Accelerating the Stochastic Simulation Algorithm

David Jenkins, Gregory D. Peterson
Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, Tennessee, USA
[djenki11, gdp]@utk.edu

Abstract— **In order for scientists to learn more about molecular biology, it is imperative that they have the ability to construct accurate models that predict the reactions of species of molecules. Generating these models using deterministic approaches is not feasible as these models may violate some of the assumptions underlying classical differential equations models (e.g., small populations with discrete values). 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 on the species and reactions. However, the algorithm tends to be computationally heavy and leads to long simulation runtimes for large systems. In this paper, we provide an approach to running these simulations using MPI and NVIDIA graphics processing units using CUDA in order to parallelize these simulations, reducing the total amount of time needed for multiple simulations to run in a more reasonable time scale.**

I. Introduction

The advancement of biology and chemistry to study the molecular models and reactions of species molecules has spurred a growth of research in the area. Chemically reacting species can often be modeled in a deterministic manner using ordinary differential equations. Although one can accurately model large systems with differential equations, they are inaccurate when modeling small species populations and with expressing transient behaviors. Therefore, we use a stochastic approach to accurately analyze these systems.

Dan Gillespie formulated an approach called the stochastic simulation algorithm (SSA) to perform the modeling of these complex systems [1,2]. This SSA technique stochastically predicts the execution time and identity of each reaction for a spatially homogeneous chemical system. In order to get accurate statistical representations of the system, one must run many simulations using different initial seeds. In practice, this can be a very time consuming process due to the need for running simulations many times.

Prior research addressed how to speed up individual simulations by using different calculation methods [1-6]. However, even with improvements to SSA performing multiple simulations can still take a great of time, especially when the model size is large. Therefore, we explore the use of NVIDIA graphics processing units, using the CUDA programming language, to provide an approach to speeding up multiple simulations by running a large number of simulations in parallel. This is possible because each simulation is independent.

This work was partially supported by the National Science Foundation, grant NSF CHE-0625598.

II. Methodology

Gillespie's SSA approach takes an initial set of species populations and reactions, calculates the likelihood of each reaction, determines the time and type of the next reaction, and applies this reaction to update the species set. This is done until a certain end time or reaction limit has been reached. There are a number of methods that previous researchers have formulated to accelerate the SSA approach; for this project we use the Direct Method [1,2]. This method limits the amount of random numbers to be generated, as well as the number of expensive floating-point divisions.

As mentioned before, the stochastic simulation algorithm calculates the species population, $X(t) = \{X_1(t),X_2(t),...,X_M(t)\}$, at each time period until the end time, t_{end}, is reached. We being with a set of M species, $S = \{S_1,S_2,...,S_M\}$, with initial populations $X(t0) = \{X_1(t_0), X_2(t_0),..., X_M(t_0)\}$, and N chemical reactions, $R=\{R_1,R_2,...,R_N\}$. Since SSA stipulates that the system must be spatially homogenous, the algorithm is only concerned with the populations of the species X(t), ignoring the position of each molecule. Each reaction R_i is defined by a stochastic constant, k_i, and a stoichiometry represented by M reactant and product coefficients, $r = \{r_1,r_2,...,r_M\}$ and $p = \{p_1,p_2,...,p_M\}$. The probability that a given reaction, R_i, occurs in a given period of time, *dt*, is given by the following equation,

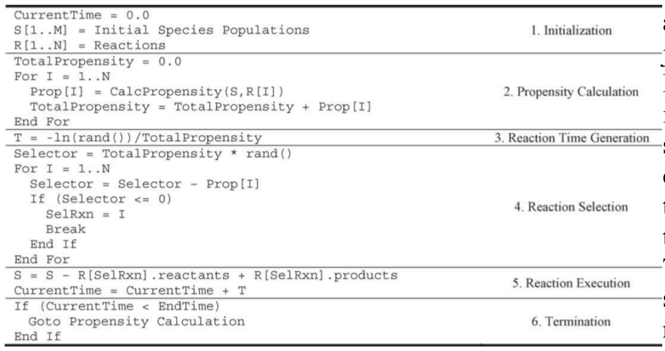
$$P_i(dt) = \alpha_i(X(t))\, dt + o(dt)$$

where α_i is a propensity function that describes the likelihood of a reaction occurring with 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 a dimerization example [5,6], an example reaction could be in the form of $S_4+S_4 \to S_5$ and the propensity would be calculated by

$$\alpha_i(X(t)) = k_i\, X_4(t)(X_4(t)-1)/2$$

Next, the occurrence time of the next reaction is found:

