Monte Carlo Methods for Parallel Processing of Diffusion Equations

Cyrus Vafadari

SUBMITTED TO THE DEPARTMENT OF NUCLEAR SCIENCE AND ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF BACHELOR OF SCIENCE IN NUCLEAR SCIENCE AND ENGINEERING AT THE MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Signature of author: ______________________________

Cyrus Vafadari
Department of Nuclear Science and Engineering
17 May 2013

Certified by ______________________________

Benoit Forget
Associate Professor of Nuclear Science and Engineering
Thesis Supervisor

Accepted by ______________________________

Dennis Whyte
Professor of Nuclear Science and Engineering
Chairman, NSE Committee for Undergraduate Students

Monte Carlo Methods for Parallel Processing of Diffusion Equations

Cyrus Vafadari

Submitted to the Department of Nuclear Science and Engineering
on 17 May 2013 in Partial Fulfillment of the Requirements for the
Degree of Bachelor of Science in Nuclear Science and Engineering

Abstract

A Monte Carlo algorithm for solving simple linear systems using a random walk is demonstrated and analyzed. The described algorithm solves for each element in the solution vector independently. Furthermore, it is demonstrated that this algorithm is easily parallelized. To reduce error, each processor can compute data for an independent element of the solution, or part of the data for a given element for the solution, allowing for larger samples to decrease stochastic error. In addition to parallelization, it is also shown that a probabilistic chain termination can decrease the runtime of the algorithm while maintaining accuracy. Thirdly, a tighter lower bound for the required number of chains given a desired error is determined.

Thesis Supervisor: Benoit Forget
Title: Associate Professor of Nuclear Science and Engineering

Contents

1 Introduction

The analysis of nuclear reactors has led to the development of sophisticated computational methods for estimating neutron flux in a reactor. The neutron flux in a reactor is determined by the either the neutron diffusion equation or the transport equation, systems of partial differential equations. Linear approximations of these equation is easily obtained with finite difference methods. The resulting matrix equation is typically solved deterministically, often with an iterative solver such as Krylov [6]. Such solvers can be found in existing software such as MATLAB or optimized specifically for a particular type of matrix. As spatial and temporal resolution on the equation increases, computational software is faced with the challenge of solving increasingly large matrix equations. Current deterministic methods of solving high resolution multigroup three-dimensional transport equations quickly overwhelm a single processor.

Various advances in software and hardware have allowed computations to be performed in parallel, dividing computation among multiple processors. Such large-scale computing, known as exascale computing, is limited by a number of challenges. Among the outstanding challenges in exascale computing for nuclear reactor analysis is the efficient division of tasks among processors. Algorithms optimal for serial processing can face bottlenecks in computing and perform suboptimally in large, parallel computer systems. Current diffusion equation solvers use a variant of successive over-relaxation algorithms. These methods are fast deterministic methods well-suited for large, sparse matrices[1].

Monte Carlo methods can easily address division of labor since each chain (i.e. random walk) is independent of the next. It can be used to solve large matrix equations (including the diffusion and transport equations) can be solved to a good approximation[4, 7]. To date, such a method has not been used to solve diffusion equations. Iterative Monte Carlo methods have many advantages in solving large, sparse systems [2]. One advantage is the ease of parallelization of a Monte Carlo solver since it eliminates bottlenecks. The application of Monte Carlo linear solvers to diffusion equations will demonstrate its feasibility in potential applications in solving very large equations.

2 Background

2.1 Theoretical Foundation [4]

A stochastic linear solver can be used to solve a linear system $Ax = b$ with unknown vector x for square, invertible matrix A. This can be rewritten as

$$x = (I - DA)x + Db$$

where D is a diagonal matrix

$$d_{ii} = \frac{\gamma}{a_{ii}}$$

where $\gamma \in (0, 1]$ is the relaxation parameter of the Jacobi overrelaxation iterative method. We can further let $f = Db$ and $L = I - DA$,

$$x = Lx + f \tag{1}$$

One method to find the vector x is to construct a random variable $X[v]$ whose expectation value is equal to x.

This is done with a discrete Markov chain. A discrete Markov chain is defined as a sequence of random variables X_1, X_2, X_3 with the Markov property: given the present state, future and past states are independent. Thus, a random walk with independent transitions between states through different elements of a matrix is a discrete Markov chain.

Consider the Markov chain

$$S = s_0 \rightarrow s_1 \rightarrow ...$$

with transitional probabilities p_{ij} from state s_i to state s_j. It can be shown that the random variable $X[v]$, defined as

$$X[v] = \frac{v_{s0}}{p_{s0}} \sum_{i=0}^{\infty} W_q f_{s_q} \tag{2}$$

where

$$W_i = \frac{l_{s_0 s_1} l_{s_1 s_2} ... l_{s_{i-1} s_1}}{p_{s_0 s_1} p_{s_1 s_2} ... p_{s_{i-1}} p_{s_i}}$$

has expectation value $EV(X) = x$.

