To the Graduate Council:

I am submitting herewith a thesis written by David Dewayne Jenkins entitled "Accelerating the Stochastic Simulation Algorithm using Emerging Architectures." 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 Computer Engineering.

Gregory D. Peterson

Major Professor

We have read this thesis
and recommend its acceptance:

Itamar Arel

Robert Harrison

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

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

Accelerating the Stochastic Simulation Algorithm using Emerging Architectures

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

David Dewayne Jenkins

December 2009

DEDICATION

This thesis is dedication to my family for all of their support, guidance and encouragement; and to my dog, Relska, who stuck by me through all the long nights and never complained (except when he was hungry).

ACKNOWLEDGMENTS

I would like to thank all of the people who helped and pushed me to complete my Master of Science degree. First and foremost, I would like to thanks my parents for all of the love, support and encouragement throughout the years. Secondly, I would not have been as successful had it not been for the best advisor that anyone could ask for, Dr. Peterson. Thirdly, I would like to thank my Dr. Arel and Dr. Harrison for taking the time to serve on my committee. Finally, I would like to thank all of my friends that supported me through all the work and sleepless nights.

ABSTRACT

In order for scientists to learn more about molecular biology, it is imperative that they have the ability to construct and evaluate models. Model statistics consistent with the chemical master equation can be obtained using Gillespie's stochastic simulation algorithm (SSA). Due to the stochastic nature of the Monte Carlo simulations, large numbers of simulations must be run in order to get accurate statistics for the species populations and reactions. However, the algorithm tends to be computationally heavy and leads to long simulation runtimes for large systems. In this research, the performance of Gillespie's stochastic simulation algorithm is analyzed and optimized using a number of techniques and architectures. These techniques include parallelizing simulations using streaming SIMD extensions (SSE), message passing interface with multicore systems and computer cluters, and CUDA with NVIDIA graphics processing units. This research is an attempt to make using the SSA a better option for modeling biological and chemical systems. Through this work, it will be shown that accelerating the algorithm in both of the serial and SSE implementations proved to be beneficial, while the CUDA implementation had lower than expected results.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chemists and biologists are continuously attempting to learn more about the interactions of different molecular species. By learning of these interactions, the scientists are able to determine ways to either change the course of reactions or inhibit future reactions. One method of obtaining this growth in knowledge is obtained by using a number of modeling techniques including, but not limited to, modeling the time evolution of spatially homogeneous mixtures of chemically reacting molecules in order to analyze and understand species populations and their interactions. This allows for scientists to study different biological and chemical networks without physically dealing with the molecules. Modeling is convenient for scientists as they do not have to physically be in a lab. Instead, the scientist could work remotely in a "virtual" lab by using a couple of approaches: deterministic and stochastic modeling.

In the deterministic approach of modeling chemical systems, ordinary differential equations are used to model a system of chemical species and reactions. For each of the species, there is a corresponding differential equation that expresses the time rate of change of population as a function of all species populations. Mathematically, this approach is attractive due to the ability to solve the equations with traditional differential equation techniques. Research has been able to describe numerous molecular models by using deterministic approaches with differential equations. This approach is sufficient for most problems; however, it does not work for all cases, such as nonlinear systems, due to the impractical fundamental assumptions. Differential equations-based modeling

does not consider the statistical fluctuations and correlations inherent in chemical reactions thereby being inaccurate for models of small populations. Also, the ordinary differential equations have no sense of discrete values for the species populations.

To defeat these limitations, a stochastic approach was formulated. Unlike the deterministic approach, there are not many equations, instead there is only one master difference equation for a probability function. This master equation measures the probability of finding species populations at any given point in time. This approach is valid in all of the cases that the deterministic approach is valid plus more. However, this comes at the cost of being mathematically and computationally more difficult to solve. This leads to the need to take advantage of implementing computer-based methods.

Dan Gillespie developed a computer-based stochastic approach called the stochastic simulation algorithm (SSA) to produce statistically exact results [6]. In the stochastic simulation algorithm, the reaction rate constants are viewed as probabilities per unit time and the temporal behavior of reactions is a Markovian random walk of N species populations in an N-dimension space [6]. Unlike the deterministic and some other stochastic approaches, SSA does not try to solve the master equation, instead it uses Monte Carlo techniques to numerically simulate the analytical Markov process. Due to this, this algorithm is extremely computationally heavy despite efforts to reduce the load [2], [4], [6], [7], [19]. Hence, runtimes for models can be very large for a single simulation. Since multiple simulations are needed to get a good statistical representation, the total runtime is then multiplied to produce an unreasonable amount of time. Hence, this shows the need for accelerated versions of this algorithm.

Implementing this algorithm on emerging architectures is an ideal way to decrease the runtimes of these algorithms. These architectures lead to the ability to target and tweak the algorithm to take full advantage of all the resources available, thereby increasing the performance of the algorithm. Examples of such architectures include, but are not limited to, graphics processing units (chapter 6), multicore CPUs, clusters of computers (chapter 5), streaming SIMD extension vectorization units (chapter 4), and field programmable gate arrays. The following are examples of previous research in applying this algorithm to emerging architectures. SSA was implemented

on a GPU by Li et. al. with relatively good results [12]. H. Li et. al. [11] and M. Langlais et. al. [10] researched implementing the stochastic simulation algorithm on clusters of computers. There have been numerous attempts to put the stochastic simulation algorithm on FPGAs such as McCollum [18] and Yoshimi [27].

For this work, the main focus is to accelerate this algorithm using some of the emerging architectures. The second chapter of this work gives a more in-depth look at the stochastic simulation algorithm and why it is the superior method for modeling. The following chapters will go into the details of accelerating the stochastic simulation for a number of architectures. Chapter 3 explains the work completed to accelerate the serial version of the code by focusing on memory usage and overall ordering of floating point operations. Next, Chapter 4 presents the work to implement the stochastic simulation algorithm on using the streaming SIMD extension vectorization units on both Intel and AMD processors. Chapter 5 describes the work performed to implement the algorithm using multicore processors, clusters of computers, and the message passing interface. Chapter 6 explains the implementation using CUDA and NVIDIA graphics processing units. For the serial implementations, the new proposed code is compared with the baseline using all of the models listed in Appendix A. Each of the other implementations will be compared to the serial implementation to view the performance speedup factor with the SCHLOGL, DIMER, and HSR models given in Appendix A. Finally, the last chapter will present an overall comparison of all the implementation methods as well as some final remarks.

Overall, this work proves to be a valuable resource to the scientists looking for a speedier way to perform the stochastic simulation algorithm for modeling systems.

Chapter 2

Modeling Molecular Species and Reactions

This chapter will present the motivation for modeling gene regulatory networks as well as methods of doing so.

2.1 Modeling of Gene Regulatory Networks

The main motivation for modeling gene regulatory networks is to learn how these interactions occur in order to formulate a way to interrupt or enhance them. Once a method is created to modify or control the processes, a scientist can then control the rest of the reactions anyway he wants, such as by inserting a foreign species for example. This means the scientist could create medicines for certain diseases and viruses.

In the past, scientists have used this modeling technique a number of times. For instance, mathematical models for cell mitosis in frog eggs were generated [15]. Also, another group used the exact stochastic simulation algorithm to model the phage λ *Escherichia coli* cells [1]. Finally, another group created models of the phage T7 virus in *Escherichia coli* [3]. The use of modeling are not limited to these three presented, but each of these are good examples of the use of modeling in the research community.

2.2 Chemical Master Equation

In order to understand modeling, the purpose of modeling must be explained. For each chemically reacting system, there is a chemical master equation that describes every aspect of the system. This equation is extremely difficult to solve completely, thus the need for modeling approaches arises. In this section, a simple overview of explanation of the derivation of the chemical master equation is given and is in no way meant to be complete derivation. More information on the derivation can be found in [6] and [8].

Chemical equations come in the form shown below.

$$l_1 r_1 + \ldots + l_\rho r_\rho \underset{k_{-1}}{\overset{k_1}{\rightleftharpoons}} l'_1 p_1 + \ldots + l'_\gamma p_\gamma \tag{2.1}$$

First, to explain some of the details of the chemical master equation (CME), a few definitions need to be defined.

Definition Let S denote the set of species molecules. S contains a total of N species and is given by $S = S_1, S_2, ..., S_N$ where S_i gives the species with the index of i. This simply differentiates certain species and is not to be confused with the species populations X described below.

Definition Let X denote the set of species populations and X_i denotes the species with the index of i. M signifies the total number of species in the given system. Given any time t, the set of species populations is given by the following expression: $X(t) = X_1(t), X_2(t), ..., X_M(t)$. In other words, the species populations at time t=0 is give as $X(0) = X_1(0), X_2(0), ..., X_M(0)$.

Definition Let R denote the set of reactions in the network. R contains a total of M reactions and is given by $R = R_1, R_2, ..., R_M$ where R_j gives the reaction with the index of j.

Definition Let r and p denote the sets of reactants and products, respectively. r and p each contain a total of M individual sets of reactants and products given by $r = r_1, r_2, ..., r_M$ and $p = p_1, p_2, ..., p_M$ where r_j and p_j gives the individual set of reactants and products for the reaction j. In equation 2.1, ρ_j and γ_j denote the total number of reactions and products in reaction j.

Definition Let l denote the set of reaction coefficients. l_j is the set of reaction coefficients for reaction j.

Definition Let k denote the set of stochastic rate constants for each reaction. k is a set of size N given by the expression $k = k_1, k_2, ..., k_N$ where k_j gives the rate constant for reaction j.

For this work, the chemically reacting system is assumed to be well-stirred and at a constant temperature, indicating the irrelevancy of spacial locations and heat fluctuations. The purpose of the CME is to describe the time evolution of all the species, $X(t)$, given an initial state of $X(t_0)$. Each molecular species in the set of $X_i(t)$ is a random variable that describes the species population. For each reaction, there is an associated probability function that describes the likelihood of a reaction occurring within the next time step called a propensity function, denoted as $\alpha_j(X)$. There is also a matrix, ν, that gives the species population net change amounts (indexed by the reaction index, j, then the species index, i). Below is the equation for the propensity calculations.

$$\alpha_j = k_j * \prod_{i \in r_j} nCr(i, l_i) \tag{2.2}$$

where nCr is the number of combinations of the argument$_1$ taken argument$_2$ at a time.

Due to this propensity, there is a time evolution equation that gives the probability

$$P(x, t|X(0), t_0) \tag{2.3}$$

that $X(t)$ will equal to some final population vector x. This is the essence of the chemical master equation. Below is the CME [8], [9]:

$$\frac{\partial}{\partial t} P(x, t|X(0), t_0) = \sum_{j=0}^{M-1} [\alpha_j(x - \nu_j) P(x - \nu_j, t|X(0), t_0) - \alpha_j(x) P(x, t|X(0), t_0)] \tag{2.4}$$

As can be seen, this equation is difficult to solve analytically or numerically due to its recursive nature and dependency on the previous reaction step taken to achieve the final population vector.

2.3 Methods to Model Networks

This section will describe a number of methods to model gene regulatory networks. Each method has different reasons why it should be used over the others.

2.3.1 Molecular Dynamics Simulation

Molecular dynamics simulation is a type of simulation used to model the interactions between atoms and molecules over a period of time. This approach approximates all of the known physics of a system including those of position, velocity, and acceleration and gives exceptionally accurate results. Molecular dynamics simulation is extremely computationally expensive due to the need to compute the kinetic and potential energies of each molecule in the system for each iteration of the program. Due to this weakness, the use of this approach is not always viable for complex systems, as it will lead to impossibly long simulation runtimes. For more information on molecular dynamics simulation, refer to [23].

2.3.2 Ordinary Differential Equations Modeling

The use of ordinary differential equations is another accurate approach to modeling a system. Unlike the previous approach, the system is assumed to be well-stirred and spatially homogenous meaning that the positions of individual molecules are irrelevant to the outcome of the system. For each species, there is a corresponding differential equation that expresses the time rate of change of population as a function of all species populations. Mathematically, this approach is attractive due to the ability to solve the equations with traditional differential equation techniques. Research has been able to describe numerous molecular models by using deterministic approaches with differential equations. This approach is sufficient for most problems; however, it does not work for all cases, such as nonlinear systems, due to the impractical fundamental assumptions. This approach does not consider the statistical fluctuations and correlations thereby being inaccurate for models of small populations. Also, the ordinary differential equations do not allow the representation of discrete values for the species populations.

2.3.3 Gillespie's Stochastic Simulation

For species with large populations, differential equations are adequate enough to generate accurate models. However, this is not the case for species with small populations such as those of species in gene regulatory networks, due to their inability to model the noise of a biochemical process. Unlike the deterministic approach, there are not many equations, instead there is only one master difference equation for a probability function. This master equation measures the probability of finding species populations at any given point in time. This approach is valid in all of the same cases that the deterministic approach is valid plus more. However, this comes at the cost of being mathematically and computationally more difficult to solve. This leads to the need to take advantage of implementing computer based methods.

In Dan Gillespie's paper, "A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions," he presents a stochastic approach, called the stochastic simulation algorithm, to model spatially homogeneous chemical systems without using ordinary differential equations [6]. In the stochastic simulation algorithm, the reaction rate constants are viewed as probabilities per unit time and the temporal behavior of reactions is a Markovian random walk of N species populations in an N-dimension space [6]. Unlike the deterministic and non-computer based stochastic approaches, SSA does not involve solving the master equation, instead it uses Monte Carlo techniques to numerically simulate the analytical Markov process.

Gillespie's approach takes an initial set of species populations and reactions, calculates the likelihood of a reaction, and applies this reaction to update the species set. This is done until a certain end time or reaction limit has been reached. Below are the steps needed to execute the stochastic simulation.

As mentioned before, the stochastic simulation algorithm calculates the species population at each time period until the specified end time is reached. This happens first by inputting a species set of size M in vector form with initial populations and a vector of chemical reactions. Since SSA stipulates that the system must be spatially homogenous, the algorithm is only concerned with the populations of the species, ignoring the position of each molecule. Each reaction in R is defined by a stochastic constant and a stoichiometry represented by M reactant and product coefficients.

The steps in figure 2.1 turn out to be very computationally expensive for large and complex reaction sets. There are a number of methods that previous researchers have formulated to help with this. Each of these methods shares the same steps, however steps 2 through 4 differ in calculations in an attempt to reduce the computational complexity. For this work, serial implementations of the First Reaction [7], Direct [7], Next Reaction [4], Optimized Direct [2], and Sorting Direct Methods [19] are explored. Parallel implementations of the Direct Method are employed and compared with its serial counterpart. This method limits the amount of random numbers to be generated, as well as the number of expensive floating-point divisions. The First Reaction method was not chosen to be parallelized because of the large number of random numbers needed. The other methods do not exhibit the ability to be easily parallelizable to take advantage of the SIMD architectures inherent in the SSE and CUDA. The other methods would cause divergent threads and serialized memory loading and storing due to the different reaction dependency graphs.

A description of the Direct Method is given throughout the rest of this chapter due to its extensive use throughout this work. Further explanations of this and the other methods can be found in the cited papers previously.

Above is the pseudocode for this algorithm that will be described in detail during the remainder of this chapter.

Step 1: Initialization First, the program must be initialized, in which the data is read in and stored into memory to be used later. The input file contains the initial species populations,

1. Initialization : Read the model file and initialize all data structures.
2. Propensity Calculation : Calculate the propensity function for each reaction.
3. Reaction Time Generation : Generate random sample of the occurrence time of the next reaction.
4. Reaction Selection : Select which reaction will occur next.
5. Reaction Execution : Update the current simulation time and species variables to reflect the execution of the selected reaction.
6. Termination : If the simulation time < t_{end}, go to step 2.

Figure 2.1: Steps to execute the Stochastic Simulation Algorithm

Algorithm 1 Pseudocode for the Direct Method. Courtesy of [17]

```
CurrentTime = 0.0                                          1. Initialization
X[1...M] = Initial Species Populations
R[1...N] = Reactions
TotalPropensity = 0.0
                                                           2. Propensity Calculation
for I = 1...N do
   Prop[I] = CalcPropensity(S,R[I])
   TotalPropensity += Prop[I]
end for
T = -ln(rand())/TotalPropensity                            3. Reaction Time Generation
Selector = TotalPropensity * rand()                        4. Reaction Selection
for I = 1...N do
   Selector - Prop[I]
   if Selector ≤ 0 then
      SelRxn = I
      break
   end if
end for
X = X - R[SelRxn].reactants + R[SelRxn].products           5.Reaction Execution
CurrentTime += T;
                                                           6. Termination
if CurrentTime < EndTime then
   GOTO Propensity Calculation
end if
```

reactions, and reactant and product coefficients.

Step 2 : Propensity Calculations For this algorithm, a propensity function must be calculated for each reaction. The probability that the given reaction occurs in a given period of time, dt, is given by the following equation,

$$P_i(dt) = \alpha_i(X(t))dt + o(dt)$$

where α_i is a propensity function that describes the likelihood of a reaction occurring on species $X(t)$. The propensity function is the product of the reaction rate constant and the number of ways the reaction can occur depending on the populations of reactants. Using the dimerization example from McCollum et. al. [19], an example reaction could be in the form of $S_4 + S_4 \rightarrow S_5$ and the propensity would be calculated by

$$\alpha_i(X(t)) = k_i \frac{X_4(t)(X_4(t) - 1)}{2}.$$

Step 3 : Reaction Time Generation The occurrence time of the next reaction is estimated, next. This is calculated by dividing an exponentially distributed random number by the sum of the propensities from the previous step, as shown in the equation below.

$$\tau_{next} = \frac{-ln(UNIFORMRAND)}{\sum_{i=1}^{N} \alpha_i(X(t))}$$

Step 4 : Reaction Selection The next reaction that will occur is selected in this step. This is accomplished by multiplying a uniformly distributed random number by the total sum of the propensities. This number is then incrementally subtracted by the propensity of each reaction until the value reaches less than or equal to 0. The index of the reaction whose propensity causes the value to reach this bound is the index of the next reaction.

Step 5 : Reaction Execution The reaction is executed by updating the species population. To do this, each of the reaction coefficients is subtracted from the current population. After that, the product coefficients are added to the population. This gives the updated population values that represent $X(t)$. This is given in the equation below.

$$X(t) = X(t-1) - r_{selectedReaction} + p_{selectedReaction}$$

Step 6 : Termination At this step, if the current calculated time has not reached the specified end time, t_{end}, steps 2 through 5 are repeated again. If the current calculated time is the same or larger than the upper bound t_{end}, the program will terminate

2.4 Conclusion

Since the algorithm is a type of Monte Carlo application with stochastic properties, many runs must be completed in order to get a good statistical, representative model. These numerous runs increase the computational complexity of this algorithm potentially leading to extremely long run times for large networks. This paves the way for the need of accelerating this application by parallelizing multiple simulations. Parallel approaches will be given, discussed, and analyzed in the following

chapters.

Chapter 3

Serial Implementation

This chapter will discuss the serial implementations used as a comparison for the acceleration methods in the following chapters. First, the implementations for the algorithm are discussed and compared. Then the results of these implementations are given to use as comparisons for the methods discussed later.

3.1 Implementations

As mentioned before, this algorithm was implemented in both C and C++. The original C++ code, named Exact Stochastic Simulation (ESS), was written by James Michael McCollum for his thesis and dissertation research [17] and [18]. This algorithm was then optimized in C for this thesis research. Each of these implementations will be described below.

3.1.1 McCollum's Exact Stochastic Simulation - ESS

At the time of its release, McCollum's C++ implementation, ESS, was one of the most efficient available. It is implemented using C++ classes. This was done to make use of inheritance and polymorphism to implement multiple stochastic simulation methods including but not limited to the First Reaction [7], Direct [7], Next Reaction [4], Optimized Direct [2], and Sorting Direct [19]. However, for this research, only the Direct Method is examined. The details of all his implementations can be found in [19] and the source can be found in [17].

This implementation followed exactly with the algorithm outlined in Section 2.3.3. The C++ implementation for each method included several classes. These classes include, but are not limited to, a custom Vector class, Reaction Element and Reaction classes, a Model class, a Random number class, and a class for each method. Each of these is described below.

Vector Instead of using a large number of general C style arrays, a class was written to generate vectors to hold data. This class contains a few functions to add and retrieve data, resize the vector, and search the vector for a given value. As for the class members, there is a general C style array that holds the data and a variable to hold the size of the vector.

ReactionElement and Reaction To hold all of the information about a reaction, a class was created. A Reaction Element class that contains the species index and coefficient describes each species involved in a reaction. These are all linked together as products and reactants in the Reaction class by inputting them into a vector. This Reaction class also contains the rate constant of the reaction.

Model This class holds all of the information about the model including a set of three vectors holding the species counts, reactions, and species indices to be outputted.

Random In order to generate random numbers, a random number generation class was created. This class contains two functions: one to generate a uniformly distributed number and one to generate an exponentially distributed number.

The following classes are the heart of the implementations for each method. Each contain a pointer to the model, at least two vectors holding the species counts and reactions counts, and variables to hold the reaction index, current reaction time, and buffers for the random numbers. There are also other variables that are unique to the method. Each of these classes also contain three common functions called init (standing for initialize), step, and execute, but each do something different based on the method.

FirstReactionMethod This method is based on the proposed First Reaction Method by Gillespie [6]. This method has no extra variables that are unique to this implementation. This initialization function simply associates the model class and species vector to the FirstReactionMethod class variable. Next, the step function increments through each of the reactions calculating the propensity for each. Then, estimated times for each reaction are computed using random numbers [6]. The reaction with the smallest time is picked to be the reaction that is the first to occur and the one that will be executed. Finally, the reaction that is picked is executed in the execute function where the reaction count and species vectors are updated based on the reaction stoichiometry.

DirectMethod This method is based on the proposed Direct Method by Gillespie [6]. For this method, there are two extra variables that hold the following values: a double for the total propensity and a vector to hold the propensities of all the reactions. The initialization function for this method is the same as the First Reaction Method class described above. This class also contains all the functions needed to execute a single reaction in the algorithm: step and execute. The step function calculates all the propensities for each reaction, determines the reaction that will occur next, and calculates the current reaction time using a random number from a distribution parameterized by the total propensity. The execute function updates the species based on the reaction index that was chosen in the step function.

NextReactionMethod This method is based on the Next Reaction Method given by Gibson and Bruck [4]. Next, the NextReactionMethod class contains a two vectors that contain the propensity values and putative times for each reaction as well as a minimum heap. This minimum heap holds the putative times for each reaction so that the minimum of the times can be efficiently determined. The initialization function in this method does the same as the previous two as well as pre-calculates the propensities and the putative times for each reaction. The putative times are calculated by using a random number from a distribution parameterized by the total propensity. During this time, the putative times are also inputted into the minimum heap. Next, the step function grabs the reaction index corresponding to the minimum value in the heap as the reaction index, updates the reaction counts, and stores the putative time as the current reaction time. Finally, the execute

function updates the species populations and updates the putative times and the minimum heap based on the reaction dependencies.

OptimizedDirectMethod This method performs the Optimized Direct Method developed by Cao et.al. [2]. The OptimizedDirectMethod class is similar in content as the DirectMethod class. The main differences are that the initialization function pre-calculates the propensities and sum of propensities for all of the reactions as well as determining dependencies between reactions. Second, the step function only selects the reaction that will occur next, it does not calculate the propensities for all the reactions. Instead, in the execute function, after the species populations are updated, the dependent reaction propensities are recalculated. Beyond these differences, it is the same as the DirectMethod.

SortingDirectMethod This method performs the Sorting Direct Method developed by McCollum et.al. [19]. Finally, the SortingDirectMethod class is same as the OptimizedDirectMethod class with two differences. First, there is a new data structure that stores the reaction indices in approximately sorted order based on the occurrence of execution. Second, this sorting is completed in the step function after the reaction selection has occurred. Each time a reaction is selected, its position in the array is moved up, thereby reducing the search depth needed to find the reaction to be selected.

All of this is driven by a main function that continuously calls the step and execute function until either the simulation time is reached or a certain number of reactions occur. These limits are passed in as a command line argument. For each reaction, the program has the ability to output the species counts as directed by another command line argument.

The results and analysis of this implementation will be given in section 3.2

3.1.2 Accelerated Exact Stochastic Simulation - AESS

For this work, a new, optimized version of ESS is created. This new implementation is called Accelerated Exact Stochastic Simulation (referred to as AESS for the remainder of the thesis). AESS, implemented using the C programming language, is essentially the same as the ESS from

the previous section. However, all of the classes are flattened out to be a struct of variables and arrays that hold all the data needed. Also, all the class functions were flattened out to be standard functions that can be called anywhere. Doing this eliminates any of the typical C++ overhead costs that may be encountered with using a large number of classes.

As mentioned, all the data needed for the computations are stored into a struct. This struct contains the species populations, reactants, products, and rate constants. Each of these is stored as standard C arrays instead of creating a special data structure as in the previous implementation.

Also, the ESS implementation had a number of memory leaks as detected using Valgrind [20]. Using this application, a large number of the memory leaks could be detected and traced to determine where the problems were occurring. To address this problem, the AESS was written to avoid leaks or other memory problems. This was also verified using Valgrind.

This implementation was improved by not only changing the language but also the way some calculations were executed. In the ESS implementation, there were a large number of double precision calculations that could have been reduced down to only a few calculations. For instance, in the calculation of the propensities, a double precision variable is initialized with the scaled rate of the reaction then multiplied by all the species populations (of type long) that were type casted to be doubles. To reduce this, all the species populations are first multiplied together as longs then multiplied by the scaled rate at the end. In other words, for a reaction that contains 3 reactants, the ESS code would perform at least three double precision multiplications. With the optimized version, there would be one double precision multiplication after all the species populations are multiplied.

Each of the methods that were previously mentioned were implemented in this way. However, due to the transformation into the C programming language, the packaging of the implementations is not as structured. Therefore, there is a significant increase of repetitious code that slightly varies among the different methods that was avoided in the ESS version. This inherently leads to a larger total file size for the entire package. This is the downfall of doing this conversion.

All of the methods from ESS were implemented in the AESS package with the addition of the Logarithmic Method from Li and Petzold [13]. This method is the same as the Optimized Direct

Method except for the reaction selection stage. Instead of taking the scaled total propensity and subtracting it by each propensity until the result is zero, an array of partial sums is maintained so that a binary search for the following

$$subtotal[j1]scaledTotalPropensity < subtotal[j]$$

. This reduces the number floating point arithmetic operations by doing comparisons instead of subtractions. More details about this implementation and performance evaluation can be found in [13].

Besides these changes, everything else is the same. As will be shown in section 3.2, these improvements make a significant difference in performance.

3.1.3 StochKit

As a comparison, the AESS implementations are compared to a software package, StochKit, released by Linda Petzold's team at the University of California Santa Barbara [26]. This implementation is written in C++ and includes a number of different methods. These methods include the Direct Method, Optimized Direct Method, Slow-Scale SSA, and explicit and implicit tau-leaping. The StochKit website explains that this implementation is an efficient and extensible framework that scientists can use in their research.

In order to use this implementation, a new ProblemDefinition.cpp file must be generated for each model. This C++ file contains a number of functions that set up matrices of initial species populations, stoichiometry coefficients, and propensity equations. In order to feed in a different model, a new ProblemDefinition.cpp file must be created and the program must be recompiled.

Next, there must be a driver file that sets up the stochastic simulation environment. This driver file sets up the number of simulations to runs, the reaction time limit, initial seeds for the random number generation, and which method to use. The program must be recompiled if any of these settings are changed.

The StochKit is distributed to handle running until a specified reaction time is reached. In order to make an accurate comparison to the other implementations, a slight modification had to

be made to the code. In the StochRxn.cpp file that is included with the package, there is a function called StochRxn that runs through one simulation until a specified reaction time has been reached. To modify this, the portion of code that breaks the loop is altered to contain a counter to count the number of reactions occurrences and once that number is reach, the loop will break. Except for this change, the code comes as-is from the UCSB distribution page.

To help with the generation of these files, the StochKit team did provide a Java program to convert a file in the format of a Level 1 or 2 Systems Biology Markup Language [5] into these two files. However at the time of download, the Java program was not functioning, therefore each of the models needed to be converted by hand.

For this work, only the Direct Method will be examined and compared to the other implementations. A comparison of the Optimized Direct method would have been completed; however, at the time of writing, the StochKit came with a README file that explained there were some bugs in the Optimized Direct method implementation.

More information on StochKit can be found at the project's webpage [26].

3.2 Results

Each of these implementations are executed on a 2.26GHz Intel Xeon X5520 with 8192 KB cache. The runtimes of these simulations are timed using the built in C function, $gettimeofday$. Each of the given runtimes is an average of ten runs to ensure an accurate, average runtime is reported.

Each implementation is compiled using gcc version 4.1.2 with the highest optimization flags, -O3.

First, a run of 32768 simulations with 10000 reactions each for the First Reaction, Direct, Next Reaction, Optimized Direct, and Sorting Direct methods was performed for both of the ESS and the AESS implementations using all of the models in Appendix A. Tables 3.1 and 3.2 show these runtimes and the corresponding speedup of the AESS over the ESS. Figure 3.1 gives a graphical representation of these numbers to better see the comparison between the implementations.

As can be seen from the graphs, there is a significant speedup for AESS with all of the models in most of the methods. The smallest speedup (approximately 1.7x) was with the First Reaction

Table 3.1: Comparison of ESS and AESS implementations of each method with 32768 simulations, 10000 reactions each. Runtimes given in seconds.

(a) Results of AESS over ESS with 32768 Simulations, 10000 reactions each for 20Gene and AutoReg

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for 20Gene and AutoReg						
Models	20Gene			AutoReg		
	ESS	AESS	Speedup	ESS	AESS	Speedup
First	4605.65	2810.06	1.64	432.21	259.95	1.66
Direct	1565.85	226.25	6.92	195.82	54.07	3.62
Next	166.93	69.89	2.39	200.17	79.30	2.52
Optimized	405.44	57.36	7.07	172.76	49.00	3.53
Sorting	287.66	58.75	4.90	154.26	58.75	2.63
Logarithmic	N/A	182.72	N/A	N/A	52.19	N/A

(b) Results of AESS over ESS with 32768 Simulations, 10000 reactions each for Diffusion and DIMER

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for Diffusion and DIMER						
Models	Diffusion			DIMER		
	ESS	AESS	Speedup	ESS	AESS	Speedup
First	3107.37	1303.20	2.38	401.72	363.04	1.11
Direct	543.41	105.27	5.16	150.43	60.79	2.47
Next	687.32	275.45	2.50	99.95	72.17	1.38
Optimized	343.12	103.38	3.32	225.28	50.23	4.49
Sorting	355.77	71.62	4.97	159.13	49.98	3.18
Logarithmic	N/A	180.59	N/A	N/A	43.30	N/A

(c) Results of AESS over ESS with 32768 Simulations, 10000 reactions each for HSR and LambdaPhage

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for HSR and LambdaPhage						
Models	HSR			LambdaPhage		
	ESS	AESS	Speedup	ESS	AESS	Speedup
First	3071.27	1711.84	1.79	5868.37	3551.64	1.65
Direct	1009.36	204.54	4.93	2163.48	533.28	4.06
Next	498.42	235.04	2.12	1114.78	671.31	1.66
Optimized	318.12	74.76	4.26	706.94	130.69	5.41
Sorting	279.22	75.03	3.72	541.50	134.16	4.04
Logarithmic	N/A	142.38	N/A	N/A	270.86	N/A

Table 3.2: Comparison of ESS and AESS implementations of each method with 32768 simulations, 10000 reactions each. Runtimes given in seconds.

(a) Results of AESS over ESS with 32768 Simulations, 10000 reactions each for QS8 and SCHLOGL

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for QS8 and SCHLOGL						
Models	QS8			SCHLOGL		
	ESS	AESS	Speedup	ESS	AESS	Speedup
First	9296.95	5739.95	1.62	227.45	134.44	1.69
Direct	3033.48	547.19	5.54	126.12	44.52	2.83
Next	438.39	160.56	2.73	164.09	62.45	2.63
Optimized	728.54	76.62	9.51	151.58	48.32	3.14
Sorting	370.89	81.29	4.56	147.18	48.91	3.01
Logarithmic	N/A	342.01	N/A	N/A	47.38	N/A

(b) Results of AESS over ESS with 32768 Simulations, 10000 reactions each for Trans15Gene and Trans1Gene

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for Trans15Gene and Trans1Gene						
Models	Trans15Gene			Trans1Gene		
	ESS	AESS	Speedup	ESS	AESS	Speedup
First	2999.18	1838.93	1.63	423.96	259.22	1.64
Direct	1065.27	169.01	6.30	192.70	52.53	3.67
Next	185.79	77.82	2.39	119.38	55.41	2.15
Optimized	311.61	58.80	5.30	133.62	44.19	3.02
Sorting	267.09	60.17	4.44	115.72	45.39	2.55
Logarithmic	N/A	133.67	N/A	N/A	48.11	N/A

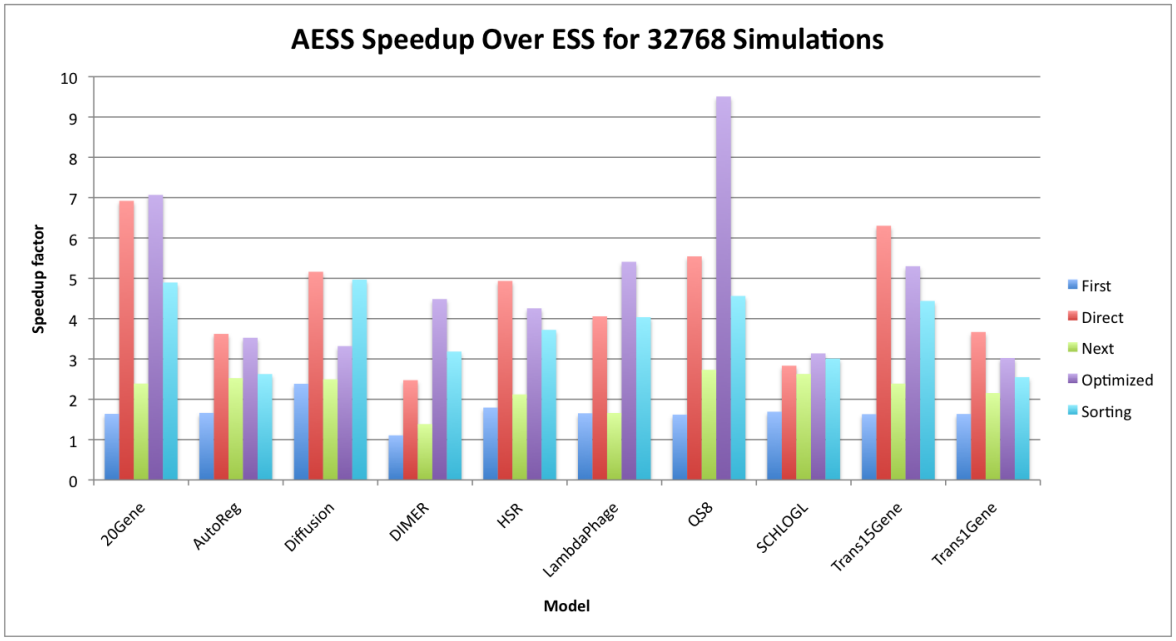


Figure 3.1: Speedup of AESS compared to ESS when running all the methods for 32768 simulations, 10000 reactions each.

method. This is due to there not being much room for optimization due to the large amount of exponentially distributed random numbers and floating point divisions. There are a total of $R * S * T$ (where R is the number of reactions in the model, S is the number of simulations, and T is the number of reactions per simulation) exponentially distributed random numbers and divisions each needed for the method to complete. For example for the HSR model, there were $61*32768*10000 = 19,988,480,000$ exponentially distributed random numbers and $19,988,480,000$ divisions. Due to this large number, this obviously dominates the performance of the algorithm.

The greatest speedup (approximately 9.5x) was with the Optimized Direct method implementation with the extremely large and complicated QS8 model. With this method, the number of exponentially distributed random numbers and floating point divisions was reduced by a factor of R because there is only one exponentially distributed random number and floating point division for each reaction. Also, this method maintains a dependency graph that allows for the recalculation of propensities for only the affected reactions. That leads the way for the optimizations making a significant difference as they are not inhibited as in the First Reaction method.

The other AESS methods still out-performed the ESS implementations; however, they were limited by other factors. For instance, the Next Reaction method had the extra load of keeping track of a minimum heap thereby offsetting the effects of the optimization techniques described earlier. Likewise, there was also the need to keep track of a list of dependencies for each reaction. Doing this reduced the amount of floating point operations needed for species updates; however, it increased the memory usage.

Also, it must be pointed out that each of the implementations has its own strengths and weaknesses when it comes to different types of models. For instance, the Optimized and Sorting Direct methods tend to perform the best with the larger models as the use of a dependency graph reduced the number of computations needed. However, with the smaller, less complicated models, it failed to achieve much of a speedup over the other models due to the need to manage the dependency graph as well as compute the small amount of calculations. When it comes to the smaller models, using either the Logarithmic or Direct Methods is more beneficial, not only for a runtime enhancement, but also for the lower memory usage.