$\tau_{next} = -\ln(URN)/\alpha$ where $\alpha = \sum_{1}^{n} \alpha_i$ and URN is a uniformly distributed random number. The next reaction is selected with each reaction having a probability of (α_i/α). With the reaction type determined, species populations are updated accordingly along with propensity values. Below is the pseudo code for this algorithm [5]:



```
CurrentTime = 0.0                                          1. Initialization
S[1..M] = Initial Species Populations
R[1..N] = Reactions
TotalPropensity = 0.0                                      2. Propensity Calculation
For I = 1..N
  Prop[I] = CalcPropensity(S,R[I])
  TotalPropensity = TotalPropensity + Prop[I]
End For
T = -ln(rand())/TotalPropensity                            3. Reaction Time Generation
Selector = TotalPropensity * rand()                        4. Reaction Selection
For I = 1..N
  Selector = Selector - Prop[I]
  If (Selector <= 0)
    SelRxn = I
    Break
  End If
End For
S = S - R[SelRxn].reactants + R[SelRxn].products           5. Reaction Execution
CurrentTime = CurrentTime + T
If (CurrentTime < EndTime)                                 6. Termination
  Goto Propensity Calculation
End If
```

III. Reconfigururable computing architecture

In recent years, graphics processing units (GPUs) have become extremely popular in the parallel computing community, due to their efficient parallel pipelines and floating point performance. For this project, we use NVIDIA Tesla C870 GPUs with 1.5GB RAM and 16KB of shared memory. In order to code a program onto these GPUs, NVIDIA has created its own language, compute unified device architecture (CUDA). This language is built onto C in order to easily incorporate into existing programs.

Implementing SSA on the GPU turns out to be quite difficult. This is because individual SSA simulations are not easily parallelized due to data dependencies. However, multiple simulation runs are needed when trying to use the algorithm to get a statistically sound model of the chemical system, so the GPU runs multiple SSA simulations at a time as below.

First, random number generation is offloaded to the GPU in order to generate the large number of random numbers for all the simulations. 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 difficult to implement, we use the Mersenne twister algorithm implemented in the NVIDIA SDK.

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.

Next, the third kernel performs a reduction on the array of propensities for each simulation. This reduction finds the sum of all the propensities for each simulation and reduces the array down to one propensity per simulation. The fourth kernel determines the index of the next reaction, assuming each reaction has a probability of (α_i/α).

Fifth, a kernel calculates the time of the next reaction. Finally, the sixth kernel updates the species populations. After these kernels have executed, control is returned back to the CPU to get the species counts.

IV. Results

After implementing SSA using CUDA, a parallel implementation was also explored using MPI. Once again, the target of the parallelized version was to speed up the amount of time for multiple simulation runs, as opposed to just one simulation.

Each of the slaves split up the simulations to do. For instance, if there were 16 slaves and a total 8192 simulations, each slave would do 8192/16=512 simulations. Once a slave has completed all of its simulations, it sends a packet indicating the completion of the simulations to the master process with the rank of 0.

To explore performance, we use a dimerization model with 8 species and 13 reactions between them and a quorum sensing model for *Vibrio fischeri* [5] with 122 species and 201 reactions.

All three implementations were implemented in C++ and compiled with the maximum optimization flags using the GNU C++ compiler and, in the case of the CUDA implementation, the NVIDIA CUDA compiler. We use one to four GPUs to examine the performance improvements. The MPI and serial implementations were executed on the *battlecat* cluster comprised of Intel Core 2 Duo 2.13 GHz CPUs with 2MB cache and 2GB RAM. The MPI code was run on 16 processors.

For each implementation, multiple numbers of simulations are run to get a trend of the speedup compared to the serial version. These numbers of simulations vary by powers of two from 1024 up to 65536. For each increment in the number of simulations, we complete 10 runs in order to get an accurate average simulation runtime.

To validate the CUDA and MPI implementations, their outputs are compared to the C++ implementation for each reaction of each simulation. Since these are stochastic processes, each implementation starts with the same seed for the random number generator as the one used in the C++ program. Then, during each step of the simulation, the species population is compared to the output of the C++ program.

Notice that as the number of simulations doubles, the time approximately doubles, as expected, for each implementation. Figures 1 and 2 contain graphs of these values. It can be seen from this graph that both the CUDA and MPI implementations outperformed the serial C++ code. Also, the runtimes of the programs are linearly related to the number of simulations run.