Proof :

$$EV(X) = EV\left(\frac{v_{s_0}}{p_{s_0}} \sum_{i=0}^{\infty} W_q f_{s_q}\right) = \sum_{s=1}^{m} \frac{v_{x_0}}{P_{x_0}} \frac{l_{s_0 s_1} l_{s_1 s_2} ... l_{s_{i-1} s_1}}{p_{s_0 s_1} p_{s_1 s_2} ... p_{s_{i-1}} p_{s_i}} f_{s_q} p_{s_0 s_1} p_{s_1 s_2 ... p_{s_{q-1} s_1}}$$

$$= \sum_{s_0=1}^{m} v_{s_0} \sum_{s_1=1}^{m} \cdots \sum_{s_{q-1}=1}^{m} l_{s_0 s_1} l_{s_1 s_2} \ldots l_{s_{q-2} s_{q-1}} \sum_{s=1}^{m} l_{s_{q-1} s_q} f_{s_q}$$

$$= \sum_{s_0=1}^{m} v_{s_0} (L^q f)_{s_0}$$

Therefore $EV(X) = x$.

2.2 Monte Carlo Algorithm [4]

A Monte Carlo algorithm can be designed to calculate the expected value given in equation 2.

```
1. Generate L
   L(i,j) = 1-gamma for diagonal entries
   L(i,j) = -gamma*A(i,j)/A(i,i) otherwise
2. Generate f
   f(i) = b(i)/A(i,i)
3. Generate RowSumL
   RowSumL(i) = sum of absolute values of elements of row of L
4. Generate Probability matrix P
   P(i,j) = |L(i,j)|/RowSumL(i)
5. for each element of the solution vector x(i)
   for each chain to nchains
       u = 0, w = 0
       u = u + w*f(i)
       Select randomly a column j from row i probability P(i,j)
           w = w * sign(L(i,j)) * RowSumL(i)
           u = u + w*f(i)
           if |w| < w_cutoff
               x(i) = x(i) + u
               break
           i = j
   x(i) = x(i) / nchains
```

3 Analysis of Linear Solvers

Linear solvers of simple matrix equations with Monte Carlo methods give non-deterministic approximations for solutions. The non-deterministic behavior demands an understanding of the sources of error, both stochastic and truncation, and how this error affects the distribution of results and convergence to the true value. Error in the Monte Carlo algorithm can arise from two distinct sources: stochastic error and truncation error. Stochastic error refers to the error associated with the randomness of the Markov chains, resulting in uncertainty in estimates. Truncation error refers to the truncation of the Markov chain (i.e. the summatoin of equation 2 cannot be infinite), generally resulting in biased approximations.

3.1 Distribution of Chain Values

Thc cstimate for the value of a matrix element is equal to the mean of the chain values recorded at the end of the truncation of each Markov chain. The chain values are distributed non-normally in general. The shape of the distribution, however, is the result of a sample from a constant, characteristic shape for a given matrix equation $Ax = b$. The distributions of calculated chain values for simulations with 50, 500, and 5000 chains are shown in Figure 1.

