* performs well for real models, we now execute a set of four biological models developed by Dr. Chris Cox at the University of

Tennessee. The first, called *ENG*, is a model of an engineered bioreporter, consisting of 32 species and 49 reactions. The second model, *TB*, consists of 17 species and 23 reactions and models tuberculosis. The third model, *QS8*, is a model of quorum sensing in eight *Vibrio fischeri* cells. *QS8* consists of 122 species and 201 reactions. The final model, *DIMER*, is a simple model of a gene whose protein undergoes dimerization. This model consists of 8 species and 13 reactions. An SBML descriptions of these models are supplied in Appendix C.

The *Adaptive*, *DirectDG2*, and *Next* simulators are run on the same Sun machine used in chapter 4. Each model is run for 5,000,000 reactions. The results of averaging 4 runs are given in figure 5-16. The data shows that for models *DIMER* and *TB*, the simulator with the smallest execution time is *DirectDG2*. The data also shows that *Next* is the optimal simulator for executing the *QS8* and *ENG* models. As expected, the *Adaptive* simulator closely matches this performance of the best performing simulator for each model. Performing such analysis on real models also helps to validate the assumptions made when analyzing simulator performance on the artificial model set generated in chapter 4.

5.6 Conclusion

By monitoring update factor as the simulation progresses, the *Adaptive Method* attempts to select either the *DirectDG2* or *Next* simulation algorithms for predicting the time-evolution of the chemical species in a particular model. By testing the performance of the *Adaptive Method* on the artificial model set developed in chapter 4 and on several

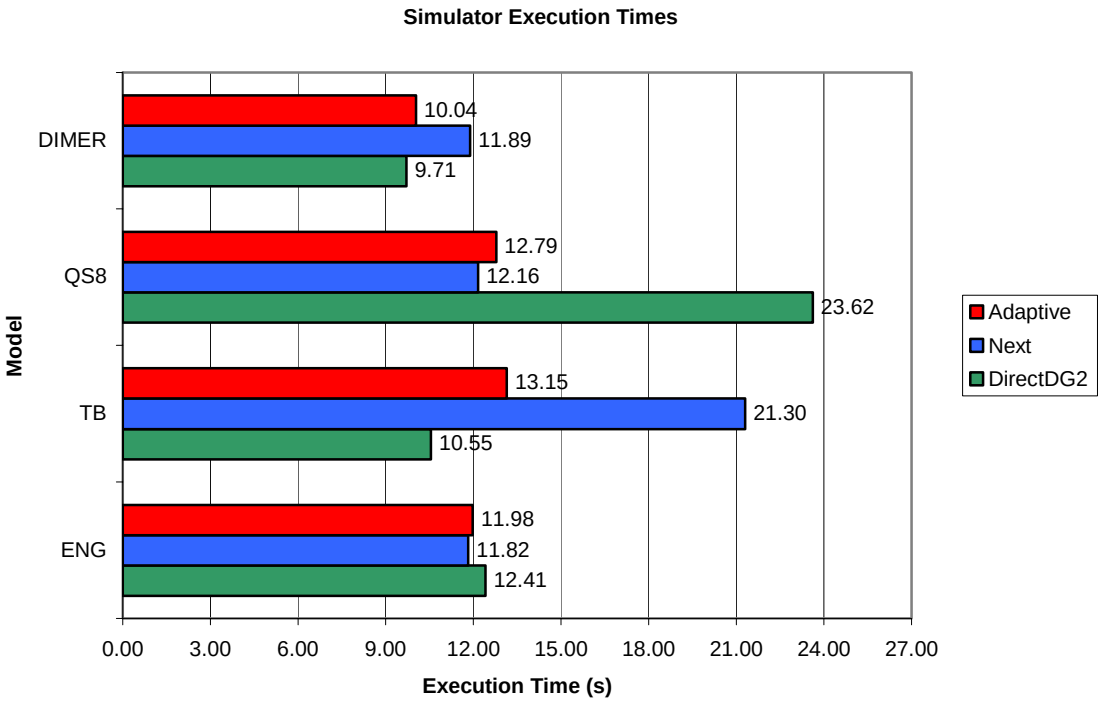

Figure 5-16: Real Model Execution Times

real gene regulatory network models, it demonstrates that the *Adaptive Method* closely matches the performance of the optimal algorithm. To tune the performance of the simulator on a particular machine, it is recommended that the software run a suite of models to determine the points at which the simulator should use a particular algorithm.

Chapter 6

Fast Propensity Calculation

In this chapter, we investigate another possible technique for accelerating Gillespie's stochastic simulation methods by modifying the propensity calculation step to reduce the number of multiplications that need to be performed per iteration of the algorithm's main loop. Several simulators are developed to measure the performance of this technique and the results are compared to the execution times of the original Gillespie simulators. Finally, the applicability of this technique is discussed with regards to accelerating both software and hardware-accelerated simulators.

6.1 Propensity Calculation

Recall from the discussions of Gillespie's algorithms in chapter three that in the *First Reaction Method*, the reaction propensity is the exponential distribution parameter used when estimating the putative time for each reaction. In the *Direct Method*, the reaction propensity values are summed to produce the total propensity for the entire chemical system, which is then used to estimate the putative time for the entire system. In either case, the propensities for each reaction must be recalculated for each iteration of the algorithm's main loop. Gibson and Bruck's dependency graph enhancement significantly reduces the number of propensity calculations that must be performed, but at least a few propensities still must be recalculated for each loop iteration.

Calculation of the propensity values is performed by taking the rate constant for the reaction and multiplying it by the number of possible ways the reaction could occur

given the current populations of the chemical species. Since Gillespie's methods only use simple chemical equations consisting of at most two reactants, only three possible cases exist for the composition of the reactants:

```
Type 1:   A -> Products
Type 2:   A + B -> Products
Type 3:   A + A -> Products
```

To calculate the propensity a_1 of a type 1 reaction, we multiply the rate constant for the reaction k by the current species population of A. This is given in the equation below.

$$a_1(t) = kA(t) \tag{1}$$

To calculate the propensity a_2 of a type 2 reaction, we multiply the rate constant for the reaction k by the current species population of A and the current species population of B. This is because any molecule of A can react with any molecule of B, therefore by combinatorics, the propensity equation would be

$$a_2(t) = kA(t)B(t) \tag{2}$$

The calculation of the propensity a_3 of a type 3 reaction may be expected to look similar to equation 2, but we must remember that a reaction can not occur between a

given molecule and itself. Therefore we determine the equation for a type 3 reaction using combinatorics as

$$a_3(t) = kA(t)\frac{(A(t)-1)}{2} \quad (3)$$

Using these three equations we can calculate the propensity for a given reaction based on the current state of the chemical system.

6.2 Fast Propensity Calculation

To calculate the propensity values more quickly, we notice that for a given system state we can compute the current propensity values at time *t* based on the values for the previous state at time t-t_0, where t_0 is the putative time of the last reaction executed.

Suppose that at time t-t_0, we executed a reaction *j* that changes the value of the species population *A(t)* by some value ΔA_j. We could represent this relation using the following equation.

$$A(t) = A(t - t_0) + \Delta A_j \quad (4)$$

Suppose *A(t)* affected the propensity value of a type 1 reaction *i*. We could calculate the propensity of reaction *i* using equation 1. We could also substitute equation 4 into equation 1 to give us

$$a_i(t) = k_i(A(t - t_0) + \Delta A_j) \quad (5)$$

Rearranging equation 5 and realizing that $a_i(t - t_0) = k_i A(t - t_0)$, we can write the following equation

$$a_i(t) = a_i(t - t_0) + k_i \Delta A_j \quad (6)$$

When using Gibson and Bruck's enhancements to Gillespie's methods, we use a dependency graph to determine the value for ΔA_j for each reaction. To utilize fast propensity calculation we can precompute the value of $k_i \Delta A_j$ during this initialization step and use it to calculate the propensity value of reaction *i* each time reaction *j* is executed. Notice that using equation 6 to calculate the propensity requires that we only perform a single addition for each iteration of the algorithm, which is more efficient than the multiplication that would have been performed using equation 1.

Similarly we can apply this technique to a type 2 chemical equation. In this scenario, three different situations exist: reaction *j* could change the value of the species population *A(t)* by some value ΔA_j, reaction *j* could change the value of the species population *B(t)* by some value ΔB_j, or reaction *j* could change the value of both species populations. In the first case,

$$B(t) = B(t - t_0) \quad (7)$$

and

$$A(t) = A(t - t_0) + \Delta A_j \tag{8}$$

Substituting equations 7 and 8 into equation 2, gives us.

$$a_i(t) = k_i(A(t - t_0) + \Delta A_j)B(t - t_0) \tag{9}$$

Rearranging equation 9 and realizing that $a_i(t - t_0) = k_i A(t - t_0)B(t - t_0)$, we can write the following equation

$$a_i(t) = a_i(t - t_0) + k_i \Delta A_j B(t) \tag{10}$$

Once again, we can precompute and store the value of $k\Delta A_j$ during the initialization step. Using equation 10 to calculate the propensity therefore uses one multiplication and one addition, instead of two multiplications used by equation 2.

Using a similar derivation, we can determine that in the second case, where the species population of *B* changes and the species population of *A* remains the same during the execution of reaction *j*, we can simplify the calculation of the propensity to the following equation.

$$a_i(t) = a_i(t - t_0) + k_i \Delta B_j A(t) \tag{11}$$

Once again we precompute and store the values we can determine during initialization, $k\Delta B_j$ and reduce one of the multiplication steps in equation 2 to an addition.

In the third case, where both the species populations of A and B change because of the reaction *j*, we can try to apply the same multiplication reduction techniques, but we find that we must do the same number of multiplications as we would have using equation 2. Therefore when both A and B change, we still use equation 2.

A type 3 chemical reaction can also be reduced by a single multiply. If the species population of A changes, we get the following equation.

$$A(t) = A(t - t_0) + \Delta A_j \tag{12}$$

When we substitute this value into equation 3, we get the following equation.

$$a_i(t) = k_i (A(t - t_0) + \Delta A_j) \frac{(A(t - t_0) + \Delta A_j - 1)}{2} \tag{13}$$

Simplifying this equation, and remembering that

$$a_i(t) = k_i A(t - t_0) \frac{(A(t - t_0) - 1)}{2} \tag{14}$$

we can simplify equation 13 to the following equation.

$$a_i(t) = a_i(t - t_0) + k_i \Delta A_j A(t) - \frac{k_i}{2}(\Delta A_j^2 + \Delta A_j) \quad (15)$$

We can precompute and store the values of $k_i \Delta A_j$ and $-k(\Delta A_j^2 + \Delta A_j)/2$ during initialization and reduce the one addition and two multiplications using in equation 3 to two additions and one multiplication.