To get another idea of the performance of these algorithms, table 3.3 shows the number of reactions that can be executed per second with each of the implementation methods. This table is put into a graphical view in figure 3.2 to get a more solid understanding of the performance of the improved AESS implementations.

As can be seen from figure 3.2, the AESS implementation is more efficient. What sticks out the most in that graph is the performance of the Direct Method for the SCHLOGL dataset with 7.1 million reactions per second. However, it is important to note that this data set is a very basic model that only has one species. To be more fair to the algorithms, a more realistic model such as the HSR performs at a rate of 1.6 million reactions per second for the AESS and 0.33 million reactions for the ESS.

Next, the Direct Method implementations are compared to the StochKit implementation. Each implementation is run for a varying number of simulations with 10000 reactions each. Table 3.5 gives the runtimes of these implementations while table 3.6 gives the speedup factor of the AESS over the other two. Likewise, figures 3.3(a) and 3.3(b) put these tables in graphical form. The

speedup values that are given show the speed up of AESS over the corresponding implementation.

Once again, it can be seen from the graphs that the AESS gives a significant improvement over the ESS program. This is contributed to the reasons given in section 3.1.2. This speedup was achieved with relatively little effort. The most difficult part of doing this was to flatten the C++ classes into standard C data structures. Once that was completed, all the functions were essentially the same with the exception of reordering the multiplications to reduce the number of double precision operations.

It can be seen that the StochKit is faster than the ESS implementation in all cases, however it is only faster than the AESS implementation when there is only one simulation being performed of the HSR model. In all the other simulations, the AESS implementation is at least 1.7 times faster and no more than 2.1 times faster. In it important to note that the StochKit and AESS are very close in performance with the realistic HSR model.

Finally, tables 3.3 and 3.4 show the approximate number of reactions per second each implementation can perform. This is important because it can really give a feel as to how efficient the AESS implementation is compared to the other two in a form that is more applicable to a scientist. Figure 3.4 shows these numbers for a test run of 32768 simulations to give a visual perspective on the numbers.

From all these results, the AESS provides the fastest implementation available.

3.3 Conclusions

This chapter has shown how making simple changes to the way the SSA code is written can speed up the original serial implementation. As mentioned, this is completed with essentially little effort. The main aspect to be considered when optimizing the C++ implementation was the memory usage. Memory needs to be allocated and accessed in an efficient way; otherwise, the performance of the program will be effected. By considering the way the memory is accessed and stored, the performance can be significantly improved. Also, using more fundamental programming techniques rather than relying on the convenience of class constructs of other languages can prove to be beneficial to the performance. Finally, the comparison between the AESS code and the StochKit

Table 3.3: Number of reactions per second of each method with 32768 simulations, 10000 reactions each

(a) Reactions per second of AESS and ESS with 32768 Simulations, 10000 reactions each for 20Gene and AutoReg

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for 20Gene and AutoReg				
Models	20Gene		AutoReg	
	ESS	AESS	ESS	AESS
First	7.11E4	1.17E5	7.58E5	1.26E6
Direct	2.09E5	1.45E6	1.67E6	6.06E6
Next	1.96E6	4.69E6	1.64E6	4.13E6
Optimized	8.08E5	5.71E6	1.90E6	6.69E6
Sorting	1.14E6	5.58E6	2.12E6	5.58E6
Logarithmic	N/A	1.79E6	N/A	6.28E6

(b) Reactions per second of AESS and ESS with 32768 Simulations, 10000 reactions each for Diffusion and DIMER

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for Diffusion and DIMER				
Models	Diffusion		DIMER	
	ESS	AESS	ESS	AESS
First	1.05E5	2.51E5	8.16E5	9.03E5
Direct	6.03E5	3.11E6	2.18E6	5.39E6
Next	4.77E5	1.19E6	3.28E6	4.54E6
Optimized	9.55E5	3.17E6	1.45E6	6.52E6
Sorting	9.21E5	4.58E6	2.06E6	6.56E6
Logarithmic	N/A	1.81E6	N/A	7.57E6

(c) Reactions per second of AESS and ESS with 32768 Simulations, 10000 reactions each for HSR and LambdaPhage

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for HSR and LambdaPhage				
Models	HSR		LambdaPhage	
	ESS	AESS	ESS	AESS
First	1.07E5	1.91E5	5.58E4	9.23E4
Direct	3.25E5	1.60E6	1.51E5	6.14E5
Next	6.57E5	1.39E6	2.94E5	4.88E5
Optimized	1.03E6	4.38E6	4.64E5	2.51E6
Sorting	1.17E6	4.37E6	6.05E5	2.44E6
Logarithmic	N/A	2.30E6	N/A	1.21E6

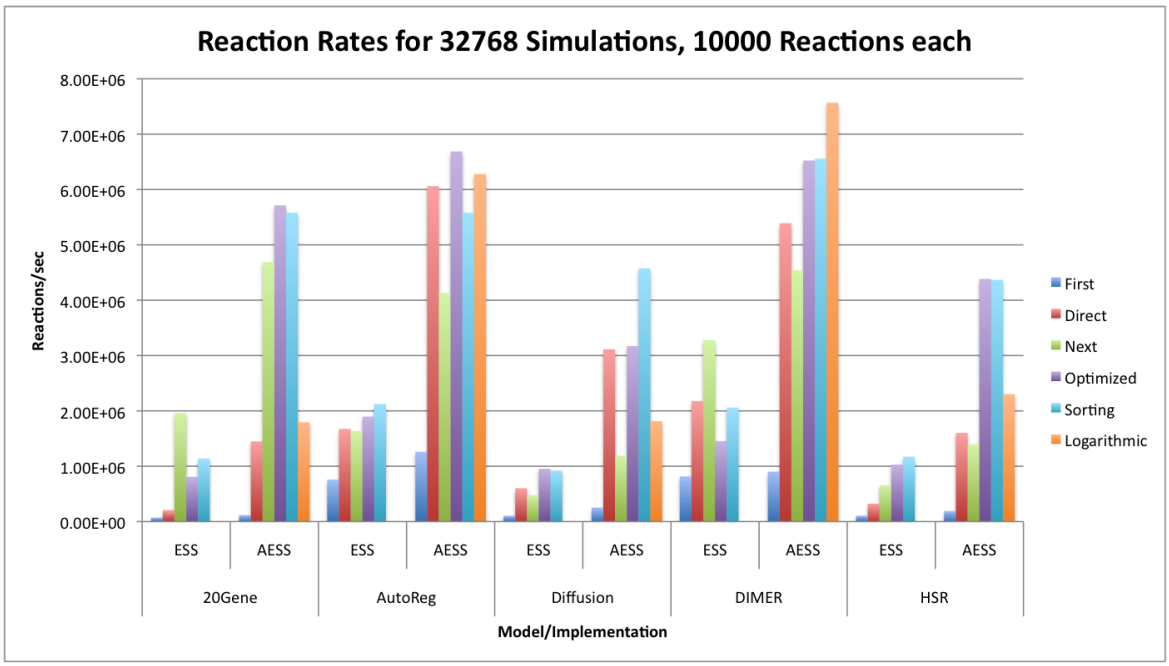
Table 3.4: Number of reactions per second of each method with 32768 simulations, 10000 reactions each

(a) Reactions per second of AESS and ESS with 32768 Simulations, 10000 reactions each for QS8 and SCHLOGL

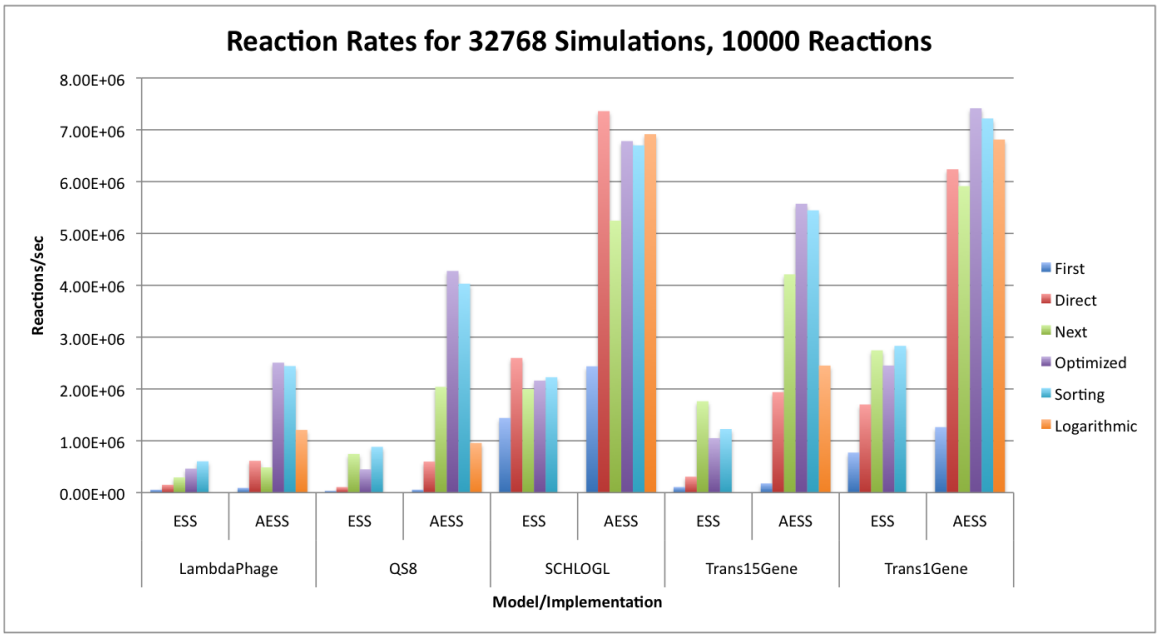
Results of AESS over ESS with 32768 Simulations, 10000 reactions each for QS8 and SCHLOGL				
Models	QS8		SCHLOGL	
	ESS	AESS	ESS	AESS
First	3.52E4	5.71E4	1.44E6	2.44E6
Direct	1.08E5	5.99E5	2.60E6	7.36E6
Next	7.47E5	2.04E6	2.00E6	5.25E6
Optimized	4.50E5	4.28E6	2.16E6	6.78E6
Sorting	8.83E5	4.03E6	2.23E6	6.70E6
Logarithmic	N/A	9.58E5	N/A	6.92E6

(b) Reactions per second of AESS and ESS with 32768 Simulations, 10000 reactions each for Trans15Gene and Trans1Gene

Results of AESS over ESS with 32768 Simulations, 10000 reactions each for Trans15Gene and Trans1Gene				
Models	Trans15Gene		Trans1Gene	
	ESS	AESS	ESS	AESS
First	1.09E5	1.78E5	7.73E5	1.26E6
Direct	3.08E5	1.94E6	1.70E6	6.24E6
Next	1.76E6	4.21E6	2.74E6	5.91E6
Optimized	1.05E6	5.57E6	2.45E6	7.42E6
Sorting	1.23E6	5.45E6	2.83E6	7.22E6
Logarithmic	N/A	2.45E6	N/A	6.81E6



(a) Reaction rates of each of the serial methods



(b) Reaction rates of each of the serial methods

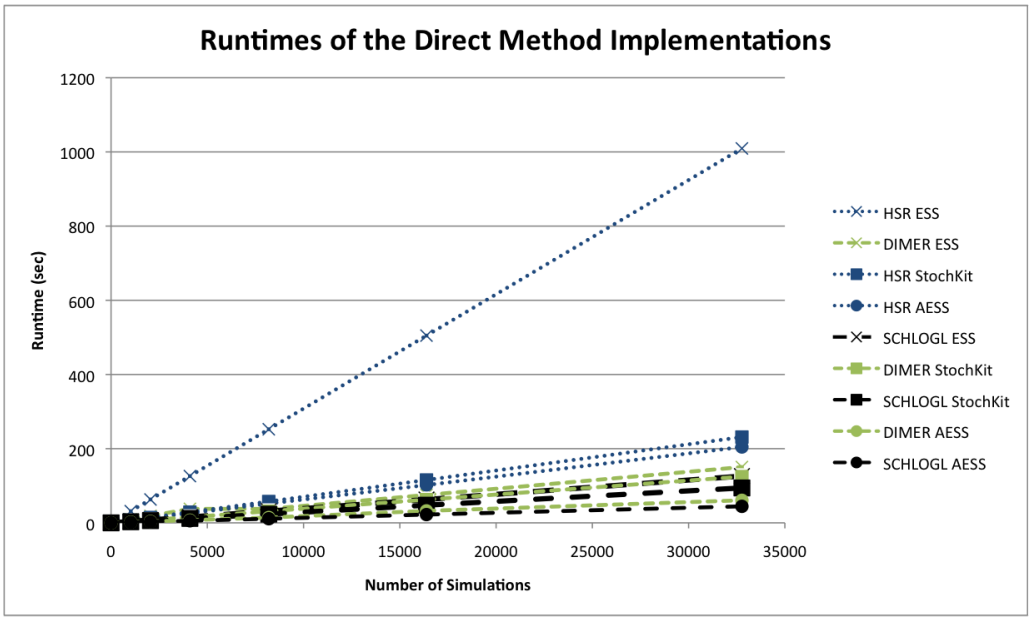
Figure 3.2: Reaction rates of running all the methods for 32768 simulations, 10000 reactions each, of the ESS and AESS implementations.

Table 3.5: Runtimes of StochKit, ESS, and AESS Direct Method implementations. Runtimes given in seconds.

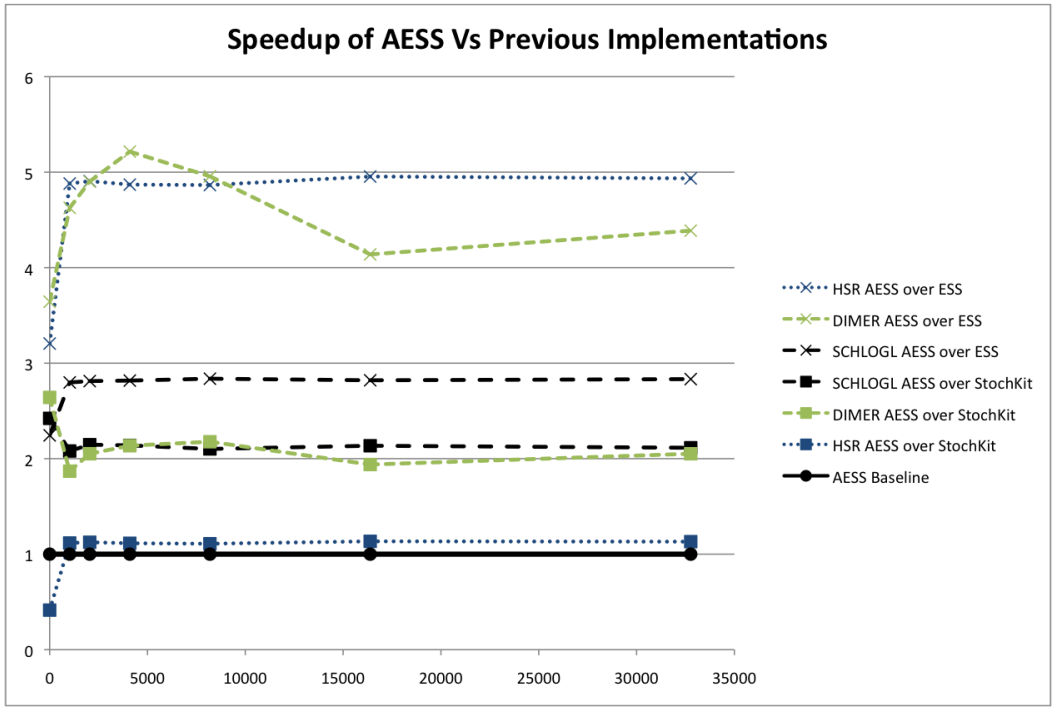
Runtimes for Multiple Implementations using 10000 reactions per simulation									
Simulations	SCHLOGL			DIMER			HSR		
	StochKit	ESS	AESS	StochKit	ESS	AESS	StochKit	ESS	AESS
1	5.09 E-3	4.72 E-3	2.10 E-3	7.07 E-3	9.75 E-3	2.68 E-3	4.57 E-3	3.54 E-2	1.10 E-2
1024	2.92	3.92	1.40	3.80	9.40	2.03	7.21	31.53	6.46
2048	5.99	7.85	2.79	7.86	18.79	3.83	14.45	63.04	12.85
4096	11.93	15.71	5.58	15.40	37.61	7.21	28.84	126.07	25.89
8192	23.41	31.60	11.14	31.28	71.20	14.36	57.54	252.34	51.87
16384	47.64	62.94	22.31	62.60	133.66	32.29	115.59	504.80	101.89
32768	94.12	126.12	44.52	124.71	266.74	60.79	231.33	1009.36	204.54

Table 3.6: Speedup factors of the AESS implementation over the StochKit and ESS

Speedup of AESS Implementation									
Simulations	SCHLOGL			DIMER			HSR		
	StochKit	ESS	AESS	StochKit	ESS	AESS	StochKit	ESS	AESS
1	2.42	2.24	1	2.64	3.64	1	0.41	3.21	1
1024	2.08	2.80	1	1.87	4.63	1	1.12	4.88	1
2048	2.15	2.81	1	2.05	4.90	1	1.12	4.91	1
4096	2.14	2.82	1	2.14	5.22	1	1.11	4.87	1
8192	2.10	2.84	1	2.18	4.96	1	1.11	4.86	1
16384	2.14	2.82	1	1.94	4.14	1	1.13	4.95	1
32768	2.11	2.83	1	2.05	4.39	1	1.13	4.93	1



(a) Runtimes of the Direct method of the StochKit, ESS, and AESS implementations.



(b) Speedup of the AESS implementation over the StochKit and ESS.

Figure 3.3: Results from running the StochKit, ESS, and AESS runs.

Table 3.7: Number of Reactions per Second with 32768 simulations, 10000 reactions each

Number of Reactions per Second with 32768 simulations, 10000 reactions each									
Simulations	SCHLOGL			DIMER			HSR		
	StochKit	ESS	AESS	StochKit	ESS	AESS	StochKit	ESS	AESS
1	1.96 E6	2.12 E6	4.76 E6	1.42 E6	1.03 E6	3.74 E6	2.19 E6	2.83 E5	9.06 E5
1024	3.51 E6	2.61 E6	7.30 E6	2.70 E6	1.09 E6	5.04 E6	1.42 E6	3.25 E5	1.59 E6
2048	3.42 E6	2.61 E6	7.33 E6	2.60 E6	1.09 E6	5.34 E6	1.42 E6	3.25 E5	1.59 E6
4096	3.43 E6	2.61 E6	7.35 E6	2.66 E6	1.09 E6	5.68 E6	1.42 E6	3.25 E5	1.58 E6
8192	3.50 E6	2.59 E6	7.36 E6	2.62 E6	1.15 E6	5.70 E6	1.42 E6	3.25 E5	1.58 E6
16384	3.44 E6	2.60 E6	7.34 E6	2.62 E6	1.23 E6	5.07 E6	1.42 E6	3.25 E5	1.61 E6
32768	3.48 E6	2.60 E6	7.36 E6	2.63 E6	1.23 E6	5.39 E6	1.42 E6	3.25 E5	1.60 E6

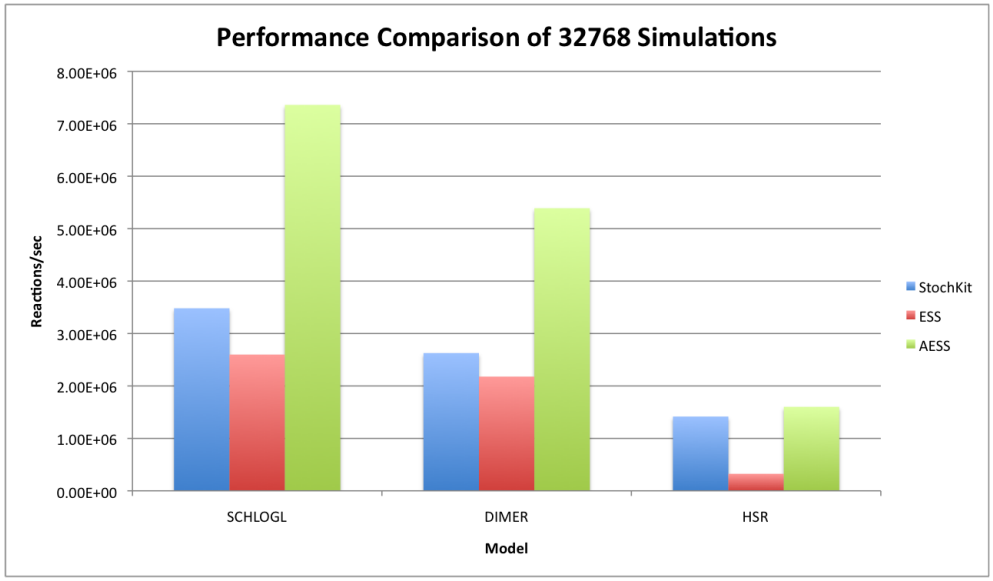


Figure 3.4: Number of reactions per second each implementation can execute.

proves that even the best algorithms can be improved, even if it is a slight change. Based on all these comparisons, the AESS version is the fastest available. Since the AESS implementation proved be such a significant speedup, it will be the basis of comparison for all of the other implementations presented later.

Chapter 4

Streaming SIMD Extensions Implementation

This chapter will discuss the streaming SIMD extensions (SSE) implementation of the stochastic simulation algorithm. First, background information of SSE is presented, then the implementation is examined. Finally, the results of this implementation are given and compared with the optimized AESS implementation described in chapter 3.

4.1 Streaming SIMD Extensions - SSE

Intel released a set of instructions and registers with their processors that allowed for vectorization of operations. This new technology is called streaming SIMD extensions (referred to as SSE for the rest of the thesis). With the use of these new vectorization capabilities, a programmer could achieve a possible 4 times speedup with their algorithms.

These special registers, as can be seen in figure 4.1, can hold many items at one time. For instance, the register is able to hold up to four floats, two doubles, or four 32-bit integers.

Once these registers are loaded with data, numerous arithmetic and logic operations can be performed with all the numbers at the same time, such as add, subtract, multiply, less than, or greater than. This is the single instruction, multiple data (SIMD) aspect of the SSE that allows for algorithm acceleration. Figure 4.2 shows how the SIMD multiplication works with these

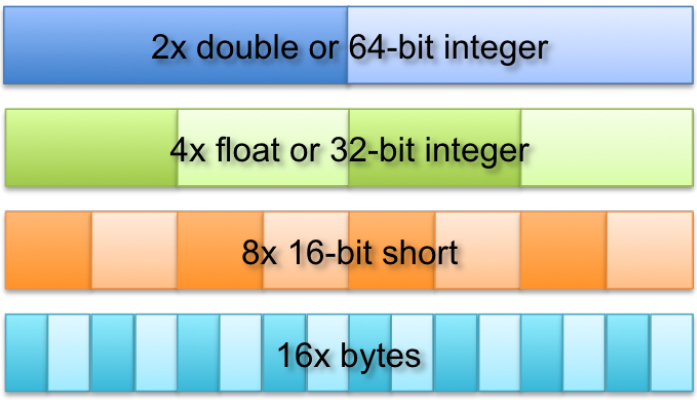


Figure 4.1: SSE Register layout

registers. Essentially, all four floats are multiplied together and stored into the resultant register simultaneously. Therefore, if a program were to only multiply two arrays of size N, only $N/4$ multiplications would be needed, leading to a four times speedup.

Originally, in order to take advantage of these vectorization techniques, the programmer would have to write their SSE code in assembly. However, there are now intrinsic functions that are available to ease the programming headaches [16]. These intrinsics allow for the programmer to call C functions that will load or store data into registers or perform arithmetic or logic operations without having to dive into the assembly language.

4.2 Implementation

As mentioned before, the use of SSE instructions has the potential to have a benefit on the stochastic simulation algorithm. The Direct Method is chosen for this implementation due to the reasons give in Section 2.3.3. To start this implementation out, the stochastic simulation will be vectorized by performing four simulations at one time (similar to the CUDA implementation examined later in chapter 6).

First, the memory had to be set up in a way so that one data point for four simulations is stored consecutively. For instance, the propensities are stored as shown in figure 4.3. This was done for the propensity, species population, reaction index, and propensity sum arrays. To load the data

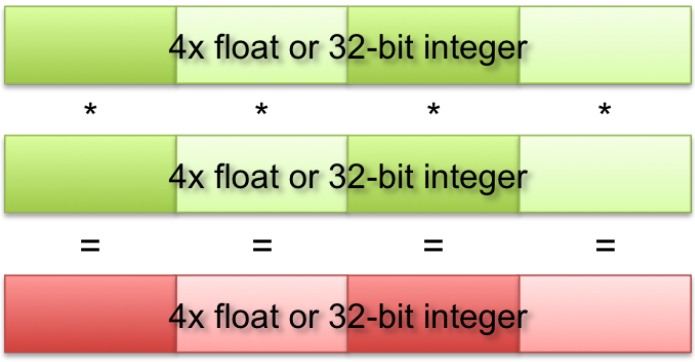


Figure 4.2: SSE multiplication

into the SSE registers requires the data to be stored into memory consecutively, therefore requiring this reconstruction of memory layout.

Next, the propensities are calculated. This is typically the most expensive step due to the number of multiplications needed for each reaction. To do these calculations for all of the simulations concurrently, the rate constant is first loaded into all four entries of the register, the species populations are stored into a register, and the coefficient count is stored into the register. The populations are subtracted by the coefficient count then multiplied by the data in the register that originally stored the rate constant. Once all of the calculations for a reaction are completed, the data is then stored back into memory. All of this can be see in figure 4.4 containing a snippet of code that calculates the propensities for all reactions of all four simulations.

Once the propensity is calculated, it must then be summed with the other propensities. This is done with the $total = \_mm\_add\_ps(total, a)$; line. Once all the propensities have been calculated and summed together, the sum is then stored back to memory to be used in later calculations.

Next, the reaction selection step is executed. Unfortunately, there was no clever way to take full advantage of SSE for this step of the algorithm. However, SSE could be used in order to multiply the propensity totals by uniformly distributed random numbers. Once these were multiplied, the reaction index for each simulation is calculated serially in the same way as the serial AESS code. Figure 4.5 is a snippet of code that takes advantage of SSE.

Finally, the species are updated based of the reaction selection. Since there is no way to guarantee the selected reaction's data to be stored in memory consecutively, there is no way to use

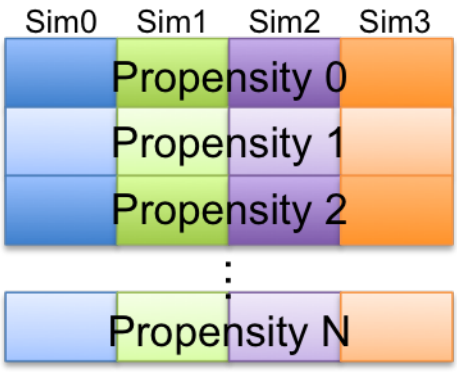


Figure 4.3: SSE propensity calculation

```
void indProp(ESS* d, Prop* p, int i){
          int j;
          float k;
          float prop[SIM];
          a = _mm_load1_ps(&d->rateConstant[i]);
          //calculate the propensity for a reaction for 4 simulations
          for(j=0; j<d->reactSize[i]; j++) {
                    for(k=0; k<d->reactCoeff[i][j]; k++) {
                              b = _mm_loadu_ps(d->species[d->reactIndex[i][j]]);
                              c = _mm_load1_ps(&k);
                              e = _mm_sub_ps(b,c);
                              a = _mm_mul_ps(a,e);
                    }
          }
          _mm_storeu_ps(p->prop[i],a);
}

void calcProp(ESS* d, Prop* p){
          int i;
          float temp = 0;
          total = _mm_load1_ps(&temp);
          //calculate all the propensities
          for(i=0;i<d->reactionSize;i++){
                    indProp(d,p,i);
                    total = _mm_add_ps(total, a);
          }
          _mm_store_ps(p->total,total);
}
```

Figure 4.4: SSE propensity calculation C code

```
...
float temp[SIM] __attribute__((aligned(16)));
for(i=0;i<SIM;i++){
          temp[i] = rand()*randmaxRecip;
}
float scaledRate[SIM] __attribute__((aligned(16)));
a = _mm_load_ps(temp);
b = _mm_load_ps(p->total);
_mm_store_ps(temp,b);
c = _mm_mul_ps(a,b);
_mm_store_ps(scaledRate,c);
...
```

Figure 4.5: SSE Reaction Selection C code

SSE to update the species populations. Therefore, all simulations are updated serially in the same way as the serial AESS code.

The following section will show the results of this implementation in comparison to the serial AESS code.

4.3 Results

For this implementation, the algorithm was run on a few different processors in order to get comparisons of different architecture. First, the serial AESS and SSE codes were run on a 2.26GHz Intel Xeon X5520 with 8192 KB cache. Next, to compare with Intel's SSE, the AMD SSE code was run on a 1.8GHz Dual Core AMD Opteron Processor 265 with 1024 KB cache. Finally, to compare the difference in the ICC and GCC compilers for SSE, the code was run on a 2.67GHz Intel Core i7 with 8192 KB cache. The runtimes of these simulations are timed using the built in C function, $gettimeofday$. Each of the given runtimes is an average of ten runs to ensure an accurate, average runtime is reported.

All of the following runtimes have been observed with the programs being compiled with gcc version 4.1.2 with the highest optimization flags, -O3, unless otherwise noted. The Intel compiler used is icc version 11.1 using the highest optimization flags, -O3.

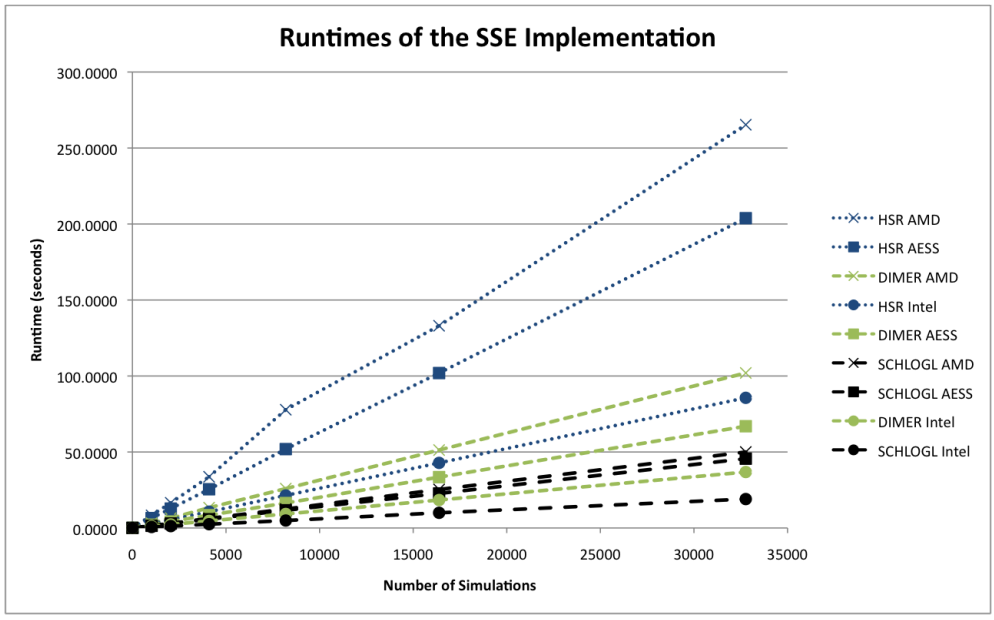
First, the runtimes and speedups of the AESS and SSE code are given in tables 4.1 and 4.2, followed by the corresponding graphs in figures 4.6.

Table 4.1: SSE runtimes for 10000 reactions - AMD vs Intel

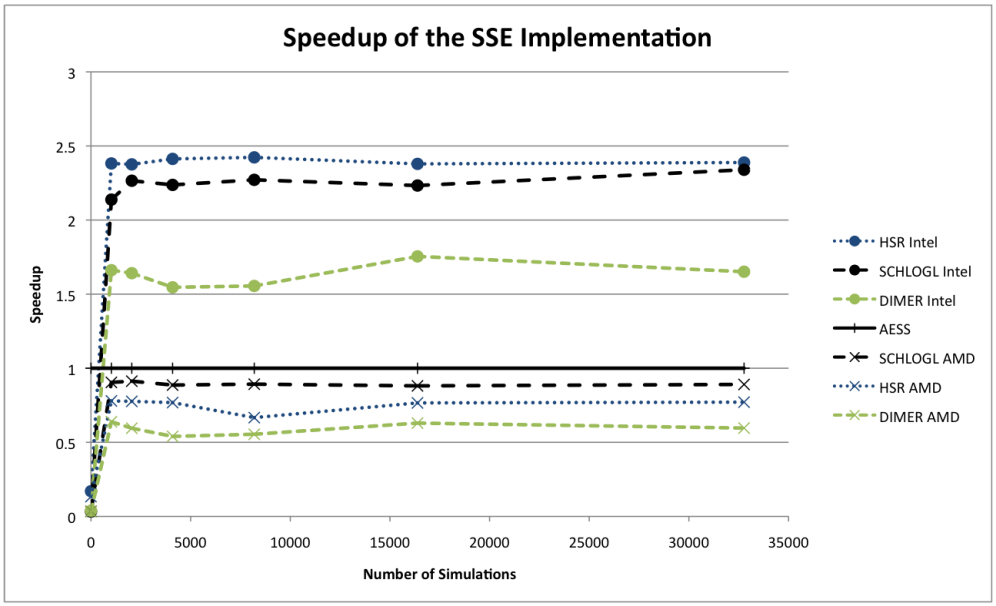
Runtimes of SSE with 10000 Reactions per Simulation									
Simulations	SCHLOGL			DIMER			HSR		
	AESS	AMD	Intel	AESS	AMD	Intel	AESS	AMD	Intel
1024	1.40	1.55	0.66	2.03	3.18	1.22	6.46	8.29	2.71
2048	2.79	3.06	1.23	3.83	6.44	2.33	12.85	16.55	5.41
4096	5.58	6.29	2.49	7.21	13.36	4.66	25.89	33.71	10.73
8192	11.14	12.48	4.90	14.36	25.91	9.24	51.87	77.81	21.41
16384	22.31	25.33	9.99	32.29	51.29	18.41	101.89	133.04	42.83
32768	44.52	49.99	19.03	60.79	102.00	36.82	204.54	265.31	85.64

Table 4.2: SSE speedups for 10000 reactions - AMD vs Intel

Speedup of SSE Implementation									
Simulations	SCHLOGL			DIMER			HSR		
	AESS	AMD	Intel	AESS	AMD	Intel	AESS	AMD	Intel
1024	1	0.90	2.14	1	0.64	1.66	1	0.78	2.38
2048	1	0.91	2.27	1	0.60	1.64	1	0.78	2.38
4096	1	0.89	2.24	1	0.54	1.55	1	0.77	2.41
8192	1	0.89	2.27	1	0.55	1.56	1	0.67	2.42
16384	1	0.88	2.23	1	0.63	1.75	1	0.77	2.38
32768	1	0.89	2.34	1	0.60	1.65	1	0.77	2.39



(a) Runtimes of all three models



(b) Speedup of SSE implementation

Figure 4.6: SSE results for 10000 reactions - AMD vs Intel

From the graphs, it can be seen that the SSE implementation was successful in providing a speedup over the serial AESS implementation on the Intel processor. Due to some of the complexities of not being able to use SSE for all of the calculations for the simulations (such as reaction selection), it was impossible to achieve a speedup closer to the 4 times possible. The important thing to note here is that the SSE ran on average more than twice as fast as the serial AESS implementation on the same processor. The AMD processor was actually slower than the serial AESS implementation possibly due to the difference in processing power compare to the Intel. Also, the baseline AESS implementation was run on an Intel process. Therefore this comparison may not be a fair one because of these factors.

Looking at figure 4.6(b), it can be seen that the realistic model, HSR, is almost 2.5x faster than the serial AESS implementation. This is significant because it means that this implementation can potentially be applied to more real world models and still achieve a speedup. Even though this is not an enormous speedup, this is still sufficient for the amount of work needed to implement it.

Table 4.3 shows the number of a reactions per second that each of the implementations achieves. Figure 4.7 gives a graphical view of the data corresponding to 32768 simulations. It can be seen that the Intel SSE implementation gives speedup over both of the serial AESS and AMD SSE implementations.

Next, table 4.4 gives the runtimes of using the same code on a 2.67GHz Intel Core i7 with 8192 KB cache compiled with two different compilers: GNU's gcc (version 4.3.3) and Intel's icc (version

Table 4.3: SSE number of reactions per second each implementation can execute.