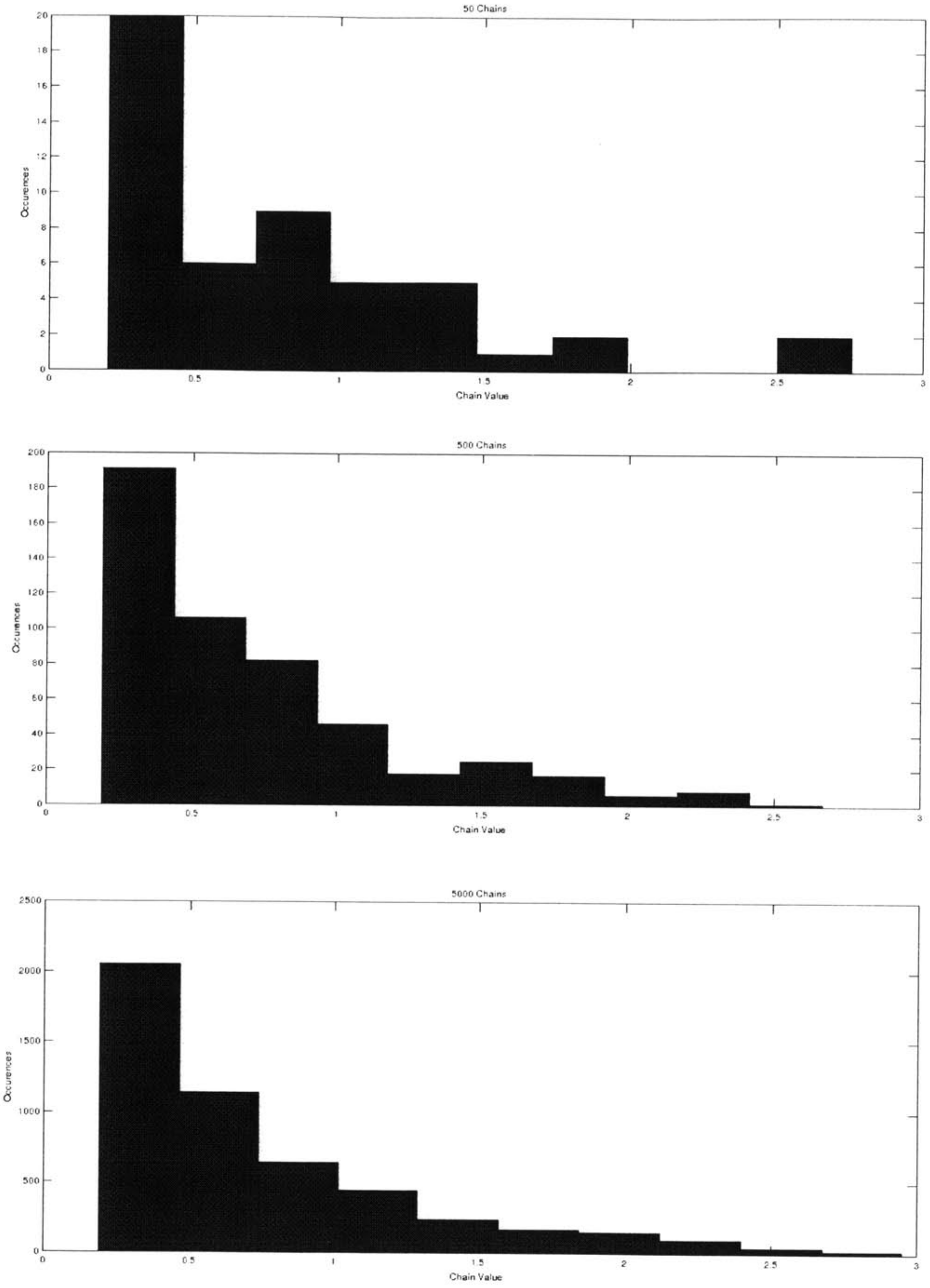

Figure 1: Distribution of Chain Values

3.2 Truncation Error

Truncation error can be reduced naively by applying a more strict cutoff weight for terminating the Markov Chain. This method, however, increases the runtime of the algorithm arbitrarily as the error is decreased. An algorithm where chains are terminated probabilistically has superior accuracy on average and much better algorithmic efficiency.

The method of probabilistic termination, also known as "Russian roulette," randomly selects some chains to continue after they have reached the truncation crieterion while others are truncated. More specifically, when the weight of a chain falls below the cutoff weight, it will be truncated with probability $p = \frac{w_{cutoff}}{w_{ave}}$, assigned a new weight $w = w_{ave}$, and continued. An implementation of probabilistic termination is used to tabulate the tradeoff between accuracy and run-time in Table 1. The control value always truncates after w falls below w_{cutoff}.

		w_{cutoff}				
		1e-2	**1e-3**	**1e-4**	**1e-5**	**1e-6**
	Control	0.0244	0.0280	0.0222	0.0223	0.0191
w_{ave}	**0.8**	0.0321	0.0252	0.0169	0.0259	0.0236
	0.4	0.0204	0.0264	0.0162	0.0258	0.0236
	0.2	0.0287	0.0298	0.0151	0.0256	0.0236
	0.1	0.0213	0.0190	0.0142	0.0251	0.0237

		w_{cutoff}				
		1e-2	**1e-3**	**1e-4**	**1e-5**	**1e-6**
	Control	214	305	392	484	565
w_{ave}	**0.8**	224	305	387	478	555
	0.4	222	302	394	479	555
	0.2	225	303	396	480	558
	0.1	226	304	391	468	559

Table 1: L^2-Norm of residual (top) and runtime (bottom) using probabilistic truncation (50,000 chains)

The simulations in Table 1 confirm the prediction that probabilistic termination of Markov chains can decrease the error of a simulation if the parameters w_{ave} and w_{cutoff} are chosen carefully. The probability $p = \frac{w_{cutoff}}{w_{ave}}$ must not be so small that the chain is never recovered from termination. It is clear that in the case of setting $w_{cutoff} = 10^{-3}$, the error in the estimated solution was as accurate as the estimation with $w_{cutoff} = 10^{-6}$, with a fraction of the runtime. Further optimization must be explored, as optimal choice for w_{ave} and w_{cutoff} depend on many factors, including total number of chains. In a trial with 500,000 chains is shown in Table 2.