6.3 Implementation Details

Because floating point numbers are an approximation of real numbers, subtracting or adding two values may produce a small error in the least significant digits of the result. When using equations 6, 10, 11, and 15 to calculate propensity values for species populations that have recently fallen to zero, the propensity values can become non-zero, which is incorrect. This error can then cause the simulator to execute a reaction that does not have a sufficient number of reactants and possibly cause species populations to fall below zero.

To avoid this, an implemention of fast propensity calculation should check to ensure that the reactants' population values are non-zero. If they are zero, the propensity value should be immediately set to zero. In the case of a type 3 reaction, if the reactant's population value is 1, the propensity value should be set to zero, because this type of reaction requires the reactant's population to be at least 2.

6.4 Performance Analysis

To analyze the performance of this enhancement, two new simulators are coded in C++ that implement fast propensity calculation. The first simulator is the Next Reaction Method with Fast Propensity Calculation simulator, or *NextFPC*, which is derived from the *Next* simulator defined in chapter 3. The second simulator is the Direct Method with Fast Propensity Calculation simulator, or *DirectFPC*, which is derived from the *DirectDG2* simulator. The performance of these complementary simulators are compared by executing the same set of model files under the same conditions as the analysis performed in chapter 4. Plots of these execution times for these simulators are given in figures 6-1 and 6-2.

A more effective way to view figures 6-1 and 6-2 would be to plot the speedup, which is ratio of the execution time of the enhanced simulator (i.e. *NextFPC*) and the execution time of the original simulator (i.e. *Next*). A speedup of greater than 1.0 shows that the enhancement improves simulator performance and a speedup of less than 1.0 shows that the enhancement degrades simulator performance. Speedup plots are given in figures 6-3 and 6-4.

Figures 6-3 and 6-4 show that by using fast propensity calculation, we see no performance improvement. The several points that show significant speedup are outliers caused by machine load imbalance. These speedup results are consistent with results gathered from running models with other update factors. Even though fast propensity calculation reduces many multiplications to additions, the overhead associated with storing and retrieving the parameters values required to do fast propensity calculation outweighs this improvement.

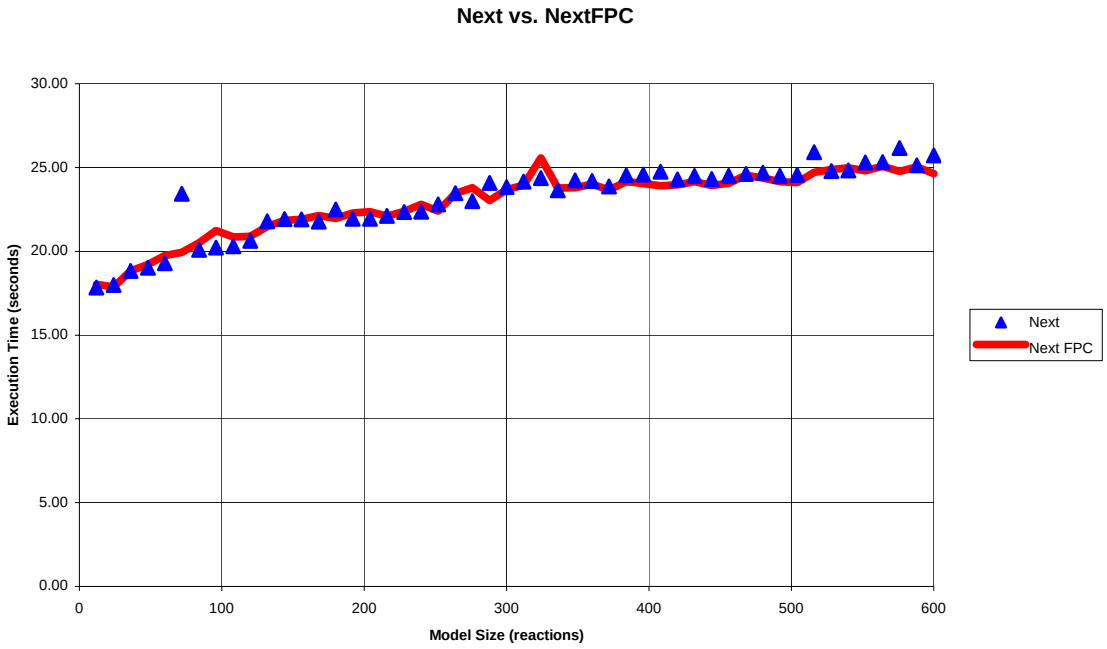

Figure 6-1: Execution Times for *Next* and *NextFPC* (Update Factor = 6)

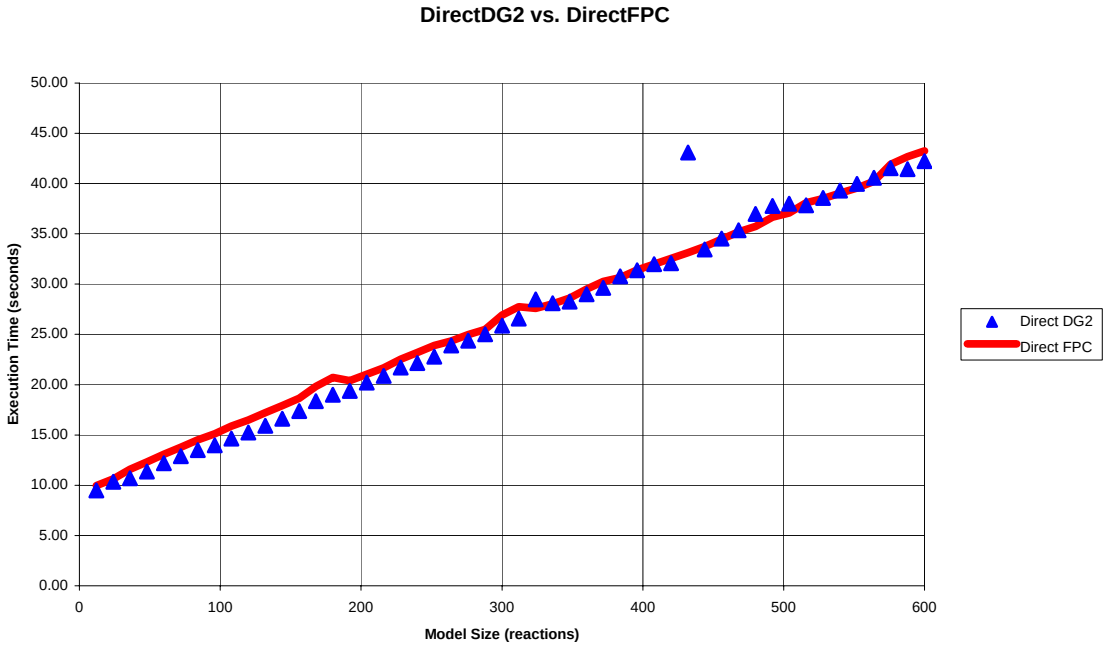

Figure 6-2: Execution Times for *DirectDG2* and *DirectFPC* (UF = 6)

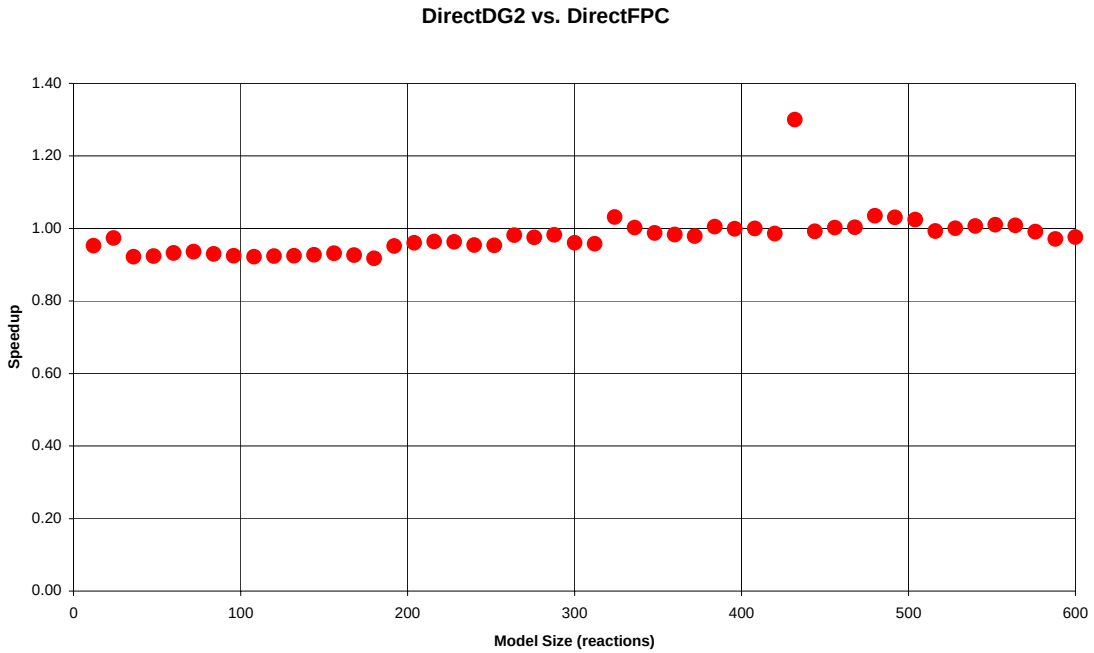

Figure 6-3: Speedup for *DirectDG2 vs. DirectFPC* (Update Factor = 6)

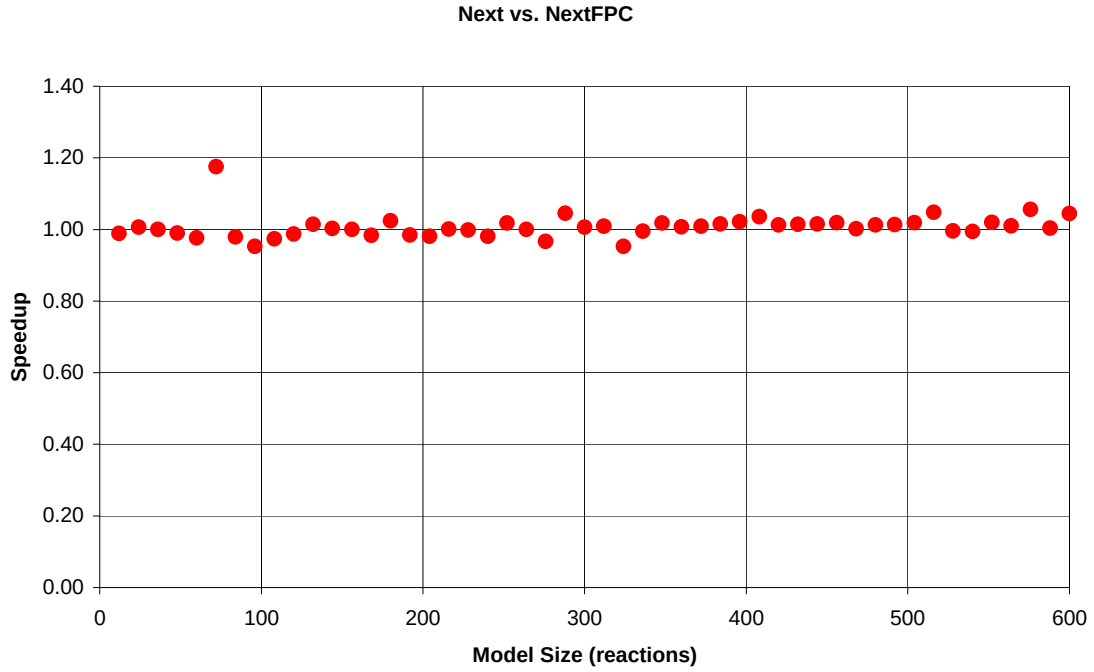

Figure 6-4: Speedup for *NextFPC* vs. *Next* (Update Factor = 6)