Reactions per Second of SSE Implementation									
Simulations	SCHLOGL			DIMER			HSR		
	AESS	AMD	Intel	AESS	AMD	Intel	AESS	AMD	Intel
1	1.53E5	1.61E5	1.57E5	1.25E5	1.21E5	1.44E5	1.58E5	1.21E5	1.54E5
1024	6.99E6	6.61E6	1.56E7	4.80E6	3.22E6	8.38E6	1.57E6	1.24E6	3.78E6
2048	6.98E6	6.69E6	1.66E7	4.81E6	3.18E6	8.77E6	1.60E6	1.24E6	3.79E6
4096	7.13E6	6.51E6	1.64E7	4.85E6	3.07E6	8.78E6	1.61E6	1.22E6	3.82E6
8192	7.05E6	6.56E6	1.67E7	4.42E6	3.16E6	8.87E6	1.58E6	1.05E6	3.83E6
16384	7.21E6	6.47E6	1.64E7	4.90E6	3.19E6	8.90E6	1.61E6	1.23E6	3.83E6
32768	7.18E6	6.55E6	1.72E7	4.89E6	3.21E6	8.90E6	1.61E6	1.24E6	3.83E6

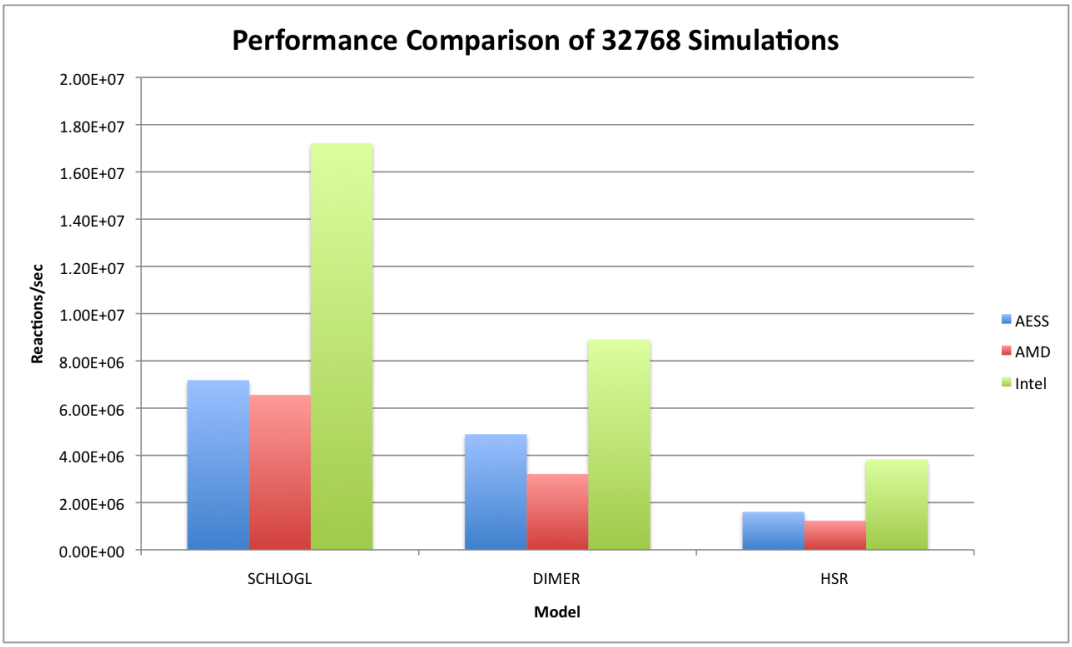


Figure 4.7: SSE number of reactions per second each implementation can execute.

11.1). Once again, the highest optimization flags, -O3, were used when compiling. Figure 4.8 shows these runtimes and the speedup of icc over gcc in a graph form.

From these figures, it can be seen that simply using the Intel compiler over the GNU compiler, one can achieve a greater speedup when running on an Intel processor. We suspect this is due to Intel having more optimizing schemes for the code with compiling than that of GNU due to its underlying expansive knowledge of the architectural implementation of SSE on its processors.

4.4 Conclusion

This chapter has shown how performing multiple simulations concurrently can optimize the serial AESS implementation of the stochastic simulation algorithm. With the help of streaming SIMD extensions, this was possible to accomplish with a relatively small amount of effort. The most difficult part of this implementation was reconstructing the memory so that the data for each simulation would be consecutive. Once that was completed, the reset was simply calling a few SSE functions to perform the operations needed. Although this implementation did not provide a

Table 4.4: AESS SSE comparison between the GCC and ICC compilers. The speed ups given are the ICC compiler speedup over the GCC compiler

(a) SSE results for 10000 reactions of the SCHLOGL model

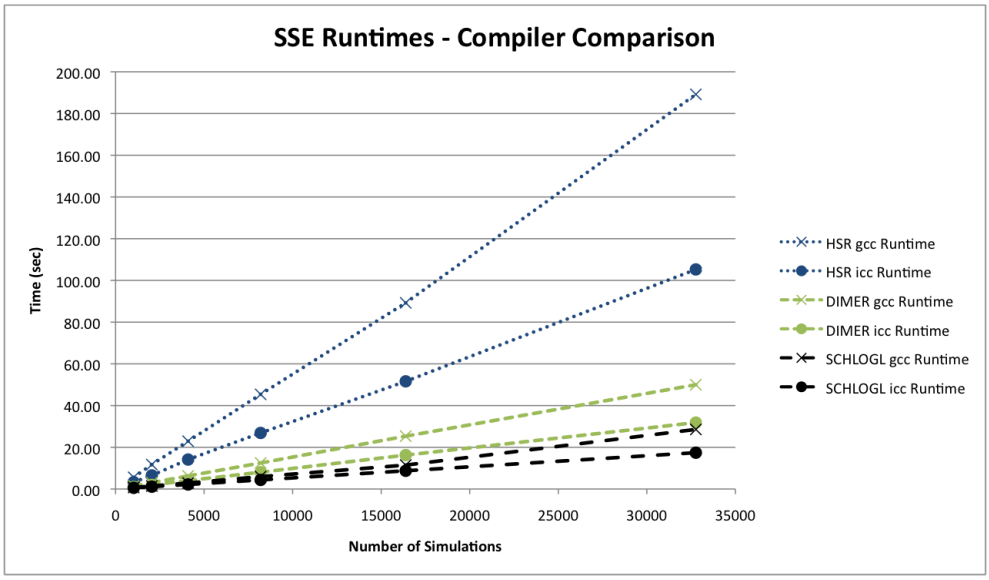
SCHLOGL for 10000 reactions			
Simulations	gcc Runtime	icc Runtime	Speedup
1024	0.73	0.56	1.30
2048	1.44	1.12	1.29
4096	2.87	2.18	1.32
8192	5.94	4.36	1.36
16384	11.47	8.74	1.31
32768	28.57	17.42	1.64

(b) SSE results for 10000 reactions of the DIMER model

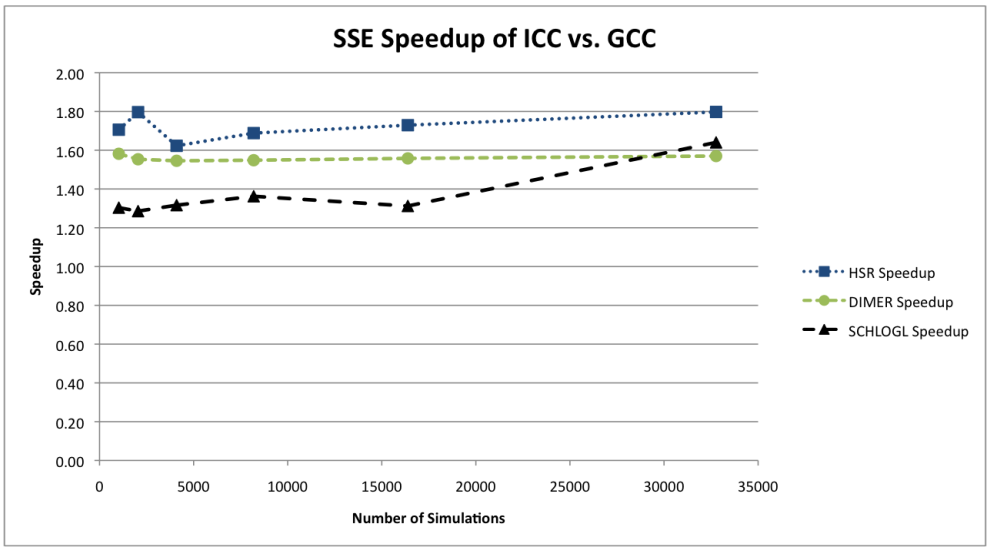
DIMER for 10000 reactions			
Simulations	gcc Runtime	icc Runtime	Speedup
1024	1.55	0.98	1.58
2048	3.06	1.97	1.55
4096	6.29	4.07	1.55
8192	12.48	8.06	1.55
16384	25.33	16.26	1.56
32768	49.99	31.84	1.57

(c) SSE results for 10000 reactions of the HSR model

HSR for 10000 reactions			
Simulations	gcc Runtime	icc Runtime	Speedup
1024	3.29	1.85	1.78
2048	6.53	3.68	1.77
4096	14.13	7.56	1.87
8192	26.88	15.03	1.79
16384	51.61	29.39	1.76
32768	105.26	58.57	1.80



(a) Runtimes of all three models



(b) Speedup of SSE implementation

Figure 4.8: SSE results for 10000 reactions - ICC vs. GCC

tremendous speedup, it is definitely an approach a programmer should consider due to its ease of use and little effort to get working code.

Something else that was interesting was the speedup achieved on an Intel processor by simply switching compilers. The use of the Intel compiler proved to be beneficial with literally no extra effort needed to achieve a speedup. Therefore, it is safe to say that if one were using an Intel processor, it would be useful to use Intel's icc compiler to ensure as much automated optimization as possible.

Chapter 5

Message Passing Interface Implementation

This chapter will discuss how a multicore implementation of the stochastic simulation algorithm with the use of clusters of computers and the message passing interface specification (MPI) [25]. The results of this implementation is then compared to the serial implementation.

5.1 Multicore CPU Systems

As demand for more efficient systems increases, the need for multicore CPU systems becomes more prevalent. However, in order to take full advantage of these multicore systems, programs need to be rewritten for parallelization to ensure the use of as many cores as possible. However, this is sometimes not an easy task. With the use of some programming language libraries and specifications, this tasked is simplified to a point.

There are a couple types of multicore CPU systems. First, there are integrated circuits that contain more than one core. Each of these cores have their own L1 cache and can communicate with the other cores through the use of a shared memory in an L2 cache as shown in figure 5.1.

The next type of multicore system to be described is a computer cluster. Computer clusters are actually a distributed system in which multiple computers are connected through network connections. Each of the computer nodes can contain multiple multicore processors and each can

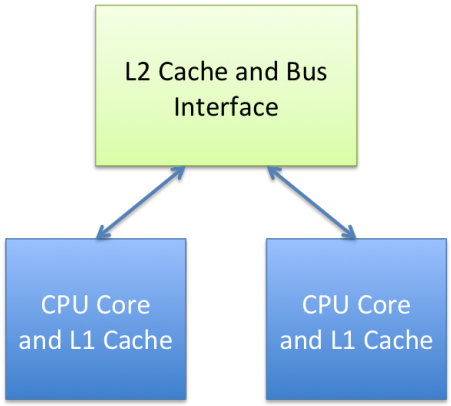


Figure 5.1: Example of the integrated circuit multicore CPU.

communicate with all the other nodes in the network. Figure 5.2 shows an example of a computer cluster.

One possible way to write a program to work with this distributed system of computer clusters is use a specification and library called the message passing interface (MPI). MPI allows for multiple processes to be fork off simultaneously while giving each process the ability to communicate with the other processes. This communication can be used to share data with each process or can be used for process synchronization. MPI can be used not only for distributed systems, but also between cores on a single system. Referring back to figure 5.2, each core can actually communicate the other cores in its corresponding node, as well as with the cores across the entire network. These communication lines were left out in the diagram for simplicity.

5.2 Implementation

For this implementation, the parallelization is attempting to speedup ensembles of simulations, not a single simulation. To accomplish this, the AESS serial implementation was modified to distribute the load across all the processes within the system in a master-slave setup using MPI. At the beginning of the program, the total number of simulations is divided by the number of processes to get Z:

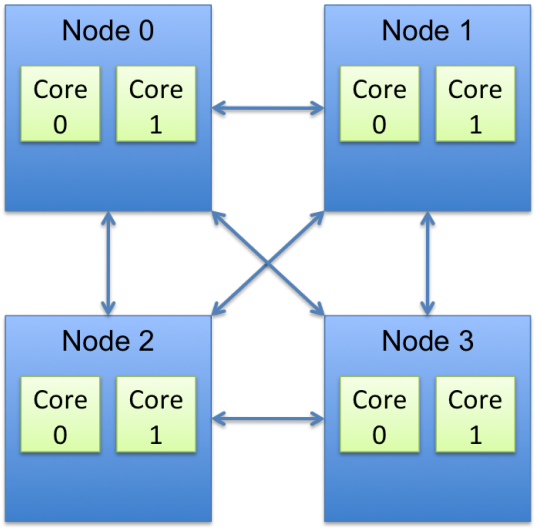


Figure 5.2: Example of a computer cluster.

$$Z = \frac{\text{Total number of simulations}}{\text{Total number of processes}}$$

Therefore, if there were 16 processes and a total 8192 simulations, each process would do 8192/16=512 simulations. Each process then serially executes Z simulations in the same way the AESS serial implementation was explained in Section 3.1.2. When each slave process has completed its distribution of simulations, it sends a confirmation packet to the master process with rank 0 and ends. Once the master process completes its share of simulations and receives all of the confirmations from all the processes, the final time is printed and the program exits successfully.

5.3 Results

This implementation is executed on a cluster of four nodes each containing two 2.26GHz quad core Intel Xeon X5520s with 8192 KB cache. The runtimes of these simulations are timed using the built in C function, *gettimeofday*. Each of the given runtimes is an average of ten runs to ensure an accurate, average runtime is reported.

Each implementation is compiled using gcc version 4.1.2 with the highest optimization flags, -O3.

First the runtimes and speedups of the serial AESS and MPI implementations are given in table 5.1 and visualized in figure 5.3.

It can be seen from the figures that the speedup of the MPI implementation is roughly a linear relationship with the number of processes used. This is what was expected because each process is only doing a subset of simulations using the same serial implementation as the AESS.

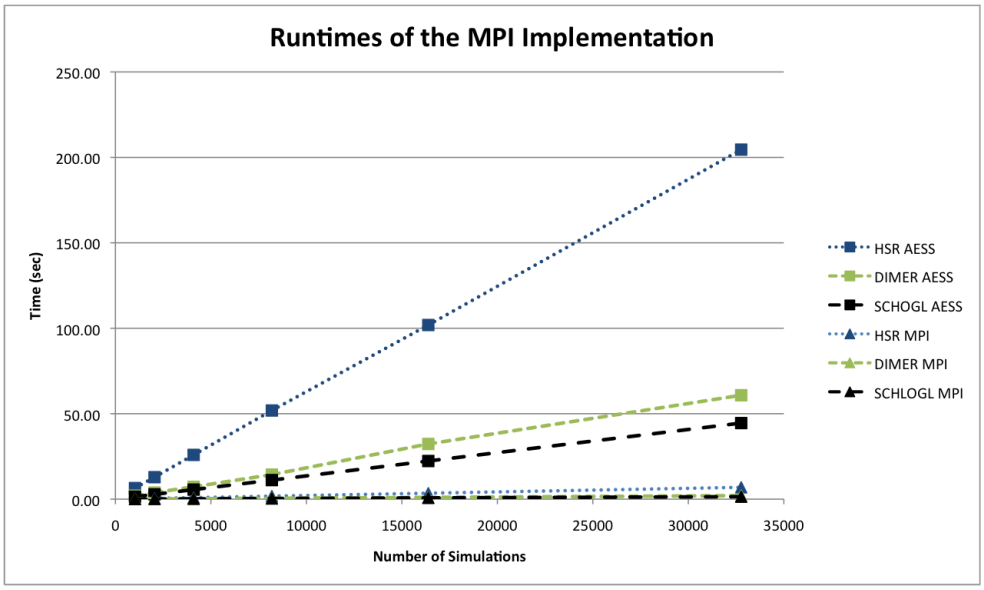
There are a couple of interesting points to note about this implementation. There are some variations in the speedup that go above and below the expected linear speedup factor. This is due to each run sharing information in the cache without a large number of conflicts. Since each node has two quad-core processors, there are a total of eight simulations being executed per node meaning four per socket. Each quad core processor has shared memory that the four simulations are accessing simultaneously. Therefore, with the DIMER model, the information needed for all four simulations was likely to be in cache at the time of access thereby meaning there were less cache misses and more cache hits. However, with the HSR model, the four simulations needed information that was not stored in cache resulting in more cache misses, increasing the total runtime.

Next, table 5.2 gives the performance of the MPI implementation compared with the AESS implementation in terms of reactions per second for 32768 simulations, 10000 reactions each. Figure 5.4 puts this data in a visual form.

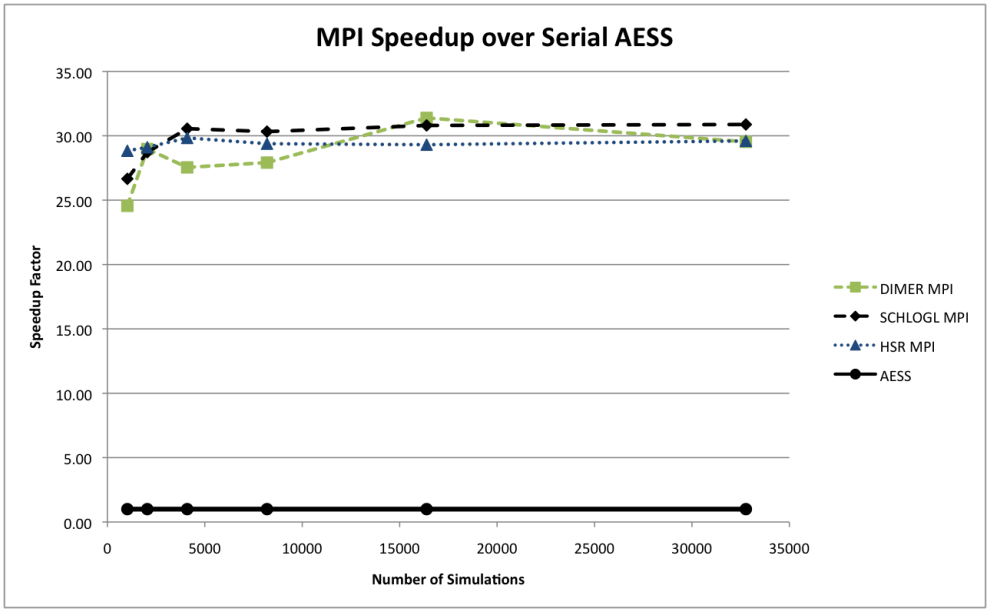
It can be seen that the MPI implementation is significantly more efficient than the serial implementation. The SCHLOGL model results in more than 227 million reactions per second for the MPI implementation, whereas the AESS serial implementation only reaches 7.36 million reactions

Table 5.1: Results of the MPI implementations with 10000 reactions per simulation.

Comparison of Serial AESS and MPI implementations with 10000 reactions per simulation									
Simulations	SCHLOGL			DIMER			HSR		
	AESS	MPI	Speedup	AESS	MPI	Speedup	AESS	MPI	Speedup
1024	1.40	0.05	26.66	2.03	0.08	24.56	6.46	0.22	28.84
2048	2.79	0.10	28.72	3.83	0.13	28.97	12.85	0.44	29.11
4096	5.58	0.18	30.55	7.21	0.26	27.54	25.89	0.87	29.83
8192	11.14	0.37	30.32	14.36	0.51	27.92	51.87	1.77	29.38
16384	22.31	0.72	30.80	32.29	1.03	31.39	101.89	3.48	29.31
32768	44.52	1.44	30.88	60.79	2.06	29.53	204.54	6.91	29.60



(a) Runtimes of all three models.



(b) Speedup of MPI implementation.

Figure 5.3: MPI results for 10000 reactions.

Table 5.2: MPI performance in terms of reactions per second for 10000 reactions per simulation.

MPI Reactions Per Second Performance			
Simulations	SCHLOGL	DIMER	HSR
1024	1.95 E8	1.24 E8	4.57 E7
2048	2.11 E8	1.55 E8	4.64 E7
4096	2.24 E8	1.56 E8	4.72 E7
8192	2.23 E8	1.59 E8	4.64 E7
16384	2.26 E8	1.59 E8	4.71 E7
32768	2.27 E8	1.59 E8	4.74 E7

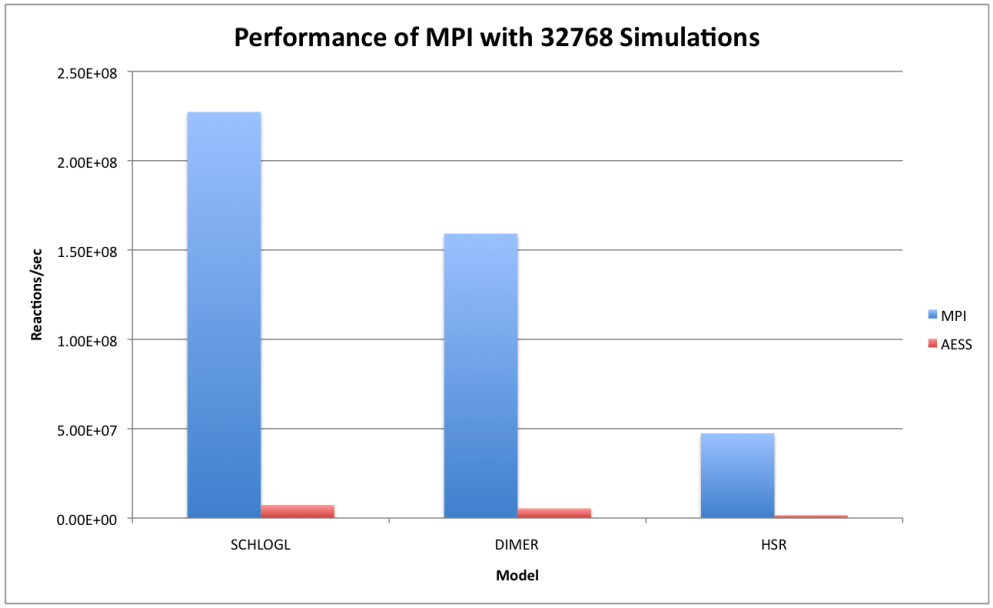


Figure 5.4: MPI performance in terms of reactions per second for 32768 simulations, 10000 reactions each.

per second. As can be seen from table 5.1, this is approximately a 30.9 times speedup, nearly an ideal speedup. Likewise, the other models are very similar.

5.4 Conclusions

This chapter has shown how performing multiple simulations concurrently can optimize the serial AESS implementation of the stochastic simulation algorithm. Using a cluster of computers with multicore processors to distribute the load of the simulations proved to be very advantageous and worth the small amount of extra effort. It must be noted that as the model becomes larger, the less shared information will fit into cache therefore causing more cache misses as each of the cores attempts to perform its computations. However, even with the larger model, the performance did not suffer much and is still competitive for an efficient approach to accelerating the stochastic simulation.

Chapter 6

Compute Unified Device Architecture Implementation

This chapter will first discuss the background of the computer unified device architecture (CUDA) on the NVIDIA graphics processing units in Section 6.1. Next, the stochastic simulation implementations (both preliminary and optimized versions) on these GPUs will be presented in Section 6.2. Finally, the results of this application will be compared to the serial AESS approach in Section 6.3.

6.1 Compute Unified Device Architecture - CUDA

Within recent years, graphics processing units (GPUs) have become extremely popular in the parallel computing community, due to their ability to be programmed for massively parallel applications with less overhead and more execution ability than previous methods. These GPUs have more transistors devoted to data processing and high memory bandwidth (as shown in figure 6.1) allowing for many instructions being executed at the same time. Because of this, single instruction, multiple data (SIMD) applications benefit the most from this parallelism.

In the case of NVIDIA GPUs, like the ones used in this project, there are a large number of multiprocessors that allow for this parallelization, as shown in figure 6.2. Each multiprocessor has its own shared memory, cache, and a set of processors which each contain registers. All of

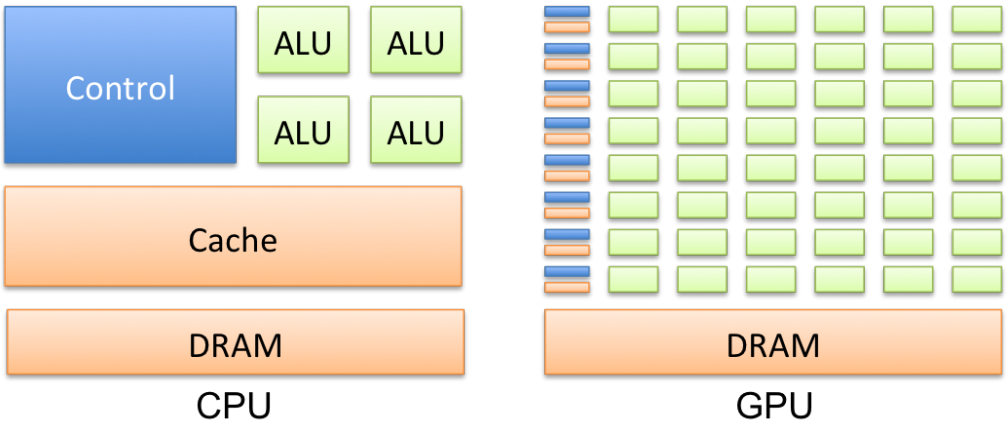


Figure 6.1: CPU architecture versus GPU architecture [21].

the multiprocessors have access to the device memory; however, this comes at a high access cost. Access to the shared memory is about 100 times faster than global device memory access. This needs to be taken into consideration when writing a program for the GPUs.

In order to code a program onto these GPUs, NVIDIA has created its own language, compute unified device architecture (CUDA) [21]. This language is built onto C in order to easily incorporate into existing programs. This language also has a couple of libraries that implement BLAS and FFT functions called, CUBLAS and CUFFT respectively. However, to produce code for the GPU that functions outside of these libraries, one needs to program custom kernels. Each kernel specifies a grid of blocks of threads shown in figure 6.3. There are a maximum of 65535 blocks and 512 threads per block allowed for each kernel. Each thread/block performs one instruction on different sets of data. This is the core of the parallelism available when using the GPU.

In order to store information onto the GPUs, memory must first be allocated using the built-in function, cudaMalloc. Then, data from an array stored on the CPU is copied to the newly allocated pointer on the GPU by using a function called cudaMemcpy. This function copies every element into the global device memory. Likewise, this is a similar process to pull data off the GPU back onto the CPU.

Due to the large delay in accessing global memory, it is for the best to limit the amount of access to it as much as possible. As can be seen from figure 6.2, there are few type of memory

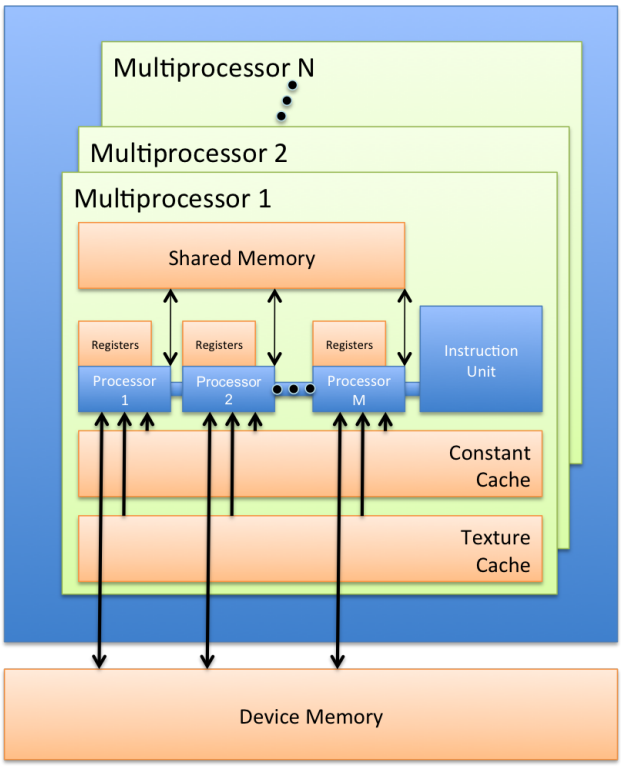


Figure 6.2: Multiprocessors in an NVIDIA GPU [21].

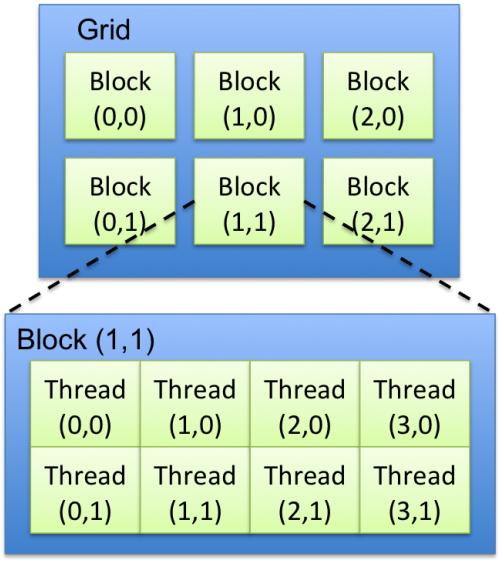


Figure 6.3: Grid, block, and thread description [21].

that allow for faster access times than global. These memories include (in order of speed) registers, shared, constant, and texture.

Each processor has a set of 32-bit registers. These registers allow for accessing memory with zero clock cycle penalty per instruction. These are obviously the fastest form of memory available, however, there is a very limited number of the registers available. Only kernels running on the GPU can load memory into a register. To do so, simply declare a variable and store whatever you need to from there. For instance,

$$intreg = globalvar[0]$$

will declare an integer variable and store the value from the global memory variable into the newly formed register.

The next type of memory is the shared memory that allows for fast memory access. In order to store items into the faster shared memory, a variable must first be declared using the following syntax,

$$\_\_shared\_\_\ DATATYPE\ sharedvar[size]$$

where DATATYPE can be any single precision C data type such as int, char, or float. There is not a set of functions like *cudaMalloc* and *cudaMemcpy* to put information directly into shared memory, bypassing the slow device memory. Therefore, to store data into the shared memory, one approach is to have each thread in the kernel store one entry of the array into global memory, as shown below.

$$sharedvar[threadIdx.x] = globalvar[threadIdx.x]$$

Doing this will cause a one time read from global memory, allowing shared memory to be accessed for the remainder of the kernel execution.

Constant memory is the next type of memory available on the GPU. The constant memory is limited to 65536 bytes and is cached for fast access. Reading from the constant memory costs one global device memory read on a cache miss, and one constant memory read on a cache hit. Therefore, on a cache miss, the data requested is read from device memory and stored into cache for future reference. This type of memory is a read only memory from inside the GPU kernels. The

constant memory must be statically declared as a global variable in the code so the compiler can map it. In order to store data into constant memory, the user must do a $cudaMemcpyToSymbol$. At which point, the kernels can read the data in constant memory.

One downside of using the GPU is that there is no support for double precision on most of NVIDIA's GPUs. Therefore, any double precision numbers used on the CPU must be converted to single precision before passing it to the GPU. However, on the higher end GPUs from NVIDIA, double precision is supported, but the performance is dramatically decreased. Another downside is the small amount of shared memory space. On the GPU used in this project, the shared memory space is 16KB per block. This is not much space, so it must be used wisely to allow for the highest efficiency. Similarly, the lack of constant memory to provide speedy access to data is a major limitation.

6.2 Implementations

Implementing AESS on the GPU turns out to be quite difficult. This is because there is not a great way to parallelize just one simulation at a time that results in a speed up of individual runs due to the dependency of one reaction on the previous one as well as the relatively small amount of work per reaction. In fact, with the overhead of memory copying back and forth, there is actually an increase in run time. However, as mentioned before, multiple simulation runs are needed when trying to use the algorithm to get a statistically sound model of the chemical system. Therefore, SSA is implemented on the GPU to run multiple simulations at a time to reduce the amount of memory copies between CPU and GPU. This also spreads the runtime across multiple simulations to achieve an overall speed up of the entire process. In the following sections, details of two implementations developed for CUDA are given with. This work is based on the previous work of Li and Petzold

6.2.1 Preliminary Implementation

On a first run through without thoughts of overall optimizations, a logical choice for implementing multiple simulations on the GPU would be to have one block per simulation. In other words, one

block would do all the calculations needed for one simulation. This was an obvious choice do to its embarrassing parallel nature since each simulation is independent of the others.

Figure 6.4 shows a diagram of one reaction step and how the CPU and GPU share the computations to speedup up the program.

Following the pseudo code given in section 2.3.3, Dan Gillespie's stochastic simulation algorithm using the Direct Method was implemented on the GPU. The Direct Method is chosen for this implementation due to the reasons give in Section 2.3.3. To do this, the input file containing the initial species populations, reactions, and reactant and product coefficients are read and stored into arrays using basic C++ functions. These arrays are then copied to the GPU using $cudaMemcpy$ and the procedure described earlier. Next, there are six kernels that are used to accomplish some of the calculations.

First, random number generation is offloaded to the GPU in order to generate the large number of random numbers for all the simulations. This is done for a couple of reasons. First, for one reaction on one simulation, there are two random numbers that are needed. Therefore, for one reaction of S simulations, there is a total of 2*S numbers needed, making the generation on the CPU expensive for large numbers of simulations. Second, if the CPU were to do the generation, the numbers would have to be copied to the GPU for every reaction for calculations, which would increase the total amount of time as previous mentioned. Offloading this task improves the speed of the program by generating large sets of numbers at a time on the GPU, with no need for copying them back and forth between the CPU and GPU. Since random number generation is extremely difficult to implement, this work used the Mersenne twister algorithm that is implemented in the NVIDIA SDK code collection. For implementation details on this design, refer to the SDK documentation provided with the code.

The second kernel calculates the propensities by distributing the calculations for the propensity of each reaction across all the threads. In other words, each block calculates the set of propensities for each simulation. Within a block, each thread calculates the propensity for each reaction. This can be shown in figure 6.5.

Next, the third kernel performs a reduction on the array of propensities for each simulation.

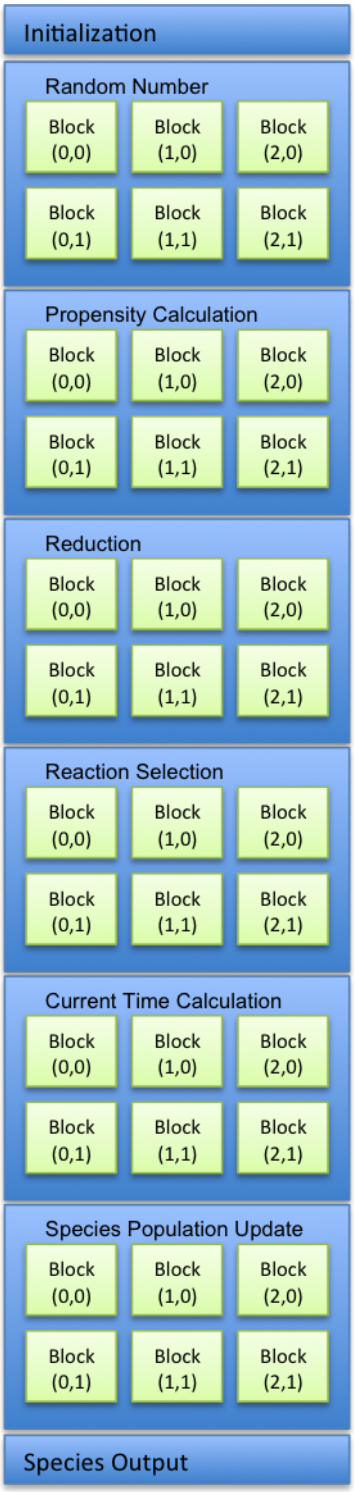


Figure 6.4: Layout of the CPU and GPU interaction of one reaction step [21].

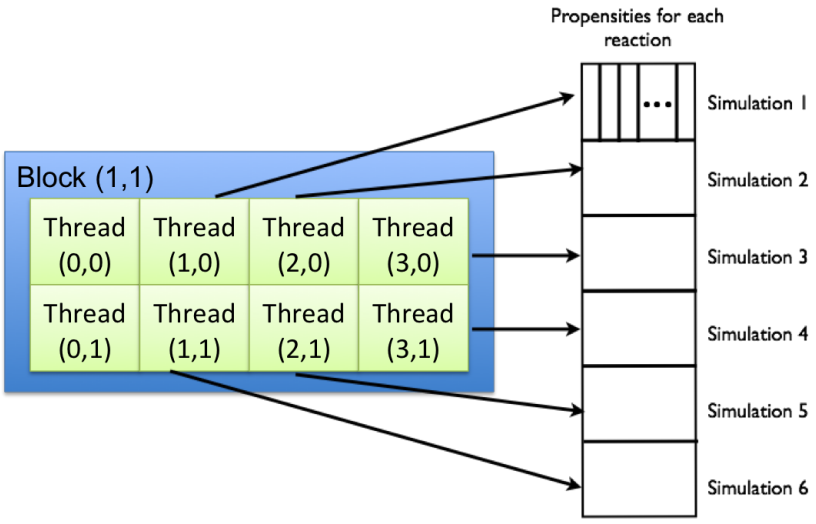


Figure 6.5: Propensity calculation distribution across the grid.