		w_{cutoff}				
		1e-2	**1e-3**	**1e-4**	**1e-5**	**1e-6**
	Control	0.0132	0.0070	0.0055	0.0089	0.0095
w_{ave}	**0.8**	0.0093	0.0054	0.0113	0.0044	0.0080
	0.4	0.0067	0.0053	0.0113	0.0044	0.0080
	0.2	0.0062	0.0058	0.0111	0.0045	0.0080
	0.1	0.0064	0.0052	0.0105	0.0046	0.0080

		w_{cutoff}				
		1e-2	**1e-3**	**1e-4**	**1e-5**	**1e-6**
	Control	2233	3106	3875	4667	5610
w_{ave}	**0.8**	2154	2987	3833	4716	5444
	0.4	2167	2988	4431	4674	5448
	0.2	2188	2994	3916	4697	5473
	0.1	2222	3000	5516	4689	5447

Table 2: L^2-Norm of residual (top) and runtime (bottom) using probabilistic truncation (500,000 chains)

3.3 Stochastic Error

As the total number of chains increases, the stochastic error of the estimate will decrease. Though the distribution of chain values is non-normal, the means of random samples from the aggregate data is guaranteed to be normally distributed by the Central Limit Theorem. While we expect the stochastic error to depend only on the total number of chains analyzed, separating the computation into independent batches allows for predictive statistics. Table 3 confirms that the total number of chains affects accuracy of the estimate, where each sample is run computationally as an independent batch, with a specified number of chains per batch.

		Chains			
		50	**100**	**200**	**1000**
	50	0.0452	0.0325	0.0247	0.0095
Batches	**100**	0.0326	0.0197	0.0164	0.0050
	200	0.0196	0.0174	0.0081	0.0044
	1000	0.0105	0.0055	0.0047	0.0037

Table 3: L^2-Norm of residual with $w_{cutoff} = 0.001, w_{ave} = 0.8$

To be confident to a given level of confidence that the true mean is within a cut-off ϵ of the calculated mean, the total number of chains must be at least $n \geq (\frac{.6745}{\epsilon(1-||L||_1)})^2$ according to Dimov [3]. Using Dimov's method, Table 4 shows the lower bound for the number of chains for a given norm, as well as the observed number of chains required to obtain the given norm.

Norm	0.0452	0.0325	0.0247	0.0095	0.0050	0.0044
Actual Chains	2500	5000	10,000	50,000	100,000	200,000
Predicted Number of Chains	7.32e6	1.02e7	1.34e7	3.48e7	6.62e7	7.52e7

Table 4: Dimov estimate for number of chains required for a minimum L^2-Norm

This estimate is quite conservative. A tighter upperbound can be determined by calculating the standard deviation of the chain values on the fly. The $n = (t * \frac{\sigma}{\epsilon})^2$, where $t*$ is the critical value for the desired confidence of a Student distribution and σ is the standard deviation of a distribution like those in Figure 1. The runtime of the algorithm increases as the square of the desired margin of error is reduced.

Norm	0.0452	0.0325	0.0247	0.00947	0.0050	0.0044
Actual Chains	2500	5000	10,000	50,000	100,000	200,000
Predicted Chains for given norm	4680	9060	15,680	106,743	382,648	494,122

Table 5: Improved estimate for umber of chains required for a minimum L^2-Norm at a 99% confidence

4 Parallel Processes

The stochastic process can be parallelized by simply assigning each node to do any given number of chains for any given element of the solution vector. Using the Message Passing Interface (MPI) standard for inter-process and inter-memory communication [5], each node was assigned an even fraction of chains to compute. Assigning each node to calculate some chains for each element of the solution vector assures that each processor does nearly even amounts of work, where assigning different processors to different elements of the solution matrix may cause uneven computational workload. The source code, in Appendix D, simply aggregates the data caluclated from each node and takes a simple mean to find the aggregate estimate. The use of multiple processors significantly decreases the runtime of the Monte-Carlo solver without compromising accuracy. A strong-scaling study was run, and results are shown in Figure 2.

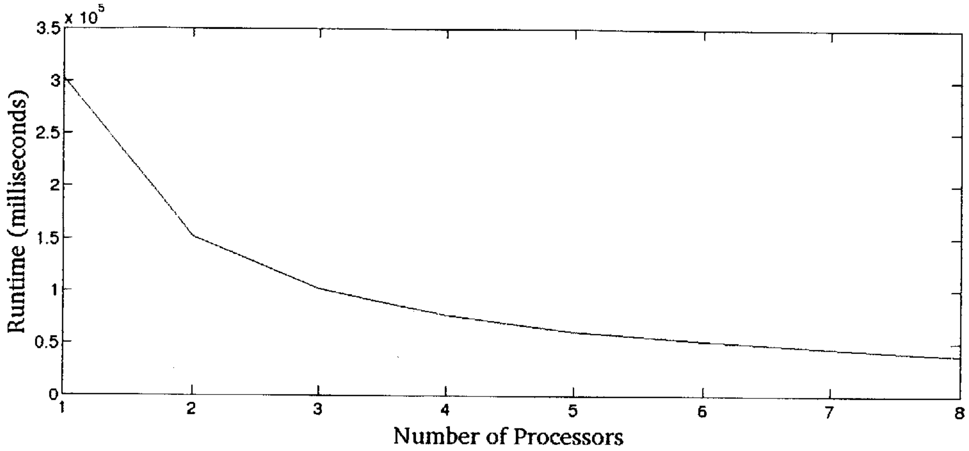

Figure 2: Runtime for Parallelized Code

The speed-up plot in Figure 3 demonstrates that some parallelization overhead is observed, most probably because of the root node's calculation of the initial matrices. Further scaling requires significantly larger matrices to fully demonstrate the parallelization of the Monte Carlo solver.