6.5 Implications

Although the software version of fast propensity calculation shows no significant gain versus the original simulation algorithms, this work is significant when discussing the possibility of implementing Gillespie's algorithms in hardware. New computer architectures are currently being developed which allow reconfigurable hardware co-processors to exist near or on the microprocessor with fast interconnect. Because the reconfigurable hardware devices can be customized and parallelized to meet the specific need of the software problem, they can demonstrate a significant performance enhancement over simple microprocessor implementations. These algorithms could be further accelerated by developing a custom application specific integrated circuit that performs the algorithm. In both situations, using the fast propensity calculation techniques described in this chapter could be used to reduce both the size and latency of the hardware, because an addition module uses less area and delay than a multiplication module.

6.6 Conclusion

The fast propensity calculation techniques presented in this chapter are an effective alternative to the propensity calculation techniques proposed by Gillespie's original algorithms. Although these techniques may not show significant performance improvement for software versions of the simulators, they could be used to reduce the area and delay of a hardware accelerated version of the simulator significantly.

Chapter 7

Conclusions and Future Work

Gene regulatory networks are the complex chemical interactions between biochemicals that allow cells to control their behavior and sustain life. A detailed understanding of gene regulatory networks is critical to the development of new gene therapy techniques and pharmaceuticals. One tool that has already been successful in helping to analyze gene regulatory networks is computer modeling. Cell models can be used to predict cell behavior, validate published results, test treatment strategies, and help develop nanoscale biological tools.

One of the major factors limiting the widespread use of modeling in the biological community is the efficiency and accuracy of the algorithms used to simulate such models. This work has focused on analyzing and enhancing the performance of the exact stochastic simulation algorithms developed by Gillespie and by Gibson and Bruck. These techniques are widely accepted as being accurate, but are normally not used because they are inefficient.

To analyze the performance of the various simulation algorithms, an artificial model set was developed that allowed the simulators to produce consistent and predictable performance results. Through this analysis it was discovered that by modifying Gillespie's *Direct Method*, we could outperform the *Next Reaction Method* for models with high update factor and low reaction count. This discovery is contrary to the belief that the *Next Reaction Method* is the most efficient method for exactly simulating all stochastic biological models .

Also through this analysis of the artificial model set, we were able to characterize the set of models that would perform well for a particular model. To monitor these factors and adaptively control the simulation algorithm used to simulate a particular model, the *Adaptive Method* was developed. This method performed similarly to the best performing algorithm for each model in the artificial set. To validate the assumptions made in developing the artificial model set and to validate that the *Adaptive Method* performed well under real conditions, a set of actual gene regulatory network models were analyzed. The performance results showed that the *Adaptive Method* is an efficient technique for performing stochastic simulation of coupled chemical reactions.

In an attempt to further enhance performance, a new technique called *Fast Propensity Calculation* was developed to reduce several multiplications to additions. When implemented, this technique did not show significant performance improvement over existing methods in software, but it is possible that *Fast Propensity Calculation* could be used to reduce the area and delay of custom hardware accelerators for stochastic simulation. This hardware acceleration approach is an area of further research.

Further analysis could also be performed to examine adaptively controlling the `PERIOD` variable in the *Adaptive Method* to further improve performance. The *Adaptive Method* could also possibly combine the use of new approximate techniques like Gillespie's tau-leaping algorithm to further improve performance. Investigations into using parallel algorithms or distributed computing to accelerate these simulation algorithms would also be useful.

The development of the *Adaptive Method* and *Fast Propensity Calculation* serve as additional steps in the creation of an accurate and efficient platform for the development of computer models for gene regulatory networks.

REFERENCES

References

[1] D. Pederson, “A Historical View of Circuit Simulation,” *IEEE Transactions on Circuits and Systems*, vol. 31, pp. 103-111, 1984.

[2] Wall, Hlavacek and Savageau, “Design of gene circuits: Lessons from bacteria,” *Nature Reviews Genetics*, vol. 5, pp. 34-42, 2004.

[3] H. De Jong, “Modeling and Simulation of Genetic Regulatory Systems: A Literature Review,” *Journal of Computational Biology*, vol. 9, pp. 67-103, 2002.

[4] Endy, D., and R. Brent. 2001. Modeling cellular behaviour. *Nature*, vol. 409, pp. 391-395.

[5] D. Gillespie, “Exact Stochastic Simulation of Coupled Chemical Reactions,” *Journal of Physical Chemistry*, vol. 81, pp. 2340-2361, 1977.

[6] M. Gibson and J. Bruck, “Efficient Exact Stochastic Simulation of Chemical Systems with Many Species and Many Channels,” *Journal of Physical Chemistry A*, vol. 104, pp. 1876-1889, 2000.

[7] K. Talaro and A. Talaro, *Foundations in Microbiology: Basic Principles*. Boston: McGraw Hill, 2002.

[8] T. Brock, D. Smith, and M. Madigan, *Biology of Microorganisms*. Englewood Cliffs: Prentice Hall, 1984.

[9] J. Setubal and J Meidanis, *Introduction to Computational Molecular Biology*. Boston: PWS, 1997.

[10] S. James, P. Nilsson, G. James, S. Kjelleberg, and T. Fagerstrom, "Luminescence Control in the Marine Bacterium Vibrio fischeri: An Analysis of the Dynamics of lux Regulation," *Journal of Molecular Biology*, vol. 296, pp. 1127-1137, 2000.

[11] J. Dockery and J. Kenner, “A mathematical model for quorum sensing in Pseudomonas aeruginosa,” *Bulletin of Mathematical Biology*, vol. 63, pp. 95-116, 2001.

[12] G. Marlovits, C. J. Tyson, B. Novak and J. J. Tyson, “Modeling M-phase control in Xenopus oocyte extracts: the surveillance mechanism for unreplicated DNA,” *Biophysical Chemistry*, vol. 72, pp. 169-184, 1998.

[13] B. Novak and J. J. Tyson. “Modeling the Cell-Division Cycle - M-Phase Trigger, Oscillations, and Size Control,” *Journal of Theoretical Biology*, vol. 165, pp. 101-134, 1993.

[14] W. Sha, J. Moore, K. Chen, A. D. Lassaletta, C. S. Yi, J. J. Tyson, and J. C. Sible, "Hysteresis drives cell-cycle transitions in Xenopus laevis egg extracts," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 100, pp. 975-980, 2003

[15] A. Arkin, J. Ross, and H. H. McAdams, "Stochastic kinetic analysis of developmental pathway bifurcation in phage lambda-infected Escherichia coli cells," *Genetics*, vol. 149, pp. 1633-1648, 1998.

[16] C. D. Cox, G. D. Peterson, M. S. Allen, J. M. Lancaster, J. M. McCollum, D. Austin, L. Yan, G. S. Sayler, and M. L. Simpson, "Analysis of Noise in Quorum Sensing," *OMICS: A Journal of Integrative Biology*, vol. 9, pp. 317-334, 2003.

[17] Endy, D., D. Kong, and J. Yin, "Intracellular kinetics of a growing virus: A genetically structured simulation for bacteriophage T7," *Biotechnology and Bioengineering*, vol. 55, pp. 375-389, 1997.

[18] D. Rapaport, *The Art of Molecular Dynamics Simulation*. Cambridge: Cambridge Press, 1995.

[19] M. Allen and D. Tildesley, *Computer Simulation of Liquids*. Oxford: Clarendon Press, 1987.

[20] D. Gillespie, "Approximate accelerated stochastic simulation of chemically reacting systems," *Journal of Chemical Physics*, vol. 115, pp. 1716-1733, 2001.

[21] D. Gillespie and L. Petzold, "Improved leap-size selection for accelerated stochastic simulation," *Journal of Chemical Physics*, vol. 119, pp. 8229-8234, 2003.

APPENDICES

Appendix A – Simulator Source Code

The following pages include the source code used to generate each of the simulators.

Adaptive.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"
#include "DependencyGraph.h"
#include "MinHeap.h"
#include <iostream>

using namespace std;

#define NEXT 1
#define DIRECT 2

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);
  DependencyGraph dependencyGraph(model);
  MinHeap minHeap;

  double currentTime = 0.0;
  double *propensities = new double[model.reactionCount];
  double *putativeTimes = new double[model.reactionCount];
  double totalPropensity = 0.0;
  int i;
  long reactionCount = 0;
  long updateCount = 0;
  int selectedReaction;

  double updateFactorThreshold = ((double)(model.reactionCount-
60))/30.0/2.0;
  int mode = NEXT;

  for(i=0; i<model.reactionCount; i++) {
    propensities[i] = model.reactions[i].getPropensity();
    putativeTimes[i] = random.getExponential()/propensities[i];
    minHeap.add(putativeTimes[i]);
  }

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    if (mode == DIRECT) {
      double scaledRand = totalPropensity * random.getUniform();
      for(i=0; i<model.reactionCount; i++) {
      if (propensities[i] != 0.0) {
        selectedReaction = i;
        scaledRand -= propensities[i];
        if (scaledRand <= 0.0) break;
      }
      }

      model.reactions[selectedReaction].execute();
```

```
      currentTime += random.getExponential()/totalPropensity;

      IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
      updateCount += dependencies->size();
      for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      totalPropensity -= propensities[index];
      propensities[index] = model.reactions[index].getPropensity();
      totalPropensity += propensities[index];
      }
    } else {
      selectedReaction = minHeap.getMin();

      model.reactions[selectedReaction].execute();

      currentTime = putativeTimes[selectedReaction];

      IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
      updateCount += dependencies->size();
      for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      double oldPropensity = propensities[index];
      propensities[index] = model.reactions[index].getPropensity();
      if (index != selectedReaction) {
        if (oldPropensity == 0.0) {
          putativeTimes[index] = currentTime +
random.getExponential()/propensities[index];
        } else {
          putativeTimes[index] = currentTime + oldPropensity /
propensities[index] * (putativeTimes[index] - currentTime);
        }
        minHeap.update(index,putativeTimes[index]);
      }
      }
      putativeTimes[selectedReaction] = currentTime +
random.getExponential()/propensities[selectedReaction];
      minHeap.update(selectedReaction,putativeTimes[selectedReaction]);
    }

    if (parameters.output) model.printState(currentTime);

    reactionCount++;
    if (reactionCount == 5000) {
      double updateFactor =
(double)updateCount/(double)reactionCount/2.0;

      if ((updateFactor > updateFactorThreshold) && (mode == NEXT)) {
      mode = DIRECT;
      totalPropensity = 0.0;
      for(i=0; i<model.reactionCount; i++) {
        totalPropensity += propensities[i];
      }
```

```
      } else if ((updateFactor < updateFactorThreshold) && (mode ==
DIRECT)) {
      mode = NEXT;
      for(i=0; i<model.reactionCount; i++) {
        putativeTimes[i] = currentTime +
random.getExponential()/propensities[i];
        minHeap.update(i,putativeTimes[i]);
      }
      }

      updateCount = 0;
      reactionCount = 0;
    }
  }
}
```

DependencyGraph.h

```
#ifndef DEPENDENCYGRAPH_H
#define DEPENDENCYGRAPH_H

#include "IntVector.h"
#include "Model.h"

class DependencyGraph {
 private:
  IntVector *dependencies;

 public:
  DependencyGraph(const Model &model);
  IntVector* getDependencies(int reactionIndex);
};

#endif
```

DependencyGraph.cc

```
#include "DependencyGraph.h"

DependencyGraph::DependencyGraph(const Model &model) {
  dependencies = new IntVector[model.reactionCount];

  for(int i=0; i<model.reactionCount; i++) {
    Reaction *reaction1 = &model.reactions[i];
    for(int j=0; j<model.reactionCount; j++) {
      Reaction *reaction2 = &model.reactions[j];
      for(int k=0; k<reaction1->reactantCount; k++) {
      int reactantIndex = reaction1->reactants[k].speciesIndex;
      int change = 0;
      int z;
      for(z=0; z<reaction2->reactantCount; z++) {
        if (reaction2->reactants[z].speciesIndex == reactantIndex) {
          change -= reaction2->reactants[z].coefficient;
        }
      }
      for(z=0; z<reaction2->productCount; z++) {
        if (reaction2->products[z].speciesIndex == reactantIndex) {
          change += reaction2->products[z].coefficient;
        }
      }
      if (change != 0) {
        dependencies[j].add(i);
        break;
      }
      }
    }
  }
}

IntVector* DependencyGraph::getDependencies(int reactionIndex) {
  return &dependencies[reactionIndex];
}
```

Direct.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);

  double currentTime = 0.0;
  double *propensities = new double[model.reactionCount];
  double totalPropensity = 0.0;
  int i;

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    totalPropensity = 0.0;
    for(i=0; i<model.reactionCount; i++) {
      propensities[i] = model.reactions[i].getPropensity();
      totalPropensity += propensities[i];
    }

    double scaledRand = totalPropensity * random.getUniform();
    int selectedReaction = -1;
    for(i=0; i<model.reactionCount; i++) {
      if (propensities[i] != 0.0) {
      selectedReaction = i;
      scaledRand -= propensities[i];
      if (scaledRand <= 0.0) break;
      }
    }

    model.reactions[selectedReaction].execute();
    currentTime += random.getExponential()/totalPropensity;

    if (parameters.output) model.printState(currentTime);
  }
}
```

DirectDG.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"
#include "DependencyGraph.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);
  DependencyGraph dependencyGraph(model);

  double currentTime = 0.0;
  double *propensities = new double[model.reactionCount];
  double totalPropensity = 0.0;
  int i;

  for(i=0; i<model.reactionCount; i++) {
    propensities[i] = model.reactions[i].getPropensity();
  }

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    totalPropensity = 0.0;
    for(i=0; i<model.reactionCount; i++) {
      totalPropensity += propensities[i];
    }

    double scaledRand = totalPropensity * random.getUniform();
    int selectedReaction = -1;
    for(i=0; i<model.reactionCount; i++) {
      if (propensities[i] != 0.0) {
      selectedReaction = i;
      scaledRand -= propensities[i];
      if (scaledRand <= 0.0) break;
      }
    }

    model.reactions[selectedReaction].execute();
    currentTime += random.getExponential()/totalPropensity;

    if (parameters.output) model.printState(currentTime);

    IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
    for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      propensities[index] = model.reactions[index].getPropensity();
    }
  }
}
```

DirectDG2.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"
#include "DependencyGraph.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);
  DependencyGraph dependencyGraph(model);

  double currentTime = 0.0;
  double *propensities = new double[model.reactionCount];
  double totalPropensity = 0.0;
  int i;

  for(i=0; i<model.reactionCount; i++) {
    propensities[i] = model.reactions[i].getPropensity();
    totalPropensity += propensities[i];
  }

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    double scaledRand = totalPropensity * random.getUniform();
    int selectedReaction = -1;
    for(i=0; i<model.reactionCount; i++) {
      if (propensities[i] != 0.0) {
      selectedReaction = i;
      scaledRand -= propensities[i];
      if (scaledRand <= 0.0) break;
      }
    }

    model.reactions[selectedReaction].execute();
    currentTime += random.getExponential()/totalPropensity;

    if (parameters.output) model.printState(currentTime);

    IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
    for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      totalPropensity -= propensities[index];
      propensities[index] = model.reactions[index].getPropensity();
      totalPropensity += propensities[index];
    }
  }
}
```

DirectFPC.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"
#include "DependencyGraph.h"
#include "FastPropensity.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);
  DependencyGraph dependencyGraph(model);

  double currentTime = 0.0;
  double *propensities = new double[model.reactionCount];
  FastPropensity fastPropensity(model,propensities,&dependencyGraph);
  double totalPropensity = 0.0;
  int i;

  for(i=0; i<model.reactionCount; i++) {
    propensities[i] = model.reactions[i].getPropensity();
    totalPropensity += propensities[i];
  }

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    double scaledRand = totalPropensity * random.getUniform();
    int selectedReaction = -1;
    for(i=0; i<model.reactionCount; i++) {
      if (propensities[i] != 0.0) {
      selectedReaction = i;
      scaledRand -= propensities[i];
      if (scaledRand <= 0.0) break;
      }
    }

    model.reactions[selectedReaction].execute();
    currentTime += random.getExponential()/totalPropensity;

    if (parameters.output) model.printState(currentTime);

    IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
    for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      fastPropensity.update(selectedReaction,i,&totalPropensity);
    }
  }
}
```

EfficientDirect.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"
#include "DependencyGraph.h"
#include "SumTree.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);
  DependencyGraph dependencyGraph(model);
  SumTree sumTree(model.reactionCount);

  double currentTime = 0.0;
  double *propensities = new double[model.reactionCount];
  double totalPropensity = 0.0;
  int i;

  for(i=0; i<model.reactionCount; i++) {
    propensities[i] = model.reactions[i].getPropensity();
    sumTree.update(i,propensities[i]);
  }

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    totalPropensity = sumTree.getSum();
    double scaledRand = totalPropensity * random.getUniform();
    int selectedReaction = sumTree.selectReaction(scaledRand);
    model.reactions[selectedReaction].execute();
    currentTime += random.getExponential()/totalPropensity;

    if (parameters.output) model.printState(currentTime);

    IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
    for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      propensities[index] = model.reactions[index].getPropensity();
      sumTree.update(index,propensities[index]);
    }
  }
}
```

FastPropensity.h

```
#ifndef FASTPROPENSITY_H
#define FASTPROPENSITY_H

#include "FastPropensityEntry.h"
#include "Model.h"
#include "DependencyGraph.h"

class FastPropensity {
 private:
  FastPropensityEntry ***data;
  int *entryCount;
  DependencyGraph *dependencyGraph;

 public:
  FastPropensity(const Model &model, double *propensities,
DependencyGraph *dependencyGraph);
  void update(int selectedReaction, int index);
  void update(int selectedReaction, int index, double
*totalPropensity);
};

#endif
```

FastPropensity.cc

```
#include "FastPropensity.h"
#include "DependencyGraph.h"

FastPropensity::FastPropensity(const Model &model, double
*propensities, DependencyGraph *dependencyGraph) {
  int i,j;
  data = new (FastPropensityEntry **)[model.reactionCount];
  entryCount = new int[model.reactionCount];
  for(i=0; i<model.reactionCount; i++) {
    IntVector *dependencies = dependencyGraph->getDependencies(i);
    data[i] = new (FastPropensityEntry *)[dependencies->size()];
    entryCount[i] = dependencies->size();
    for(j=0; j<dependencies->size(); j++) {
      data[i][j] = new
FastPropensityEntry(&model.reactions[i],&model.reactions[dependencies-
>get(j)],&propensities[dependencies->get(j)]);
    }
  }
}

void FastPropensity::update(int selectedReaction, int index) {
  data[selectedReaction][index]->execute();
}

void FastPropensity::update(int selectedReaction, int index, double
*totalPropensity) {
  data[selectedReaction][index]->execute(totalPropensity);
}
```

FastPropensityEntry.h

```
#ifndef FASTPROPENSITYENTRY_H
#define FASTPROPENSITYENTRY_H

#include "Reaction.h"

class FastPropensityEntry {
 private:
  int type;
  double *propensity;
  Reaction *reaction;
  long *species1;
  long *species2;
  double alpha;
  double beta;

 public:
  FastPropensityEntry(Reaction *executedReaction,
                  Reaction *affectedReaction,
                  double *propensityValue);
  void execute();
  void execute(double *totalPropensity);
};

#endif
```