This reduction finds the sum of all the propensities for each simulation and reduces the array down to one number per simulation indicating the sum.

The fourth kernel determines the index of the next reaction. This kernel takes the sums of the previous kernel and multiplies them by a uniformly distributed random number. Each of these scaled random numbers is then subtracted by each of the propensities for its corresponding reaction. The index of the propensity that causes this subtraction to be less than or equal to 0 is stored as the index of the next reaction that will occur.

Fifth, a kernel is also created to calculate the time of occurrence of the next reaction. This is accomplished by simply having each block divide an exponentially distributed random number by the sum of the propensities for each simulation. This kernel is only executed when the output flag is set indicating the species population is to be outputted at the end of the program.

Finally, the sixth kernel updates the species populations in a similar manner as updating the propensities. However, this time each thread corresponds to one species and updates its population. This is similar to the diagram in figure 6.5.

After these kernels have executed, control is returned back to the CPU so that the species counts can be outputted to a file to check their validity. This option can be turned on and off, as outputting this number can be very costly to the runtime of the program.

6.2.2 Optimized Implementation

As will be seen and explained in the results, the previous implementation was not as optimized as it could have been. Therefore, a new approach had to be taken that better utilized the hardware characteristics of the GPU.

In this new optimized version, the approach was similar to the previous with six kernels; however, each kernel is set up differently to make optimized use of warps, shared memory, and constant memory.

First, in the previous implementation, using one block per simulation was not the optimal setup due to the fact that it was not guaranteed to occupy an entire warp of 32 threads. Because of this, the performance was hurt tremendously. Therefore, a more optimal setup was to do one simulation per thread, despite the lack of parallelism within individual simulations. Now, each kernel can be initiated with enough threads to fill up a warp or multiple warps. This now stipulates the number of simulations to be run to be a multiple of 32 in order to provide higher performance. For the propensity calculation and species update kernels, this means that each thread is going to run through more loops; however, all of the loops within each thread are doing the exact same calculations taking advantage of the SIMD architecture.

Second, the previous implementation did not take use of the shared memory. Because of this, each time any information was read or stored from memory, it was being taken from global memory adding 400 to 600 clock cycles per read or write to the execution time of the program. Therefore, shared memory needs to be used to reduce the memory access penalty to four clock cycles per read or write [21]. To do this, each kernel has its own scheme for storing the information into shared memory, but each uses it as a buffer to the global memory. Any time calculations are done in global memory, they will be done in shared memory first then later stored back to global memory as described in section 6.1.

Finally, constant memory was not being used in the previous implementation. Because of this, all of the data was being read from global memory, resulting in a major penalty in performance. In this new implementation, all of the reactant indices and coefficients, reaction rate constants, and species updates are now stored into the global memory. This helps the performance a tremendous

amount because of the information being stored into cache for future use. As mentioned before, when there is a cache hit, all of the threads in a half warp read the cached data as fast as reading from a register [21]. Obviously, this leads to a potential significant speedup as the kernel has been constructed for each thread to be reading from the same address as much as possible.

6.3 Results

Both of these CUDA implementations were run on three different NVIDIA GPUs to examine the performance differences. These cards include:

1. GeForce 8600M
2. Tesla c870
3. Tesla c1060

All of the specifications of these cards are given in Appendix B.

The runtimes of these simulations are timed using the built in C function, $gettimeofday$. CUDA has a timer function built in as well, however, it unfortunately gave incorrect times. For example, sometimes the CUDA timers would return indicating the program ran for milliseconds when it was apparent the program ran for seconds. After talking with other CUDA programmers, this seems to be a reccurring problem without any apparent cause or fix. Also, as this C function is used to measure all the other previous implementations, it is used so that every runtime has the same metric.

Below, table 6.1 shows all of the runtimes of the CUDA implementations while varying the number of simulations by powers of two for the models given in Appendix A.

Below in figures 6.6(a) through 6.6(c), these runtimes are visualized in the form of graphs.

The first thing to notice from these graphs is the optimized implementation is a significant improvement over the original, preliminary implementation. All of the optimizations techniques work successfully. Out of all the optimizations, ensuring that a warp did not contain any stagnant threads or divergent threads was the most effective. This is due to the reasons listed in Section 6.2.2.

Table 6.1: CUDA runtimes for 10000 reactions.

(a) CUDA runtimes for 10000 reactions of the SCHLOGL model

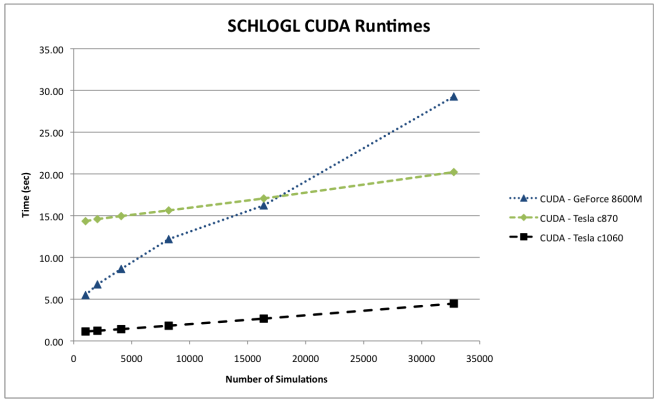
SCHLOGL for 10000 reactions						
Simulations	Preliminary			Optimized		
	Tesla c870	Tesla c1060	GeForce 8600M	Tesla c870	Tesla c1060	GeForce 8600M
1024	13.73	1.84	7.56	14.34	1.13	5.49
2048	15	2.52	10.41	14.59	1.22	6.77
4096	17.71	3.87	14.81	14.95	1.41	8.62
8192	23.02	6.67	24.25	15.63	1.82	12.19
16384	33.44	12.16	44.09	17.06	2.67	16.23
32768	54.94	23.19	82.19	20.22	4.47	29.27

(b) CUDA runtimes for 10000 reactions of the DIMER model

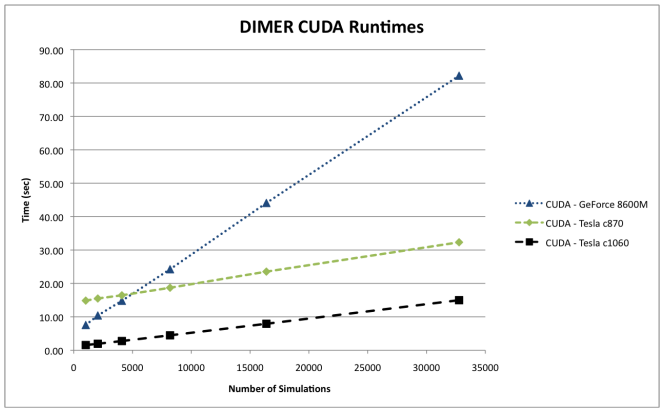
Dimerization for 10000 reactions						
Simulations	Preliminary			Optimized		
	Tesla c870	Tesla c1060	GeForce 8600M	Tesla c870	Tesla c1060	GeForce 8600M
1024	15.13	2.88	15.26	14.86	1.56	7.56
2048	17.94	4.56	25.81	15.49	1.94	10.41
4096	23.66	8.01	45.77	16.45	2.76	14.81
8192	35.67	14.88	86.6	18.71	4.47	24.25
16384	58.09	28.55	169.24	23.54	7.93	44.09
32768	103.89	56.04	333.49	32.32	14.99	82.19

(c) CUDA runtimes for 10000 reactions of the HSR model

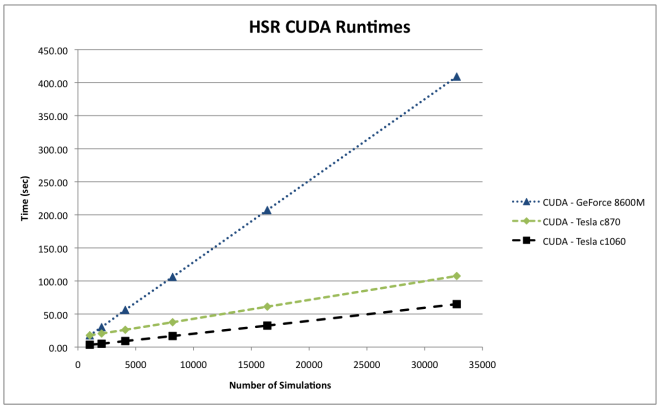
HSR for 10000 reactions						
Simulations	Preliminary			Optimized		
	Tesla c870	Tesla c1060	GeForce 8600M	Tesla c870	Tesla c1060	GeForce 8600M
1024	18.66	5.08	36.54	17.76	3.55	17.79
2048	25.22	8.77	66.34	20.49	5.01	30.11
4096	35.83	16.59	124.97	26.13	9.13	56.25
8192	57.47	31.75	241.39	37.57	16.71	106.12
16384	107.63	62.23	473.99	61.12	32.48	207.22
32768	209.57	122.99	943.55	107.51	64.93	409.09



(a) SCHLOGL results



(b) DIMER results



(c) HSR results

Figure 6.6: CUDA runtimes results

Next, by using the using different GPUs and the same code with no modifications, we were able to achieve speedups based purely on the quality and specifications of the cards. This may be an easy way to achieve a speedup out of the CUDA code, however, this could be expensive for the user and may not be a feasible alternative for all.

On the Tesla c870 GPUs, there was a relatively large runtime for the smaller number of simulations due to some problems. First off, there was an extremely high initialization time for the graphics cards to start up, on average of about 6-7 seconds. Compared to the other two cards, this is definitely some sort of hardware issue. Whether it is the installation of the cards or defective cards, the reason is unclear. Second, these cards were not completely reliable. For instance, sometimes running a code would be successful while other times the programs would quit giving an "unspecified kernel launch" error. This not only happened with my code, but also the code supplied in the NVIDIA SDK. Once again, the cause for this is unknown.

Below in table 6.2 and figure 6.7 are the speedups over the optimized serial AESS implementation.

From these figures, we can see that both of the implementations were successful in generating a speedup over the serial code for larger numbers of simulations. However, it is apparent that the optimized implementation is approximately 10 times faster than the preliminary implementation.

Also, these figures show something interesting: the preliminary implementation offers a relatively constant speedup while the optimized gives an increasing speedup. The preliminary implementation runtimes double as the number of simulations doubles, just as the serial code. However, the optimized does not follow this same trend. Instead, as the number of simulations increase, the speedup increases as well. This can be contributed to keeping more of the GPU resources busy while keeping less idle, including the streaming multiprocessors.

Lastly, it can be seen that as the model becomes larger and more convoluted (as with the HSR model), the performance decreases. This is due to the larger number of calculations need per simulation.

In table 6.3, a list of the number of reactions per second can be seen in order to get a different visualization of the performance over the serial AESS implementation. Also, figure 6.8 gives a

Table 6.2: CUDA speedups for 10000 reactions

(a) CUDA speedups for 10000 reactions of the SCHLOGL model

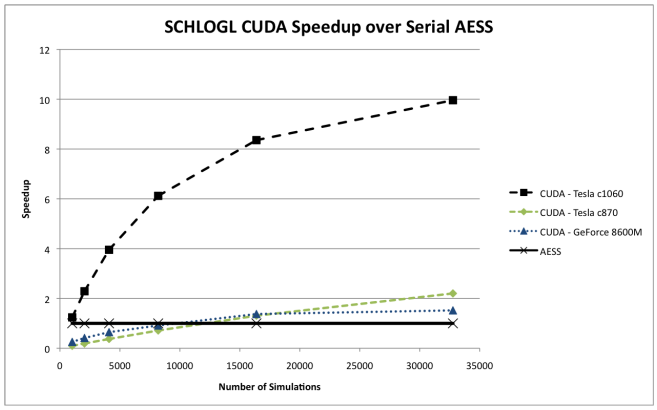
SCHLOGL for 10000 reactions						
Simulations	Preliminary			Optimized		
	Tesla c870	Tesla c1060	GeForce 8600M	Tesla c870	Tesla c1060	GeForce 8600M
1024	0.10	0.76	0.19	0.10	1.24	0.26
2048	0.19	1.11	0.27	0.19	2.29	0.41
4096	0.31	1.44	0.38	0.37	3.95	0.65
8192	0.48	1.67	0.46	0.71	6.12	0.91
16384	0.67	1.83	0.51	1.31	8.36	1.37
32768	0.81	1.92	0.54	2.20	9.96	1.52

(b) CUDA speedups for 10000 reactions of the DIMER model

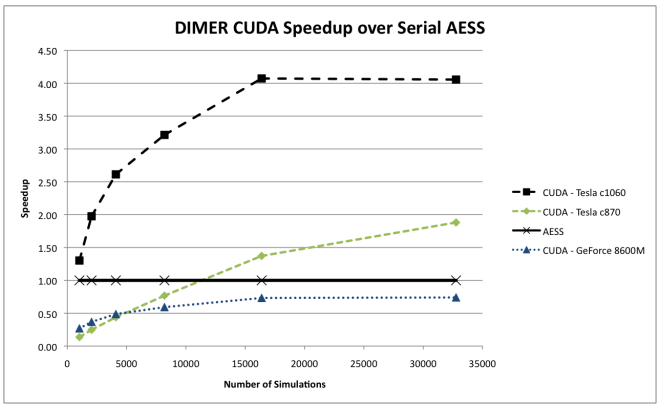
Dimerization for 10000 reactions						
Simulations	Preliminary			Optimized		
	Tesla c870	Tesla c1060	GeForce 8600M	Tesla c870	Tesla c1060	GeForce 8600M
1024	0.13	0.71	0.13	0.14	1.30	0.27
2048	0.21	0.84	0.15	0.25	1.98	0.37
4096	0.30	0.90	0.16	0.44	2.61	0.49
8192	0.40	0.97	0.17	0.77	3.21	0.59
16384	0.56	1.13	0.19	1.37	4.07	0.73
32768	0.59	1.08	0.18	1.88	4.06	0.74

(c) CUDA speedups for 10000 reactions of the HSR model

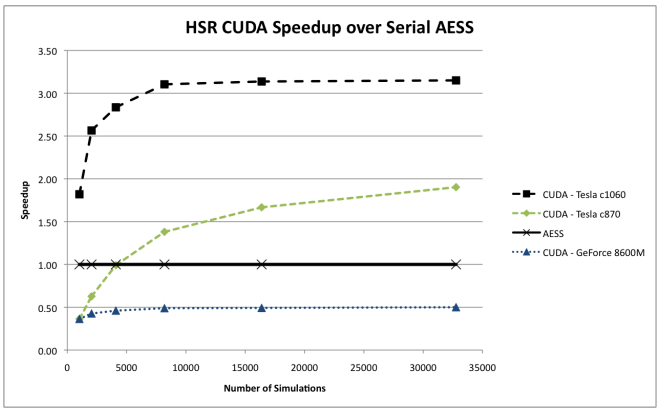
HSR for 10000 reactions						
Simulations	Preliminary			Optimized		
	Tesla c870	Tesla c1060	GeForce 8600M	Tesla c870	Tesla c1060	GeForce 8600M
1024	0.35	1.27	0.18	0.36	1.82	0.36
2048	0.51	1.47	0.19	0.63	2.56	0.43
4096	0.72	1.56	0.21	0.99	2.84	0.46
8192	0.90	1.63	0.21	1.38	3.10	0.49
16384	0.95	1.64	0.21	1.67	3.14	0.49
32768	0.98	1.66	0.22	1.90	3.15	0.50



(a) SCHLOGL results



(b) DIMER results



(c) HSR results

Figure 6.7: CUDA speedup results

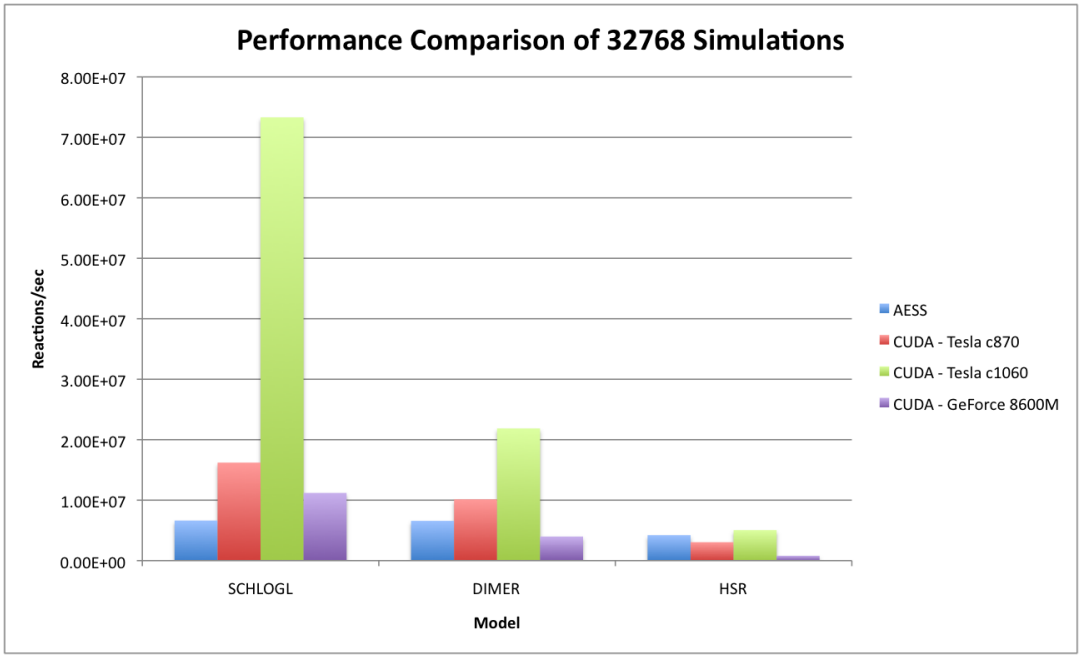


Figure 6.8: CUDA number of reactions per second each implementation can execute.

visual representation of 32768 simulations with the information in the table.

6.4 Conclusion

Overall, using the NVIDIA graphics processing units proved to be efficient enough to be considered as a viable option to accelerating the stochastic simulation algorithm. However, this is only the case when extra steps were taken in order to make a more optimized version. Once the CUDA syntax and architecture has been studied, implementing the algorithm onto the GPUs was easy enough to warrant its use.

Saying this, however, comparing the results of these implementations with other CUDA programs, these do not reach the potential of high speedups possible with the GPUs. Because of this, this algorithm is not well suited for GPUs and similar results could be achieved with other technologies such as SSE or clusters of computers. This can be contributed to the amount of memory accesses needed for each reaction and the huge penalty for doing so.

Table 6.3: CUDA number of Reactions per Second with 32768 simulations, 10000 reactions each

(a) SCHLOGL - Number of Reactions per Second with 32768 simulations, 10000 reactions each

Reactions Per Second SCHLOGL for 10000 reactions						
Simulations	Tesla c870	Tesla c1060	8600M	Tesla c870	Tesla c1060	8600M
1024	7.46E5	5.57E6	1.35E6	7.14E5	9.06E6	1.87E6
2048	1.37E6	8.13E6	1.97E6	1.40E6	1.68E7	3.03E6
4096	2.31E6	1.06E7	2.77E6	2.74E6	2.90E7	4.75E6
8192	3.56E6	1.23E7	3.38E6	5.24E6	4.50E7	6.72E6
16384	4.90E6	1.35E7	3.72E6	9.60E6	6.14E7	1.01E7
32768	5.96E6	1.41E7	3.99E6	1.62E7	7.33E7	1.12E7

(b) DIMER - Number of Reactions per Second with 32768 simulations, 10000 reactions each

Reactions Per Second DIMER for 10000 reactions						
Simulations	Tesla c870	Tesla c1060	8600M	Tesla c870	Tesla c1060	8600M
1024	6.77E5	3.56E6	6.71E5	6.89E5	6.56E6	1.35E6
2048	1.14E6	4.49E6	7.93E5	1.32E6	1.06E7	1.97E6
4096	1.73E6	5.11E6	8.95E5	2.49E6	1.48E7	2.77E6
8192	2.30E6	5.51E6	9.46E5	4.38E6	1.83E7	3.38E6
16384	2.82E6	5.74E6	9.68E5	6.96E6	2.07E7	3.72E6
32768	3.15E6	5.85E6	9.83E5	1.01E7	2.19E7	3.99E6

(c) HSR - Number of Reactions per Second with 32768 simulations, 10000 reactions each

Reactions Per Second HSR for 10000 reactions						
Simulations	Tesla c870	Tesla c1060	8600M	Tesla c870	Tesla c1060	8600M
1024	5.49E5	2.02E6	2.80E5	5.77E5	2.88E6	5.76E5
2048	8.12E5	2.34E6	3.09E5	1.00E6	4.09E6	6.80E5
4096	1.14E6	2.47E6	3.28E5	1.57E6	4.49E6	7.28E5
8192	1.43E6	2.58E6	3.39E5	2.18E6	4.90E6	7.72E5
16384	1.52E6	2.63E6	3.46E5	2.68E6	5.04E6	7.91E5
32768	1.56E6	2.66E6	3.47E5	3.05E6	5.05E6	8.01E5

Chapter 7

Summary and Conclusion

7.1 Overall Comparisons

As an overview, each of the CPU implementations was executed on a 2.26GHz Intel Xeon X5520 with 8192 KB cache and compiled with GCC version 4.1.2 with the highest optimization flags, -O3, with a couple of exceptions. The SSE implementation was also executed on a 1.8GHz Dual Core AMD Opteron Processor 265 with 1024 KB cache, GCC version 4.3.3. The comparisons of the ICC version 11.1 and GCC version 4.3.3 compilers were executed on a 2.67GHz Intel Core i7 with 8192 KB cache. The MPI implementation was executed on a computer cluster of four nodes, each with two 2.26GHz quad core Intel Xeon X5520 processors. The CUDA implementations executed on the following three graphics processing units: GeForce 8600M, Tesla c870, and Tesla c1060 (appendix B).

Below in figures 7.1 through 7.3 are a set of graphs that plot the speedups of all the implementations over the ESS code will the three main models used: SCHLOGL, DIMER, and HSR. Figure 7.4 gives graphical view of the reactions per second each implementation reached to get a comparison of the performance.

First, it is important to optimize the base serial case in order to see what kind of speedup is possible without doing any parallelization. By optimizing the serial code with C, AESS was boosted by a speedup of approximately 4.7 times with the realistic HSR model. Doing this, however, makes

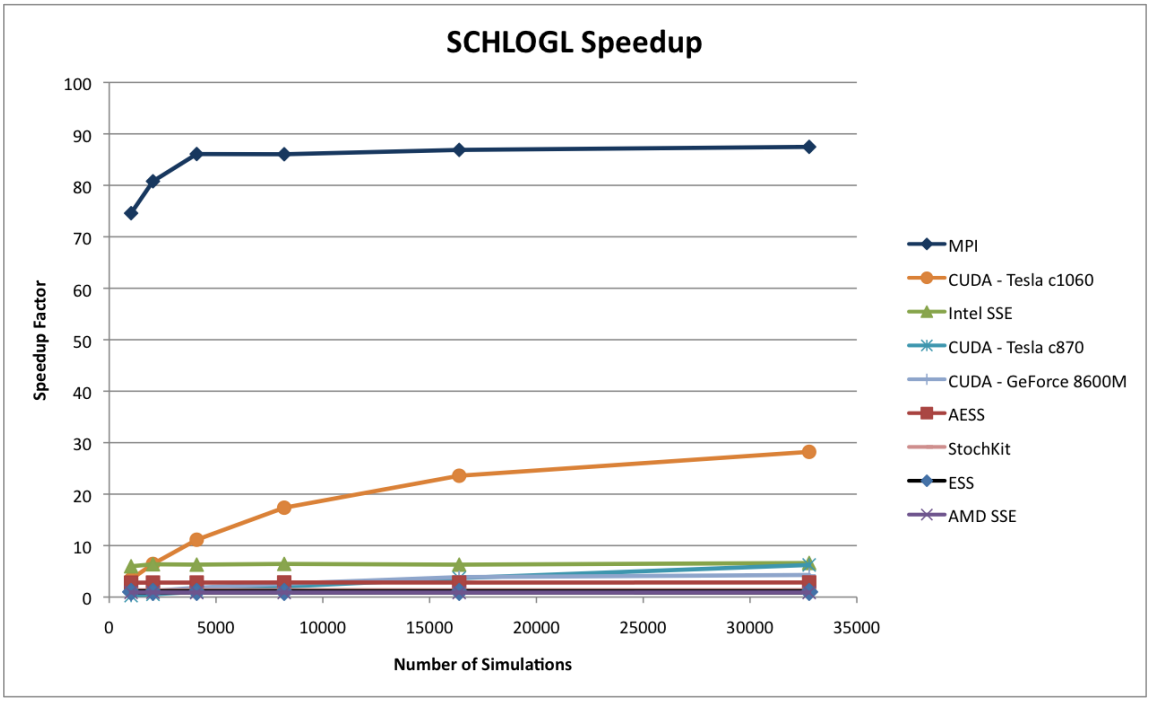


Figure 7.1: Speedup of all implementations with the SCHLOGL model with ESS being the baseline.

it more difficult for the parallelization of the algorithm to reach high speedups. But, it is only fair to optimize the base case if the parallel cases are being optimized as well in order to give a fair comparison.

Out of all of the implementations, the MPI implementation by far provided the greatest speedup for all three models. This implementation is definitely one that a scientist would consider using due to its significant speedup over all of the other implementations. With all of the models, the speedup is corresponds relatively linearly to the number of processes/cores used. Since the code was run on a cluster of 4 nodes with 8 cores each, the maximal number of processes is 32. Therefore the speedup was around 32 times faster than the serial AESS code and up to approximately 150 times faster than the ESS implementation.

If a cluster of computers is not feasible, the CUDA implementation running on the Tesla c1060 is the next best choice. With smaller models, this implementation can reach up to 28 times speedup over the ESS implementation and almost 10 times faster than the AESS code. Even with more realistics models such as the HSR set, a 15 times speedup is possible over the ESS and

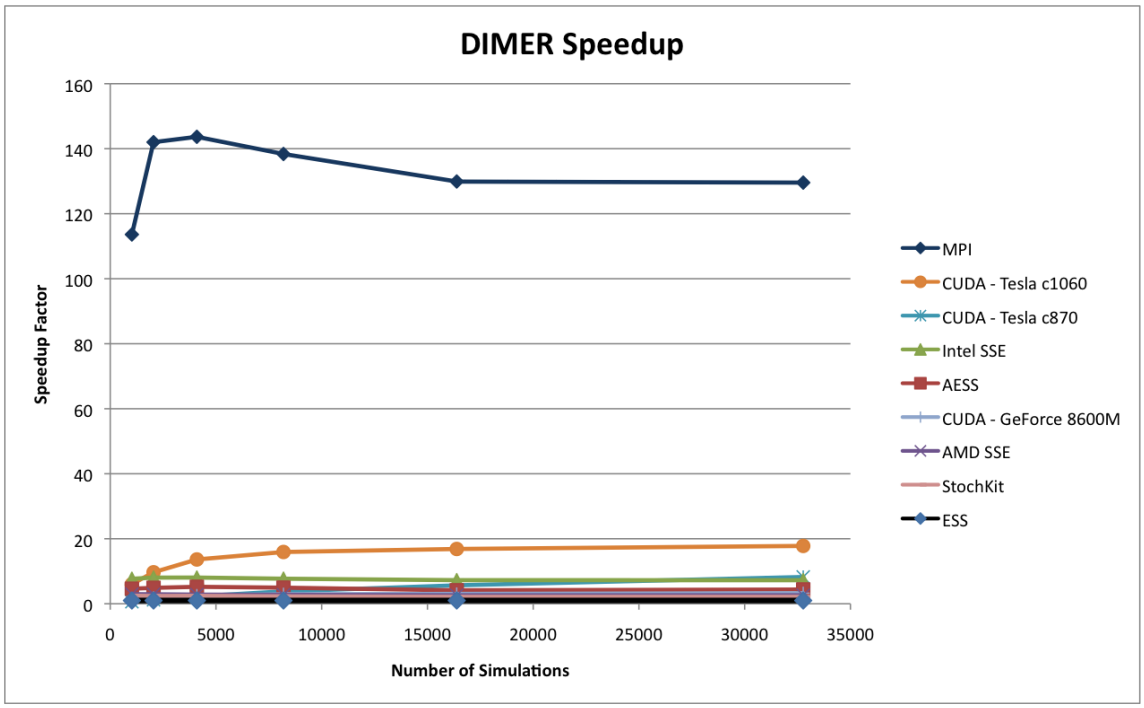


Figure 7.2: Speedup of all implementations with the DIMER model with ESS being the baseline.

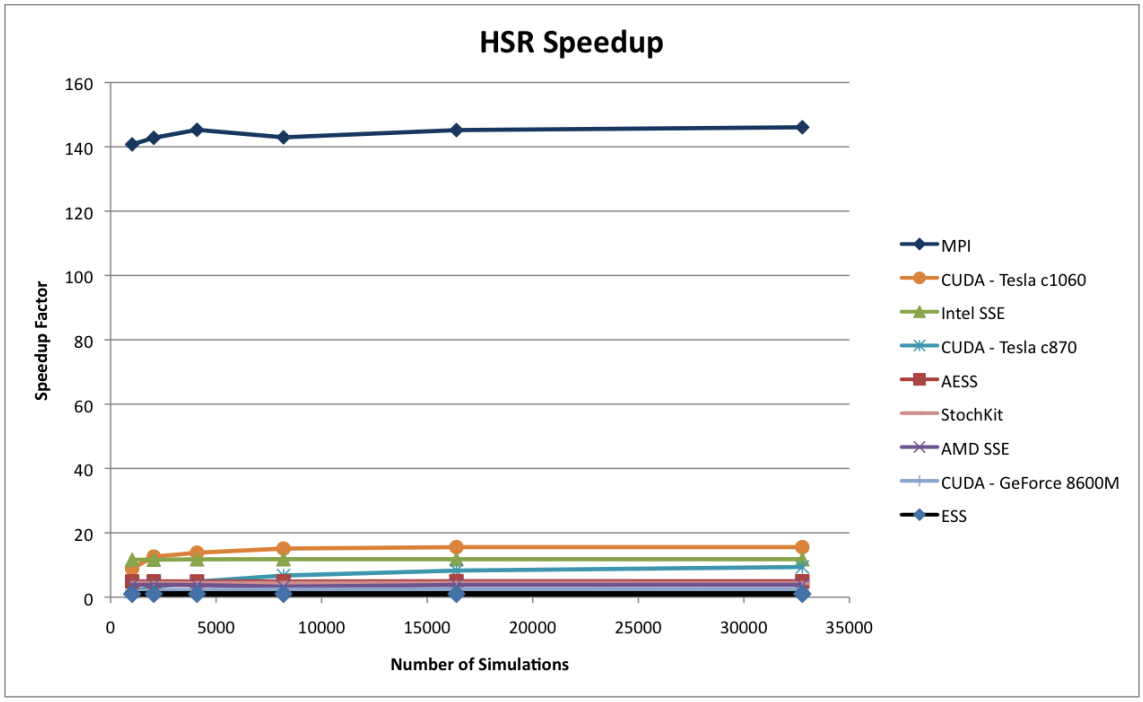


Figure 7.3: Speedup of all implementations with the HSR model with ESS being the baseline.

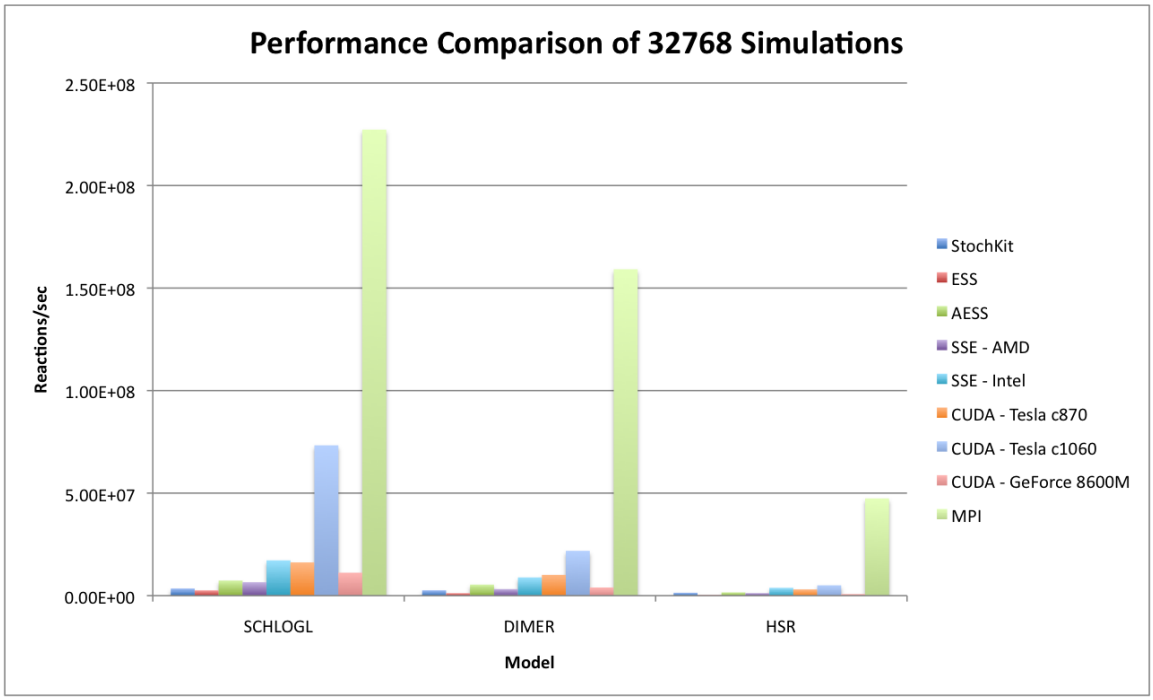


Figure 7.4: Reactions per second of all implementations with all models.

3.14 times speedup over the AESS code. Although the CUDA implementation proves to have high performance, it comes at the price of a complex implementation. Also, to get a decent performance, a high end card is needed, thereby increasing the total price for the computing system.

However, if a graphics processor were not available to the scientist, it would be advantageous to implement the algorithm using SSE on an Intel processor. This implementation gives an approximate 11 times speedup over the unoptimized ESS code and a 2.3 times speedup of the optimized AESS. Also, if available, using Intel's icc compiler should be used in order to achieve as much optimization out of the SSE as possible.

7.2 Future Work

Despite the progress this work has accomplished in accelerating the stochastic simulation, there is still room for future work. First off, different methods could be implemented in all the ways presented in the paper to examine which is the most optimal for the targeted architecture. For instance, one of the other stochastic simulation algorithm methods may provide a more optimal

and better approach on the GPUs than the Direct Method that was used here.

Also, hardware acceleration could potentially provide a substantial speedup to this algorithm as well. Implementing this in a hardware description language and synthesizing it down on a field programmable gate array (FPGA) would be the best way to approach this. There have been implementations in the past to do this ([14], [24], [18], [27]), however, each of these approaches required resynthesis in order to use a different model. The designs were not generic enough to handle multiple models while still providing adequate speedups. This leaves the door open for a new approach to hardware acceleration that is generic to execute the algorithm on many models, not just one.

7.3 Conclusion

This work has provided a number of approaches to accelerating Gillepie's stochastic simulation algorithm. Each approach provides a speedup from the original serial implementation and could be useful for any scientist or engineer in need of this algorithm. Even though the CUDA implementation performed the best, the other approaches may work better in other situations. For instance, if a scientist does not have access to an NVIDIA GPU, the CUDA implementation would be unusable. Instead, the SSE version would be the next best approach to use. Whichever implementation the scientist chooses to use, it can be expected to achieve a speedup in order to receive results in a more reasonable amount of time.

Bibliography

[1] 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.

[2] Y. Cao, H. Li, L.R. Petzold, "Efficient formulation of the stochastic simulation algorithm for chemically reacting systems", *J. Chem. Phys.*, vol. 121, no. 9, pp. 4059-4067, September 2004.

[3] D. Endy, 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.

[4] M. Gibson and J. Bruck, "Efficient exact stochastic simulation of chemical systems with many species and many channels", *J. Phys. Chem. A*, vol. 104 no. 9, pp. 1876-1889, 2000.

[5] M. Hucka, S. Hoops, S. Keating, N. Le Novre, S. Sahle, and D. Wilkinson, "Systems Biology Markup Language (SBML) Level 2: Structures and Facilities for Model Definitions", 2008. *Available from Nature Precedings*, http://dx.doi.org/10.1038/npre.2008.2715.1.