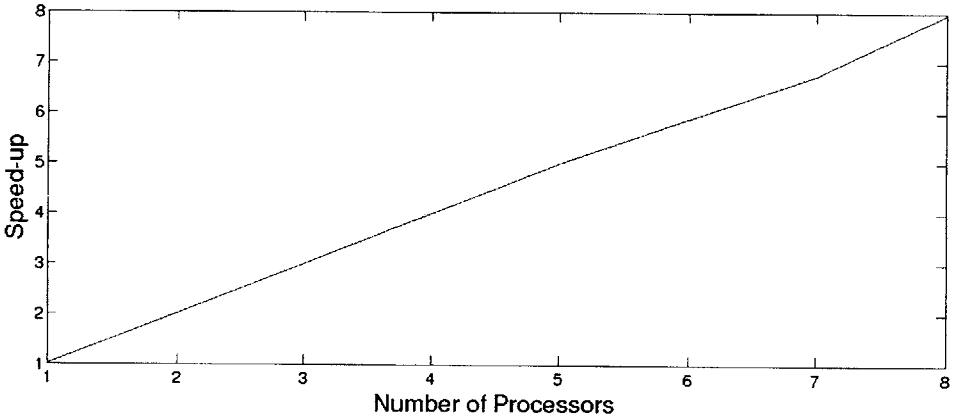

Figure 3: Speed-up for Parallelized Code

5 Conclusion and Future Work

The parallelization of Monte Carlo methods to quickly approximate solutions to linear systems of equations is demonstrated. It is shown that error can be reduced for a given runtime by properly selecting parameters to probabilistically terminate Markov chains. The optimization for parameters chosen for probabilistic termination of chains are dependent on the number of total chains. Further analysis is necessary to find a more quantitative relationship between number of chains and the parameters for probabilistic termination.

Statistical Monte Carlo algorithms present further opportunities in parallel computing research and optimization. Parallel computing architectures are subject to computation errors in a single node. The stochastic nature of the Monte Carlo solver may allow for error detection by handling outlier data. While deterministic solvers would otherwise propogate such an error, Monte Carlo solvers have the potential to be robust to such computational errors.

References

[1] Jalili Behabadi Mohammad Hasan Abadi Peyvand Ali, Pazirandeh. Finite difference method for solving neutron diffusion equation in hexagonal geometry. In *Nuclear Energy for New Europe*, 2009.

[2] J. H. Curtiss. Monte carlo methods for the iteration of linear operators. *J. Math Phys*, 32(4):209–232, 1954.

[3] Ivan T. Dimov. *Monte Carlo methods for applied scientists.* World Scientific Publishing Company, Incorporated, 2008.

[4] T.T. Dimov, T. T. Dimov, and T.V. Gurov. A new iterative monte carlo approach for inverse matrix problem. *Journal of Computational and Applied Mathematics*, 92:15–35, 1998.

[5] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA, 1994.

[6] H. Knibbe, C. W. Oosterlee, and Cornelis Vuik. Gpu implementation of a helmholtz krylov solver preconditioned by a shifted laplace multigrid method. *J. Computational Applied Mathematics*, 236(3):281–293, 2011.

[7] Ashok Srinivasan and Vikram Aggarwal. Improved monte carlo linear solvers through non-diagonal splitting. In *Proceedings of the 2003 international conference on Computational science and its applications: PartIII*, ICCSA'03, pages 168–177, Berlin, Heidelberg, 2003. Springer-Verlag.

Appendix A: Diffusion Equation

A general, two-group diffusion equation is written as follows:

$$-\nabla\cdot D_1(\overrightarrow{r})\nabla\phi_1(\overrightarrow{r})+[\Sigma_{a1}(\overrightarrow{r})+\Sigma_{s12}(\overrightarrow{r})]\phi_1(\overrightarrow{r}) = \nu\Sigma_{f1}(\overrightarrow{r})\phi_1(\overrightarrow{r})+\nu\Sigma_{f2}(\overrightarrow{r})\phi_2(\overrightarrow{r})+s_1(\overrightarrow{r})$$

$$-\nabla\cdot D_2(\overrightarrow{r})\nabla\phi_2(\overrightarrow{r}) + \Sigma_{a2}(\overrightarrow{r})\phi_2(\overrightarrow{r}) = \Sigma_{s12}(\overrightarrow{r})\phi_1 + s_s(\overrightarrow{r})$$