FastPropensityEntry.cc

```
#include "FastPropensityEntry.h"
#include <stdlib.h>

FastPropensityEntry::FastPropensityEntry(Reaction *executedReaction,
                                Reaction *affectedReaction,
                                double *propensityValue) {
  reaction = affectedReaction;
  propensity = propensityValue;

  if (affectedReaction->reactantCount == 1) {
    if (affectedReaction->reactants[0].coefficient == 1) {
      type = 1;
      species1 = affectedReaction->reactants[0].species;
      int delta = executedReaction->getDelta(affectedReaction-
>reactants[0].speciesIndex);
      alpha = affectedReaction->rate*delta;
    } else if (affectedReaction->reactants[0].coefficient == 2) {
      type = 5;
      species1 = affectedReaction->reactants[0].species;
      int delta = executedReaction->getDelta(affectedReaction-
>reactants[0].speciesIndex);
      alpha = affectedReaction->rate*delta;
      beta = -affectedReaction->rate/2.0*(delta*delta+delta);
    } else {
      type = 4;
    }
  } else if (affectedReaction->reactantCount == 2) {
    if ((affectedReaction->reactants[0].coefficient != 1) ||
      (affectedReaction->reactants[1].coefficient != 1)) {
      type = 4;
    } else {
      int delta1 = executedReaction->getDelta(affectedReaction-
>reactants[0].speciesIndex);
      int delta2 = executedReaction->getDelta(affectedReaction-
>reactants[1].speciesIndex);
      if ((delta1 != 0) && (delta2 == 0)) {
      type = 2;
      species1 = affectedReaction->reactants[0].species;
      species2 = affectedReaction->reactants[1].species;
      alpha = affectedReaction->rate*delta1;
      } else if ((delta1 == 0) && (delta2 != 0)) {
      type = 3;
      species1 = affectedReaction->reactants[0].species;
      species2 = affectedReaction->reactants[1].species;
      alpha = affectedReaction->rate*delta2;
      } else {
      type = 4;
      }
    }
  } else {
    type = 4;
  }
```

```
}

void FastPropensityEntry::execute() {
  if (type == 1) {
    *propensity = *propensity + alpha;
  } else if (type == 2) {
    *propensity = *propensity + alpha*(*species2);
  } else if (type == 3) {
    *propensity = *propensity + alpha*(*species1);
  } else if (type == 4) {
    *propensity = reaction->getPropensity();
  } else {
    *propensity = *propensity + alpha*(*species1) + beta;
  }
}

void FastPropensityEntry::execute(double *totalPropensity) {
  if (type == 1) {
    *propensity += alpha;
    *totalPropensity += alpha;
  } else if (type == 2) {
    double delta = alpha*(*species2);
    *propensity += delta;
    *totalPropensity += delta;
  } else if (type == 3) {
    double delta = alpha*(*species1);
    *propensity += delta;
    *totalPropensity += delta;
  } else if (type == 4) {
    *totalPropensity -= *propensity;
    *propensity = reaction->getPropensity();
    *totalPropensity += *propensity;
  } else {
    double delta = alpha*(*species1) + beta;
    *propensity += delta;
    *totalPropensity += delta;
  }
}
```

First.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);

  double currentTime = 0.0;
  double putativeTime;
  double temp;
  int selectedReaction;
  int i;

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    putativeTime =
random.getExponential()/model.reactions[0].getPropensity();
    selectedReaction = 0;
    for(i=0; i<model.reactionCount; i++) {
      temp =
random.getExponential()/model.reactions[i].getPropensity();
      if (temp < putativeTime) {
      putativeTime = temp;
      selectedReaction = i;
      }
    }

    model.reactions[selectedReaction].execute();
    currentTime += putativeTime;

    if (parameters.output) model.printState(currentTime);
  }
}
```

FirstDG.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"
#include "DependencyGraph.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);
  DependencyGraph dependencyGraph(model);

  double currentTime = 0.0;
  double *propensities = new double[model.reactionCount];
  double putativeTime;
  double temp;
  int selectedReaction;
  int i;

  for(i=0; i<model.reactionCount; i++) {
    propensities[i] = model.reactions[i].getPropensity();
  }

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    putativeTime = random.getExponential()/propensities[0];
    selectedReaction = 0;
    for(i=0; i<model.reactionCount; i++) {
      temp = random.getExponential()/propensities[i];
      if (temp < putativeTime) {
      putativeTime = temp;
      selectedReaction = i;
      }
    }

    model.reactions[selectedReaction].execute();
    currentTime += putativeTime;

    if (parameters.output) model.printState(currentTime);

    IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
    for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      propensities[index] = model.reactions[index].getPropensity();
    }
  }
}
```

IntVector.h

```
#ifndef INTVECTOR_H
#define INTVECTOR_H

class IntVector {
 private:
  int *data;
  int count;

 public:
  IntVector();
  void add(int value);
  int size() const;
  int get(int index) const;
};

#endif
```

IntVector.cc

```
#include "IntVector.h"
#include <stdlib.h>

IntVector::IntVector() {
  count = 0;
  data = NULL;
}

void IntVector::add(int value) {
  int *temp = new int[count+1];
  for(int i=0; i<count; i++) {
    temp[i] = data[i];
  }
  temp[count] = value;
  if (data != NULL) delete data;
  data = temp;
  count++;
}

int IntVector::size() const {
  return count;
}

int IntVector::get(int index) const {
  return data[index];
}
```

makefile

```
all: bin/PrintModel bin/Direct bin/First bin/DirectDG bin/FirstDG
bin/DirectDG2 bin/Next bin/EfficientDirect bin/NextFPC bin/DirectFPC
bin/Adaptive

bin/EfficientDirect: EfficientDirect.cc SpeciesUpdate.o Reaction.o
Model.o Parameters.o Random.o IntVector.o DependencyGraph.o SumTree.o
      c++ -O3 -o bin/EfficientDirect EfficientDirect.cc SpeciesUpdate.o
Reaction.o Model.o Parameters.o Random.o IntVector.o DependencyGraph.o
SumTree.o

bin/Next: Next.cc SpeciesUpdate.o Reaction.o Model.o Parameters.o
Random.o IntVector.o DependencyGraph.o MinHeap.o
      c++ -O3 -o bin/Next Next.cc SpeciesUpdate.o Reaction.o Model.o
Parameters.o Random.o IntVector.o DependencyGraph.o MinHeap.o

bin/Adaptive: Adaptive.cc SpeciesUpdate.o Reaction.o Model.o
Parameters.o Random.o IntVector.o DependencyGraph.o MinHeap.o
      c++ -O3 -o bin/Adaptive Adaptive.cc SpeciesUpdate.o Reaction.o
Model.o Parameters.o Random.o IntVector.o DependencyGraph.o MinHeap.o

bin/DirectDG2: DirectDG2.cc SpeciesUpdate.o Reaction.o Model.o
Parameters.o Random.o IntVector.o DependencyGraph.o
      c++ -O3 -o bin/DirectDG2 DirectDG2.cc SpeciesUpdate.o Reaction.o
Model.o Parameters.o Random.o IntVector.o DependencyGraph.o

bin/DirectDG: DirectDG.cc SpeciesUpdate.o Reaction.o Model.o
Parameters.o Random.o IntVector.o DependencyGraph.o
      c++ -O3 -o bin/DirectDG DirectDG.cc SpeciesUpdate.o Reaction.o
Model.o Parameters.o Random.o IntVector.o DependencyGraph.o

bin/FirstDG: FirstDG.cc SpeciesUpdate.o Reaction.o Model.o Parameters.o
Random.o IntVector.o DependencyGraph.o
      c++ -O3 -o bin/FirstDG FirstDG.cc SpeciesUpdate.o Reaction.o
Model.o Parameters.o Random.o IntVector.o DependencyGraph.o

bin/NextFPC: NextFPC.cc SpeciesUpdate.o Reaction.o Model.o Parameters.o
Random.o IntVector.o DependencyGraph.o FastPropensityEntry.o
FastPropensity.o MinHeap.o
      c++ -O3 -o bin/NextFPC NextFPC.cc SpeciesUpdate.o Reaction.o
Model.o Parameters.o Random.o IntVector.o DependencyGraph.o
FastPropensityEntry.o FastPropensity.o MinHeap.o

bin/DirectFPC: DirectFPC.cc SpeciesUpdate.o Reaction.o Model.o
Parameters.o Random.o IntVector.o DependencyGraph.o
FastPropensityEntry.o FastPropensity.o
      c++ -O3 -o bin/DirectFPC DirectFPC.cc SpeciesUpdate.o Reaction.o
Model.o Parameters.o Random.o IntVector.o DependencyGraph.o
FastPropensityEntry.o FastPropensity.o

bin/Direct: Direct.cc SpeciesUpdate.o Reaction.o Model.o Parameters.o
Random.o
```

```
	c++ -O3 -o bin/Direct Direct.cc SpeciesUpdate.o Reaction.o
Model.o Parameters.o Random.o

bin/First: First.cc SpeciesUpdate.o Reaction.o Model.o Parameters.o
Random.o
	c++ -O3 -o bin/First First.cc SpeciesUpdate.o Reaction.o Model.o
Parameters.o Random.o

bin/PrintModel: PrintModel.cc SpeciesUpdate.o Reaction.o Model.o
	c++ -O3 -o bin/PrintModel PrintModel.cc SpeciesUpdate.o
Reaction.o Model.o

SumTree.o: SumTree.h SumTree.cc
	c++ -O3 -o SumTree.o -c SumTree.cc

IntVector.o: IntVector.h IntVector.cc
	c++ -O3 -o IntVector.o -c IntVector.cc

DependencyGraph.o: DependencyGraph.h DependencyGraph.cc IntVector.h
	c++ -O3 -o DependencyGraph.o -c DependencyGraph.cc

MinHeap.o: MinHeap.h MinHeap.cc
	c++ -O3 -o MinHeap.o -c MinHeap.cc

Random.o: Random.h Random.cc
	c++ -O3 -o Random.o -c Random.cc

Parameters.o: Parameters.h Parameters.cc
	c++ -O3 -o Parameters.o -c Parameters.cc

SpeciesUpdate.o: SpeciesUpdate.h SpeciesUpdate.cc
	c++ -O3 -o SpeciesUpdate.o -c SpeciesUpdate.cc

Reaction.o: Reaction.h Reaction.cc SpeciesUpdate.h
	c++ -O3 -o Reaction.o -c Reaction.cc

Model.o: Model.h Model.cc Reaction.h
	c++ -O3 -o Model.o -c Model.cc

FastPropensityEntry.o: FastPropensityEntry.h FastPropensityEntry.cc
Reaction.h
	c++ -O3 -o FastPropensityEntry.o -c FastPropensityEntry.cc

FastPropensity.o: FastPropensity.h FastPropensity.cc
FastPropensityEntry.h Model.h DependencyGraph.h
	c++ -O3 -o FastPropensity.o -c FastPropensity.cc
```

MinHeap.h

```
#ifndef MINHEAP_H
#define MINHEAP_H

class MinHeap {
 private:
  class Node;

 public:
  MinHeap();
  ~MinHeap();
  int add(double value);
  void update(int id, double value);
  int getMin() const;

 private:
  void add(Node *n, Node *tree);
  void swap(int id1, int id2);
  void update_aux(Node *n);
  int nodeCount(Node *tree) const;

 private:
  Node *d_top;
  int d_nodeCount;
  Node **d_nodes;
};

class MinHeap::Node {
 public:
  Node *left;
  Node *right;
  Node *parent;
  double value;
  int id;
};


#endif
```