[6] D. T. Gillespie, "A General Method for Numerically Simulating the Stochastic Time Evolution of Coupled Chemical Reactions", *J. Comp. Phys.*, vol. 22, no. 4, pp. 403-434, December 1976.

[7] D. T. Gillespie, "Exact Stochastic Simulation of Coupled Chemical Reactions", *J. Phys. Chem.*, vol. 81, no. 25, pp. 2340-2361, 1977.

[8] D. T. Gillespie, "A rigorous derivation of the chemical master equation", *Physica A: Statistical and Theoretical Physics*, vol. 188, no. 1-3, pp. 404-425, September 1992.

[9] D. T. Gillespie, "Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems", *J. Chem. Phys.*, vol. 115, pp. 1716-1733, 2001.

[10] M. Langlais and G. Latu and J. Roman and P. Silan, "Performance analysis and qualitative results of an efficient parallel stochastic simulator for a marine host-parasite system",

Concurrency and Computation-Practice & Experience, vol. 15, no. 11-12, pp. 1133–1150, September 2003.

[11] H. Li, Y. Cao, L. R. Petzold, and D. T. Gillespie, "Algorithms and software for stochastic simulation of biochemical reacting systems", Biotechnol. Prog., September 2007. Online. http://dx.doi.org/10.1021/bp070255h.

[12] H. Li and L. Petzold, "Efficient Parallelization of Stochastic Simulation Algorithm for Chemically Reacting Systems on the Graphics Processing Unit", Technical report, Dept. Computer Science, University of California, Santa Barbara, 2007.

[13] H. Li and L. Petzold, "Logarithmic Direct Method for Discrete Stochastic Simulation of Chemically Reacting Systems", Technical report, Dept. Computer Science, University of California, Santa Barbara, 2006.

[14] L. Lok, "The need for speed in stochastic simulation", *Nature Biotechnology*, vol. 22, No. 8, pp. 964-965, 2004.

[15] 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. 179-184, 1998.

[16] Microsoft, "MMX, SSE, and SSE2 Intrinsics", 2009. Microsoft. 16 Oct 2009. http://msdn.microsoft.com/en-us/library/y0dh78ez%28VS.80%29.aspx.

[17] J. M. McCollum, "Accelerating Exact Stochastic Simulation", MS thesis, University of Tennessee. 2004.

[18] J. M. McCollum, "Accelerating Exact Stochastic Simulation of Biochemical Systems", Ph.D. thesis, University of Tennessee. 2006.

[19] J. M. McCollum, et. al., "The sorting direct method for stochastic simulation of biochemical systems with varying reaction execution behavior", *Computational Biology and Chemistry*, vol. 30, No. 1, pp. 39-49, February 2006.

[20] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation", *ACM Conference on Programming Language Design and Implementation*, pp. 89-100, 2007.

[21] NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide", June 2008 [Online], http://developer.download.nvidia.com.

[22] J. Plank, "CS140 Lecture notes – Fields", 2007. University of Tennessee. 13 July 2009. http://www.cs.utk.edu/~plank/plank/classes/cs140/Notes/Fields/.

[23] D. C. Rapaport, The Art of Molecular Dynamics Simulation, Cambridge: Cambridge University Press, 2004.

[24] L. Salwinski, D. Eisenberg, "In silico simulation of biochemical network dynamics", *Nature Biotechnology*,

[25] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra, "MPI: The Complete Reference", MIT Press, Cambridge, MA, 1996.

[26] "StochKit: Stochastic Simulation Kit", *StochKit*, University of California Santa Barbara, Accessed: 1 April 2009. http://engineering.ucsb.edu/~cse/StochKit/index.html.

[27] M. Yoshimi, Y. Osana, T. Fukushima, H. Amano, "Stochastic simulation for biochemical reactions on FPGA", *Lecture Notes in Computer Science*, vol. 3203, pp. 105-114, 2004.

Appendix

Appendix A

Models

Each model is in the form described below.

1. Integer indicating the number of species.
2. Integers denoting the populations for each species.
3. Integer indicating the number of reactions.
4. Each reaction
   (a) Integer indicating the number of reactants
   (b) Each reactant listed with its coefficient (integer) followed by its index (integer)
   (c) Integer indicating the number of products
   (d) Each product listed with its coefficient (integer) followed by its index (integer)
   (e) A real number that is the rate constant for the reaction
5. Integer indicating the number of species to output.
6. Integers that represent the species index to output.

A.1 20Gene

This model contains 100 species and 100 reactions [17].

```
100
1 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
1 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
100
1 1 0 2 1 0 1 1 10.0
1 1 1 2 1 1 1 2 10.0
1 1 1 0 1.0
1 1 2 0 1.0
1 1 3 2 1 3 1 4 5.0
1 1 4 2 1 4 1 5 5.0
1 1 4 0 1.0
1 1 5 0 1.0
2 1 3 1 2 1 1 6 0.0001
1 1 6 2 1 6 1 4 100.0
1 1 10 2 1 10 1 11 10.0
1 1 11 2 1 11 1 12 10.0
1 1 11 0 1.0
1 1 12 0 1.0
1 1 13 2 1 13 1 14 5.0
1 1 14 2 1 14 1 15 5.0
1 1 14 0 1.0
1 1 15 0 1.0
2 1 13 1 11 1 1 16 0.0001
1 1 16 2 1 16 1 14 100.0
1 1 20 2 1 20 1 21 10.0
1 1 21 2 1 21 1 22 10.0
1 1 21 0 1.0
1 1 22 0 1.0
1 1 23 2 1 23 1 24 5.0
1 1 24 2 1 24 1 25 5.0
1 1 24 0 1.0
1 1 25 0 1.0
2 1 23 1 21 1 1 26 0.0001
1 1 26 2 1 26 1 24 100.0
1 1 30 2 1 30 1 31 10.0
1 1 31 2 1 31 1 32 10.0
1 1 31 0 1.0
1 1 32 0 1.0
1 1 33 2 1 33 1 34 5.0
1 1 34 2 1 34 1 35 5.0
1 1 34 0 1.0
1 1 35 0 1.0
2 1 33 1 31 1 1 36 0.0001
1 1 36 2 1 36 1 34 100.0
1 1 40 2 1 40 1 41 10.0
1 1 41 2 1 41 1 42 10.0
1 1 41 0 1.0
1 1 42 0 1.0
1 1 43 2 1 43 1 44 5.0
1 1 44 2 1 44 1 45 5.0
1 1 44 0 1.0
1 1 45 0 1.0
2 1 43 1 41 1 1 46 0.0001
```

```
1 1 46 2 1 46 1 44 100.0
1 1 50 2 1 50 1 51 10.0
1 1 51 2 1 51 1 52 10.0
1 1 51 0 1.0
1 1 52 0 1.0
1 1 53 2 1 53 1 54 5.0
1 1 54 2 1 54 1 55 5.0
1 1 54 0 1.0
1 1 55 0 1.0
2 1 53 1 51 1 1 56 0.0001
1 1 56 2 1 56 1 54 100.0
1 1 60 2 1 60 1 61 10.0
1 1 61 2 1 61 1 62 10.0
1 1 61 0 1.0
1 1 62 0 1.0
1 1 63 2 1 63 1 64 5.0
1 1 64 2 1 64 1 65 5.0
1 1 64 0 1.0
1 1 65 0 1.0
2 1 63 1 61 1 1 66 0.0001
1 1 66 2 1 66 1 64 100.0
1 1 70 2 1 70 1 71 10.0
1 1 71 2 1 71 1 72 10.0
1 1 71 0 1.0
1 1 72 0 1.0
1 1 73 2 1 73 1 74 5.0
1 1 74 2 1 74 1 75 5.0
1 1 74 0 1.0
1 1 75 0 1.0
2 1 73 1 71 1 1 76 0.0001
1 1 76 2 1 76 1 74 100.0
1 1 80 2 1 80 1 81 10.0
1 1 81 2 1 81 1 82 10.0
1 1 81 0 1.0
1 1 82 0 1.0
1 1 83 2 1 83 1 84 5.0
1 1 84 2 1 84 1 85 5.0
1 1 84 0 1.0
1 1 85 0 1.0
2 1 83 1 81 1 1 86 0.0001
1 1 86 2 1 86 1 84 100.0
1 1 90 2 1 90 1 91 10.0
1 1 91 2 1 91 1 92 10.0
1 1 91 0 1.0
1 1 92 0 1.0
1 1 93 2 1 93 1 94 5.0
1 1 94 2 1 94 1 95 5.0
1 1 94 0 1.0
1 1 95 0 1.0
2 1 93 1 91 1 1 96 0.0001
1 1 96 2 1 96 1 94 100.0
10
6 16 26 36 46 56 66 76 86 96
```

A.2 AutoReg

This model contains 5 species and 8 reactions [17].

```
5
1 0 0 0 0
8
1 1 0 2 1 0 1 1 10.0
1 1 1 0 1.0
1 1 1 2 1 1 1 2 10.0
1 1 2 0 1.0
1 2 2 1 1 3 1.0
1 1 3 1 2 2 10.0
2 1 3 1 0 1 1 4 0.1
1 1 4 2 1 3 1 0 10.0
5
0 1 2 3 4
```

A.3 Diffusion

This model contains 33 species and 64 reactions [17].

```
33
10000 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
64
1 1 0 1 1 1 10.0
1 1 0 1 1 2 10.0
1 1 0 1 1 3 10.0
1 1 0 1 1 4 10.0
1 1 0 1 1 5 10.0
1 1 0 1 1 6 10.0
1 1 0 1 1 7 10.0
1 1 0 1 1 8 10.0
1 1 0 1 1 9 10.0
1 1 0 1 1 10 10.0
1 1 0 1 1 11 10.0
1 1 0 1 1 12 10.0
1 1 0 1 1 13 10.0
1 1 0 1 1 14 10.0
1 1 0 1 1 15 10.0
1 1 0 1 1 16 10.0
1 1 0 1 1 17 10.0
1 1 0 1 1 18 10.0
1 1 0 1 1 19 10.0
1 1 0 1 1 20 10.0
1 1 0 1 1 21 10.0
1 1 0 1 1 22 10.0
1 1 0 1 1 23 10.0
1 1 0 1 1 24 10.0
1 1 0 1 1 25 10.0
1 1 0 1 1 26 10.0
1 1 0 1 1 27 10.0
1 1 0 1 1 28 10.0
1 1 0 1 1 29 10.0
1 1 0 1 1 30 10.0
1 1 0 1 1 31 10.0
1 1 0 1 1 32 10.0
1 1 1 1 1 0 10.0
1 1 2 1 1 0 10.0
1 1 3 1 1 0 10.0
1 1 4 1 1 0 10.0
1 1 5 1 1 0 10.0
1 1 6 1 1 0 10.0
1 1 7 1 1 0 10.0
1 1 8 1 1 0 10.0
1 1 9 1 1 0 10.0
1 1 10 1 1 0 10.0
1 1 11 1 1 0 10.0
1 1 12 1 1 0 10.0
1 1 13 1 1 0 10.0
1 1 14 1 1 0 10.0
1 1 15 1 1 0 10.0
1 1 16 1 1 0 10.0
1 1 17 1 1 0 10.0
1 1 18 1 1 0 10.0
```

```
1 1 19 1 1 0 10.0
1 1 20 1 1 0 10.0
1 1 21 1 1 0 10.0
1 1 22 1 1 0 10.0
1 1 23 1 1 0 10.0
1 1 24 1 1 0 10.0
1 1 25 1 1 0 10.0
1 1 26 1 1 0 10.0
1 1 27 1 1 0 10.0
1 1 28 1 1 0 10.0
1 1 29 1 1 0 10.0
1 1 30 1 1 0 10.0
1 1 31 1 1 0 10.0
1 1 32 1 1 0 10.0
3
0 1 2
```

A.4 DIMER

This model contains 8 species and 13 reactions [17].

```
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
```

A.5 HSR

This model contains 28 species and 61 reactions [17].

```
28
0 0 0 0 1 4645670 1324 80 16 3413 29 584 1 22 0 171440 9150 2280 6 596 0 13 3 3 7 0 260 0
61
2 1 0 1 1 1 1 2 2.54
1 1 2 2 1 0 1 1 1
2 1 3 1 0 1 1 4 0.254
1 1 4 2 1 3 1 0 1
2 1 0 1 5 1 1 6 0.0254
1 1 6 2 1 0 1 5 10
2 1 3 1 13 1 1 14 254
1 1 14 2 1 3 1 13 10000
2 1 15 1 13 1 1 16 0.000254
1 1 16 2 1 15 1 13 0.01
2 1 2 1 5 1 1 7 0.000254
1 1 7 2 1 2 1 5 1
2 1 4 1 5 1 1 8 0.000254
1 1 8 2 1 4 1 5 1
2 1 2 1 9 1 1 11 2.54
1 1 11 2 1 2 1 9 1
2 1 4 1 10 1 1 12 2540
1 1 12 2 1 4 1 10 1000
2 1 14 1 17 1 1 18 0.0254
1 1 18 2 1 14 1 17 1
1 1 12 2 1 21 1 12 6.62
1 1 21 0 0.5
1 1 21 2 1 13 1 21 20
1 1 13 0 0.03
1 1 16 1 1 15 0.03
1 1 14 1 1 3 0.03
1 1 18 2 1 3 1 17 0.03
1 1 27 2 1 3 1 26 0.03
1 1 12 2 1 22 1 12 1.67
1 1 22 0 0.5
1 1 22 2 1 17 1 22 20
1 1 17 0 0.03
1 1 18 1 1 14 0.03
1 1 11 2 1 24 1 11 0.00625
1 1 24 0 0.5
1 1 24 2 1 3 1 24 7
1 1 3 0 0.03
1 1 18 2 1 17 1 13 3
1 1 20 1 1 19 0.7
1 1 27 2 1 13 1 26 0.5
1 1 12 2 1 23 1 12 1
1 1 23 0 0.5
1 1 23 2 1 19 1 23 20
1 1 19 0 0.03
1 1 20 1 1 3 0.03
2 1 3 1 19 1 1 20 2.54
1 1 20 2 1 3 1 19 10000
1 1 12 2 1 25 1 12 0.43333
1 1 25 0 0.5
1 1 25 2 1 26 1 25 20
```

```
1 1 26 0 0.03
1 1 27 1 1 14 0.03
2 1 14 1 26 1 1 27 2.54
1 1 27 2 1 14 1 26 10000
1 1 4 1 1 0 0.03
1 1 12 2 1 0 1 10 0.03
1 1 8 1 1 6 0.03
1 1 14 1 1 13 0.03
1 1 18 2 1 13 1 17 0.03
1 1 27 2 1 13 1 26 0.03
1 1 20 1 1 19 0.03
28
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
```

A.6 LambdaPhage

This model contains 61 species and 117 reactions [17].

```
61
0 0 12 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 100 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
40 30 1 1 0 0 0 0 0 0 1 0
117
2 1 53 1 2 3 1 53 1 2 10 9 0.011
2 1 50 1 14 1 1 54 0.01
1 1 54 2 1 50 1 14 1.0
3 1 50 1 0 1 12 1 1 6 0.02569
1 1 6 3 1 50 1 0 1 12 1.0
3 1 50 1 0 1 12 1 1 35 0.00967
1 1 35 3 1 50 1 0 1 12 1.0
2 1 26 1 9 1 1 5 0.2
1 1 5 2 1 26 1 9 1.0
2 1 49 1 36 1 1 3 0.01
1 1 3 2 1 49 1 36 0.01
1 1 43 0 7.0E-4
2 1 0 1 12 1 1 10 0.2165
1 1 10 2 1 0 1 12 1.0
3 1 2 1 28 1 60 4 1 2 1 28 1 60 10 55 0.014
3 1 34 1 2 1 41 4 1 34 1 2 1 41 10 43 0.0010
3 1 35 1 2 1 33 4 1 35 1 2 1 33 10 55 0.014
3 1 30 1 5 1 2 4 1 30 1 5 1 2 10 55 0.014
2 1 51 1 9 1 1 60 0.2
1 1 60 2 1 51 1 9 1.0
3 1 34 1 2 1 6 4 1 34 1 2 1 6 10 43 0.0010
1 2 31 1 1 1 0.1
1 1 1 1 2 31 0.5
4 1 1 1 50 1 0 1 12 1 1 19 8.0E-5
1 1 19 4 1 1 1 50 1 0 1 12 1.0
2 1 9 1 52 1 1 33 0.2
1 1 33 2 1 9 1 52 1.0
3 1 53 1 2 1 27 4 10 36 1 53 1 2 1 27 0.0022
2 2 1 1 40 1 1 42 0.0316
1 1 42 2 2 1 1 40 1.0
1 1 56 1 1 21 0.6
3 1 51 1 2 1 28 4 1 51 1 2 1 28 10 55 0.0070
3 2 1 1 50 1 12 1 1 32 5.2E-4
1 1 32 3 2 1 1 50 1 12 1.0
2 1 50 1 12 1 1 28 0.69422
1 1 28 2 1 50 1 12 1.0
1 1 18 1 1 21 0.0010
2 1 0 1 40 1 1 29 0.2025
1 1 29 2 1 0 1 40 1.0
2 1 49 1 55 1 1 37 2.0E-4
1 1 37 2 1 49 1 55 0.05
1 1 9 0 0.00231
2 1 14 1 55 1 1 38 0.00726
1 1 38 2 1 14 1 55 1.0
3 1 1 1 0 1 12 1 1 4 0.1779
1 1 4 3 1 1 1 0 1 12 1.0
1 1 37 1 1 49 0.6
3 1 20 1 2 1 59 4 1 20 1 2 1 59 10 55 0.0070
2 1 2 1 24 3 1 2 10 43 1 24 0.015
```

```
2 1 30 1 2 3 10 31 1 30 1 2 0.014
3 1 50 1 0 1 12 1 1 57 0.0019
1 1 57 3 1 50 1 0 1 12 1.0
1 2 43 1 1 0 0.1
1 1 0 1 2 43 0.5
3 1 35 1 2 1 52 4 1 35 1 2 1 52 10 55 0.0070
3 1 19 1 2 1 7 4 1 19 1 2 10 43 1 7 0.011
2 3 0 1 12 1 1 23 0.00486
1 1 23 2 3 0 1 12 1.0
2 2 1 1 12 1 1 44 0.06684
1 1 44 2 2 1 1 12 1.0
2 2 0 1 40 1 1 46 0.116
1 1 46 2 2 0 1 40 1.0
3 1 53 1 47 1 2 4 10 36 1 53 1 47 1 2 0.011
3 1 2 1 45 1 7 4 1 2 1 45 10 43 1 7 0.011
3 1 1 1 50 1 12 1 1 25 0.01186
1 1 25 3 1 1 1 50 1 12 1.0
2 1 35 1 2 3 1 35 10 31 1 2 0.014
3 1 1 1 0 1 40 1 1 39 0.014
1 1 39 3 1 1 1 0 1 40 1.0
2 1 2 1 54 3 1 2 10 43 1 54 4.0E-5
2 1 9 1 27 1 1 47 0.2
1 1 47 2 1 9 1 27 1.0
3 1 26 1 30 1 2 4 1 26 1 30 1 2 10 55 0.0070
3 1 34 1 2 1 25 4 1 34 1 2 10 43 1 25 0.0010
2 1 50 1 12 1 1 41 0.1362
1 1 41 2 1 50 1 12 1.0
3 1 1 2 0 1 12 1 1 8 0.04266
1 1 8 3 1 1 2 0 1 12 1.0
3 1 1 1 50 1 12 1 1 30 0.25123
1 1 30 3 1 1 1 50 1 12 1.0
3 1 50 1 14 1 55 1 1 24 0.00161
1 1 24 3 1 50 1 14 1 55 1.0
2 1 1 1 14 1 1 11 1.0E-5
1 1 11 2 1 1 1 14 0.1
3 1 34 1 2 1 15 4 1 34 1 2 1 15 10 43 0.0010
1 1 31 0 0.0025
2 1 21 1 55 1 1 56 2.5E-4
1 1 56 2 1 21 1 55 0.065
2 2 0 1 12 1 1 22 0.13136
1 1 22 2 2 0 1 12 1.0
1 1 3 1 1 49 0.0010
2 2 50 1 12 1 1 20 0.1891
1 1 20 2 2 50 1 12 1.0
3 2 1 1 0 1 12 1 1 48 0.00644
1 1 48 3 2 1 1 0 1 12 1.0
3 1 20 1 34 1 2 4 1 20 1 34 1 2 10 43 0.0010
2 1 36 1 21 1 1 18 0.01
1 1 18 2 1 36 1 21 0.01
2 1 1 1 12 1 1 13 0.449
1 1 13 2 1 1 1 12 1.0
2 3 1 1 12 1 1 58 0.00414
1 1 58 2 3 1 1 12 1.0
2 1 1 1 40 1 1 17 0.4132
1 1 17 2 1 1 1 40 1.0
3 1 32 1 34 1 2 4 1 32 1 34 1 2 10 43 0.0010
```

```
4 1 1 1 50 1 0 1 12 1 1 15 0.00112
1 1 15 4 1 1 1 50 1 0 1 12 1.0
3 1 50 2 0 1 12 1 1 45 0.0158
1 1 45 3 1 50 2 0 1 12 1.0
2 1 9 1 59 1 1 16 0.2
1 1 16 2 1 9 1 59 1.0
2 1 2 1 28 3 10 31 1 2 1 28 0.014
3 1 20 1 2 1 16 4 1 20 1 2 1 16 10 55 0.014
2 1 20 1 2 3 10 31 1 20 1 2 0.014
3 1 2 1 57 1 7 4 1 57 1 2 10 43 1 7 0.011
2 1 50 1 40 1 1 53 0.6942
1 1 53 2 1 50 1 40 1.0
61
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
```

A.7 QS8

This model contains 122 species and 201 reactions [17].

```
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
```

A.8 SCHLOGL

This model contains 1 species and 4 reactions [11].

```
1
250
4
1 2 0 1 3 0 0.03
1 3 0 1 2 0 0.0001
0 1 1 0 200
1 1 0 0 3.5
1
0
```

A.9 Trans15Gene

This model contains 32 species and 64 reactions [17].

```
32
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
64
0 1 1 0 784.0
1 1 0 0 2.0
0 1 1 1 78.0
1 1 1 0 2.0
1 1 0 2 1 0 1 2 2.0
1 1 2 0 58.0
2 1 1 1 2 3 1 1 1 2 1 3 22.0
1 1 3 0 3.0
1 1 0 2 1 0 1 4 2.0
1 1 4 0 58.0
2 1 1 1 4 3 1 1 1 4 1 5 22.0
1 1 5 0 3.0
1 1 0 2 1 0 1 6 2.0
1 1 6 0 58.0
2 1 1 1 6 3 1 1 1 6 1 7 22.0
1 1 7 0 3.0
1 1 0 2 1 0 1 8 2.0
1 1 8 0 58.0
2 1 1 1 8 3 1 1 1 8 1 9 22.0
1 1 9 0 3.0
1 1 0 2 1 0 1 10 2.0
1 1 10 0 58.0
2 1 1 1 10 3 1 1 1 10 1 11 22.0
1 1 11 0 3.0
1 1 0 2 1 0 1 12 2.0
1 1 12 0 58.0
2 1 1 1 12 3 1 1 1 12 1 13 22.0
1 1 13 0 3.0
1 1 0 2 1 0 1 14 2.0
1 1 14 0 58.0
2 1 1 1 14 3 1 1 1 14 1 15 22.0
1 1 15 0 3.0
1 1 0 2 1 0 1 16 2.0
1 1 16 0 58.0
2 1 1 1 16 3 1 1 1 16 1 17 22.0
1 1 17 0 3.0
1 1 0 2 1 0 1 18 2.0
1 1 18 0 58.0
2 1 1 1 18 3 1 1 1 18 1 19 22.0
1 1 19 0 3.0
1 1 0 2 1 0 1 20 2.0
1 1 20 0 58.0
2 1 1 1 20 3 1 1 1 20 1 21 22.0
1 1 21 0 3.0
1 1 0 2 1 0 1 22 2.0
1 1 22 0 58.0
2 1 1 1 22 3 1 1 1 22 1 23 22.0
1 1 23 0 3.0
1 1 0 2 1 0 1 24 2.0
1 1 24 0 58.0
```

```
2 1 1 1 24 3 1 1 1 24 1 25 22.0
1 1 25 0 3.0
1 1 0 2 1 0 1 26 2.0
1 1 26 0 58.0
2 1 1 1 26 3 1 1 1 26 1 27 22.0
1 1 27 0 3.0
1 1 0 2 1 0 1 28 2.0
1 1 28 0 58.0
2 1 1 1 28 3 1 1 1 28 1 29 22.0
1 1 29 0 3.0
1 1 0 2 1 0 1 30 2.0
1 1 30 0 58.0
2 1 1 1 30 3 1 1 1 30 1 31 22.0
1 1 31 0 3.0
4
3 5 7 9
```

A.10 Trans1Gene

This model contains 4 species and 8 reactions [11].

```
4
0 0 0 0
8
0 1 1 0 784.0
1 1 0 0 2.0
1 1 0 2 1 0 1 1 2.0
1 1 1 0 58.0
0 1 1 2 78.0
1 1 2 0 2.0
2 1 1 1 2 3 1 1 1 2 1 3 22.0
1 1 3 0 3.0
4
0 1 2 3
```

Appendix B

NVIDIA GPU Specifications

This section gives the output of the deviceQuery and bandwidthTest programs from the NVIDIA CUDA SDK.

B.1 GeForce 8600M

From the deviceQuery program:

```
Device 0: "GeForce 8600M GT"
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         1
  Total amount of global memory:                 268238848 bytes
  Number of multiprocessors:                     4
  Number of cores:                               32
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    0.75 GHz
  Concurrent copy and execution:                 No
  Run time limit on kernels:                     Yes
  Integrated:                                    No
  Support host page-locked memory mapping:       No
```

From the bandwidthTest program:

```
Running on......
      device 0:GeForce 8600M GT
Quick Mode
Host to Device Bandwidth for Pageable memory
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               263.6
```

```
Quick Mode
Device to Host Bandwidth for Pageable memory
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               1180.0

Quick Mode
Device to Device Bandwidth
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               10105.0

&&&& Test PASSED
```

B.2 Tesla c870

From the deviceQuery program:

```
Device 0: "Tesla C870"
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         0
  Total amount of global memory:                 1610350592 bytes
  Number of multiprocessors:                     16
  Number of cores:                               128
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 8192
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.19 GHz
  Concurrent copy and execution:                 No
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       No
```

From the bandwidthTest program:

```
Running on......
      device 0:Tesla C870
Quick Mode
Host to Device Bandwidth for Pageable memory
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               1215.6

Quick Mode
Device to Host Bandwidth for Pageable memory
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               942.8

Quick Mode
Device to Device Bandwidth
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               51433.8

&&&& Test PASSED
```

B.3 Tesla c1060

From the deviceQuery program:

```
Device 0: "Tesla C1060"
  CUDA Capability Major revision number:         1
  CUDA Capability Minor revision number:         3
  Total amount of global memory:                 4294705152 bytes
  Number of multiprocessors:                     30
  Number of cores:                               240
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          262144 bytes
  Texture alignment:                             256 bytes
  Clock rate:                                    1.30 GHz
  Concurrent copy and execution:                 Yes
  Run time limit on kernels:                     No
  Integrated:                                    No
  Support host page-locked memory mapping:       Yes
```

From the bandwidthTest program:

```
Running on......
      device 0:Tesla C1060
Quick Mode
Host to Device Bandwidth for Pageable memory
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               5337.5

Quick Mode
Device to Host Bandwidth for Pageable memory
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               4454.2

Quick Mode
Device to Device Bandwidth
.
Transfer Size (Bytes)   Bandwidth(MB/s)
 33554432               73657.5

&&&& Test PASSED
```

Appendix C

Source Code

This appendix will list all of the source code for the work done in this research.

C.1 fields.h

This is the header file for the fields library that is used to read in the model files. Courtesy of [22].

```
#ifndef _FIELDS_H_
#define _FIELDS_H_

#define MAXLEN 1001
#define MAXFIELDS 1000

typedef struct inputstruct {
  char *name;                /* File name */
  FILE *f;                   /* File descriptor */
  int line;                  /* Line number */
  char text1[MAXLEN];        /* The line */
  char text2[MAXLEN];        /* Working -- contains fields */
  int NF;                    /* Number of fields */
  char *fields[MAXFIELDS];   /* Pointers to fields */
  int file;                  /* 1 for file, 0 for popen */
} *IS;

extern IS new_inputstruct(char* filename);
extern IS pipe_inputstruct(char* command);
extern int get_line(IS is); /* returns NF, or -1 on EOF.  Does not
                               close the file */
extern void jettison_inputstruct(IS is);  /* frees the IS and fcloses
                                             the file */
#endif
```

C.2 fields.c

This is the source file for the fields library that is used to read in the model files. Courtesy of [22].

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "fields.h"

#define talloc(ty, sz) (ty *) malloc (sz * sizeof(ty))
#define strdup(s) ((char *) strcpy(talloc(char, strlen(s)+1), s))

static IS make_inputstruct(char* filename, char* key)
{
  IS is;
  int file;

  if (strcmp(key, "f") == 0) {
    file = 1;
  } else if (strcmp(key, "p") == 0) {
    file = 0;
  } else {
    return NULL;
  }

  is = talloc(struct inputstruct, 1);

  is->text1[MAXLEN-1] = '\0';
  is->NF = 0;
  is->line = 0;
  if (filename == NULL) {
    is->name = "stdin";
    is->f = stdin;
  } else {
    is->name = filename;
    is->file = file;
    if (file) {
      is->f = fopen(filename, "r");
    } else {
      is->f = popen(filename, "r");
    }
    if (is->f == NULL) {
      free(is);
      return NULL;
    }
  }
  return is;
}



IS new_inputstruct(char *filename)   /* use NULL for stdin.  Calls malloc */
{
  return make_inputstruct(filename, "f");
}

IS pipe_inputstruct(char *command)
{
  return make_inputstruct(command, "p");
}

int get_line(IS is)
{
  int i, len;
  int f;
```

```
  char *tmp;
  char lastchar;
  char *line;

  is->NF = 0;

  if (fgets(is->text1, MAXLEN-1, is->f) == NULL) {
    is->NF = -1;
    return -1;
  }

  is->line++;
  strcpy(is->text2, is->text1);

  line = is->text2;
  lastchar = ' ';
  for (i = 0; line[i] != '\0' && i < MAXLEN-1; i++) {
    if (isspace(line[i])) {
      lastchar = line[i];
      line[i] = '\0';
    } else {
      if (isspace(lastchar)) {
        is->fields[is->NF] = line+i;
        is->NF++;
      }
      lastchar = line[i];
    }
  }
  return is->NF;
}

void jettison_inputstruct(IS is)
{
  if (is->f != stdin) {
    if (is->file) {
      fclose(is->f);
    } else {
      pclose(is->f);
    }
  }
  free(is);
  return;
}
```

C.3 firstReaction.h

This is the header file for the serial First Reaction Method implementation.

```
#ifndef _FIRST_H_
#define _FIRST_H_
#include <math.h>

#define MAX(x,y) (x>y?x:y)


typedef struct {
  long speciesSize;
  long reactionSize;
  long numToBePrinted;
  long selectedReaction;
  double MYINFINITY;
  long* netUpdate;
  long* species;
  double* rateConstant;
  long* speciesToOutput;
  long* reactSize;
  long** reactIndex;
  long** reactCoeff;
  long* prodSize;
  long** prodIndex;
  long** prodCoeff;
  double currentTime;
}ESS;

typedef struct{
  double* prop;
  double total;
}Prop;

#endif
```

C.4 firstReaction.c

This is the source file for the serial First Reaction Method implementation.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include "firstReaction.h"
#include "fields.h"

long factorial(long x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

void Initialize(char* inputFile, ESS* d, Prop* p){
  int i=0,j,k=0,l, m=-1,n=0;
  int maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (long*)malloc(sizeof(long)*d->speciesSize);
        if(d->species==NULL){
          perror("In initialize, d->species");
          exit(1);
        }
        m=0;
      }else if(i==1){
        //store the reaction size
        d->reactionSize = atol(is->fields[0]);
        d->netUpdate = (long*)malloc(sizeof(long)*d->speciesSize*d->reactionSize);
        if(d->netUpdate==NULL){
          perror("In initialize, d->netUpdate");
          exit(1);
        }
        d->rateConstant = (double*)malloc(sizeof(double)*d->reactionSize);
        if(d->rateConstant==NULL){
          perror("In initialize, d->rateConstant");
          exit(1);
        }
        //allocate memory for the reactants
        d->reactSize = (long*)malloc(sizeof(long)*d->reactionSize);
        if(d->reactSize==NULL){
          perror("In initialize, d->reactSize");
          exit(1);
        }
        d->reactCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
        if(d->reactCoeff==NULL){
          perror("In initialize, d->reactCoeff");
          exit(1);
        }
        d->reactIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
```

```
    if(d->reactIndex==NULL){
      perror("In initialize, d->reactIndex");
      exit(1);
    }
    //allocate memory for the products
    d->prodSize = (long*)malloc(sizeof(long)*d->reactionSize);
    if(d->prodSize==NULL){
      perror("In initialize, d->prodSize");
      exit(1);
    }
    d->prodCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodCoeff==NULL){
      perror("In initialize, d->prodCoeff");
      exit(1);
    }
    d->prodIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodIndex==NULL){
      perror("In initialize, d->prodIndex");
      exit(1);
    }
    //allocate memory for the propensities
    p->prop = (double*)malloc(sizeof(double)*d->reactionSize);
    if(p->prop==NULL){
      perror("In initialize, p->prop");
      exit(1);
    }
    memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(long));
  }else if(i==2){
    //store the number to be printed
    d->numToBePrinted = atol(is->fields[0]);
    d->speciesToOutput = (long*)malloc(sizeof(long)*d->numToBePrinted);
    if(d->speciesToOutput==NULL){
      perror("In initialize, d->speciesToOutput");
      exit(1);
    }
    m= 2;
  }else{
    fprintf(stderr,"Malformed File!!! Try again\n");
    exit(1);
  }
  i++;
}else{
  if(m==0){
    //Store the inital species populations
    for(j=0;j<is->NF;j++){
      d->species[j] = atol(is->fields[j]);
    }
    m++;
  }else{
    if(k==d->reactionSize){
      //store the indices to be printed
      for(j=0;j<d->numToBePrinted;j++){
        d->speciesToOutput[j] = atol(is->fields[j]);
      }
    }else{
      maxCoeff=0;
      d->reactSize[k] = atol(is->fields[0]);

      //store the reactants
      d->reactCoeff[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
      if(d->reactCoeff[k]==NULL){
        perror("In initialize, d->reactCoeff[k]");
        exit(1);
      }
      d->reactIndex[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
```

```
            if(d->reactIndex[k]==NULL){
              perror("In initialize, d->reactIndex[k]");
              exit(1);
            }
            for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
              d->reactCoeff[k][l] = atol(is->fields[j+1]);
              maxCoeff = MAX(maxCoeff,d->reactCoeff[k][l]);
              d->reactIndex[k][l] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->reactIndex[k][l]] -= d->reactCoeff[k][l];

            }
            j++;
            d->prodSize[k] = atol(is->fields[j]);
            d->prodCoeff[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodCoeff[k]==NULL){
              perror("In initialize, d->prodCoeff[k]");
              exit(1);
            }
            d->prodIndex[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodIndex[k]==NULL){
              perror("In initialize, d->prodIndex[k]");
              exit(1);
            }
            //store the products
            for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
              d->prodCoeff[k][n] = atol(is->fields[j+1]);
              d->prodIndex[k][n] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->prodIndex[k][n]] += d->prodCoeff[k][n];
            }
            j++;
            //store the rate constants
            if(d->reactSize[k]!=0){
              d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
            }else{
              d->rateConstant[k] = atof(is->fields[j]);
            }
          }
          k++;
        }
      }
    }
    jettison_inputstruct(is);
    d->currentTime = 0.0;
    d->MYINFINITY = -log(0.0);
}

double indProp(ESS* d, Prop* p, long i){
    long j,k;
    long prop=1;
    //calculate the propensity for a reaction
    for(j=0; j<d->reactSize[i]; j++) {
      for(k=0; k<d->reactCoeff[i][j]; k++) {
        // if (d->species[d->reactIndex[i][j]] - k == 0){
        //   return 0.0;
        // }
        prop *= (d->species[d->reactIndex[i][j]] - k);
      }
    }
    p->prop[i] = prop*d->rateConstant[i];
    // return prop*d->rateConstant[i];
}

void calcProp(ESS* d, Prop* p, double randmaxRecip){
    long i;
    double t=d->MYINFINITY,temp;
```

```
  p->total=0.0;
  //calculate all the propensities
  for(i=0;i<d->reactionSize;i++){
    indProp(d,p,i);
    temp = -log(rand()*randmaxRecip)/p->prop[i];
    if(temp < t){
      t = temp;
      d->selectedReaction = i;
    }
  }
  d->currentTime += t;
}

void updateSpecies(ESS* d){
  long i;
  for(i=0;i<d->reactSize[d->selectedReaction];i++){
    d->species[d->reactIndex[d->selectedReaction][i]] -=
      d->reactCoeff[d->selectedReaction][i];
  }

  for(i=0;i<d->prodSize[d->selectedReaction];i++){
    d->species[d->prodIndex[d->selectedReaction][i]] +=
      d->prodCoeff[d->selectedReaction][i];
  }

}

void printSpecies(ESS* d, double currentTime){
  long i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%d ", d->species[d->speciesToOutput[i]]);
  }
  printf("\n");
}

void destroyESS(ESS* d, Prop* p){
  free(p->prop);
  free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
  free(d->speciesToOutput);
  free(d->reactSize);
  free(d->prodSize);
  for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
    free(d->reactIndex[d->reactionSize]);
    free(d->reactCoeff[d->reactionSize]);
    free(d->prodIndex[d->reactionSize]);
    free(d->prodCoeff[d->reactionSize]);
  }
  free(d->reactIndex);
  free(d->reactCoeff);
  free(d->prodIndex);
  free(d->prodCoeff);
}

int main(int argc, char* argv[]){
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions #Simulations\n");
    // fprintf(stderr, "\toutput - 1 to output the species, 0 to suppress outputs\n");
    exit(1);
  }
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);
```

```
  ESS d;
  Prop p;
  long numReactions=atol(argv[2]);
  long numSimulations = atol(argv[3]);
  // int output = atol(argv[3]);
  long selectedReaction;
  double randmaxRecip = (double)1/RAND_MAX;
  srand(0);

  double currentTime=0.0;
  //Read in the file and initialize everything
  Initialize(argv[1],&d,&p);
  long species[d.speciesSize];
  int i,j=numReactions,k,l;
  memcpy(species,d.species,d.speciesSize*sizeof(long));
  for(i=0;i<numSimulations;i++){
    srand(i);
    memcpy(d.species,species,d.speciesSize*sizeof(long));
    d.currentTime=0.0;
    // while(currentTime <= .5){
    for(numReactions=j;numReactions>=0;numReactions--){
      //calculate the propensities
      calcProp(&d, &p, randmaxRecip);
      //update the species
      updateSpecies(&d);
    }
  }
//  printSpecies(&d,d.currentTime);
  gettimeofday(&after,NULL);
  double totalTime = after.tv_sec - before.tv_sec;
  totalTime += ((double)after.tv_usec)/1000000.0;
  totalTime -= ((double)before.tv_usec)/1000000.0;

  printf("Total Time: %f\n", totalTime);

  destroyESS(&d, &p);

  return 0;
}
```