We assume macroscopic cross section to be constant. Furthermore, we assume that there are no neutron sources. Then, a diffusion equation in one dimension becomes:

$$-D_1\partial_x^2\phi_1(\overrightarrow{r}) + [\Sigma_{a1} + \Sigma_{s12}]\phi_1(\overrightarrow{r}) = \nu\Sigma_{f1}\phi_1(\overrightarrow{r}) + \nu\Sigma_{f2}\phi_2(\overrightarrow{r})$$

$$-D_2\partial_x^2\phi_2(\overrightarrow{r}) + \Sigma_{a2}\phi_2(\overrightarrow{r}) = \Sigma_{s12}\phi_1(\overrightarrow{r})$$

Appendix B: Selected Matrix Equation for Simulation Testing

The matrix equation $Ax = b$ was solved. The numerical values of A and b were chosen by solving a 1-dimensional diffusion equation with 10 spatial discretizations, 2 groups, fission cross sections (.0025, .08125), removal cross sections (0.0318, 0.114), ν of 2.4, and down-scattering cross section (0.022).

A = (1,1) 2.0318 (2,1) -1.0000 (11,1) -0.0220 (1,2) -1.0000 (2,2) 2.0318 (3,2) -1.0000 (12,2) -0.0220 (2,3) -1.0000 (3,3) 2.0318 (4,3) -1.0000 (13,3) -0.0220 (3,4) -1.0000 (4,4) 2.0318 (5,4) -1.0000 (14,4) -0.0220 (4,5) -1.0000 (5,5) 2.0318 (6,5) -1.0000 (15,5) -0.0220 (5,6) -1.0000 (6,6) 2.0318 (7,6) -1.0000 (16,6) -0.0220 (6,7) -1.0000 (7,7) 2.0318 (8,7) -1.0000 (17,7) - 0.0220 (7,8) -1.0000 (8,8) 2.0318 (9,8) -1.0000 (18,8) -0.0220 (8,9) -1.0000 (9,9) 2.0318 (10,9) -1.0000 (19,9) -0.0220 (9,10) -1.0000 (10,10) 2.0318 (11,10) -1.0000 (20,10) -0.0220 (10,11) -1.0000 (11,11) 2.1140 (12,11) -1.0000 (11,12) -1.0000 (12,12) 2.1140 (13,12) -1.0000 (12,13) -1.0000 (13,13) 2.1140 (14,13) -1.0000 (13,14) -1.0000 (14,14) 2.1140 (15,14) -1.0000 (14,15) -1.0000 (15,15) 2.1140 (16,15) -1.0000 (15,16) -1.0000 (16,16) 2.1140 (17,16) -1.0000 (16,17) -1.0000 (17,17) 2.1140 (18,17) -1.0000 (17,18) -1.0000 (18,18) 2.1140 (19,18) -1.0000 (18,19) -1.0000 (19,19) 2.1140 (20,19) -1.0000 (19,20) -1.0000 (20,20) 2.1140

b = 0.1000 0.1000 0.1000 0.1000 0.1000 0.1000 0.1000 0.1000 0.1000 0.1000 0 0 0 0 0 0 0 0 0 0