MinHeap.cc

```
#include "MinHeap.h"
#include <sstream>

MinHeap::MinHeap() {
  d_top = NULL;
  d_nodeCount = 0;
}

MinHeap::~MinHeap() {
  int i;
  for (i=0; i<d_nodeCount; i++) {
    delete d_nodes[i];
  }
}

int MinHeap::add(double value) {
  Node *n = new Node();
  n->id = d_nodeCount;
  n->value = value;
  n->left = NULL;
  n->right = NULL;
  n->parent = NULL;

  Node **temp = new (Node *)[d_nodeCount+1];
  for(int i=0; i<d_nodeCount; i++) {
    temp[i] = d_nodes[i];
  }
  temp[d_nodeCount] = n;
  if (d_nodeCount != 0) {
    delete d_nodes;
  }
  d_nodes = temp;
  d_nodeCount++;

  if (d_top == NULL) {
    d_top = n;
  } else {
    add(n,d_top);
  }
  return d_nodeCount-1;
}

void MinHeap::update(int id, double value) {
  d_nodes[id]->value = value;
  update_aux(d_nodes[id]);
}

int MinHeap::getMin() const {
  return d_top->id;
}

void MinHeap::update_aux(Node *n) {
```

```
  if ((n->parent != NULL) && (n->value < n->parent->value)) {
    swap(n->id,n->parent->id);
    update_aux(n->parent);
  }

  Node *minchild = NULL;
  if ((n->left != NULL) && (n->right != NULL)) {
    if (n->left->value < n->right->value) {
      minchild = n->left;
    } else {
      minchild = n->right;
    }
  } else if (n->left != NULL) {
    minchild = n->left;
  } else if (n->right != NULL) {
    minchild = n->right;
  } else {
    return;
  }

  if (minchild->value < n->value) {
    swap(minchild->id,n->id);
    update_aux(minchild);
  }
}

int MinHeap::nodeCount(Node *tree) const {
  if (tree == NULL) {
    return 0;
  } else {
    return 1+nodeCount(tree->left)+nodeCount(tree->right);
  }
}

void MinHeap::add(Node *n, Node *tree) {
  if (tree->left == NULL) {
    tree->left = n;
    n->parent = tree;
    update(n->id,n->value);
    return;
  }
  if (tree->right == NULL) {
    tree->right = n;
    n->parent = tree;
    update(n->id,n->value);
    return;
  }
  int leftCount = nodeCount(tree->left);
  int rightCount = nodeCount(tree->right);
  if (leftCount <= rightCount) {
    add(n,tree->left);
  } else {
    add(n,tree->right);
  }
}
```

```
void MinHeap::swap(int id1, int id2) {
  double temp = d_nodes[id1]->value;
  d_nodes[id1]->value = d_nodes[id2]->value;
  d_nodes[id2]->value = temp;
  d_nodes[id1]->id = id2;
  d_nodes[id2]->id = id1;
  Node *n = d_nodes[id1];
  d_nodes[id1] = d_nodes[id2];
  d_nodes[id2] = n;
}
```

Model.h

```
#ifndef MODEL_H
#define MODEL_H

#include "Reaction.h"

class Model {
 public:
  long *species;
  Reaction *reactions;
  int speciesCount;
  int reactionCount;

  Model(char *filename);
  void output() const;
  void printState(double currentTime) const;
};

#endif
```

Model.cc

```
#include <iostream>
#include <fstream>
#include "Model.h"

using namespace std;

Model::Model(char *filename) {
  int i,j;
  ifstream in;
  in.open(filename);

  in >> speciesCount;
  species = new long[speciesCount];
  for(i=0; i<speciesCount; i++) {
    in >> species[i];
  }

  in >> reactionCount;
  reactions = new Reaction[reactionCount];
  for(i=0; i<reactionCount; i++) {
    Reaction *reaction = &reactions[i];
    in >> reaction->reactantCount;
    reaction->reactants = new SpeciesUpdate[reaction->reactantCount];
    for(j=0; j<reaction->reactantCount; j++) {
      SpeciesUpdate *reactant = &reaction->reactants[j];
      in >> reactant->coefficient;
      in >> reactant->speciesIndex;
      reactant->species = &species[reactant->speciesIndex];
    }
    in >> reaction->productCount;
    reaction->products = new SpeciesUpdate[reaction->productCount];
    for(j=0; j<reaction->productCount; j++) {
      SpeciesUpdate *product = &reaction->products[j];
      in >> product->coefficient;
      in >> product->speciesIndex;
      product->species = &species[product->speciesIndex];
    }
    in >> reaction->rate;
  }
}

void Model::output() const {
  int i;
  for(i=0; i<speciesCount; i++) {
    cout << "s" << i << ": " << species[i] << endl;
  }
  for(i=0; i<reactionCount; i++) {
    reactions[i].output();
    cout << endl;
  }
}
```

```
void Model::printState(double currentTime) const {
  cout << currentTime;
  for(int i=0; i<speciesCount; i++) {
    cout << " " << species[i];
  }
  cout << endl;
}
```

Next.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"
#include "DependencyGraph.h"
#include "MinHeap.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);
  DependencyGraph dependencyGraph(model);
  MinHeap minHeap;

  double currentTime = 0.0;
  double *propensities = new double[model.reactionCount];
  double *putativeTimes = new double[model.reactionCount];
  int selectedReaction;
  int i;

  for(i=0; i<model.reactionCount; i++) {
    propensities[i] = model.reactions[i].getPropensity();
    putativeTimes[i] = random.getExponential()/propensities[i];
    minHeap.add(putativeTimes[i]);
  }

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    selectedReaction = minHeap.getMin();
    model.reactions[selectedReaction].execute();
    currentTime = putativeTimes[selectedReaction];
    if (parameters.output) model.printState(currentTime);

    IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
    for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      double oldPropensity = propensities[index];
      propensities[index] = model.reactions[index].getPropensity();
      if (index != selectedReaction) {
      if (oldPropensity == 0.0) {
        putativeTimes[index] = currentTime +
random.getExponential()/propensities[index];
      } else {
        putativeTimes[index] = currentTime + oldPropensity /
propensities[index] * (putativeTimes[index] - currentTime);
      }
      minHeap.update(index,putativeTimes[index]);
      }
    }

    putativeTimes[selectedReaction] = currentTime +
random.getExponential()/propensities[selectedReaction];
```

```
      minHeap.update(selectedReaction,putativeTimes[selectedReaction]);
    }
  }
```

NextFPC.cc

```
#include "Model.h"
#include "Parameters.h"
#include "Random.h"
#include "DependencyGraph.h"
#include "MinHeap.h"
#include "FastPropensity.h"

int main(int argc,char **argv) {
  Parameters parameters(argc,argv);
  Random random(parameters.seed);
  Model model(parameters.filename);
  DependencyGraph dependencyGraph(model);
  MinHeap minHeap;

  double *propensities = new double[model.reactionCount];
  FastPropensity fastPropensity(model,propensities,&dependencyGraph);
  double *putativeTimes = new double[model.reactionCount];
  int selectedReaction;
  int i;
  double currentTime = 0.0;

  for(i=0; i<model.reactionCount; i++) {
    propensities[i] = model.reactions[i].getPropensity();
    putativeTimes[i] = random.getExponential()/propensities[i];
    minHeap.add(putativeTimes[i]);
  }

  if (parameters.output) model.printState(currentTime);
  for( ; parameters.reactions > 0; parameters.reactions--) {
    selectedReaction = minHeap.getMin();
    model.reactions[selectedReaction].execute();
    currentTime = putativeTimes[selectedReaction];
    if (parameters.output) model.printState(currentTime);

    IntVector *dependencies =
dependencyGraph.getDependencies(selectedReaction);
    for(i=0; i<dependencies->size(); i++) {
      int index = dependencies->get(i);
      double oldPropensity = propensities[index];
      fastPropensity.update(selectedReaction,i);
      if (index != selectedReaction) {
      if (oldPropensity == 0.0) {
        putativeTimes[index] = currentTime +
random.getExponential()/propensities[index];
      } else {
        putativeTimes[index] = currentTime + oldPropensity /
propensities[index] * (putativeTimes[index] - currentTime);
      }
      minHeap.update(index,putativeTimes[index]);
      }
    }
```

```
    putativeTimes[selectedReaction] = currentTime +
random.getExponential()/propensities[selectedReaction];
    minHeap.update(selectedReaction,putativeTimes[selectedReaction]);
  }
}
```

Parameters.h

```
#ifndef PARAMETERS_H
#define PARAMETERS_H

class Parameters {
 public:
  bool output;
  long reactions;
  char *filename;
  long seed;

  Parameters(int argc, char **argv);
  void printUsage(char *executableName);
};

#endif
```

Parameters.cc

```
#include "Parameters.h"
#include <stdlib.h>
#include <iostream>

using namespace std;

Parameters::Parameters(int argc, char **argv) {
  output = false;
  reactions = -1;
  filename = NULL;
  seed = 1;

  for(int i=1; i<argc; i++) {
    if (strcmp(argv[i],"-o") == 0) {
      output = true;
    } else if (strcmp(argv[i],"-i") == 0) {
      filename = argv[++i];
    } else if (strcmp(argv[i],"-seed") == 0) {
      seed = strtol(argv[++i],NULL,10);
    } else if (strcmp(argv[i],"-rxns") == 0) {
      reactions = strtol(argv[++i],NULL,10);
    } else {
      cout << "Invalid switch: " << argv[i] << endl;
      printUsage(argv[0]);
    }
  }

  if (filename == NULL) {
    cout << "Input model not specified." << endl;
    printUsage(argv[0]);
  }

  if (reactions == -1) {
    cout << "Reaction count not specified." << endl;
    printUsage(argv[0]);
  }
}

void Parameters::printUsage(char *executableName) {
  cout << "usage: " << executableName << " [-o] [-seed <seed>] -i
<model> -rxns <reactionCount>" << endl;
  exit(1);
}
```

PrintModel.cc

```
#include "Model.h"
#include <iostream>

using namespace std;

int main(int argc, char** argv) {
  if (argc != 2) {
    cout << "usage: PrintModel <filename>" << endl;
  }
  Model model(argv[1]);
  model.output();
}
```

Random.h

```
#ifndef RANDOM_H
#define RANDOM_H

class Random {
 private:
  unsigned long myrand;

 public:
  Random(long seed);
  double getUniform();
  double getExponential();
};

#endif
```

Random.cc

```
#include <math.h>
#include "Random.h"

Random::Random(long seed) {
  myrand = (unsigned long)seed;
}

double Random::getUniform() {
  myrand = myrand * 0x41c64e6d + 0x3039;
  long temp = myrand & 0x7fffffff;
  return (double)temp/2147483647.0;
}

double Random::getExponential() {
  double temp = getUniform();
  while (temp == 0.0) {
    temp = getUniform();
  }
  return -log(temp);
}
```

Reaction.h

```
#ifndef REACTION_H
#define REACTION_H

#include "SpeciesUpdate.h"

class Reaction {
 public:
  int reactantCount;
  int productCount;
  SpeciesUpdate *reactants;
  SpeciesUpdate *products;
  double rate;

  void output() const;
  void execute();
  double getPropensity() const;
  int getDelta(int speciesIndex) const;
};