C.5 direct.h

This is the header file for the direct method implementation.

```
#ifndef _DIRECT_H_
#define _DIREC_H_

#define MAX(x,y) (x>y?x:y)

typedef struct {
  long speciesSize;
  long reactionSize;
  long numToBePrinted;
  long* netUpdate;
  long* species;
  double* rateConstant;
  long* speciesToOutput;
  long* reactSize;
  long** reactIndex;
  long** reactCoeff;
  long* prodSize;
  long** prodIndex;
  long** prodCoeff;
}ESS;

typedef struct{
  double* prop;
  double total;
}Prop;

#endif
```

C.6 direct.c

This lists the c code that implements the Direct Method.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include "direct.h"
#include "fields.h"

long factorial(long x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

void Initialize(char* inputFile, ESS* d, Prop* p){
  int i=0,j,k=0,l, m=-1,n=0;
  int maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (long*)malloc(sizeof(long)*d->speciesSize);
        if(d->species==NULL){
          perror("In initialize, d->species");
          exit(1);
        }
        m=0;
      }else if(i==1){
        //store the reaction size
        d->reactionSize = atol(is->fields[0]);
        d->netUpdate = (long*)malloc(sizeof(long)*d->speciesSize*d->reactionSize);
        if(d->netUpdate==NULL){
          perror("In initialize, d->netUpdate");
          exit(1);
        }
        d->rateConstant = (double*)malloc(sizeof(double)*d->reactionSize);
        if(d->rateConstant==NULL){
          perror("In initialize, d->rateConstant");
          exit(1);
        }
        //allocate memory for the reactants
        d->reactSize = (long*)malloc(sizeof(long)*d->reactionSize);
        if(d->reactSize==NULL){
          perror("In initialize, d->reactSize");
          exit(1);
        }
        d->reactCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
        if(d->reactCoeff==NULL){
          perror("In initialize, d->reactCoeff");
          exit(1);
        }
        d->reactIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
```

```
      if(d->reactIndex==NULL){
        perror("In initialize, d->reactIndex");
        exit(1);
      }
      //allocate memory for the products
      d->prodSize = (long*)malloc(sizeof(long)*d->reactionSize);
      if(d->prodSize==NULL){
        perror("In initialize, d->prodSize");
        exit(1);
      }
      d->prodCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
      if(d->prodCoeff==NULL){
        perror("In initialize, d->prodCoeff");
        exit(1);
      }
      d->prodIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
      if(d->prodIndex==NULL){
        perror("In initialize, d->prodIndex");
        exit(1);
      }
      //allocate memory for the propensities
      p->prop = (double*)malloc(sizeof(double)*d->reactionSize);
      if(p->prop==NULL){
        perror("In initialize, p->prop");
        exit(1);
      }
      memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(long));
    }else if(i==2){
      //store the number to be printed
      d->numToBePrinted = atol(is->fields[0]);
      d->speciesToOutput = (long*)malloc(sizeof(long)*d->numToBePrinted);
      if(d->speciesToOutput==NULL){
        perror("In initialize, d->speciesToOutput");
        exit(1);
      }
      m= 2;
    }else{
      fprintf(stderr,"Malformed File!!! Try again\n");
      exit(1);
    }
    i++;
  }else{
    if(m==0){
      //Store the inital species populations
      for(j=0;j<is->NF;j++){
        d->species[j] = atol(is->fields[j]);
      }
      m++;
    }else{
      if(k==d->reactionSize){
        //store the indices to be printed
        for(j=0;j<d->numToBePrinted;j++){
          d->speciesToOutput[j] = atol(is->fields[j]);
        }
      }else{
        maxCoeff=0;
        d->reactSize[k] = atol(is->fields[0]);

        //store the reactants
        d->reactCoeff[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
        if(d->reactCoeff[k]==NULL){
          perror("In initialize, d->reactCoeff[k]");
          exit(1);
        }
        d->reactIndex[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
```

```
            if(d->reactIndex[k]==NULL){
              perror("In initialize, d->reactIndex[k]");
              exit(1);
            }
            for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
              d->reactCoeff[k][l] = atol(is->fields[j+1]);
              maxCoeff = MAX(maxCoeff,d->reactCoeff[k][l]);
              d->reactIndex[k][l] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->reactIndex[k][l]] -= d->reactCoeff[k][l];

            }
            j++;
            d->prodSize[k] = atol(is->fields[j]);
            d->prodCoeff[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodCoeff[k]==NULL){
              perror("In initialize, d->prodCoeff[k]");
              exit(1);
            }
            d->prodIndex[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodIndex[k]==NULL){
              perror("In initialize, d->prodIndex[k]");
              exit(1);
            }
            //store the products
            for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
              d->prodCoeff[k][n] = atol(is->fields[j+1]);
              d->prodIndex[k][n] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->prodIndex[k][n]] += d->prodCoeff[k][n];
            }
            j++;
            //store the rate constants
            if(d->reactSize[k]!=0){
              d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
            }else{
              d->rateConstant[k] = atof(is->fields[j]);
            }
          }
          k++;
        }
      }
    }
    jettison_inputstruct(is);
}

double indProp(ESS* d, long i){
  long j,k;
  long prop=1;
  //calculate the propensity for a reaction
  for(j=0; j<d->reactSize[i]; j++) {
    for(k=0; k<d->reactCoeff[i][j]; k++) {
      if (d->species[d->reactIndex[i][j]] - k == 0){
        return 0.0;
      }
      prop *= (d->species[d->reactIndex[i][j]] - k);
    }
  }

  return prop*d->rateConstant[i];
}

void calcProp(ESS* d, Prop* p){
  long i;
  p->total=0.0;
  //calculate all the propensities
  for(i=0;i<d->reactionSize;i++){
```

```
    p->prop[i] = indProp(d,i);
    p->total+=p->prop[i];
  }
}

long reactionSelect(Prop* p, long reactionSize, double randmaxRecip){
  long i;
  double scaledRate = rand()*randmaxRecip*p->total;
  for(i=0;i<reactionSize;i++){
    scaledRate -= p->prop[i];
    if(scaledRate <= 0){
      return i;
    }
  }
  return i-1;
}

void updateSpecies(ESS* d, long selectedReaction){
  long i;

  for(i=0;i<d->reactSize[selectedReaction];i++){
    d->species[d->reactIndex[selectedReaction][i]] -= d->reactCoeff[selectedReaction][i];
  }

  for(i=0;i<d->prodSize[selectedReaction];i++){
    d->species[d->prodIndex[selectedReaction][i]] += d->prodCoeff[selectedReaction][i];
  }

}

void printSpecies(ESS* d, double currentTime){
  long i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%d ", d->species[d->speciesToOutput[i]]);
  }
  printf("\n");
}

void destroyESS(ESS* d, Prop* p){
  free(p->prop);
  free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
  free(d->speciesToOutput);
  free(d->reactSize);
  free(d->prodSize);
  for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
    free(d->reactIndex[d->reactionSize]);
    free(d->reactCoeff[d->reactionSize]);
    free(d->prodIndex[d->reactionSize]);
    free(d->prodCoeff[d->reactionSize]);
  }
  free(d->reactIndex);
  free(d->reactCoeff);
  free(d->prodIndex);
  free(d->prodCoeff);
}

int main(int argc, char* argv[]){
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions #Simulations\n");
    // fprintf(stderr, "\toutput - 1 to output the species, 0 to suppress outputs\n");
    exit(1);
  }
```

```
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);

  ESS d;
  Prop p;
  long numReactions=atol(argv[2]);
  long numSimulations = atol(argv[3]);
  // int output = atol(argv[3]);
  long selectedReaction;
  double randmaxRecip = (double)1/RAND_MAX;
  srand(0);

  double currentTime=0.0;
  //Read in the file and initialize everything
  Initialize(argv[1],&d,&p);
  long species[d.speciesSize];
  int i,j=numReactions,k,l;
  memcpy(species,d.species,d.speciesSize*sizeof(long));
  for(i=0;i<numSimulations;i++){
    srand(i);
    memcpy(d.species,species,d.speciesSize*sizeof(long));
    currentTime=0.0;
    for(numReactions=j;numReactions>=0;numReactions--){
      // if(output == 1){
      //   printSpecies(&d,currentTime);
      // }
      //calculate the propensities
      calcProp(&d, &p);
      //select the reaction
      selectedReaction = reactionSelect(&p,d.reactionSize,randmaxRecip);

      //calculate the current time
      currentTime += -log(rand()*randmaxRecip)/p.total;
      //update the species
      updateSpecies(&d,selectedReaction);
    }
  }
  printSpecies(&d,currentTime);
  gettimeofday(&after,NULL);
  double totalTime = after.tv_sec - before.tv_sec;
  totalTime += ((double)after.tv_usec)/1000000.0;
  totalTime -= ((double)before.tv_usec)/1000000.0;

  printf("Total Time: %f\n", totalTime);

  destroyESS(&d, &p);

  return 0;
}
```

C.7 MinHeap.h

This is the header file for the MinHeap implementation.

```
#ifndef _MINHEAP_H
#define _MINHEAP_H

#include <stdlib.h>

typedef struct Node_s{
  struct Node_s* left;
  struct Node_s* right;
  struct Node_s* parent;
  double value;
  unsigned long id;
  unsigned long treeSize;
}Node;

typedef struct{
  Node *d_top;
  Node **d_nodes;
  unsigned long d_nodeCount;
}MinHeap;

void initNode(Node* n);
void destroyNode(Node* n);
void initMinHeap(MinHeap* m, unsigned long size);
void destroyMinHeap(MinHeap* m);
void outputMinHeap(Node *n);
void updateMinHeap(MinHeap* m, unsigned long id, double value);
unsigned long getMinMinHeap(MinHeap* m);
void update_auxMinHeap(MinHeap* m,Node *n);
unsigned long nodeCountMinHeap(Node *tree);
void addMinHeap(Node *n, Node *tree);
void swapMinHeap(MinHeap* m, unsigned long id1, unsigned long id2);

#endif
```

C.8 MinHeap.c

This is the source file for the serial MinHeap implementation.

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

void initNode(Node* n){
  n->left = NULL;
    n->right = NULL;
    n->parent = NULL;
    n->value = 0.0;
    n->id = 0;
    n->treeSize = 1;
}

void destroyNode(Node* n){
  if (n->left != NULL) free(n->left);
    if (n->right != NULL) free(n->right);
}

void initMinHeap(MinHeap* m, unsigned long size) {
  m->d_top = NULL;
  m->d_nodes = (Node**)malloc(sizeof(Node*)*size);
  if (m->d_nodes == NULL) {
    perror("MinHeap d_nodes memory allocation");
    exit(1);
  }
  m->d_nodeCount = size;
  unsigned long i;
  for(i=0; i<m->d_nodeCount; i++) {
    m->d_nodes[i] = (Node*)malloc(sizeof(Node));
    if (m->d_nodes[i] == NULL) {
      perror("MinHeap d_nodes[i] memory allocation");
      exit(1);
    }
    initNode(m->d_nodes[i]);
    m->d_nodes[i]->id = i;

    if (m->d_top == NULL) {
      m->d_top = m->d_nodes[i];
    } else {
      addMinHeap(m->d_nodes[i],m->d_top);
    }
  }
}

void destroyMinHeap(MinHeap* m) {
  if (m->d_top != NULL) free(m->d_top);
  if (m->d_nodes != NULL){
    int i;
    for(i=0;i<m->d_nodeCount;i++){
      free(m->d_nodes[i]);
    }
  }
}

void outputMinHeap(Node *n) {
  if (n != NULL) {
    printf("%f(",n->value);
    outputMinHeap(n->left);
    printf(",");
    outputMinHeap(n->right);
    printf(")");
  }
```

```
}

void updateMinHeap(MinHeap* m, unsigned long id, double value) {
  m->d_nodes[id]->value = value;
  update_auxMinHeap(m, m->d_nodes[id]);
}

unsigned long getMinMinHeap(MinHeap* m) {
  return m->d_top->id;
}

void update_auxMinHeap(MinHeap* m, Node *n) {
  if ((n->parent != NULL) && (n->value < n->parent->value)) {
    swapMinHeap(m, n->id,n->parent->id);
    update_auxMinHeap(m, 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) {
    swapMinHeap(m, minchild->id,n->id);
    update_auxMinHeap(m, minchild);
  }
}

unsigned long nodeCountMinHeap(Node *tree) {
  if (tree == NULL) {
    return 0;
  } else {
    return 1+nodeCountMinHeap(tree->left)+nodeCountMinHeap(tree->right);
  }
}

void addMinHeap(Node *n, Node *tree) {
  if (tree->left == NULL) {
    tree->left = n;
    n->parent = tree;
    Node *temp = tree;
    while (temp != NULL) {
      temp->treeSize++;
      temp = temp->parent;
    }
    return;
  }
  if (tree->right == NULL) {
    tree->right = n;
    n->parent = tree;
    Node *temp = tree;
    while (temp != NULL) {
      temp->treeSize++;
      temp = temp->parent;
    }
```

```
    return;
  }
  unsigned long leftCount = tree->left->treeSize;
  unsigned long rightCount = tree->right->treeSize;
  if (leftCount <= rightCount) {
    addMinHeap(n,tree->left);
  } else {
    addMinHeap(n,tree->right);
  }
}

void swapMinHeap(MinHeap* m, unsigned long id1, unsigned long id2) {

  double temp = m->d_nodes[id1]->value;
  m->d_nodes[id1]->value = m->d_nodes[id2]->value;
  m->d_nodes[id2]->value = temp;
  m->d_nodes[id1]->id = id2;
  m->d_nodes[id2]->id = id1;

  Node *n = m->d_nodes[id1];
  m->d_nodes[id1] = m->d_nodes[id2];
  m->d_nodes[id2] = n;
}
```

C.9 nextReaction.h

This is the header file for the serial Next Reaction Method implementation.

```
#ifndef _NEXT_H_
#define _NEXT_H_
#include <math.h>
#include "MinHeap.h"

#define MAX(x,y) (x>y?x:y)


typedef struct {
  long speciesSize;
  long reactionSize;
  long numToBePrinted;
  unsigned long selectedReaction;
  double MYINFINITY;
  long* netUpdate;
  long* species;
  double* rateConstant;
  long* speciesToOutput;
  long* reactSize;
  long** reactIndex;
  long** reactCoeff;
  long* prodSize;
  long** prodIndex;
  long** prodCoeff;
  double currentTime;
  MinHeap minHeap;
  double* putativeTimes;
  long* numDependency;
  long** dependency;
  unsigned long* reactionCounts;
}ESS;

typedef struct{
  double* prop;
  double total;
}Prop;

#endif
```

C.10 nextReaction.c

This is the source file for the serial Next Reaction Method implementation.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include "nextReaction.h"
#include "fields.h"
#include "MinHeap.h"

long factorial(long x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

double indProp(ESS* d, Prop* p, long i){
  long j,k;
  long prop=1;
  //calculate the propensity for a reaction
  for(j=0; j<d->reactSize[i]; j++) {
    for(k=0; k<d->reactCoeff[i][j]; k++) {
      prop *= (d->species[d->reactIndex[i][j]] - k);
    }
  }
  p->prop[i] = prop*d->rateConstant[i];
}

void calcProp(ESS* d, Prop* p, double randmaxRecip){
  long i;
  double t=d->MYINFINITY,temp;
  p->total=0.0;
  //calculate all the propensities
  for(i=0;i<d->reactionSize;i++){
    indProp(d,p,i);
    temp = -log(rand()*randmaxRecip)/p->prop[i];
    if(temp < t){
      t = temp;
      d->selectedReaction = i;
    }
  }
  d->currentTime += t;
}

void getDependencies(ESS* d){
  long* temp[d->reactionSize];
  long i,j,k,l,m,count=0;
  d->numDependency = (long*)malloc(sizeof(long)*d->reactionSize);
  d->dependency = (long**)malloc(sizeof(long*)*d->reactionSize);
  for(i=0;i<d->reactionSize;i++){
    d->numDependency[i] = count;
    d->dependency[i] = (long*)malloc(sizeof(long)*d->reactionSize);
    for(j=0,count=0;j<d->speciesSize;j++){
      if(d->netUpdate[d->speciesSize*i+j]!=0){
        for(k=0;k<d->reactionSize;k++){
          for(l=0;l<d->reactSize[k];l++){
            if(d->reactIndex[k][l]==j){
              for(m=0;m<count;m++){
                if(d->dependency[i][count]==k) break;
```

```
                }
                if(m==count){
                  d->dependency[i][count] = k;
                  count++;
                }
              }
            }
          }
        }
      }
      d->numDependency[i] = count;
      d->dependency[i] = realloc(d->dependency[i],count*sizeof(long));
    }
  }

void Initialize(char* inputFile, ESS* d, Prop* p){
  int i=0,j,k=0,l, m=-1,n=0;
  int maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (long*)malloc(sizeof(long)*d->speciesSize);
        if(d->species==NULL){
          perror("In initialize, d->species");
          exit(1);
        }
        m=0;
      }else if(i==1){
        //store the reaction size
        d->reactionSize = atol(is->fields[0]);
        d->netUpdate = (long*)malloc(sizeof(long)*d->speciesSize*d->reactionSize);
        if(d->netUpdate==NULL){
          perror("In initialize, d->netUpdate");
          exit(1);
        }
        d->rateConstant = (double*)malloc(sizeof(double)*d->reactionSize);
        if(d->rateConstant==NULL){
          perror("In initialize, d->rateConstant");
          exit(1);
        }
        //allocate memory for the reactants
        d->reactSize = (long*)malloc(sizeof(long)*d->reactionSize);
        if(d->reactSize==NULL){
          perror("In initialize, d->reactSize");
          exit(1);
        }
        d->reactCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
        if(d->reactCoeff==NULL){
          perror("In initialize, d->reactCoeff");
          exit(1);
        }
        d->reactIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
        if(d->reactIndex==NULL){
          perror("In initialize, d->reactIndex");
          exit(1);
        }
        //allocate memory for the products
        d->prodSize = (long*)malloc(sizeof(long)*d->reactionSize);
```

```
    if(d->prodSize==NULL){
      perror("In initialize, d->prodSize");
      exit(1);
    }
    d->prodCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodCoeff==NULL){
      perror("In initialize, d->prodCoeff");
      exit(1);
    }
    d->prodIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodIndex==NULL){
      perror("In initialize, d->prodIndex");
      exit(1);
    }
    //allocate memory for the propensities
    p->prop = (double*)malloc(sizeof(double)*d->reactionSize);
    if(p->prop==NULL){
      perror("In initialize, p->prop");
      exit(1);
    }
    memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(long));
    initMinHeap(&d->minHeap, d->reactionSize);
    d->putativeTimes = (double*)malloc(sizeof(double)*d->reactionSize);
    if(d->putativeTimes==NULL){
      perror("In initialize, d->putativeTimes");
      exit(1);
    }
    d->reactionCounts = (unsigned long*)malloc(sizeof(unsigned long)*d->reactionSize);
    if(d->reactionCounts==NULL){
      perror("In initialize, d->reactionCounts");
      exit(1);
    }
  }else if(i==2){
    //store the number to be printed
    d->numToBePrinted = atol(is->fields[0]);
    d->speciesToOutput = (long*)malloc(sizeof(long)*d->numToBePrinted);
    if(d->speciesToOutput==NULL){
      perror("In initialize, d->speciesToOutput");
      exit(1);
    }
    m= 2;
  }else{
    fprintf(stderr,"Malformed File!!! Try again\n");
    exit(1);
  }
  i++;
}else{
  if(m==0){
    //Store the inital species populations
    for(j=0;j<is->NF;j++){
      d->species[j] = atol(is->fields[j]);
    }
    m++;
  }else{
    if(k==d->reactionSize){
      //store the indices to be printed
      for(j=0;j<d->numToBePrinted;j++){
        d->speciesToOutput[j] = atol(is->fields[j]);
      }
    }else{
      maxCoeff=0;
      d->reactSize[k] = atol(is->fields[0]);

      //store the reactants
      d->reactCoeff[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
```

```
            if(d->reactCoeff[k]==NULL){
              perror("In initialize , d->reactCoeff[k]");
              exit(1);
            }
            d->reactIndex[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
            if(d->reactIndex[k]==NULL){
              perror("In initialize , d->reactIndex[k]");
              exit(1);
            }
            for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
              d->reactCoeff[k][l] = atol(is->fields[j+1]);
              maxCoeff = MAX(maxCoeff,d->reactCoeff[k][l]);
              d->reactIndex[k][l] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->reactIndex[k][l]] -= d->reactCoeff[k][l];

            }
            j++;
            d->prodSize[k] = atol(is->fields[j]);
            d->prodCoeff[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodCoeff[k]==NULL){
              perror("In initialize , d->prodCoeff[k]");
              exit(1);
            }
            d->prodIndex[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodIndex[k]==NULL){
              perror("In initialize , d->prodIndex[k]");
              exit(1);
            }
            //store the products
            for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
              d->prodCoeff[k][n] = atol(is->fields[j+1]);
              d->prodIndex[k][n] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->prodIndex[k][n]] += d->prodCoeff[k][n];
            }
            j++;
            //store the rate constants
            if(d->reactSize[k]!=0){
              d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
            }else{
              d->rateConstant[k] = atof(is->fields[j]);
            }
          }
          k++;
        }
      }
    }
    jettison_inputstruct(is);
    d->currentTime = 0.0;
    d->MYINFINITY = -log(0.0);
    double randmaxRecip = (double)1/RAND_MAX;
    for(i=0;i<d->reactionSize;i++){
      indProp(d,p,i);
      d->putativeTimes[i] = -log(rand()*randmaxRecip)/p->prop[i];
      updateMinHeap(&d->minHeap,i,d->putativeTimes[i]);
    }
    getDependencies(d);
}

void updateSpecies(ESS* d){
    long i;
    for(i=0;i<d->reactSize[d->selectedReaction];i++){
      d->species[d->reactIndex[d->selectedReaction][i]] -=
        d->reactCoeff[d->selectedReaction][i];
    }
```

```
  for(i=0;i<d->prodSize[d->selectedReaction];i++){
    d->species[d->prodIndex[d->selectedReaction][i]] +=
      d->prodCoeff[d->selectedReaction][i];
  }

}

void printSpecies(ESS* d, double currentTime){
  long i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%d ", d->species[d->speciesToOutput[i]]);
  }
  printf("\n");
}

void destroyESS(ESS* d, Prop* p){
  free(p->prop);
  free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
  free(d->speciesToOutput);
  free(d->reactSize);
  free(d->prodSize);
  for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
    free(d->reactIndex[d->reactionSize]);
    free(d->reactCoeff[d->reactionSize]);
    free(d->prodIndex[d->reactionSize]);
    free(d->prodCoeff[d->reactionSize]);
  }
  free(d->reactIndex);
  free(d->reactCoeff);
  free(d->prodIndex);
  free(d->prodCoeff);
}

void step(ESS* d){
  d->selectedReaction = getMinMinHeap(&d->minHeap);
  d->reactionCounts[d->selectedReaction]++;
  d->currentTime = d->putativeTimes[d->selectedReaction];
}

void execute(ESS* d, Prop* p, double randmaxRecip){
  unsigned long i;
  unsigned long index;
  double oldPropensity;
  updateSpecies(d);
  for(i=0; i<d->numDependency[d->selectedReaction]; i++) {
    index = d->dependency[d->selectedReaction][i];
    oldPropensity = p->prop[index];
    indProp(d,p,index);
    if (index != d->selectedReaction) {
      if (oldPropensity == 0.0) {
        d->putativeTimes[index] = d->currentTime + -log(rand()*randmaxRecip)/p->prop[index];
      } else {
        d->putativeTimes[index] = d->currentTime + oldPropensity/p->prop[index] *
          (d->putativeTimes[index] - d->currentTime);
      }
      updateMinHeap(&d->minHeap, index, d->putativeTimes[index]);
    }
  }
  d->putativeTimes[d->selectedReaction] = d->currentTime + -log(rand()*randmaxRecip)/
    p->prop[d->selectedReaction];
  updateMinHeap(&d->minHeap, d->selectedReaction, d->putativeTimes[d->selectedReaction]);
}
```

```
int main(int argc, char* argv[]){
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions #Simulations\n");
    exit(1);
  }
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);
  ESS d;
  Prop p;
  long numReactions=atol(argv[2]);
  long numSimulations = atol(argv[3]);
  double randmaxRecip = (double)1/RAND_MAX;
  srand(100);

  //Read in the file and initialize everything
  Initialize(argv[1],&d,&p);
  long species[d.speciesSize];
  double putativeTimes[d.reactionSize], prop[d.reactionSize], ptotal;
  int i,j=numReactions,k;
  ptotal=p.total;
  memcpy(species,d.species,d.speciesSize*sizeof(long));
  memcpy(putativeTimes,d.putativeTimes,d.reactionSize*sizeof(double));
  memcpy(prop,p.prop,d.reactionSize*sizeof(double));
  for(i=0;i<numSimulations;i++){
    srand(i);
    memcpy(d.species,species,d.speciesSize*sizeof(long));
    memset(d.reactionCounts,0,d.reactionSize*sizeof(long));
    memcpy(d.putativeTimes,putativeTimes,d.reactionSize*sizeof(double));
    memcpy(p.prop,prop,d.reactionSize*sizeof(double));
    p.total=ptotal;
    for(k=0;k<d.reactionSize;k++){
      updateMinHeap(&d.minHeap,k,d.putativeTimes[k]);
    }
    d.currentTime=0.0;
    // printSpecies(&d,d.currentTime);
    for(numReactions=j;numReactions>0;numReactions--){
      //In fashion of Mike's code, step and execute
      step(&d);
      execute(&d,&p,randmaxRecip);
    }
  }
//  printSpecies(&d,d.currentTime);
  gettimeofday(&after,NULL);
  double totalTime = after.tv_sec - before.tv_sec;
  totalTime += ((double)after.tv_usec)/1000000.0;
  totalTime -= ((double)before.tv_usec)/1000000.0;

  printf("Total Time: %f\n", totalTime);

  destroyESS(&d, &p);

  return 0;
}
```

C.11 odirect.h

This is the header file for the serial Optimized Direct Method implementation.

```
#ifndef _ODIRECT_H_
#define _ODIRECT_H_

#define MAX(x,y) (x>y?x:y)
#define MIN(x,y) (x<y?x:y)

typedef struct {
  long speciesSize;
  long reactionSize;
  long numToBePrinted;
  long* netUpdate;
  long* species;
  double* rateConstant;
  long* speciesToOutput;
  long* reactSize;
  long** reactIndex;
  long** reactCoeff;
  long* prodSize;
  long** prodIndex;
  long** prodCoeff;
  long* numDependency;
  long** dependency;
  long* sortedIndex;
}ESS;

typedef struct{
  double* prop;
  double total;
  double* partial;
}Prop;


#endif
```

C.12 odirect.c

This is the source file for the serial Optimized Direct Method implementation.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include "odirect.h"
#include "fields.h"


long factorial(long x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

double indProp(ESS* d, long i){
  long j,k;
  long prop=1;
  //calculate the propensity for a reaction
  for(j=0; j<d->reactSize[i]; j++) {
    for(k=0; k<d->reactCoeff[i][j]; k++) {
      if (d->species[d->reactIndex[i][j]] - k == 0){
        return 0.0;
      }
      prop *= (d->species[d->reactIndex[i][j]] - k);
    }
  }

  return prop*d->rateConstant[i];
}

//Start Presimulation functions
long reactionSelectPre(Prop* p, long reactionSize, double randmaxRecip){
  long i;
  double random = rand()*randmaxRecip;
  double scaledRate = random*p->total;
  for(i=0;i<reactionSize;i++){
    scaledRate -= p->prop[i];
    if(scaledRate <= 0){
      return i;
    }
  }
  return i-1;
}

void calcPropPre(ESS* d, Prop* p){
  long i;
  p->total=0.0;
  for(i=0;i<d->reactionSize;i++){
    p->prop[i] = indProp(d,i);
    p->total+=p->prop[i];
  }
}

void updateSpeciesPre(ESS* d, long selectedReaction){
  long i;

  for(i=0;i<d->reactSize[selectedReaction];i++){
```

```
    d->species[d->reactIndex[selectedReaction][i]] -=
      d->reactCoeff[selectedReaction][i];
  }

  for(i=0;i<d->prodSize[selectedReaction];i++){
    d->species[d->prodIndex[selectedReaction][i]] +=
      d->prodCoeff[selectedReaction][i];
  }

}

void quickSort(long arr[], ESS* d, long left, long right) {
      long i = left, j = right;
      long tmp,* tempArr;
    double tempRate;
      long pivot = arr[(left + right) / 2];
      /* partition */
      while (i <= j) {
            while (arr[i] > pivot)
                  i++;
            while (arr[j] < pivot)
                  j--;
            if (i <= j) {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
          tmp = d->sortedIndex[i];
                  d->sortedIndex[i] = d->sortedIndex[j];
                  d->sortedIndex[j] = tmp;
                  i++;
                  j--;
            }
      };
      /* recursion */
      if (left < j)
            quickSort(arr, d, left, j);
      if (i < right)
            quickSort(arr, d, i, right);
}

void preSim(ESS* d, Prop* p, int numSim){
  int i;
  long selectedReaction;
  long initSpec[d->speciesSize];
  double randmaxRecip = (double)1/RAND_MAX;
  long pickReactions[d->reactionSize];
  memset(pickReactions, 0, d->reactionSize*sizeof(long));
  memcpy(initSpec, d->species, d->speciesSize*sizeof(long));
  for(i=0;i<numSim;i++){
    //calculate the propensities
    calcPropPre(d, p);
    //select the reaction
    selectedReaction = reactionSelectPre(p,d->reactionSize,randmaxRecip);
    pickReactions[selectedReaction]++;
    //update the species
    updateSpeciesPre(d,selectedReaction);
  }
  quickSort(pickReactions, d, 0, d->reactionSize-1);
  memcpy(d->species,initSpec, d->speciesSize*sizeof(long));
}

//End Presimulation functions

void getDependencies(ESS* d){
  long* temp[d->reactionSize];
```

```
  long i,j,k,l,m,count=0;
  d->numDependency = (long*)malloc(sizeof(long)*d->reactionSize);
  d->dependency = (long**)malloc(sizeof(long*)*d->reactionSize);
  for(i=0;i<d->reactionSize;i++){
    d->numDependency[i] = count;
    d->dependency[i] = (long*)malloc(sizeof(long)*d->reactionSize);
    for(j=0,count=0;j<d->speciesSize;j++){
      if(d->netUpdate[d->speciesSize*i+j]!=0){
        for(k=0;k<d->reactionSize;k++){
          for(l=0;l<d->reactSize[k];l++){
            if(d->reactIndex[k][l]==j){
              for(m=0;m<count;m++){
                if(d->dependency[i][count]==k) break;
              }
              if(m==count){
                d->dependency[i][count] = k;
                count++;
              }
            }
          }
        }
      }
    }
    d->numDependency[i] = count;
    d->dependency[i] = realloc(d->dependency[i],count*sizeof(long));
  }
}

void Initialize(char* inputFile, ESS* d, Prop* p, int numSim){
  long i=0,j,k=0,l, m=-1,n=0;
  long maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (long*)malloc(sizeof(long)*d->speciesSize);
        if(d->species==NULL){
          perror("In initialize, d->species");
          exit(1);
        }
        m=0;
      }else if(i==1){
        //store the reaction size
        d->reactionSize = atol(is->fields[0]);
        d->netUpdate = (long*)malloc(sizeof(long)*d->speciesSize*d->reactionSize);
        if(d->netUpdate==NULL){
          perror("In initialize, d->netUpdate");
          exit(1);
        }
        d->rateConstant = (double*)malloc(sizeof(double)*d->reactionSize);
        if(d->rateConstant==NULL){
          perror("In initialize, d->rateConstant");
          exit(1);
        }
        //allocate memory for the reactants
        d->reactSize = (long*)malloc(sizeof(long)*d->reactionSize);
        if(d->reactSize==NULL){
          perror("In initialize, d->reactSize");
          exit(1);
```

```
    }
    d->reactCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->reactCoeff==NULL){
      perror("In initialize, d->reactCoeff");
      exit(1);
    }
    d->reactIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->reactIndex==NULL){
      perror("In initialize, d->reactIndex");
      exit(1);
    }
    //allocate memory for the products
    d->prodSize = (long*)malloc(sizeof(long)*d->reactionSize);
    if(d->prodSize==NULL){
      perror("In initialize, d->prodSize");
      exit(1);
    }
    d->prodCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodCoeff==NULL){
      perror("In initialize, d->prodCoeff");
      exit(1);
    }
    d->prodIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodIndex==NULL){
      perror("In initialize, d->prodIndex");
      exit(1);
    }
    d->sortedIndex = (long*)malloc(sizeof(long)*d->reactionSize);
    if(d->sortedIndex==NULL){
      perror("In initialize, d->sortedIndex");
      exit(1);
    }
    //allocate memory for the propensities
    p->prop = (double*)malloc(sizeof(double)*d->reactionSize);
    if(p->prop==NULL){
      perror("In initialize, p->prop");
      exit(1);
    }
    memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(long));
  }else if(i==2){
    //store the number to be printed
    d->numToBePrinted = atol(is->fields[0]);
    d->speciesToOutput = (long*)malloc(sizeof(long)*d->numToBePrinted);
    if(d->speciesToOutput==NULL){
      perror("In initialize, d->speciesToOutput");
      exit(1);
    }
    m= 2;
  }else{
    fprintf(stderr,"Malformed File!!! Try again\n");
    exit(1);
  }
  i++;
}else{
  if(m==0){
    //Store the inital species populations
    for(j=0;j<is->NF;j++){
      d->species[j] = atol(is->fields[j]);
    }
    m++;
  }else{
    if(k==d->reactionSize){
      //store the indices to be printed
      for(j=0;j<d->numToBePrinted;j++){
        d->speciesToOutput[j] = atol(is->fields[j]);
```

```
            }
          }else{
            maxCoeff=0;
            d->reactSize[k] = atol(is->fields[0]);
            d->sortedIndex[k] = k;
            //store the reactants
            d->reactCoeff[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
            if(d->reactCoeff[k]==NULL){
              perror("In initialize, d->reactCoeff[k]");
              exit(1);
            }
            d->reactIndex[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
            if(d->reactIndex[k]==NULL){
              perror("In initialize, d->reactIndex[k]");
              exit(1);
            }
            for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
              d->reactCoeff[k][l] = atol(is->fields[j+1]);
              maxCoeff = MAX(maxCoeff,d->reactCoeff[k][l]);
              d->reactIndex[k][l] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->reactIndex[k][l]] -= d->reactCoeff[k][l];

            }
            j++;
            d->prodSize[k] = atol(is->fields[j]);
            d->prodCoeff[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodCoeff[k]==NULL){
              perror("In initialize, d->prodCoeff[k]");
              exit(1);
            }
            d->prodIndex[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodIndex[k]==NULL){
              perror("In initialize, d->prodIndex[k]");
              exit(1);
            }
            //store the products
            for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
              d->prodCoeff[k][n] = atol(is->fields[j+1]);
              d->prodIndex[k][n] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->prodIndex[k][n]] += d->prodCoeff[k][n];
            }
            j++;
            //store the rate constants
            if(d->reactSize[k]!=0){
              d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
            }else{
              d->rateConstant[k] = atof(is->fields[j]);
            }
          }
          k++;
        }
      }
    }
    jettison_inputstruct(is);
    preSim(d,p,numSim);
    calcPropPre(d,p);
    getDependencies(d);
}

long reactionSelect(ESS* d, Prop* p, long reactionSize, double randmaxRecip){
    long i;
    double random = rand()*randmaxRecip;
    double scaledRate = random*p->total;
    for(i=0;i<reactionSize;i++){
      scaledRate -= p->prop[d->sortedIndex[i]];
```

```
    if(scaledRate <= 0){
      return d->sortedIndex[i];
    }
  }
  return i-1;
}

void updateSpecies(ESS* d, Prop* p, long selectedReaction){
  long i;
  for(i=0;i<d->reactSize[selectedReaction];i++){
    d->species[d->reactIndex[selectedReaction][i]] -= d->reactCoeff[selectedReaction][i];
  }

  for(i=0;i<d->prodSize[selectedReaction];i++){
    d->species[d->prodIndex[selectedReaction][i]] += d->prodCoeff[selectedReaction][i];
  }

  for(i=0;i<d->numDependency[selectedReaction];i++){
    p->total -= p->prop[d->dependency[selectedReaction][i]];
    p->prop[d->dependency[selectedReaction][i]] = indProp(d,
    d->dependency[selectedReaction][i]);
    p->total += p->prop[d->dependency[selectedReaction][i]];
  }
}

void printSpecies(ESS* d, double currentTime){
  long i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%d ", d->species[d->speciesToOutput[i]]);
  }
  printf("\n");
}

void destroyESS(ESS* d, Prop* p){
  free(p->prop);
  free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
  free(d->speciesToOutput);
  free(d->reactSize);
  free(d->prodSize);
  for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
    free(d->reactIndex[d->reactionSize]);
    free(d->reactCoeff[d->reactionSize]);
    free(d->prodIndex[d->reactionSize]);
    free(d->prodCoeff[d->reactionSize]);
  }
  free(d->reactIndex);
  free(d->reactCoeff);
  free(d->prodIndex);
  free(d->prodCoeff);
}

int main(int argc, char* argv[]){
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions #Simulations\n");
    // fprintf(stderr, "\toutput - 1 to output the species, 0 to suppress outputs\n");
    exit(1);
  }
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);

  ESS d;
```

```
  Prop p;
  long numReactions=atol(argv[2]);
  long numSimulations = atol(argv[3]);
  // int output = atol(argv[3]);
  long selectedReaction;
  double randmaxRecip = (double)1/RAND_MAX;
  srand(0);

  double currentTime=0.0;
  //Read in the file and initialize everything
  Initialize(argv[1],&d,&p,numSimulations);
  long species[d.speciesSize], sortedIndex[d.reactionSize];
  double prop[d.reactionSize], ptotal;
  ptotal=p.total;
  int i,j=numReactions,k,l;
  memcpy(species,d.species,d.speciesSize*sizeof(long));
  memcpy(sortedIndex,d.sortedIndex,d.reactionSize*sizeof(long));
  memcpy(prop,p.prop,d.reactionSize*sizeof(double));
  for(i=0;i<numSimulations;i++){
    srand(i);
    memcpy(d.species,species,d.speciesSize*sizeof(long));
    memcpy(d.sortedIndex,sortedIndex,d.reactionSize*sizeof(long));
    memcpy(p.prop,prop,d.reactionSize*sizeof(double));
    p.total=ptotal;
    currentTime=0.0;
    // while(currentTime <= .5){
    for(numReactions=j;numReactions>=0;numReactions--){
      //select the reaction
      selectedReaction = reactionSelect(&d,&p,d.reactionSize,randmaxRecip);
      //calculate the current time
      currentTime += -log(rand()*randmaxRecip)/p.total;
      //update the species
      updateSpecies(&d, &p, selectedReaction);
    }
  }
//  printSpecies(&d,currentTime);
  gettimeofday(&after,NULL);
  double totalTime = after.tv_sec - before.tv_sec;
  totalTime += ((double)after.tv_usec)/1000000.0;
  totalTime -= ((double)before.tv_usec)/1000000.0;

  printf("Total Time: %f\n", totalTime);

  destroyESS(&d, &p);

  return 0;
}
```