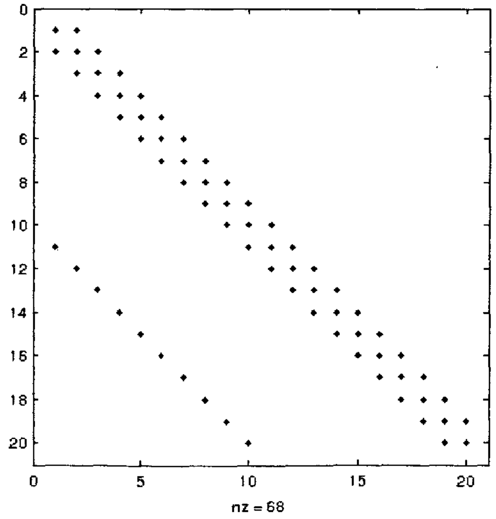

Figure 4: Elements of Sparse Matrix A

Appendix C: Sample Code for Monte Carlo Solver in Matlab

```
% Based on: "A new iterative Monte Carlo approach for inverse matrix" by I

function [result] = sparseBatchStochasticSolver(A, b, nchains, nbatches, w
    %% Sanitize Inputs
    % Make sure A is square
    dims = size(A);
    if (dims(1) ~= dims(2))st
        error('Matrix is not square. Please use square matrices only.');
    end
    n = dims(1);
    % Make sure b has the same dimension as A
    if (n ~= size(b))
        error('A and b do not have compatible dimensions.');
    end
    % Make sure cutoff for weight is valid
    if (wcutoff > 1 | wcutoff < 0)
        error('Invalid value for wcutoff');
    end
    % Make sure relaxation factor is valid
    if (relaxation > 1 | relaxation <=0)
        error('relaxation var has bad value');
    end
    % Check if diagonals dominate
    for i=1:n
        diagIndex = sum(abs(A),2)-2*diag(abs(A));
        if (diagIndex(i) > 0)
            error('Not diagonally dominant');
        end
    end
    %% Pre-Processing
    x = zeros(n,nbatches); % Solution vector
    stdev = zeros(n,nbatches);
```

```
stdevBatch = zeros(n,1);

L = sparse(n,n);
for i=1:n
    for j=1:n
        if (i == j)
            L(i,j) = 1 - relaxation;
        else
            L(i,j) = -relaxation*A(i,j)/A(i,i);
        end
    end
end
     f = zeros(n,1);
for i=1:n
    f(i) = relaxation*b(i)/A(i,i);
end
     rowSumL = zeros(n,1);
for i=1:n
    for j=1:n
        rowSumL(i) = rowSumL(i) + abs(L(i,j));
    end
    if (rowSumL(i) > 10)
            disp(['lsum is pretty high....']);
            rowSumL
    end
end
     P = sparse(n,n);
C = sparse(n,n);

for i=1:n
    for j=1:n
        if L(i,j) == 0
            continue
        end
        P(i,j) = abs(L(i,j))/rowSumL(i);
        if (j==1)
```

```
            C(i,j) = P(i,j);
        else
            C(i,j) = P(i,j) + C(i,j-1);
        end
    end
end

%% Computation
for i=1:n
    for batch=1:nbatches
        M2 = 0;
        % If we already knew the solution earlier somehow, don't bothe
        if (x(i,batch) ~= 0)
            continue
        end

        for m=1:nchains
            u = 0;
            w = 1;
            point=i;
            u = u + w*f(point);

            while (1 == 1)
                prn = rand(1);
                for k=1:n
                    if (prn <= C(point,k))
                        nextpoint = k;
                        break
                    end
                end
                w = w * sign(L(point,nextpoint)) * rowSumL(point);
                u = u + w*f(nextpoint);
                if (abs(w) < wcutoff)
                    % Online statistics for mean and stdev
                    if (rand(1) < (wcutoff/wave))
                        w = wave;
```

```
                            continue;
                        end
                        delta = u - x(i,batch);
                        x(i,batch) = x(i,batch) + delta/m;
                        M2 = M2 + delta*(u-x(i,batch));
                        break
                    end
                    point = nextpoint;
                end
            stdev(i,batch) = sqrt(M2/(nchains-1));
            end
        end
        result(i) = mean(x(i,:))
    end
end
```