#endif
```

Reaction.cc

```
#include <iostream>
#include "Reaction.h"

using namespace std;

void Reaction::output() const {
  int i;

  for(i=0; i<reactantCount; i++) {
    if (i != 0) {
      cout << " + ";
    }
    reactants[i].output();
  }
  cout << " => ";
  if (productCount == 0) {
    cout << "*" << endl;
  } else {
    for(i=0; i<productCount; i++) {
      if (i != 0) {
      cout << " + ";
      }
      products[i].output();
    }
  }

  cout << "  " << rate;
}

void Reaction::execute() {
  int i;
  for(i=0; i<reactantCount; i++) {
    *reactants[i].species -= reactants[i].coefficient;
  }
  for(i=0; i<productCount; i++) {
    *products[i].species += products[i].coefficient;
  }
}

double Reaction::getPropensity() const {
  int i,j;
  double propensity = rate;
  for(i=0; i<reactantCount; i++) {
    for(j=0; j<reactants[i].coefficient; j++) {
      propensity *= (double)(*reactants[i].species-j);
    }
    for(j=2; j<=reactants[i].coefficient; j++) {
      propensity /= (double)j;
    }
  }
  return propensity;
}
```

```
int Reaction::getDelta(int speciesIndex) const {
  int delta = 0;

  for(int i=0; i<reactantCount; i++) {
    if (reactants[i].speciesIndex == speciesIndex) {
      delta -= reactants[i].coefficient;
    }
  }
  for(int i=0; i<productCount; i++) {
    if (products[i].speciesIndex == speciesIndex) {
      delta += products[i].coefficient;
    }
  }

  return delta;
}
```

SpeciesUpdate.h

```
#ifndef SPECIESUPDATE_H
#define SPECIESUPDATE_H

class SpeciesUpdate {
 public:
  int coefficient;
  int speciesIndex;
  long *species;

  void output();
};

#endif
```

SpeciesUpdate.cc

```
#include <iostream>
#include "SpeciesUpdate.h"

using namespace std;

void SpeciesUpdate::output() {
  if (coefficient != 1) {
    cout << coefficient << " ";
  }
  cout << "s" << speciesIndex;
}
```

SumTree.h

```
class Node {
 public:
  Node *leftChild;
  Node *rightChild;
  Node *parent;
  double value;
  int id;
};

class SumTree {
public:
  SumTree(int size);
  void update(int id, double value);
  void print();
  double getSum();
  int selectReaction(double rand);

private:
  Node *root;
  Node **leafNodes;
  int leafNodeCount;
  int getNodeCount(Node *node);
  int getLeafNodeCount(Node *node);
  void assignNodeNumbers(Node *node);
  void updateHelper(Node *node);
  void add();
  Node *addHelper(Node *node);
  void printHelper(Node *node);
  Node *initNode();
  int selectReactionHelper(double rand, Node *node);
};
```

SumTree.cc

```
#include <iostream>
#include "SumTree.h"

using namespace std;

SumTree::SumTree(int size) {
  root = NULL;
  while(getLeafNodeCount(root) < size) {
    add();
  }
  leafNodes = new (Node *)[size];
  leafNodeCount = 0;
  assignNodeNumbers(root);
}

void SumTree::assignNodeNumbers(Node *node) {
  if (node == NULL) {
    return;
  } else if ((node->leftChild == NULL) && (node->rightChild == NULL)) {
    leafNodes[leafNodeCount] = node;
    node->id = leafNodeCount;
    leafNodeCount++;
  } else {
    assignNodeNumbers(node->leftChild);
    assignNodeNumbers(node->rightChild);
  }
}

int SumTree::getNodeCount(Node *node) {
  if (node == NULL) {
    return 0;
  } else {
    return 1 + getNodeCount(node->leftChild) + getNodeCount(node-
>rightChild);
  }
}

int SumTree::getLeafNodeCount(Node *node) {
  if (node == NULL) {
    return 0;
  } else if ((node->leftChild == NULL) && (node->rightChild == NULL)) {
    return 1;
  } else {
    return getLeafNodeCount(node->leftChild) + getLeafNodeCount(node-
>rightChild);
  }
}

Node *SumTree::initNode() {
  Node *node = new Node();
  node->leftChild = NULL;
  node->rightChild = NULL;
```

```
  node->parent = NULL;
  node->value = 0.0;
  node->id = -1;
  return node;
}

void SumTree::add() {
  if (root == NULL) {
    root = initNode();
  } else {
    Node *parent = addHelper(root);
    Node *n1 = initNode();
    Node *n2 = initNode();
    n1->parent = parent;
    n2->parent = parent;
    parent->leftChild = n1;
    parent->rightChild = n2;
  }
}

Node *SumTree::addHelper(Node *node) {
  if ((node->leftChild == NULL) || (node->rightChild == NULL)){
    return node;
  } else if (getNodeCount(node->leftChild) == getNodeCount(node-
>rightChild)) {
    return addHelper(node->leftChild);
  } else {
    return addHelper(node->rightChild);
  }
}

void SumTree::update(int id, double value) {
  leafNodes[id]->value = value;
  updateHelper(leafNodes[id]->parent);
}

void SumTree::updateHelper(Node *node) {
  if (node == NULL) {
    return;
  } else {
    node->value = node->leftChild->value + node->rightChild->value;
    updateHelper(node->parent);
  }
}

void SumTree::print() {
  printHelper(root);
  cout << endl;
}

void SumTree::printHelper(Node *node) {
  if (node == NULL) {
    cout << "-";
  } else {
    if ((node->leftChild == NULL) && (node->rightChild == NULL)) {
```

```
      cout << node->value;
    } else {
      cout << node->value << "(";
      printHelper(node->leftChild);
      cout << ",";
      printHelper(node->rightChild);
      cout << ")";
    }
  }
}

double SumTree::getSum() {
  return root->value;
}

int SumTree::selectReaction(double rand) {
  return selectReactionHelper(rand,root);
}

int SumTree::selectReactionHelper(double rand, Node *node) {
  if (node->id != -1) {
    return node->id;
  } else {
    if (rand < node->leftChild->value) {
      return selectReactionHelper(rand,node->leftChild);
    } else {
      return selectReactionHelper(rand - node->leftChild->value,node-
>rightChild);
    }
  }
}
```

Appendix B – Performance Analysis Source Code

The following pages include the source code used to generate models and measure the execution times of the simulators.

ModelGen.java

```
import java.io.*;
import java.util.*;

class ModelGen {
    public static void generateModel(int speciesCount,
                                     int factor,
                                     String filename) throws IOException {
      PrintStream out = new PrintStream(new
FileOutputStream(filename));
      out.println(speciesCount);
      for(int i=0; i<speciesCount; i++) {
          if (i != 0) out.print(" ");
          out.print("100");
      }
      out.println();

      out.println(speciesCount*factor);
      for(int i=0; i<speciesCount; i++) {
          for(int j=0; j<factor; j++) {
            out.println("1 1 " + i + " 1 1 " + ((i+1)%speciesCount) + "
1.0");
          }
      }
      out.close();
    }

    public static void main(String args[]) throws IOException {
      for(int modelsize=12; modelsize<=12*50; modelsize+=12) {
          generateModel(modelsize,1,"" + modelsize + "_1.in");
          generateModel(modelsize/2,2,"" + modelsize + "_2.in");
          generateModel(modelsize/3,3,"" + modelsize + "_3.in");
          generateModel(modelsize/4,4,"" + modelsize + "_4.in");
      }
    }
}
```

Performance.java

```
import java.io.*;
import java.util.*;

class Performance {
    public static void print(String s) {
      System.out.println(s);
      System.err.println(s);
    }

    public static void main(String args[])
                                throws Exception {
      String executable = args[0];
      print("File\tSize\tFactor\tRxns\tTime");

      for(int j=1; j<=4; j++) {
          for(int i=12; i<=12*50; i+=12) {
            String filename = "models/" + i + "_" + j + ".in";
            long totalReactions = 5000000;
            long t1 = Calendar.getInstance().getTimeInMillis();
            Process p2 = Runtime.getRuntime().exec("./" + executable +
" -i " + filename + " -rxns " + totalReactions);
            p2.waitFor();
            long t2 = Calendar.getInstance().getTimeInMillis();
            print(filename + "\t" + i + "\t" + j + "\t" +
totalReactions + "\t" + ((t2-t1)/1000.0));
          }
      }
    }
}
```

Appendix C – Test Models

The following pages include SBML code for the real biological models used for performance analysis.

Model: DIMER

```
8
1 1 0 0 0 0 0 0
13
1 1 0 2 1 0 1 2 0.01
1 1 2 0 6e-3
1 1 2 2 1 2 1 4 3e-2
1 1 4 0 4e-4
2 1 6 1 1 1 1 7 0.0016
1 1 7 2 1 1 1 6 0.2
1 1 1 2 1 1 1 3 0.002
1 1 7 2 1 7 1 3 0.1
1 1 3 0 6e-3
1 1 3 2 1 3 1 5 3e-2
1 1 5 0 4e-4
1 2 4 1 1 6 0.016
1 1 6 1 2 4 1
8
0 1 2 3 4 5 6 7
```

Model: ENG

```
32
0 130 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0
49
1 1 3 2 1 3 1 4 .01
1 1 4 0 .012
1 1 4 2 1 4 1 1 .1
1 1 1 0 .0006
2 1 1 1 0 1 1 2 .000001
1 1 2 2 1 1 1 0 .01
1 1 6 4 1 6 1 7 1 8 1 29 .3
1 1 7 0 .012
1 1 8 0 .012
1 1 7 2 1 7 1 9 .1
1 1 8 2 1 8 1 10 .1
1 1 9 0 0.0006
1 1 10 0 0.0006
2 1 9 1 10 1 1 11 .01
1 1 11 2 1 9 1 10 .01
1 1 11 0 0.0006
1 1 0 0 0.0005
1 1 12 2 1 12 1 0 .2
2 1 12 1 0 1 1 13 0.0000005
1 1 13 2 1 12 1 0 .00004
1 1 31 2 1 15 1 31 .08
1 1 15 0 .012
1 1 15 2 1 15 1 16 .1
1 1 16 0 0.0006
2 1 16 1 18 1 1 27 .005
1 1 27 2 1 18 1 16 1
1 1 27 2 1 27 1 20 .08
1 1 20 0 .012
1 1 20 2 1 20 1 22 .1
1 1 22 0 0.0006
1 1 17 2 1 17 1 19 0.02
1 1 19 0 0.012
1 1 19 2 1 19 1 21 .1
1 1 21 0 0.0006
1 1 22 2 1 22 1 23 85
1 1 23 1 1 26 .1
1 1 26 1 1 23 .00001
2 1 23 1 21 1 1 24 .001
1 1 24 2 1 21 1 23 .3636
1 2 24 1 1 25 .2
1 1 25 1 2 24 .433
2 1 25 1 28 1 1 6 .1
1 1 6 2 1 25 1 28 4
1 1 29 0 0.012
1 1 29 2 1 29 1 30 .03
1 1 30 0 0.0006
2 1 30 1 23 1 1 30 5
2 1 2 1 14 1 1 31 0.06
1 1 31 2 1 2 1 14 10
```