C.13 sortingDirect.h

This is the header file for the serial Sorting Direct Method implementation.

```
#ifndef _SORTING_H_
#define _SORTING_H_

#define MAX(x,y) (x>y?x:y)

typedef struct {
  long speciesSize;
  long reactionSize;
  long numToBePrinted;
  long reactionIndex;
  long* netUpdate;
  long* species;
  double* rateConstant;
  long* speciesToOutput;
  long* reactSize;
  long** reactIndex;
  long** reactCoeff;
  long* prodSize;
  long** prodIndex;
  long** prodCoeff;
  long* numDependency;
  long** dependency;
  long* sortedIndex;
  long* reactionList;
}ESS;

typedef struct{
  double* prop;
  double total;
}Prop;


#endif
```

C.14 sortingDirect.c

This is the source file for the serial Sorting Direct Method implementation.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include "sortingDirect.h"
#include "fields.h"


long factorial(long x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

double indProp(ESS* d, long i){
  long j,k;
  long prop=1;
  //calculate the propensity for a reaction
  for(j=0; j<d->reactSize[i]; j++) {
    for(k=0; k<d->reactCoeff[i][j]; k++) {
      if (d->species[d->reactIndex[i][j]] - k == 0){
        return 0.0;
      }
      prop *= (d->species[d->reactIndex[i][j]] - k);
    }
  }

  return prop*d->rateConstant[i];
}

//Start Presimulation functions
long reactionSelectPre(Prop* p, long reactionSize, double randmaxRecip){
  long i;
  double random = rand()*randmaxRecip;
  double scaledRate = random*p->total;
  for(i=0;i<reactionSize;i++){
    scaledRate -= p->prop[i];
    if(scaledRate <= 0){
      return i;
    }
  }
  return i-1;
}

void calcPropPre(ESS* d, Prop* p){
  long i;
  // printf("in here\n");
  p->total=0.0;
  for(i=0;i<d->reactionSize;i++){
    p->prop[i] = indProp(d,i);
    // printf("%f\n", p->prop[i]);
    p->total+=p->prop[i];
  }
}

void updateSpeciesPre(ESS* d, long selectedReaction){
  long i;
```

```
  for(i=0;i<d->reactSize[selectedReaction];i++){
    d->species[d->reactIndex[selectedReaction][i]] -= d->reactCoeff[selectedReaction][i];
  }

  for(i=0;i<d->prodSize[selectedReaction];i++){
    d->species[d->prodIndex[selectedReaction][i]] += d->prodCoeff[selectedReaction][i];
  }

}

void quickSort(long arr[], ESS* d, long left, long right) {
      long i = left, j = right;
      long tmp,* tempArr;
    double tempRate;
      long pivot = arr[(left + right) / 2];
      /* partition */
      while (i <= j) {
            while (arr[i] > pivot)
                  i++;
            while (arr[j] < pivot)
                  j--;
            if (i <= j) {
                  tmp = arr[i];
                  arr[i] = arr[j];
                  arr[j] = tmp;
          tmp = d->sortedIndex[i];
                  d->sortedIndex[i] = d->sortedIndex[j];
                  d->sortedIndex[j] = tmp;
                  i++;
                  j--;
            }
      };
      /* recursion */
      if (left < j)
            quickSort(arr, d, left, j);
      if (i < right)
            quickSort(arr, d, i, right);
}

void preSim(ESS* d, Prop* p, int numSim){
  int i;
  long selectedReaction;
  long initSpec[d->speciesSize];
  double randmaxRecip = (double)1/RAND_MAX;
  long pickReactions[d->reactionSize];
  memset(pickReactions, 0, d->reactionSize*sizeof(long));
  memcpy(initSpec, d->species, d->speciesSize*sizeof(long));
  for(i=0;i<numSim;i++){
    //calculate the propensities
    calcPropPre(d, p);
    //select the reaction
    selectedReaction = reactionSelectPre(p,d->reactionSize,randmaxRecip);
    pickReactions[selectedReaction]++;
    //update the species
    updateSpeciesPre(d,selectedReaction);
  }
  quickSort(pickReactions, d, 0, d->reactionSize-1);
  memcpy(d->species,initSpec, d->speciesSize*sizeof(long));
}

//End Presimulation functions

void getDependencies(ESS* d){
  long* temp[d->reactionSize];
```

```
  long i,j,k,l,m,count=0;
  d->numDependency = (long*)malloc(sizeof(long)*d->reactionSize);
  d->dependency = (long**)malloc(sizeof(long*)*d->reactionSize);
  for(i=0;i<d->reactionSize;i++){
    d->numDependency[i] = count;
    d->dependency[i] = (long*)malloc(sizeof(long)*d->reactionSize);
    for(j=0,count=0;j<d->speciesSize;j++){
      if(d->netUpdate[d->speciesSize*i+j]!=0){
        for(k=0;k<d->reactionSize;k++){
          for(l=0;l<d->reactSize[k];l++){
            if(d->reactIndex[k][l]==j){
              for(m=0;m<count;m++){
                if(d->dependency[i][count]==k) break;
              }
              if(m==count){
                d->dependency[i][count] = k;
                count++;
              }
            }
          }
        }
      }
    }
    d->numDependency[i] = count;
    d->dependency[i] = realloc(d->dependency[i],count*sizeof(long));
  }
}

void Initialize(char* inputFile, ESS* d, Prop* p, int numSim){
  long i=0,j,k=0,l, m=-1,n=0;
  long maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (long*)malloc(sizeof(long)*d->speciesSize);
        if(d->species==NULL){
          perror("In initialize, d->species");
          exit(1);
        }
        m=0;
      }else if(i==1){
        //store the reaction size
        d->reactionSize = atol(is->fields[0]);
        d->netUpdate = (long*)malloc(sizeof(long)*d->speciesSize*d->reactionSize);
        if(d->netUpdate==NULL){
          perror("In initialize, d->netUpdate");
          exit(1);
        }
        d->rateConstant = (double*)malloc(sizeof(double)*d->reactionSize);
        if(d->rateConstant==NULL){
          perror("In initialize, d->rateConstant");
          exit(1);
        }
        //allocate memory for the reactants
        d->reactSize = (long*)malloc(sizeof(long)*d->reactionSize);
        if(d->reactSize==NULL){
          perror("In initialize, d->reactSize");
          exit(1);
```

```
    }
    d->reactCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->reactCoeff==NULL){
      perror("In initialize, d->reactCoeff");
      exit(1);
    }
    d->reactIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->reactIndex==NULL){
      perror("In initialize, d->reactIndex");
      exit(1);
    }
    //allocate memory for the products
    d->prodSize = (long*)malloc(sizeof(long)*d->reactionSize);
    if(d->prodSize==NULL){
      perror("In initialize, d->prodSize");
      exit(1);
    }
    d->prodCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodCoeff==NULL){
      perror("In initialize, d->prodCoeff");
      exit(1);
    }
    d->prodIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodIndex==NULL){
      perror("In initialize, d->prodIndex");
      exit(1);
    }
    d->sortedIndex = (long*)malloc(sizeof(long)*d->reactionSize);
    if(d->sortedIndex==NULL){
      perror("In initialize, d->sortedIndex");
      exit(1);
    }
    //allocate memory for the propensities
    p->prop = (double*)malloc(sizeof(double)*d->reactionSize);
    if(p->prop==NULL){
      perror("In initialize, p->prop");
      exit(1);
    }
    d->reactionList = (long*)malloc(sizeof(long)*d->reactionSize);
    if(d->reactionList==NULL){
      perror("In initialize, d->reactionList");
      exit(1);
    }
    memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(long));
  }else if(i==2){
    //store the number to be printed
    d->numToBePrinted = atol(is->fields[0]);
    d->speciesToOutput = (long*)malloc(sizeof(long)*d->numToBePrinted);
    if(d->speciesToOutput==NULL){
      perror("In initialize, d->speciesToOutput");
      exit(1);
    }
    m= 2;
  }else{
    fprintf(stderr,"Malformed File!!! Try again\n");
    exit(1);
  }
  i++;
}else{
  if(m==0){
    //Store the inital species populations
    for(j=0;j<is->NF;j++){
      d->species[j] = atol(is->fields[j]);
    }
    m++;
```

```
    }else{
      if(k==d->reactionSize){
        //store the indices to be printed
        for(j=0;j<d->numToBePrinted;j++){
          d->speciesToOutput[j] = atol(is->fields[j]);
        }
      }else{
        maxCoeff=0;
        d->reactSize[k] = atol(is->fields[0]);
        d->sortedIndex[k] = k;
        //store the reactants
        d->reactCoeff[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
        if(d->reactCoeff[k]==NULL){
          perror("In initialize, d->reactCoeff[k]");
          exit(1);
        }
        d->reactIndex[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
        if(d->reactIndex[k]==NULL){
          perror("In initialize, d->reactIndex[k]");
          exit(1);
        }
        for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
          d->reactCoeff[k][l] = atol(is->fields[j+1]);
          maxCoeff = MAX(maxCoeff,d->reactCoeff[k][l]);
          d->reactIndex[k][l] = atol(is->fields[j+2]);
          d->netUpdate[k*d->speciesSize+d->reactIndex[k][l]] -= d->reactCoeff[k][l];

        }
        j++;
        d->prodSize[k] = atol(is->fields[j]);
        d->prodCoeff[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
        if(d->prodCoeff[k]==NULL){
          perror("In initialize, d->prodCoeff[k]");
          exit(1);
        }
        d->prodIndex[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
        if(d->prodIndex[k]==NULL){
          perror("In initialize, d->prodIndex[k]");
          exit(1);
        }
        //store the products
        for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
          d->prodCoeff[k][n] = atol(is->fields[j+1]);
          d->prodIndex[k][n] = atol(is->fields[j+2]);
          d->netUpdate[k*d->speciesSize+d->prodIndex[k][n]] += d->prodCoeff[k][n];
        }
        j++;
        //store the rate constants
        if(d->reactSize[k]!=0){
          d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
        }else{
          d->rateConstant[k] = atof(is->fields[j]);
        }
      }
      k++;
    }
  }
}
jettison_inputstruct(is);
// preSim(d,p,numSim);
calcPropPre(d,p);
getDependencies(d);
for(i=0;i<d->reactionSize;i++){
  d->reactionList[i] = i;
}
```

```
}

long reactionSelect(ESS* d, Prop* p, long reactionSize, double randmaxRecip){
  long i;
  double random = rand()*randmaxRecip;
  double scaledRate = random*p->total;
  for(i=0;i<reactionSize;i++){
    d->reactionIndex = d->reactionList[i];
    scaledRate -= p->prop[d->reactionIndex];
    if(scaledRate <= 0){
      break;
    }
  }
  if(i!=0){
    long temp = d->reactionList[i];
    d->reactionList[i] = d->reactionList[i-1];
    d->reactionList[i-1] = temp;
  }
  return d->reactionIndex;
}

void updateSpecies(ESS* d, Prop* p, long selectedReaction){
  long i;
  for(i=0;i<d->reactSize[selectedReaction];i++){
    d->species[d->reactIndex[selectedReaction][i]] -= d->reactCoeff[selectedReaction][i];
  }

  for(i=0;i<d->prodSize[selectedReaction];i++){
    d->species[d->prodIndex[selectedReaction][i]] += d->prodCoeff[selectedReaction][i];
  }

  for(i=0;i<d->numDependency[selectedReaction];i++){
    p->total -= p->prop[d->dependency[selectedReaction][i]];
    p->prop[d->dependency[selectedReaction][i]] = indProp(d,
    d->dependency[selectedReaction][i]);
    p->total += p->prop[d->dependency[selectedReaction][i]];
  }

}

void printSpecies(ESS* d, double currentTime){
  long i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%d ", d->species[d->speciesToOutput[i]]);
  }
  printf("\n");
}

void destroyESS(ESS* d, Prop* p){
  free(p->prop);
  free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
  free(d->speciesToOutput);
  free(d->reactSize);
  free(d->prodSize);
  for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
    free(d->reactIndex[d->reactionSize]);
    free(d->reactCoeff[d->reactionSize]);
    free(d->prodIndex[d->reactionSize]);
    free(d->prodCoeff[d->reactionSize]);
  }
  free(d->reactIndex);
  free(d->reactCoeff);
```

```
  free(d->prodIndex);
  free(d->prodCoeff);
}


int main(int argc, char* argv[]){
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions #Simulations\n");
    // fprintf(stderr, "\toutput - 1 to output the species, 0 to suppress outputs\n");
    exit(1);
  }
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);

  ESS d;
  Prop p;
  long numReactions=atol(argv[2]);
  long numSimulations = atol(argv[3]);
  // int output = atol(argv[3]);
  long selectedReaction;
  double randmaxRecip = (double)1/RAND_MAX;
  srand(0);

  double currentTime=0.0;
  int i,j=numReactions,k,l;
  //Read in the file and initialize everything
  Initialize(argv[1],&d,&p, numSimulations);
  //backup original data for other simulations
  long species[d.speciesSize], reactionList[d.reactionSize];
  double prop[d.reactionSize], ptotal;
  ptotal=p.total;
  memcpy(species,d.species,d.speciesSize*sizeof(long));
  memcpy(reactionList,d.reactionList,d.reactionSize*sizeof(long));
  memcpy(prop,p.prop,d.reactionSize*sizeof(double));
  for(i=0;i<numSimulations;i++){
    srand(i);
    memcpy(d.species,species,d.speciesSize*sizeof(long));
    memcpy(p.prop,prop,d.reactionSize*sizeof(double));
    memcpy(d.reactionList,reactionList,d.reactionSize*sizeof(long));
    p.total = ptotal;
    currentTime=0.0;
    // while(currentTime <= .5){
    for(numReactions=j;numReactions>=0;numReactions--){
      //select the reaction
      selectedReaction = reactionSelect(&d,&p,d.reactionSize,randmaxRecip);
      //calculate the current time
      currentTime += -log(rand()*randmaxRecip)/p.total;
      //update the species
      updateSpecies(&d, &p, selectedReaction);
    }
  }
//  printSpecies(&d,currentTime);
  gettimeofday(&after,NULL);
  double totalTime = after.tv_sec - before.tv_sec;
  totalTime += ((double)after.tv_usec)/1000000.0;
  totalTime -= ((double)before.tv_usec)/1000000.0;

  printf("Total Time: %f\n", totalTime);

  destroyESS(&d, &p);

  return 0;
}
```

C.15 logDirect.h

This is the header file for the serial Logarithmic Direct Method implementation.

```
#ifndef _LOGDIRECT_H_
#define _LOGDIRECT_H_

#define MAX(x,y) (x>y?x:y)
#define MIN(x,y) (x<y?x:y)

typedef struct {
  long speciesSize;
  long reactionSize;
  long numToBePrinted;
  long* netUpdate;
  long* species;
  double* rateConstant;
  long* speciesToOutput;
  long* reactSize;
  long** reactIndex;
  long** reactCoeff;
  long* prodSize;
  long** prodIndex;
  long** prodCoeff;
  long* numDependency;
  long** dependency;
}ESS;

typedef struct{
  double* prop;
  double total;
  double* partial;
}Prop;


#endif
```

C.16 logDirect.c

This is the source file for the serial Logarithmic Direct Method implementation.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include "odirect.h"
#include "fields.h"


long factorial(long x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

double indProp(ESS* d, long i){
  long j,k;
  long prop=1;
  //calculate the propensity for a reaction
  for(j=0; j<d->reactSize[i]; j++) {
    for(k=0; k<d->reactCoeff[i][j]; k++) {
      if (d->species[d->reactIndex[i][j]] - k == 0){
        return 0.0;
      }
      prop *= (d->species[d->reactIndex[i][j]] - k);
    }
  }

  return prop*d->rateConstant[i];
}

void calcPropPre(ESS* d, Prop* p){
  long i;
  p->total=0.0;
  for(i=0;i<d->reactionSize;i++){
    p->prop[i] = indProp(d,i);
    p->total+=p->prop[i];
  }
}

void getDependencies(ESS* d){
  long* temp[d->reactionSize];
  long i,j,k,l,m,count=0;
  d->numDependency = (long*)malloc(sizeof(long)*d->reactionSize);
  d->dependency = (long**)malloc(sizeof(long*)*d->reactionSize);
  for(i=0;i<d->reactionSize;i++){
    d->numDependency[i] = count;
    d->dependency[i] = (long*)malloc(sizeof(long)*d->reactionSize);
    for(j=0,count=0;j<d->speciesSize;j++){
      if(d->netUpdate[d->speciesSize*i+j]!=0){
  for(k=0;k<d->reactionSize;k++){
    for(l=0;l<d->reactSize[k];l++){
      if(d->reactIndex[k][l]==j){
        for(m=0;m<count;m++){
    if(d->dependency[i][count]==k) break;
        }
        if(m==count){
    d->dependency[i][count] = k;
```

```
      count++;
          }
        }
      }
    }
        }
      }
      d->numDependency[i] = count;
      d->dependency[i] = realloc(d->dependency[i],count*sizeof(long));
    }
}

void Initialize(char* inputFile, ESS* d, Prop* p, int numSim){
  long i=0,j,k=0,l, m=-1,n=0;
  long maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (long*)malloc(sizeof(long)*d->speciesSize);
        if(d->species==NULL){
          perror("In initialize, d->species");
          exit(1);
        }
        m=0;
      }else if(i==1){
        //store the reaction size
        d->reactionSize = atol(is->fields[0]);
        d->netUpdate = (long*)malloc(sizeof(long)*d->speciesSize*d->reactionSize);
        if(d->netUpdate==NULL){
          perror("In initialize, d->netUpdate");
          exit(1);
        }
        d->rateConstant = (double*)malloc(sizeof(double)*d->reactionSize);
        if(d->rateConstant==NULL){
          perror("In initialize, d->rateConstant");
          exit(1);
        }
        //allocate memory for the reactants
        d->reactSize = (long*)malloc(sizeof(long)*d->reactionSize);
        if(d->reactSize==NULL){
          perror("In initialize, d->reactSize");
          exit(1);
        }
        d->reactCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
        if(d->reactCoeff==NULL){
          perror("In initialize, d->reactCoeff");
          exit(1);
        }
        d->reactIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
        if(d->reactIndex==NULL){
          perror("In initialize, d->reactIndex");
          exit(1);
        }
        //allocate memory for the products
        d->prodSize = (long*)malloc(sizeof(long)*d->reactionSize);
        if(d->prodSize==NULL){
          perror("In initialize, d->prodSize");
          exit(1);
```

```
    }
    d->prodCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodCoeff==NULL){
      perror("In initialize, d->prodCoeff");
      exit(1);
    }
    d->prodIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodIndex==NULL){
      perror("In initialize, d->prodIndex");
      exit(1);
    }
    d->sortedIndex = (long*)malloc(sizeof(long)*d->reactionSize);
    if(d->sortedIndex==NULL){
      perror("In initialize, d->sortedIndex");
      exit(1);
    }
    //allocate memory for the propensities
    p->prop = (double*)malloc(sizeof(double)*d->reactionSize);
    if(p->prop==NULL){
      perror("In initialize, p->prop");
      exit(1);
    }
    memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(long));
    p->partial = (double*)malloc(sizeof(double)*d->reactionSize);
    if(p->partial==NULL){
      perror("In initialize, p->partial");
      exit(1);
    }
  }else if(i==2){
    //store the number to be printed
    d->numToBePrinted = atol(is->fields[0]);
    d->speciesToOutput = (long*)malloc(sizeof(long)*d->numToBePrinted);
    if(d->speciesToOutput==NULL){
      perror("In initialize, d->speciesToOutput");
      exit(1);
    }
    m= 2;
  }else{
    fprintf(stderr,"Malformed File!!! Try again\n");
    exit(1);
  }
  i++;
}else{
  if(m==0){
    //Store the inital species populations
    for(j=0;j<is->NF;j++){
      d->species[j] = atol(is->fields[j]);
    }
    m++;
  }else{
    if(k==d->reactionSize){
      //store the indices to be printed
      for(j=0;j<d->numToBePrinted;j++){
        d->speciesToOutput[j] = atol(is->fields[j]);
      }
    }else{
      maxCoeff=0;
      d->reactSize[k] = atol(is->fields[0]);
      d->sortedIndex[k] = k;
      //store the reactants
      d->reactCoeff[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
      if(d->reactCoeff[k]==NULL){
        perror("In initialize, d->reactCoeff[k]");
        exit(1);
      }
```

```
            d->reactIndex[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
            if(d->reactIndex[k]==NULL){
              perror("In initialize, d->reactIndex[k]");
              exit(1);
            }
            for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
              d->reactCoeff[k][l] = atol(is->fields[j+1]);
              maxCoeff = MAX(maxCoeff,d->reactCoeff[k][l]);
              d->reactIndex[k][l] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->reactIndex[k][l]] -= d->reactCoeff[k][l];

            }
            j++;
            d->prodSize[k] = atol(is->fields[j]);
            d->prodCoeff[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodCoeff[k]==NULL){
              perror("In initialize, d->prodCoeff[k]");
              exit(1);
            }
            d->prodIndex[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodIndex[k]==NULL){
              perror("In initialize, d->prodIndex[k]");
              exit(1);
            }
            //store the products
            for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
              d->prodCoeff[k][n] = atol(is->fields[j+1]);
              d->prodIndex[k][n] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->prodIndex[k][n]] += d->prodCoeff[k][n];
            }
            j++;
            //store the rate constants
            if(d->reactSize[k]!=0){
              d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
            }else{
              d->rateConstant[k] = atof(is->fields[j]);
            }
          }
          k++;
        }
      }
    }
    jettison_inputstruct(is);
    calcPropPre(d,p);
    p->partial[0]=p->prop[d->sortedIndex[0]];
    for(i=1;i<d->reactionSize;i++){
      p->partial[i] = p->partial[i-1] + p->prop[i];
    }
    getDependencies(d);
}

int binarySearch(int sortedArray[], int first, int last, int key) {
   while (first <= last) {
        int mid = (first + last) >> 1;  // compute mid point.
        if (key > sortedArray[mid])
            first = mid + 1;  // repeat search in top half.
        else if (key < sortedArray[mid])
            last = mid - 1; // repeat search in bottom half.
        else
            return mid;      // found it. return position /////
   }
   return -(first + 1);     // failed to find key
}

long reactionSelect(ESS* d, Prop* p, long reactionSize, double randmaxRecip){
```

```
  long i;
  double random = rand()*randmaxRecip;
  double scaledRate = random*p->total;
  long low=0,high=reactionSize-1,middle;
for(i=0;i<reactionSize;i++){
  if(scaledRate <= p->partial[i]){
    return i;
  }
}
return reactionSize-1;
  // while (low <= high) {
  //    middle = low + ((high - low)>>1);
  //    if (scaledRate < p->partial[middle] && scaledRate < p->partial[middle-1])
  //      high = middle - 1;
  //    else if (scaledRate > p->partial[middle])
  //      low = middle + 1;
  //    else
  //      return middle;
  // }
  // return middle;
}

void updateSpecies(ESS* d, Prop* p, long selectedReaction){
  long i;
  long minimum = 1e6;
  for(i=0;i<d->reactSize[selectedReaction];i++){
    d->species[d->reactIndex[selectedReaction][i]] -= d->reactCoeff[selectedReaction][i];
  }

  for(i=0;i<d->prodSize[selectedReaction];i++){
    d->species[d->prodIndex[selectedReaction][i]] += d->prodCoeff[selectedReaction][i];
  }

  for(i=0;i<d->numDependency[selectedReaction];i++){
    p->prop[d->dependency[selectedReaction][i]] = indProp(d,
    d->dependency[selectedReaction][i]);
    minimum = MIN(minimum,d->dependency[selectedReaction][i]);
  }

  // if(minimum==0){
    i=1;
    p->partial[0] = p->prop[0];
  // }else{
  //    i=minimum;
  // }
  for(;i<d->reactionSize;i++){
    p->partial[i] = p->prop[i] + p->partial[i-1];
  }
  p->total = p->partial[d->reactionSize-1];
}

void printSpecies(ESS* d, double currentTime){
  long i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%d ", d->species[d->speciesToOutput[i]]);
  }
  printf("\n");
}

void destroyESS(ESS* d, Prop* p){
  free(p->prop);
  free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
```

```
  free(d->speciesToOutput);
  free(d->reactSize);
  free(d->prodSize);
  for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
    free(d->reactIndex[d->reactionSize]);
    free(d->reactCoeff[d->reactionSize]);
    free(d->prodIndex[d->reactionSize]);
    free(d->prodCoeff[d->reactionSize]);
  }
  free(d->reactIndex);
  free(d->reactCoeff);
  free(d->prodIndex);
  free(d->prodCoeff);
}

int main(int argc, char* argv[]){
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions #Simulations\n");
    // fprintf(stderr, "\toutput - 1 to output the species, 0 to suppress outputs\n");
    exit(1);
  }
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);

  ESS d;
  Prop p;
  long numReactions=atol(argv[2]);
  long numSimulations = atol(argv[3]);
  // int output = atol(argv[3]);
  long selectedReaction;
  double randmaxRecip = (double)1/RAND_MAX;
  srand(0);

  double currentTime=0.0;
  //Read in the file and initialize everything
  Initialize(argv[1],&d,&p,numSimulations);
  long species[d.speciesSize];
  double prop[d.reactionSize];
  double propPartial[d.reactionSize], ptotal;
  ptotal=p.total;
  int i,j=numReactions,k,l;
  memcpy(species,d.species,d.speciesSize*sizeof(long));
  memcpy(prop,p.prop,d.reactionSize*sizeof(double));
  memcpy(propPartial,p.partial,d.reactionSize*sizeof(double));
  for(i=0;i<numSimulations;i++){
    srand(i);
    memcpy(d.species,species,d.speciesSize*sizeof(long));
    memcpy(p.prop,prop,d.reactionSize*sizeof(double));
    memcpy(p.partial,propPartial,d.reactionSize*sizeof(double));
    p.total=ptotal;
    currentTime=0.0;
    // while(currentTime <= .5){
    for(numReactions=j;numReactions>=0;numReactions--){
      //select the reaction
      selectedReaction = reactionSelect(&d,&p,d.reactionSize,randmaxRecip);
      //calculate the current time
      currentTime += -log(rand()*randmaxRecip)/p.total;
      //update the species
      updateSpecies(&d, &p, selectedReaction);
    }
  }
//  printSpecies(&d,currentTime);
  gettimeofday(&after,NULL);
  double totalTime = after.tv_sec - before.tv_sec;
```

```
    totalTime += ((double)after.tv_usec)/1000000.0;
    totalTime -= ((double)before.tv_usec)/1000000.0;

    printf("Total Time: %f\n", totalTime);

    destroyESS(&d, &p);

    return 0;
}
```

C.17 directSSE.h

This is the header file for the SSE implementation.

```
#ifndef _DIRECT_H_
#define _DIRECT_H_


#define MAX(x,y) (x>y?x:y)
#define SIM 4

typedef struct {
  int speciesSize;
  int reactionSize;
  int numToBePrinted;
  int* netUpdate;
  float** species;
  float* rateConstant;
  int selectedReaction[SIM];
  int* speciesToOutput;
  int* reactSize;
  int** reactIndex;
  int** reactCoeff;
  int* prodSize;
  int** prodIndex;
  int** prodCoeff;
}ESS;

typedef struct{
  float** prop;
  float total[SIM] __attribute__((aligned(16)));
}Prop;

#endif
```

C.18 directSSE.c

This is the source file for the SSE implementation.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include <xmmintrin.h>
#include "directSSE.h"
#include "fields.h"

__m128 a, b, c, e, total, randMaxRecipVec, negOne,curTime;

int factorial(int x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

void Initialize(char* inputFile, ESS* d, Prop* p){
  int i=0,j,k=0,l, m=-1,n=0;
  int maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (float**)malloc(sizeof(float*)*d->speciesSize);
        for(i=0;i<d->speciesSize;i++){
          d->species[i] = (float*)malloc(sizeof(float)*SIM);
        }
        i=0;
        if(d->species==NULL){
          perror("In initialize, d->species");
          exit(1);
        }
        m=0;
      }else if(i==1){
        //store the reaction size
        d->reactionSize = atol(is->fields[0]);
        d->netUpdate = (int*)malloc(sizeof(int)*d->speciesSize*d->reactionSize);
        if(d->netUpdate==NULL){
          perror("In initialize, d->netUpdate");
          exit(1);
        }
        d->rateConstant = (float*)malloc(sizeof(float)*d->reactionSize);
        if(d->rateConstant==NULL){
          perror("In initialize, d->rateConstant");
          exit(1);
        }
        //allocate memory for the reactants
        d->reactSize = (int*)malloc(sizeof(int)*d->reactionSize);
        if(d->reactSize==NULL){
          perror("In initialize, d->reactSize");
          exit(1);
```

```
    }
    d->reactCoeff = (int**)malloc(sizeof(int*)*d->reactionSize);
    if(d->reactCoeff==NULL){
      perror("In initialize, d->reactCoeff");
      exit(1);
    }
    d->reactIndex = (int**)malloc(sizeof(int*)*d->reactionSize);
    if(d->reactIndex==NULL){
      perror("In initialize, d->reactIndex");
      exit(1);
    }
    //allocate memory for the products
    d->prodSize = (int*)malloc(sizeof(int)*d->reactionSize);
    if(d->prodSize==NULL){
      perror("In initialize, d->prodSize");
      exit(1);
    }
    d->prodCoeff = (int**)malloc(sizeof(int*)*d->reactionSize);
    if(d->prodCoeff==NULL){
      perror("In initialize, d->prodCoeff");
      exit(1);
    }
    d->prodIndex = (int**)malloc(sizeof(int*)*d->reactionSize);
    if(d->prodIndex==NULL){
      perror("In initialize, d->prodIndex");
      exit(1);
    }
    //allocate memory for the propensities
    p->prop = (float**)malloc(sizeof(float*)*d->reactionSize);
    for(i=0;i<d->reactionSize;i++){
      p->prop[i] = (float*)malloc(sizeof(float)*SIM);
    }
    // p->total = (float*)malloc(sizeof(float)*SIM);
    i=1;
    if(p->prop==NULL){
      perror("In initialize, p->prop");
      exit(1);
    }
    memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(int));
  }else if(i==2){
    //store the number to be printed
    d->numToBePrinted = atol(is->fields[0]);
    d->speciesToOutput = (int*)malloc(sizeof(int)*d->numToBePrinted);
    if(d->speciesToOutput==NULL){
      perror("In initialize, d->speciesToOutput");
      exit(1);
    }
    m = 2;
  }else{
    fprintf(stderr,"Malformed File!!! Try again\n");
    exit(1);
  }
  i++;
}else{
  if(m==0){
    //Store the inital species populations
    for(j=0;j<is->NF;j++){
      for(m=0;m<SIM;m++){
        d->species[j][m] = atol(is->fields[j]);
      }
    }
    m++;
  }else{
    if(k==d->reactionSize){
      //store the indices to be printed
```

```
            for(j=0;j<d->numToBePrinted;j++){
              d->speciesToOutput[j] = atol(is->fields[j]);
            }
          }else{
            maxCoeff=0;
            d->reactSize[k] = atol(is->fields[0]);

            //store the reactants
            d->reactCoeff[k] = (int*)malloc(sizeof(int)*d->reactSize[k]);
            if(d->reactCoeff[k]==NULL){
              perror("In initialize, d->reactCoeff[k]");
              exit(1);
            }
            d->reactIndex[k] = (int*)malloc(sizeof(int)*d->reactSize[k]);
            if(d->reactIndex[k]==NULL){
              perror("In initialize, d->reactIndex[k]");
              exit(1);
            }
            for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
              d->reactCoeff[k][l] = atol(is->fields[j+1]);
              maxCoeff = MAX(maxCoeff,d->reactCoeff[k][l]);
              d->reactIndex[k][l] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->reactIndex[k][l]] -= d->reactCoeff[k][l];