Appendix D: Sample Code for Parallelized Monte Carlo Solver using MPI in C

```
#include "mpi.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <cs.h>
#include <stddef.h>

double getEntry(cs *A, int nnz, int row, int col)
// Gets entry from matrix A at location given by row, col
    {      int i;      for (i=0; i<nnz; i++)
        {
        if (A->i[i] == row && A->p[i] == col)
            {
            return A->x[i];
            }
        }
    return 0;
    }

double getEntry2(long *i, long *p, double *x,int nnz, long row, long col)
    // Gets entry from matrix A at location given by row, col
    {
    int c;
    for (c=0; c<nnz; c++)
        {
        if (i[c] == row && p[c] == col)
            {
            return x[c];
            }
        }
    return 0;
    }
void mcSolve(double *Px, long *Pi, long *Pp, double *Lx, long *Li, long *L
int nnz, int n,
             int nchains, int nbatches,
             double wcutoff, double wave, double relaxation)
    {
    double x[n];
    int i, batch, m;
    long c, point, nextpoint;
    double cumSoFar, u, w, prn;
    for (i=0; i<n; i++)
        {
        for (batch=0; batch < nbatches; batch++)
            {
            for (m=0; m < nchains; m++)
                {
                u = 0;
                w = 1;
                point = i;
                u = u + w*f[point];
                while (1==1)
                    {
                    prn = ((double)rand()/(double)RAND_MAX);
                    cumSoFar = 0;
                    c = 0;
                    // Assign nextpoint
                    while (prn > cumSoFar)
                        {
                        cumSoFar = cumSoFar + getEntry2(Pi, Pp, Px, nnz, p
                        nextpoint = c;
                        c++;
                        }
                    w = w * rowSumL[point] * fabs(getEntry2(Li,Lp,Lx,nnz,
                    u = u + w*f[nextpoint];
                    if (fabs(w) < wcutoff)
                        {
                            if (drand48() < (wave/wcutoff))
                                {
                                w = wave;
                                continue;
                                }
                        x[i] = x[i] + u;
                        break;
                        }
                    point = nextpoint;
                    }
                }
            }
        x[i] = x[i]/(nchains*nbatches);
        printf("%.20f\n", x[i]);
        }
    }

int main(int argc, char **argv)
    {
    double relaxation = 1.0;
    double wcutoff = 1e-3;
    int nchains = 50;
    int nbatches = 1000;
    double wave = 0.1;

    int ierr, nprocs, rank;
    int root = 0;
    ierr = MPI_Init(&argc, &argv);
    ierr = MPI_Comm_rank(MPI_COMM_WORLD, &rank);// Get id of each processor
    ierr = MPI_Comm_size(MPI_COMM_WORLD, &nprocs); // Get total number of
    srand(rank*500000); // Seed the pseudorandom number generator
    int nnz = 68, n = 20;
    cs *L;
    L = cs_spalloc (n, n, nnz, 1, 1);
    double f[n];
    cs *P;
```

```
P = cs_spalloc(n,n,nnz,1,1);
double rowSumL[n];
long Li[nnz];
long Lp[nnz];
double Lx[nnz];
long Pi[nnz];
long Pp[nnz];
double Px[nnz];
if (rank == root)
    {
    cs *A;
    A = cs_spalloc (n, n, nnz, 1, 1);
    A = readMatrixFromFile(A);
    double b[n];
    b = readVectorFromFile(b);
    double *avals = A->x;
    long *cols = A->p;
    long *rows = A->i;
    // Initialize std dev vector
    double std[n];
    // Initialize batch std dev vector
    double stdbatch[n];

                // Generate L
    int c;
    for (c=0; c<nnz; c++)
        {
        if (avals[c] == 0)
            {
            printf("Somehow a zero entry got in to the A matrix when I
            continue;
            }
        if (cols[c] == rows[c]) //if it's a diagonal
            {
            if (relaxation == 1.0)
                {
```

```
                continue;
                }
            else
                {
                cs_entry(L, rows[c], cols[c], 1-relaxation);
                }
            }
        else
            {
            int rowNum = rows[c];
            double diag;
            cs_entry(L, rowNum, cols[c], -relaxation*avals[c]/getEntry
            }
        }

    // Generate rowSumL (an array that holds the value of the sum of t
    for (c=0; c<nnz; c++)
        {
        rowSumL[L->i[c]] = 0.0;
        }
    for (c=0; c<nnz; c++)
        {
        rowSumL[L->i[c]] = rowSumL[L->i[c]]+fabs(L->x[c]);
        }

    // Generate f
    for (c=0; c<n; c++)
        {
        f[c] = 0.0;
        }
    for (c=0; c<n; c++)
        {
        f[c] = relaxation*b[c]/getEntry(A,nnz, c, c);
        }

    // Generate P
    for (c=0; c<nnz; c++)
        {
        if (L->x[c] == 0)
            {
            continue;
            }
        cs_entry(P, L->i[c], L->p[c], fabs(L->x[c])/rowSumL[L->i[c]]);
        }

    for (c=0; c<nnz; c++)
        {
        Li[c]=L->i[c];
        Lp[c]=L->p[c];
        Lx[c]=L->x[c];
        Pi[c]=P->i[c];
        Pp[c]=P->p[c];
        Px[c]=P->x[c];
        }
    }

ierr = MPI_Barrier(MPI_COMM_WORLD);
ierr = MPI_Bcast(f, n, MPI_DOUBLE, root, MPI_COMM_WORLD);
ierr = MPI_Bcast(rowSumL, n, MPI_DOUBLE, root, MPI_COMM_WORLD);
ierr = MPI_Bcast(Li, nnz, MPI_LONG, root, MPI_COMM_WORLD);
ierr = MPI_Bcast(Lp, nnz, MPI_LONG, root, MPI_COMM_WORLD);
ierr = MPI_Bcast(Lx, nnz, MPI_DOUBLE, root, MPI_COMM_WORLD);

ierr = MPI_Bcast(Pi, nnz, MPI_LONG, root, MPI_COMM_WORLD);
ierr = MPI_Bcast(Pp, nnz, MPI_LONG, root, MPI_COMM_WORLD);
ierr = MPI_Bcast(Px, nnz, MPI_DOUBLE, root, MPI_COMM_WORLD);

ierr = MPI_Barrier(MPI_COMM_WORLD);

int mod = nbatches % nprocs;
nbatches = nbatches/nprocs;
if (rank == root)
```

```
    {
    nbatches = nbatches + mod;
    }

mcSolve(Px, Pi, Pp, Lx, Li, Lp, f, rowSumL, nnz, n, nchains, nbatches,
          wcutoff, wave, relaxation);


ierr = MPI_Barrier(MPI_COMM_WORLD);
ierr = MPI_Finalize();
}
```

Uses CSparse library from:

Direct Methods for Sparse Linear Systems, T. A. Davis, SIAM, Philadelphia, Sept. 2006. Part of the SIAM Book Series on the Fundamentals of Algorithms.