```
11
0 1 2 11 16 21 22 23 26 28 30
```

Model: QS8

```
122
1 16000000 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0
1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0
0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1
0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0
201
1 1 0 1 2 0 .0001194
1 1 2 2 1 2 1 3 5.55e-3
1 1 3 0 2.78e-5
1 2 3 1 1 4 .00003
1 1 4 1 2 3 .01
1 2 4 1 1 5 .0006
1 1 5 1 2 4 .01
2 1 6 1 5 1 1 7 .02
1 1 7 2 1 6 1 5 .01
1 1 7 2 1 7 1 8 0.06
1 1 8 0 .006
1 1 8 2 1 8 1 9 .03
1 1 9 0 0.0006
1 1 1 1 1 10 4.7e-7
1 1 10 1 1 1 0.8
2 1 9 1 10 1 1 11 0.001
1 1 11 2 1 9 1 10 0.3636
1 2 11 1 1 12 0.02
1 1 12 1 2 11 0.433
2 1 12 1 13 1 1 14 0.1
1 1 14 2 1 12 1 13 4
1 1 14 2 1 14 1 15 0.3
1 1 15 0 0.006
1 1 15 2 1 15 1 16 0.03
1 1 16 0 0.0006
2 1 16 1 0 3 1 0 1 16 1 1 0.006
1 1 17 2 1 17 1 18 5.55e-3
1 1 18 0 2.78e-5
1 2 18 1 1 19 .00003
1 1 19 1 2 18 .01
1 2 19 1 1 20 .0006
1 1 20 1 2 19 .01
2 1 21 1 20 1 1 22 .02
1 1 22 2 1 21 1 20 .01
1 1 22 2 1 22 1 23 0.06
1 1 23 0 .006
1 1 23 2 1 23 1 24 .03
1 1 24 0 0.0006
1 1 1 1 1 25 4.7e-7
1 1 25 1 1 1 0.8
2 1 24 1 25 1 1 26 0.001
1 1 26 2 1 24 1 25 0.3636
1 2 26 1 1 27 0.02
1 1 27 1 2 26 0.433
2 1 27 1 28 1 1 29 0.1
1 1 29 2 1 27 1 28 4
```

```
1 1 29 2 1 29 1 30 0.3
1 1 30 0 0.006
1 1 30 2 1 30 1 31 0.03
1 1 31 0 0.0006
2 1 31 1 0 3 1 0 1 31 1 1 0.006
1 1 32 2 1 32 1 33 5.55e-3
1 1 33 0 2.78e-5
1 2 33 1 1 34 .00003
1 1 34 1 2 33 .01
1 2 34 1 1 35 .0006
1 1 35 1 2 34 .01
2 1 36 1 35 1 1 37 .02
1 1 37 2 1 36 1 35 .01
1 1 37 2 1 37 1 38 0.06
1 1 38 0 .006
1 1 38 2 1 38 1 39 .03
1 1 39 0 0.0006
1 1 1 1 1 40 4.7e-7
1 1 40 1 1 1 0.8
2 1 39 1 40 1 1 41 0.001
1 1 41 2 1 39 1 40 0.3636
1 2 41 1 1 42 0.02
1 1 42 1 2 41 0.433
2 1 42 1 43 1 1 44 0.1
1 1 44 2 1 42 1 43 4
1 1 44 2 1 44 1 45 0.3
1 1 45 0 0.006
1 1 45 2 1 45 1 46 0.03
1 1 46 0 0.0006
2 1 46 1 0 3 1 0 1 46 1 1 0.006
1 1 47 2 1 47 1 48 5.55e-3
1 1 48 0 2.78e-5
1 2 48 1 1 49 .00003
1 1 49 1 2 48 .01
1 2 49 1 1 50 .0006
1 1 50 1 2 49 .01
2 1 51 1 50 1 1 52 .02
1 1 52 2 1 51 1 50 .01
1 1 52 2 1 52 1 53 0.06
1 1 53 0 .006
1 1 53 2 1 53 1 54 .03
1 1 54 0 0.0006
1 1 1 1 1 55 4.7e-7
1 1 55 1 1 1 0.8
2 1 54 1 55 1 1 56 0.001
1 1 56 2 1 54 1 55 0.3636
1 2 56 1 1 57 0.02
1 1 57 1 2 56 0.433
2 1 57 1 58 1 1 59 0.1
1 1 59 2 1 57 1 58 4
1 1 59 2 1 59 1 60 0.3
1 1 60 0 0.006
1 1 60 2 1 60 1 61 0.03
1 1 61 0 0.0006
2 1 61 1 0 3 1 0 1 61 1 1 0.006
```

```
1 1 62 2 1 62 1 63 5.55e-3
1 1 63 0 2.78e-5
1 2 63 1 1 64 .00003
1 1 64 1 2 63 .01
1 2 64 1 1 65 .0006
1 1 65 1 2 64 .01
2 1 66 1 65 1 1 67 .02
1 1 67 2 1 66 1 65 .01
1 1 67 2 1 67 1 68 0.06
1 1 68 0 .006
1 1 68 2 1 68 1 69 .03
1 1 69 0 0.0006
1 1 1 1 1 70 4.7e-7
1 1 70 1 1 1 0.8
2 1 69 1 70 1 1 71 0.001
1 1 71 2 1 69 1 70 0.3636
1 2 71 1 1 72 0.02
1 1 72 1 2 71 0.433
2 1 72 1 73 1 1 74 0.1
1 1 74 2 1 72 1 73 4
1 1 74 2 1 74 1 75 0.3
1 1 75 0 0.006
1 1 75 2 1 75 1 76 0.03
1 1 76 0 0.0006
2 1 76 1 0 3 1 0 1 76 1 1 0.006
1 1 77 2 1 77 1 78 5.55e-3
1 1 78 0 2.78e-5
1 2 78 1 1 79 .00003
1 1 79 1 2 78 .01
1 2 79 1 1 80 .0006
1 1 80 1 2 79 .01
2 1 81 1 80 1 1 82 .02
1 1 82 2 1 81 1 80 .01
1 1 82 2 1 82 1 83 0.06
1 1 83 0 .006
1 1 83 2 1 83 1 84 .03
1 1 84 0 0.0006
1 1 1 1 1 85 4.7e-7
1 1 85 1 1 1 0.8
2 1 84 1 85 1 1 86 0.001
1 1 86 2 1 84 1 85 0.3636
1 2 86 1 1 87 0.02
1 1 87 1 2 86 0.433
2 1 87 1 88 1 1 89 0.1
1 1 89 2 1 87 1 88 4
1 1 89 2 1 89 1 90 0.3
1 1 90 0 0.006
1 1 90 2 1 90 1 91 0.03
1 1 91 0 0.0006
2 1 91 1 0 3 1 0 1 91 1 1 0.006
1 1 92 2 1 92 1 93 5.55e-3
1 1 93 0 2.78e-5
1 2 93 1 1 94 .00003
1 1 94 1 2 93 .01
1 2 94 1 1 95 .0006
```

```
1 1 95 1 2 94 .01
2 1 96 1 95 1 1 97 .02
1 1 97 2 1 96 1 95 .01
1 1 97 2 1 97 1 98 0.06
1 1 98 0 .006
1 1 98 2 1 98 1 99 .03
1 1 99 0 0.0006
1 1 1 1 1 100 4.7e-7
1 1 100 1 1 1 0.8
2 1 99 1 100 1 1 101 0.001
1 1 101 2 1 99 1 100 0.3636
1 2 101 1 1 102 0.02
1 1 102 1 2 101 0.433
2 1 102 1 103 1 1 104 0.1
1 1 104 2 1 102 1 103 4
1 1 104 2 1 104 1 105 0.3
1 1 105 0 0.006
1 1 105 2 1 105 1 106 0.03
1 1 106 0 0.0006
2 1 106 1 0 3 1 0 1 106 1 1 0.006
1 1 107 2 1 107 1 108 5.55e-3
1 1 108 0 2.78e-5
1 2 108 1 1 109 .00003
1 1 109 1 2 108 .01
1 2 109 1 1 110 .0006
1 1 110 1 2 109 .01
2 1 111 1 110 1 1 112 .02
1 1 112 2 1 111 1 110 .01
1 1 112 2 1 112 1 113 0.06
1 1 113 0 .006
1 1 113 2 1 113 1 114 .03
1 1 114 0 0.0006
1 1 1 1 1 115 4.7e-7
1 1 115 1 1 1 0.8
2 1 114 1 115 1 1 116 0.001
1 1 116 2 1 114 1 115 0.3636
1 2 116 1 1 117 0.02
1 1 117 1 2 116 0.433
2 1 117 1 118 1 1 119 0.1
1 1 119 2 1 117 1 118 4
1 1 119 2 1 119 1 120 0.3
1 1 120 0 0.006
1 1 120 2 1 120 1 121 0.03
1 1 121 0 0.0006
2 1 121 1 0 3 1 0 1 121 1 1 0.006
17
1 8 15 23 30 38 45 53 60 68 75 83 90 98 105 113 120
```

Model: TB

```
17
10 0 0 10 0 0 10 20 0 0 0 0 0 0 1 0 1
23
1 1 16 2 1 0 1 16 10
2 1 0 1 3 2 1 3 1 1 16
2 1 0 1 4 2 1 4 1 1 32
2 1 0 1 5 2 1 5 1 1 16
2 1 1 1 3 2 1 3 1 2 0.6
2 1 1 1 4 2 1 4 1 2 0.6
2 1 1 1 5 2 1 5 1 2 0.78
1 1 2 0 4
1 1 3 1 1 4 100
1 1 4 1 1 3 1
1 1 4 1 1 5 0.5
1 1 5 1 1 4 10
2 1 6 1 7 1 1 8 1
1 2 1 1 1 13 10
1 1 13 1 2 1 10
2 1 13 1 14 1 1 15 5
1 1 15 2 1 13 1 14 10
2 1 15 1 9 2 1 8 1 15 10
1 1 8 1 1 9 10
1 1 9 1 1 10 4
1 1 10 1 1 11 6
1 1 11 3 1 12 1 6 1 7 1
1 1 12 0 100
17
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

Vita

James Michael McCollum was born in Pittsburgh, PA on September 7, 1979. He was raised in Pittsburgh, PA and attended school at Montessori Centre Academy and Hampton Middle School. In 1997, he graduated from Hampton High School. From there, he went to the University of Dayton in Ohio and received a B.E.E. in Electrical Engineering and a B.S. in Computer Science in 2001. After graduation, he worked for a Neolinear Inc., a analog circuit design automation company in Pittsburgh, PA. In 2004, he received his M.S. in Electrical Engineering from the University of Tennessee, Knoxville.

Currently, James is pursuing his doctorate in Electrical Engineering from the University of Tennessee, Knoxville.