            }
            j++;
            d->prodSize[k] = atol(is->fields[j]);
            d->prodCoeff[k] = (int*)malloc(sizeof(int)*d->prodSize[k]);
            if(d->prodCoeff[k]==NULL){
              perror("In initialize, d->prodCoeff[k]");
              exit(1);
            }
            d->prodIndex[k] = (int*)malloc(sizeof(int)*d->prodSize[k]);
            if(d->prodIndex[k]==NULL){
              perror("In initialize, d->prodIndex[k]");
              exit(1);
            }
            //store the products
            for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
              d->prodCoeff[k][n] = atol(is->fields[j+1]);
              d->prodIndex[k][n] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->prodIndex[k][n]] += d->prodCoeff[k][n];
            }
            j++;
            //store the rate constants
            if(d->reactSize[k]!=0){
              d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
            }else{
              d->rateConstant[k] = atof(is->fields[j]);
            }
          }
          k++;
        }
      }
    }
    jettison_inputstruct(is);
}

void indProp(ESS* d, Prop* p, int i){
  int j;
  float k;
  float prop[SIM];
  a = _mm_load1_ps(&d->rateConstant[i]);
  //calculate the propensity for a reaction for 4 simulations
  for(j=0; j<d->reactSize[i]; j++) {
```

```
    for(k=0; k<d->reactCoeff[i][j]; k++) {
      b = _mm_loadu_ps(d->species[d->reactIndex[i][j]]);
      c = _mm_load1_ps(&k);
      e = _mm_sub_ps(b,c);
      a = _mm_mul_ps(a,e);
    }
  }
  _mm_storeu_ps(p->prop[i],a);
}

void calcProp(ESS* d, Prop* p){
  int i;
  float temp = 0;
  total = _mm_load1_ps(&temp);
  //calculate all the propensities
  for(i=0;i<d->reactionSize;i++){
    indProp(d,p,i);
    total = _mm_add_ps(total, a);
  }
  _mm_store_ps(p->total,total);
}

void reactionSelect(ESS* d, Prop* p, int reactionSize, float randmaxRecip){
  int i,j;
  float temp[SIM] __attribute__((aligned(16)));
  for(i=0;i<SIM;i++){
    temp[i] = rand()*randmaxRecip;
  }
  float scaledRate[SIM] __attribute__((aligned(16)));
  a = _mm_load_ps(temp);
  b = _mm_load_ps(p->total);
  _mm_store_ps(temp,b);
  c = _mm_mul_ps(a,b);
  _mm_store_ps(scaledRate,c);
  for(j=0;j<SIM;j++){
    for(i=0;i<reactionSize;i++){
      scaledRate[j] -= p->prop[i][j];
      if(scaledRate[j] <= 0){
        d->selectedReaction[j] = i;
        break;
      }
    }
    if(i==reactionSize){
      d->selectedReaction[j] = i-1;
    }
  }
}

void updateSpecies(ESS* d){
  int i,j;
  for(j=0;j<SIM;j++){
    // if(j==0){
    //   printf("%d\n", d->selectedReaction[j]);
    // }
    for(i=0;i<d->reactSize[d->selectedReaction[j]];i++){
      d->species[d->reactIndex[d->selectedReaction[j]][i]][j] -=
        d->reactCoeff[d->selectedReaction[j]][i];
    }

    for(i=0;i<d->prodSize[d->selectedReaction[j]];i++){
      d->species[d->prodIndex[d->selectedReaction[j]][i]][j] +=
        d->prodCoeff[d->selectedReaction[j]][i];
    }
  }
}
```

```
void printSpecies(ESS* d, float currentTime, int simulation){
  int i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%.0f ", d->species[d->speciesToOutput[i]][simulation]);
  }
  printf("\n");
}

void destroyESS(ESS* d, Prop* p){
  free(p->prop);
  free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
  free(d->speciesToOutput);
  free(d->reactSize);
  free(d->prodSize);
  for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
    free(d->reactIndex[d->reactionSize]);
    free(d->reactCoeff[d->reactionSize]);
    free(d->prodIndex[d->reactionSize]);
    free(d->prodCoeff[d->reactionSize]);
  }
  free(d->reactIndex);
  free(d->reactCoeff);
  free(d->prodIndex);
  free(d->prodCoeff);
}

int main(int argc, char* argv[]){
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions #Simulations\n");
    // fprintf(stderr, "\toutput - 1 to output the species, 0 to suppress outputs\n");
    exit(1);
  }
  int numSim = atoi(argv[3]);
  if(numSim%4!=0){
    fprintf(stderr, "Number of Simulations must be divisible by 4!\n");
    exit(1);
  }else{
    numSim /= 4;
  }
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);

  ESS d;
  Prop p;
  int numReactions=atol(argv[2]);
  // int output = atol(argv[3]);
  int selectedReaction;
  float randmaxRecip = (float)1/RAND_MAX;
  float negonetemp = -1;
  randMaxRecipVec = _mm_load1_ps(&randmaxRecip);
  negOne = _mm_load1_ps(&negonetemp);
  negonetemp = 0;
  curTime = _mm_load1_ps(&negonetemp);
  srand(0);

  float currentTime=0.0;
  //Read in the file and initialize everything
  Initialize(argv[1],&d,&p);
  int i,j=numReactions,k=0,l;
  // float species[d.speciesSize][SIM];
```

```
  float** species;
  species = (float**)malloc(sizeof(float*)*d.speciesSize);
  for(i=0;i<d.speciesSize;i++){
    species[i] = (float*)malloc(sizeof(float)*SIM);
  }
  printSpecies(&d,currentTime,0);
  float temp[SIM];

  for(k=0,i=0;k<SIM;k++){
    srand(i);
    for(l=0;l<d.speciesSize;l++){
      species[l][k] = d.species[l][k];
    }
  }

  for(i=0;i<numSim;i++){
    for(k=0;k<SIM;k++){
      for(l=0;l<d.speciesSize;l++){
        d.species[l][k] = species[l][k];
      }
    }

    memset(temp,0,SIM*sizeof(float));
    currentTime =0;
//    currentTime=0.0;
    for(numReactions=j;numReactions>=0;numReactions--){
      // if(output == 1){
      //   printSpecies(&d,currentTime);
      // }
      //calculate the propensities
      calcProp(&d, &p);
      //select the reaction
      reactionSelect(&d, &p,d.reactionSize,randmaxRecip);
      //calculate the current time
      for(k=0;k<SIM;k++){
         temp[k] += -log(rand()*randmaxRecip)/p.total[k];
      }

      //update the species
      updateSpecies(&d);

    }
  }

  printSpecies(&d,currentTime,0);
  gettimeofday(&after,NULL);
  float totalTime = after.tv_sec - before.tv_sec;
  totalTime += ((float)after.tv_usec)/1000000.0;
  totalTime -= ((float)before.tv_usec)/1000000.0;

  printf("Total Time: %f\n", totalTime);

  destroyESS(&d, &p);

  return 0;
}
```

C.19 directMPI.c

This is the source file for the MPI Direct Method implementation.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include <mpi.h>
#include "direct.h"
#include "fields.h"

long factorial(long x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

void Initialize(char* inputFile, ESS* d, Prop* p){
  int i=0,j,k=0,l, m=-1,n=0;
  int maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (long*)malloc(sizeof(long)*d->speciesSize);
        if(d->species==NULL){
          perror("In initialize, d->species");
          exit(1);
        }
        m=0;
      }else if(i==1){
        //store the reaction size
        d->reactionSize = atol(is->fields[0]);
        d->netUpdate = (long*)malloc(sizeof(long)*d->speciesSize*d->reactionSize);
        if(d->netUpdate==NULL){
          perror("In initialize, d->netUpdate");
          exit(1);
        }
        d->rateConstant = (double*)malloc(sizeof(double)*d->reactionSize);
        if(d->rateConstant==NULL){
          perror("In initialize, d->rateConstant");
          exit(1);
        }
        //allocate memory for the reactants
        d->reactSize = (long*)malloc(sizeof(long)*d->reactionSize);
        if(d->reactSize==NULL){
          perror("In initialize, d->reactSize");
          exit(1);
        }
        d->reactCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
        if(d->reactCoeff==NULL){
          perror("In initialize, d->reactCoeff");
          exit(1);
        }
```

```
    d->reactIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->reactIndex==NULL){
      perror("In initialize, d->reactIndex");
      exit(1);
    }
    //allocate memory for the products
    d->prodSize = (long*)malloc(sizeof(long)*d->reactionSize);
    if(d->prodSize==NULL){
      perror("In initialize, d->prodSize");
      exit(1);
    }
    d->prodCoeff = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodCoeff==NULL){
      perror("In initialize, d->prodCoeff");
      exit(1);
    }
    d->prodIndex = (long**)malloc(sizeof(long*)*d->reactionSize);
    if(d->prodIndex==NULL){
      perror("In initialize, d->prodIndex");
      exit(1);
    }
    //allocate memory for the propensities
    p->prop = (double*)malloc(sizeof(double)*d->reactionSize);
    if(p->prop==NULL){
      perror("In initialize, p->prop");
      exit(1);
    }
    memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(long));
  }else if(i==2){
    //store the number to be printed
    d->numToBePrinted = atol(is->fields[0]);
    d->speciesToOutput = (long*)malloc(sizeof(long)*d->numToBePrinted);
    if(d->speciesToOutput==NULL){
      perror("In initialize, d->speciesToOutput");
      exit(1);
    }
    m= 2;
  }else{
    fprintf(stderr,"Malformed File!!! Try again\n");
    exit(1);
  }
  i++;
}else{
  if(m==0){
    //Store the inital species populations
    for(j=0;j<is->NF;j++){
      d->species[j] = atol(is->fields[j]);
    }
    m++;
  }else{
    if(k==d->reactionSize){
      //store the indices to be printed
      for(j=0;j<d->numToBePrinted;j++){
        d->speciesToOutput[j] = atol(is->fields[j]);
      }
    }else{
      maxCoeff=0;
      d->reactSize[k] = atol(is->fields[0]);

      //store the reactants
      d->reactCoeff[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
      if(d->reactCoeff[k]==NULL){
        perror("In initialize, d->reactCoeff[k]");
        exit(1);
      }
```

```
            d->reactIndex[k] = (long*)malloc(sizeof(long)*d->reactSize[k]);
            if(d->reactIndex[k]==NULL){
              perror("In initialize, d->reactIndex[k]");
              exit(1);
            }
            for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
              d->reactCoeff[k][l] = atol(is->fields[j+1]);
              maxCoeff = MAX(maxCoeff,d->reactCoeff[k][l]);
              d->reactIndex[k][l] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->reactIndex[k][l]] -=
                d->reactCoeff[k][l];

            }
            j++;
            d->prodSize[k] = atol(is->fields[j]);
            d->prodCoeff[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodCoeff[k]==NULL){
              perror("In initialize, d->prodCoeff[k]");
              exit(1);
            }
            d->prodIndex[k] = (long*)malloc(sizeof(long)*d->prodSize[k]);
            if(d->prodIndex[k]==NULL){
              perror("In initialize, d->prodIndex[k]");
              exit(1);
            }
            //store the products
            for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
              d->prodCoeff[k][n] = atol(is->fields[j+1]);
              d->prodIndex[k][n] = atol(is->fields[j+2]);
              d->netUpdate[k*d->speciesSize+d->prodIndex[k][n]] +=
                d->prodCoeff[k][n];
            }
            j++;
            //store the rate constants
            if(d->reactSize[k]!=0){
              d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
            }else{
              d->rateConstant[k] = atof(is->fields[j]);
            }
          }
          k++;
        }
      }
    }
    jettison_inputstruct(is);
}

double indProp(ESS* d, long i){
  long j,k;
  long prop=1;
  //calculate the propensity for a reaction
  for(j=0; j<d->reactSize[i]; j++) {
    for(k=0; k<d->reactCoeff[i][j]; k++) {
      if (d->species[d->reactIndex[i][j]] - k == 0){
        return 0.0;
      }
      prop *= (d->species[d->reactIndex[i][j]] - k);
    }
  }

  return prop*d->rateConstant[i];
}

void calcProp(ESS* d, Prop* p){
  long i;
```

```
  p->total=0.0;
  //calculate all the propensities
  for(i=0;i<d->reactionSize;i++){
    p->prop[i] = indProp(d,i);
    p->total+=p->prop[i];
  }
  // printf("%f\n",p->total);
}

long reactionSelect(Prop* p, long reactionSize, double randmaxRecip){
  long i;
  double scaledRate = rand()*randmaxRecip*p->total;
  for(i=0;i<reactionSize;i++){
    scaledRate -= p->prop[i];
    if(scaledRate <= 0){
      return i;
    }
  }
  return i-1;
}

void updateSpecies(ESS* d, long selectedReaction){
  long i;

  for(i=0;i<d->reactSize[selectedReaction];i++){
    d->species[d->reactIndex[selectedReaction][i]] -=
      d->reactCoeff[selectedReaction][i];
  }

  for(i=0;i<d->prodSize[selectedReaction];i++){
    d->species[d->prodIndex[selectedReaction][i]] +=
      d->prodCoeff[selectedReaction][i];
  }

}

void printSpecies(ESS* d, double currentTime){
  long i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%d ", d->species[d->speciesToOutput[i]]);
  }
  printf("\n");
}

void destroyESS(ESS* d, Prop* p){
  free(p->prop);
  free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
  free(d->speciesToOutput);
  free(d->reactSize);
  free(d->prodSize);
  for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
    free(d->reactIndex[d->reactionSize]);
    free(d->reactCoeff[d->reactionSize]);
    free(d->prodIndex[d->reactionSize]);
    free(d->prodCoeff[d->reactionSize]);
  }
  free(d->reactIndex);
  free(d->reactCoeff);
  free(d->prodIndex);
  free(d->prodCoeff);
}
```

```
int main(int argc, char* argv[]){
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions #Simulations\n");
    // fprintf(stderr, "\toutput - 1 to output the species, 0 to suppress outputs\n");
    exit(1);
  }

  int rank, size;
  MPI_Init (&argc, &argv);  /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);  /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);  /* get number of processes */
  //  printf("Hello from rank %d\n", rank);
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);
  //  if(rank!=0){
    //    gettimeofday(&before,NULL);

    ESS d;
    Prop p;
    long numReactions=atol(argv[2]);
    long numSimulations = atol(argv[3])/(size);
    // int output = atol(argv[3]);
    long selectedReaction;
    double randmaxRecip = (double)1/RAND_MAX;
    srand(rank);

    double currentTime=0.0;
    //Read in the file and initialize everything
    Initialize(argv[1],&d,&p);
//    printf("Rank %d Initialized!",rank);
    long species[d.speciesSize];
    int i,j=numReactions;


    memcpy(species,d.species,d.speciesSize*sizeof(long));
    for(i=0;i<numSimulations;i++){
      srand(i*rank);
      memcpy(d.species,species,d.speciesSize*sizeof(long));
      currentTime=0.0;
      for(numReactions=j;numReactions>=0;numReactions--){
        // if(output == 1){
        //   printSpecies(&d,currentTime);
        // }
        //calculate the propensities
        //MPI_Barrier(MPI_COMM_WORLD);
        calcProp(&d, &p);
        //select the reaction
        selectedReaction = reactionSelect(&p,d.reactionSize,randmaxRecip);
        // printf("%f\n", p.total);
        //calculate the current time
        currentTime += -log(rand()*randmaxRecip)/p.total;
        //update the species
        updateSpecies(&d,selectedReaction);
        // if(currentTime >= 10){
        //   break;
        // }
      }
    }
    //    printSpecies(&d,currentTime);
    gettimeofday(&after,NULL);
    double totalTime = after.tv_sec - before.tv_sec;
    totalTime += ((double)after.tv_usec)/1000000.0;
    totalTime -= ((double)before.tv_usec)/1000000.0;
    //  printf("Node %d: %f seconds\n",rank, totalTime);
```

```
    MPI_Send((void*)&totalTime, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    destroyESS(&d, &p);
    //  }else{
       if(rank==0){
    int i;
    double temp=0;
    //    gettimeofday(&before,NULL);
    //receive times from each slave
    for(i=1;i<size;i++){
      MPI_Recv((void*)&temp,1,MPI_DOUBLE,i,1,MPI_COMM_WORLD,NULL);
    }
    gettimeofday(&after,NULL);
    double totalTime = after.tv_sec - before.tv_sec;
    totalTime += ((double)after.tv_usec)/1000000.0;
    totalTime -= ((double)before.tv_usec)/1000000.0;
    //print out the total amount of time taken to receive everything
    //cout << "Total Time " << totalTime << endl;
    printf("%f\n", totalTime);
  }
  MPI_Finalize();
  return 0;
}
```

C.20 directCUDA.h

This is the header file for the CUDA implementation.

```
#ifndef _DIRECT_H_
#define _DIREC_H_


#define MAX(x,y) (x>y?x:y)

typedef struct {
  int speciesSize;
  int reactionSize;
  int reactionPow2up;
  int numToBePrinted;
  int* netUpdate;
  int* species;
  float* rateConstant;
  int* speciesToOutput;
  int* reactSize;
  int* reactIndex;
  int* reactCoeff;
  int* prodSize;
  int* prodIndex;
  int* prodCoeff;
}ESS;

typedef struct{
  float* prop;
  float total;
}Prop;

typedef struct {
  int speciesSize;
  int reactionSize;
  int numToBePrinted;
  int* netUpdate;
  int* species;
  float* rateConstant;
  int* speciesToOutput;
  int* reactSize;
  int* reactIndex;
  int* reactCoeff;
  int* prodSize;
  int* prodIndex;
  int* prodCoeff;
}ESScuda;

typedef struct{
  float* prop;
  float* total;
}Propcuda;

#endif
```

C.21 directCUDA.c

This is the source file for the SSE implementation.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <sys/time.h>
#include <time.h>
#include <cuda.h>
#include <cutil.h>
#include <cuda_runtime.h>
#include <cutil_inline.h>
#include "directCUDA.h"
#include "fields.h"
#include "sharedmem.cuh"
#include "MersenneTwister.h"
#include "MersenneTwister_kernel.cu"

#define FUNCVERSION(x, y) x ## _ ## y
#define XFUNCVERSION(x, y) FUNCVERSION(x, y)
#define FUNC(NAME) XFUNCVERSION(NAME, SMVERSION)
#ifdef __DEVICE_EMULATION__
#define EMUSYNC __syncthreads()
#else
#define EMUSYNC
#endif

#define MAXREACTIONS 256
#define MAXSPECIES 58
// #define NUMSIMS 32768
// #define NUMSIMPERBLOCK 128
// #define NUMBLOCKS NUMSIMS/NUMSIMPERBLOCK

__constant__ int reactSizec[MAXREACTIONS];
__constant__ int reactCoeffc[MAXREACTIONS*2];
__constant__ int reactIndexc[MAXREACTIONS*2];
__constant__ float rateConstantc[MAXREACTIONS];
__constant__ int netUpdatec[MAXSPECIES*MAXREACTIONS];

extern "C" int iDivUp(int a, int b){
    return ((a % b) != 0) ? (a / b + 1) : (a / b);
}

//floor(a / b)
extern "C" int iDivDown(int a, int b){
    return a / b;
}

//Align a to nearest higher multiple of b
extern "C" int iAlignUp(int a, int b){
    return ((a % b) != 0) ?  (a - a % b + b) : a;
}

//Align a to nearest lower multiple of b
extern "C" int iAlignDown(int a, int b){
    return a - a % b;
}


__global__ void calcPropKernel(int* species, int* prodSize, int* prodCoeff,
                int* prodIndex, float* prop, int speciesSize,
                int reactionSize, int pow2up){
  int i,k,j;
```

```
  int spec = (threadIdx.x+blockIdx.x*blockDim.x)*speciesSize;
  int index = (threadIdx.x+blockIdx.x*blockDim.x)*pow2up;
  int reaction = threadIdx.x*2;
  SharedMemory<int> smem;
  int* sprop = smem.getPointer();
  for(i=0;i<reactionSize;i++,index++){
    sprop[threadIdx.x] = 1;
    for(j=0; j<reactSizec[i]; j++) {
      for(k=0; k<reactCoeffc[i*2+j]; k++) {
        if (species[spec+reactIndexc[i*2+j]] - k == 0){
          sprop[threadIdx.x]=0;
          break;
        }
        sprop[threadIdx.x] *= (species[spec+reactIndexc[i*2+j]] - k);
      }
      if(k!=reactCoeffc[i*2+j]){
        break;
      }
    }
    prop[index] = sprop[threadIdx.x]*rateConstantc[i];
  }
}

__global__ void updateSpeciesKernel(int* species, int speciesSize,
                    unsigned int* selectedReaction){
  int i;
  int spec = (threadIdx.x+blockIdx.x*blockDim.x)*speciesSize;
  int index = (threadIdx.x+blockIdx.x*blockDim.x);
  for(i=0;i<speciesSize;i++){
    species[spec+i]+=netUpdatec[selectedReaction[index]*speciesSize+i];
  }
}

__global__ void
reactionSelectKernel(float* prop, float* sumProp, unsigned int reactionSize,
          unsigned int reactionPow2up, unsigned int* reactionIndex,
          float* random){
  int i;
  int reaction = (threadIdx.x+blockIdx.x*blockDim.x)*reactionPow2up;
  float temp = sumProp[reaction+reactionSize-1]*
        random[(threadIdx.x+blockIdx.x*blockDim.x)];
  SharedMemory<unsigned int> smem;
  unsigned int* rindex = smem.getPointer();

  for(i=0; i<reactionSize; i++) {
    if (sumProp[i+reaction] >= temp) {
      rindex[threadIdx.x]=i;
      break;
    }
  }

  reactionIndex[(threadIdx.x+blockIdx.x*blockDim.x)] = rindex[threadIdx.x];
}

__global__ void
currentTimeCalc(float* sumProp, unsigned int reactionPow2up,
          float* currentTimes, float* random, unsigned int reactionSize){
  unsigned int reaction = (threadIdx.x+blockIdx.x*blockDim.x);
  currentTimes[reaction] += -log(random[reaction])/
        sumProp[reaction*reactionPow2up+reactionSize-1];
}

bool isPow2(unsigned int x)
{
    return ((x&(x-1))==0);
```

```
}

unsigned int nextPow2( unsigned int x ) {
    --x;
    x |= x >> 1;
    x |= x >> 2;
    x |= x >> 4;
    x |= x >> 8;
    x |= x >> 16;
    return ++x;
}

__global__ void myReduce(float* in,float* sums, int reactionSize){
  if(threadIdx.x < reactionSize){
    SharedMemory<float> smem;
    float* s_in = smem.getPointer();
    int i = blockIdx.x*reactionSize + threadIdx.x;
    s_in[threadIdx.x] = in[i];
    for(unsigned int s = 1;s<reactionSize;s++){
      if(threadIdx.x == s){
        s_in[threadIdx.x] += s_in[threadIdx.x-1];
      }
    }
    sums[blockIdx.x*reactionSize+threadIdx.x] = s_in[threadIdx.x];
  }
}

__global__ void myReduce2(float* in,float* sums, int reactionSize){
  int i;
  int simulation = (threadIdx.x+blockIdx.x*blockDim.x)*reactionSize;
  float prev;
  prev = in[0];
  sums[0] = prev;
  for(i=1;i<reactionSize;i++){
    prev += in[simulation+i];
    sums[simulation+i] = prev;
  }
}

int factorial(int x){
  if(x==1){
    return 1;
  }else{
    return factorial(x-1)*x;
  }
}

void Initialize(char* inputFile, ESS* d, ESScuda* dc, Prop* p, Propcuda* pc,
      int NUMSIMS){
  int i=0,j,k=0,l, m=-1,n=0;
  int maxCoeff;
  IS is;
  is = new_inputstruct(inputFile);
  while(get_line(is) >= 0){
    //ignore blank lines
    if(is->NF == 0){
      continue;
    }else if(is->NF == 1 && m != 0 && m != 2){
      if(i==0){
        //store the species size
        d->speciesSize = atol(is->fields[0]);
        d->species = (int*)malloc(sizeof(int)*d->speciesSize);
        CUDA_SAFE_CALL( cudaMalloc((void**)&dc->species,
              sizeof(int)*d->speciesSize*NUMSIMS) );
        if(d->species==NULL){
```

```
    perror("In initialize, d->species");
    exit(1);
  }
  m=0;
}else if(i==1){
  //store the reaction size
  d->reactionSize = atol(is->fields[0]);
  d->reactionPow2up = nextPow2(d->reactionSize);
  d->netUpdate = (int*)malloc(sizeof(int)*d->speciesSize*d->reactionSize);
  if(d->netUpdate==NULL){
    perror("In initialize, d->netUpdate");
    exit(1);
  }
  d->rateConstant = (float*)malloc(sizeof(float)*d->reactionSize);
  if(d->rateConstant==NULL){
    perror("In initialize, d->rateConstant");
    exit(1);
  }
  //allocate memory for the reactants
  d->reactSize = (int*)malloc(sizeof(int)*d->reactionSize);
  if(d->reactSize==NULL){
    perror("In initialize, d->reactSize");
    exit(1);
  }
  d->reactCoeff = (int*)malloc(sizeof(int)*d->reactionSize*2);
  if(d->reactCoeff==NULL){
    perror("In initialize, d->reactCoeff");
    exit(1);
  }
  d->reactIndex = (int*)malloc(sizeof(int)*d->reactionSize*2);
  if(d->reactIndex==NULL){
    perror("In initialize, d->reactIndex");
    exit(1);
  }
  //allocate memory for the products
  d->prodSize = (int*)malloc(sizeof(int)*d->reactionSize);
  CUDA_SAFE_CALL( cudaMalloc((void**)&dc->prodSize,
      sizeof(int)*d->reactionSize) );
  if(d->prodSize==NULL){
    perror("In initialize, d->prodSize");
    exit(1);
  }
  d->prodCoeff = (int*)malloc(sizeof(int)*d->reactionSize*2);
  CUDA_SAFE_CALL( cudaMalloc((void**)&dc->prodCoeff,
      sizeof(int)*d->reactionSize*2) );
  if(d->prodCoeff==NULL){
    perror("In initialize, d->prodCoeff");
    exit(1);
  }
  d->prodIndex = (int*)malloc(sizeof(int)*d->reactionSize*2);
  CUDA_SAFE_CALL( cudaMalloc((void**)&dc->prodIndex,
      sizeof(int)*d->reactionSize*2) );
  if(d->prodIndex==NULL){
    perror("In initialize, d->prodIndex");
    exit(1);
  }
  //allocate memory for the propensities
  p->prop = (float*)malloc(sizeof(float)*d->reactionPow2up);
  CUDA_SAFE_CALL( cudaMalloc((void**)&pc->prop,
      sizeof(int)*d->reactionPow2up*NUMSIMS) );
  if(p->prop==NULL){
    perror("In initialize, p->prop");
    exit(1);
  }
  memset(d->netUpdate,0,d->speciesSize*d->reactionSize*sizeof(int));
```

```
      memset(p->prop,0,d->reactionPow2up*sizeof(float));
    }else if(i==2){
      //store the number to be printed
      d->numToBePrinted = atol(is->fields[0]);
      d->speciesToOutput = (int*)malloc(sizeof(int)*d->numToBePrinted);
      if(d->speciesToOutput==NULL){
        perror("In initialize, d->speciesToOutput");
        exit(1);
      }
      m= 2;
    }else{
      fprintf(stderr,"Malformed File!!! Try again\n");
      exit(1);
    }
    i++;
  }else{
    if(m==0){
      //Store the inital species populations
      for(j=0;j<is->NF;j++){
        d->species[j] = atol(is->fields[j]);
      }
      m++;
    }else{
      if(k==d->reactionSize){
        //store the indices to be printed
        for(j=0;j<d->numToBePrinted;j++){
          d->speciesToOutput[j] = atol(is->fields[j]);
        }
      }else{
        maxCoeff=0;
        d->reactSize[k] = atol(is->fields[0]);

        for(j=0,l=0;j<d->reactSize[k]*2;j+=2,l++){
          d->reactCoeff[k*2+l] = atol(is->fields[j+1]);
          maxCoeff = MAX(maxCoeff,d->reactCoeff[k*2+l]);
          d->reactIndex[k*2+l] = atol(is->fields[j+2]);
          d->netUpdate[k*d->speciesSize+d->reactIndex[k*2+l]] -=
            d->reactCoeff[k*2+l];

        }
        j++;
        d->prodSize[k] = atol(is->fields[j]);

        //store the products
        for(l=0,n=0;l<d->prodSize[k]*2;l+=2,j+=2,n++){
          d->prodCoeff[k*2+n] = atol(is->fields[j+1]);
          d->prodIndex[k*2+n] = atol(is->fields[j+2]);
          d->netUpdate[k*d->speciesSize+d->prodIndex[k*2+n]] +=
            d->prodCoeff[k*2+n];
        }
        j++;
        //store the rate constants
        if(d->reactSize[k]!=0){
          d->rateConstant[k] = atof(is->fields[j])/factorial(maxCoeff);
        }else{
          d->rateConstant[k] = atof(is->fields[j]);
        }
      }
      k++;
    }
  }
}


CUDA_SAFE_CALL(cudaMemcpyToSymbol(netUpdatec,d->netUpdate,
```

```
    d->speciesSize*d->reactionSize*sizeof(int),0,cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpyToSymbol(rateConstantc,d->rateConstant,
    d->reactionSize*sizeof(int),0,cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpyToSymbol(reactSizec,d->reactSize,
    d->reactionSize*sizeof(int),0,cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpyToSymbol(reactCoeffc,d->reactCoeff,
    d->reactionSize*2*sizeof(int),0,cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpyToSymbol(reactIndexc,d->reactIndex,
    d->reactionSize*2*sizeof(int),0,cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dc->prodSize,d->prodSize,
    d->reactionSize*sizeof(int),cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dc->prodCoeff,d->prodCoeff,
    d->reactionSize*2*sizeof(int),cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(dc->prodIndex,d->prodIndex,
    d->reactionSize*2*sizeof(int),cudaMemcpyHostToDevice));

  for(i=0;i<NUMSIMS;i++){
    CUDA_SAFE_CALL(cudaMemcpy(pc->prop+i*d->reactionPow2up,p->prop,
      d->reactionPow2up*sizeof(float),cudaMemcpyHostToDevice));
    CUDA_SAFE_CALL(cudaMemcpy(dc->species+i*d->speciesSize,d->species,
      d->speciesSize*sizeof(int),cudaMemcpyHostToDevice));
  }

  jettison_inputstruct(is);
}

int reactionSelect(Prop* p, int reactionSize, float randmaxRecip){
  int i;
  float scaledRate = rand()*randmaxRecip*p->total;
  for(i=0;i<reactionSize;i++){
    scaledRate -= p->prop[i];
    if(scaledRate <= 0){
      return i;
    }
  }
  return i-1;
}

void printSpecies(ESS* d, float currentTime){
  int i;
  printf("%f ",currentTime);
  for(i=0;i<d->numToBePrinted;i++){
    printf("%d ", d->species[d->speciesToOutput[i]]);
  }
  printf("\n");
}

void destroyESS(ESS* d, ESScuda* dc, Prop* p, Propcuda* pc){
  free(p->prop);
  // free(d->netUpdate);
  free(d->species);
  free(d->rateConstant);
  free(d->speciesToOutput);
  // free(d->reactSize);
  free(d->prodSize);
  // for(d->reactionSize--;d->reactionSize>=0;d->reactionSize--){
  //   free(d->reactIndex[d->reactionSize]);
  //   free(d->reactCoeff[d->reactionSize]);
  //   free(d->prodIndex[d->reactionSize]);
  //   free(d->prodCoeff[d->reactionSize]);
  // }
  // free(d->reactIndex);
  // free(d->reactCoeff);
  free(d->prodIndex);
  free(d->prodCoeff);
```

```
  cudaFree(dc->species);
  cudaFree(dc->prodSize);
  cudaFree(dc->prodIndex);
  cudaFree(dc->prodCoeff);
}

int main(int argc, char* argv[]){
  int NUMSIMPERBLOCK = 128;
  if(argc!=4){
    fprintf(stderr, "Usage: ./ess inputFile #Reactions NUMSIMS\n");
    // fprintf(stderr, "\toutput - 1 to output the species, 0 to suppress outputs\n");
    fprintf(stderr, "\t NUMSIMS must be divisible by %d\n", NUMSIMPERBLOCK);
    exit(1);
  }


  ESS d;
  ESScuda dc;
  Prop p;
  Propcuda pc;
  int NUMSIMS = atoi(argv[3]);
  // int NUMSIMPERBLOCK = 128;
  int NUMBLOCKS = NUMSIMS/NUMSIMPERBLOCK;
  int numReactions=atol(argv[2]);
  // int output = atol(argv[3]);
  unsigned int* selectedReaction;
  cudaMalloc((void**)&selectedReaction,sizeof(int)*NUMSIMS);
  float randmaxRecip = (float)1/RAND_MAX;
  srand(0);

  float currentTime=0.0;

  //Read in the file and initialize everything
  Initialize(argv[1],&d,&dc,&p,&pc, NUMSIMS);
  struct timeval before;
  struct timeval after;
  gettimeofday(&before,NULL);
  int seed = 77;
  int    PATH_N = NUMSIMS*2;
  int N_PER_RNG = iAlignUp(iDivUp(PATH_N, MT_RNG_COUNT), 2);
  int    RAND_N = MT_RNG_COUNT * N_PER_RNG;
  float* total;
  float* d_total;
  float* sums;
  float* random;
  float* currentTimes;
  const char *dat_path = "./data/MersenneTwister.dat";
  loadMTGPU(dat_path);
  // total = (float*)malloc(sizeof(float)*d.reactionSize);
  cudaMallocHost((void**)&total,sizeof(float)*NUMSIMS);
  cudaMalloc((void**)&d_total,sizeof(float)*NUMSIMS*d.reactionPow2up);
  cudaMalloc((void**)&sums,sizeof(float)*NUMSIMS*d.reactionPow2up);
  cudaMalloc((void**)&random,sizeof(float)*RAND_N);
  cudaMalloc((void**)&currentTimes,sizeof(float)*NUMSIMS);
  float r;
  int i;
  printSpecies(&d,currentTime);
  memset(total, 0, sizeof(float)*NUMSIMS);
  CUDA_SAFE_CALL(cudaMemcpy(currentTimes,total,
    sizeof(float)*NUMSIMS,cudaMemcpyHostToDevice));
  for(;numReactions>=0;numReactions--){

    //generate Random Numbers
    seedMTGPU(seed*numReactions);
    RandomGPU <<< 32, 128 >>> (random, N_PER_RNG);
```

```
    //calculate the propensities
    calcPropKernel <<< NUMBLOCKS, NUMSIMPERBLOCK, NUMSIMPERBLOCK*sizeof(float) >>>
  (dc.species,dc.prodSize,dc.prodCoeff,dc.prodIndex,pc.prop,
  d.speciesSize,d.reactionSize,d.reactionPow2up);

    myReduce <<< NUMSIMS, d.reactionPow2up, d.reactionPow2up*sizeof(float) >>>
      (pc.prop, sums, d.reactionPow2up);

    //select the reaction
    // r = rand()*randmaxRecip;
    reactionSelectKernel <<< NUMBLOCKS, NUMSIMPERBLOCK, NUMSIMPERBLOCK*sizeof(float) >>>
      (pc.prop, sums, d.reactionSize, d.reactionPow2up, selectedReaction, random);
    //calculate the current time
    currentTimeCalc <<< NUMBLOCKS, NUMSIMPERBLOCK >>>
      (sums, d.reactionPow2up, currentTimes, random+NUMSIMS, d.reactionSize);

    //update the species
    updateSpeciesKernel <<< NUMBLOCKS, NUMSIMPERBLOCK >>>
      (dc.species, d.speciesSize, selectedReaction);
  }
  CUDA_SAFE_CALL(cudaMemcpy(d.species,dc.species+(NUMSIMS-1)*d.speciesSize,
    sizeof(int)*d.speciesSize,cudaMemcpyDeviceToHost));
  CUDA_SAFE_CALL(cudaMemcpy(&currentTime,currentTimes+(NUMSIMS-1),
    sizeof(float),cudaMemcpyDeviceToHost));

  printSpecies(&d,currentTime);
  gettimeofday(&after,NULL);
  float totalTime = after.tv_sec - before.tv_sec;
  totalTime += ((float)after.tv_usec)/1000000.0;
  totalTime -= ((float)before.tv_usec)/1000000.0;

  printf("Total Time: %f\n", totalTime);

  destroyESS(&d, &dc, &p, &pc);
  cudaFree(d_total);
  cudaFree(sums);
  cudaFree(random);
  cudaFree(currentTimes);
  cudaFree(selectedReaction);
  return 0;
}
```

Vita

David Dewayne Jenkins was born in Cookeville, TN on 11 June 1986. He graduated from Cookeville High School in 2004 where he went on to the University of Tennessee in Knoxville, TN. He received his Bachelor of Science in Computer Engineering in May 2008. He continued his education at UTK where he is currently pursuing his Master of Science degree in Computer Engineering.