The CUDA implementation on one GPU performed approximately 4 times faster than the serial CPU code for larger numbers of simulations. This is contributed to the massive parallelism offered by the GPU to allow for thousands of simulations to run at one time. When implemented on multiple GPUs, this average speed up over the serial code is approximately 12 times faster with larger numbers of simulations. With smaller numbers of simulations, the overhead of forking off multiple threads to initialize and run four GPUs becomes the limiting factor of achieving a greater speed up with the current implementation. The MPI implementation speedup was expected as well due to the linear fashion of the C++ serial code as the number of simulations increase. On average, this MPI code ran 14 times faster than the serial CPU code once the MPI overheads were in taken into considerations.

Next, the quorum sensing model was used in the simulations to view the performance using a larger model. Figure 2 shows the runtimes and speedups using this model. Its performance was nearly as high as with the dimerization model.

V. Conclusions

In this paper, we explore the potential for using GPUs with CUDA to accelerate the stochastic simulation algorithm for chemical species. Because of data dependencies, each simulation is not targeted for acceleration, but rather the parallel simulation of a collection of simulations to provide good statistical analysis. We find that the GPU provides modest acceleration for the SSA algorithm while maintaining the exact statistical properties of the chemical master equation. Grid-based processing is appropriate for SSA simulations, with nearly perfect speedup achieved due to the independence of simulations. Future work will explore optimizations of the mapping on the GPUs as well as FPGA implementations of the SSA method.

References

[1] 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.
[2] D. T. Gillespie, "Exact Stochastic Simulation of Coupled Chemical Reactions", *J. Phys. Chem*., vol. 81, no. 25, pp. 2340-2361, 1977.
[3] D. T. Gillespie, "Approximate Accelerated Stochastic Simulation of Chemically Reacting Systems", *J. Chem. Phys*., vol. 115, pp. 1716-1733, 2001.
[4] 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.
[5] J. M. McCollum, "Accelerating Exact Stochastic Simulation of Biochemical Systems", PhD dissertation, Electrical and Computer Engineering, University of Tennessee. 2006.
[6] 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.
[7] NVIDIA Corporation, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide", June 2008 [Online], http://developer.download.nvidia.com.

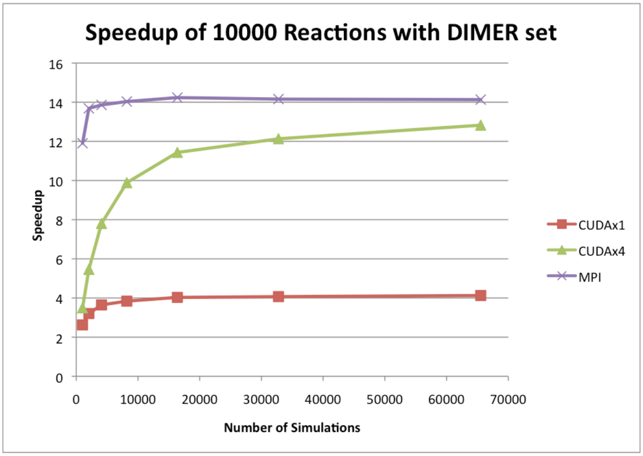


Figure 1: Performance with Dimerization Model

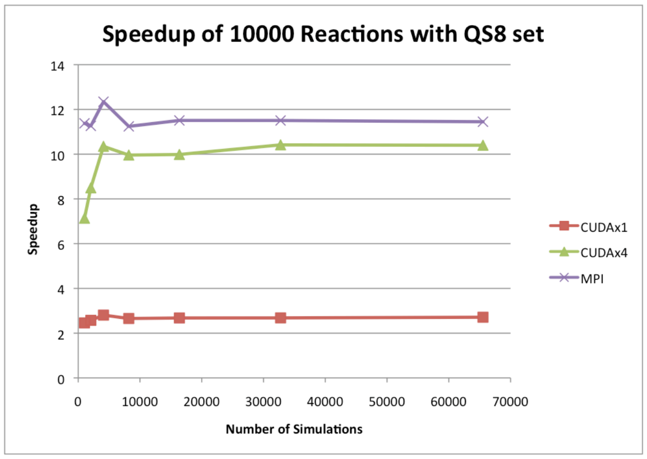


Figure 2: Performance with Quorum Sensing Model