# Comparison of Monte Carlo Linear Solvers in Load-Balancing Applications

A. Srinivasan and B. Bouta
Department of Computer Science, Florida State University,
Tallahassee FL 32306-4530, USA, Email: {asriniva,bouta}@cs.fsu.edu.

November 16, 2005

**Abstract**

Monte Carlo (MC) linear solvers can be thought of as stochastic realizations of deterministic stationary iterative processes. Different MC solvers can be constructed from different iterative processes. They vary in their effectiveness, depending on the linear system solved. We consider a class of matrices arising from applications involving dynamic load balancing of parallel computations. MC linear solvers appear promising for such applications. We compare the effectiveness three different MC linear solvers for these applications. The significance of this work lies in (i) showing how MC linear solvers can be effective for these applications and, more importantly, (ii) comparing the effectiveness of different types of MC linear solvers for this class of applications.

## 1 Introduction

Work assigned to different processors in a parallel computation can vary significantly over the course of the computation in many important applications, such as in finite element codes with adaptive meshes and in adaptive molecular dynamics simulations. In order to ensure high efficiency, the loads need to be balanced dynamically (that is, each processor needs to be assigned roughly equal work). In most popular dynamic techniques, the process of load balancing can be divided into two components: (i) determining how much data should be moved from one processor to another, in order to balance the load across the system, while minimizing the amount of data sent, and (ii) determining which data should should be sent, in order to keep the communication cost of the rest of the computation small. This paper deals with the issue of determining the amount of load that needs to be moved from one processor to another.

Hu, Blake, and Emerson proposed an optimal scheme for solving this problem. It involves solving a linear system using the Conjugate Gradient method. We propose using a MC linear solver to solve the same system. This has the potential to reduce the communication and computation costs of load balancing, as discussed later.

MC linear solvers are based on estimating a Neumann series, using a Markov chain. The series itself arises from the splitting used in a stationary iterative scheme. MC linear solvers can be constructed for different types of splittings. They vary in their effectiveness, depending on the system solved. We compare three types of MC linear solvers on the dynamic load balancing application.

The outline of the rest of this paper is as follows. In § 2, we provide background material on dynamic load balancing. In § 3, we summarize the MC linear solvers compared in this paper, including a scheme (called SDI) that we recently proposed, and a Chebyshev based MC scheme

that we introduce in this paper. We then discuss the MC scheme for dynamic load balancing in § 4, and present empirical results comparing the different types of MC linear solvers in § 5. We finally summarize our conclusions in § 6.

# 2 Load Balancing

## 2.1 Background

Efficient use of parallel computers requires that (i) the computational effort (*load*) be roughly equally shared (*balanced*) across processes, and (ii) that the communication cost be minimized. Satisfying both requirements is an NP-hard problem in any reasonable model of parallelization. But a variety of heuristics have been proposed, and software developed, for accomplishing the above two goals. These techniques are typically effective when the load and communication patterns do not change during the course of the computation.

In many computations of increasing importance, such as finite element codes with adaptive meshes and adaptive molecular dynamics simulations, the load changes with time. Then dynamic load balancing techniques, which are applied during the course of the computation, are required. Dynamic techniques need to be fast, since changes in load may occur often. They also need to ensure that the amount of data transferred due to the load balancing step is small. We next discuss the typical computational structure of these applications, and then summarize an "optimal" load balancing technique.

## 2.2 Computational Structure of the Applications

Scientific applications often have the following computational structure: they start with some specified initial state, and iteratively update the state of the system. We can think of each iteration as an increment in time. Parallelization is typically carried out by domain decomposition, where we partition the state space, and assign different portions of the state-space to different processes. Processes typically need some data "owned" by other processes, in order to update their state. Since this data changes with time, they need to communicate with certain "neighboring" processes each iteration, to obtain the missing data. Note that we consider two processes to be neighbors in a logical sense – that is, they own portions of the state space that depend on each other. We do not imply that neighboring processes run on processors that are directly linked in the communication network. The processes may also perform some global communication, depending on the application. We illustrate the computational structure on each process in algorithm 2.1.

**Algorithm 2.1:** ComputationLoop()

**for** $time \leftarrow 1$ **to** *Number of steps*

**do**
1. Update local state
2. Exchange data with neighbors
**if** (*necessary*)
  **then** 3. Balance load

If the loads on the processes do not change much, then the load balancing step is not required. However, significant changes in loads occur in many adaptive applications, such as those mentioned in § 1. For example, adaptive finite element codes dynamically create finer meshes in regions where higher is resolution required, and this leads to increased loads on processes that own portions of such regions. In these situations, it is necessary to balance the load, since the process with the largest

load will otherwise become a bottleneck to efficient parallelization. Furthermore, the time spent in dynamic load balancing should be much smaller than the time spent in the actual computation.

### 2.3 An Optimal Dynamic Load Balancing Scheme

**Terminology:** Before discussing a popular load balancing scheme, we first mention some terminology. The process *graph* is given by $G = (V, E)$, where each vertex in $V$ corresponds to a process, and an edge $\{u, v\} \in E$ indicates that processes corresponding to vertices $u$ and $v$ communicate with each other. The process graph does not consider the physical topology of the network. The *load* on process $i$ (corresponding to vertex $i$) is given by $b_i$. The *degree* of vertex $i$ is denoted by $d_i$, and the degree $d$ of the graph is defined by $d = \max_i d_i$. Associated with any graph is a matrix called its *Laplacian*, which contains the connectivity information of the graph. We let $L$ denote the Laplacian of G. Its entries are given by $l_{ij} = -1$ if $\{i, j\} \in E$, $l_{ii} = d_i$, and $l_{ij} = 0$ otherwise. It is a positive semidefinite matrix. For a connected graph (our discussion deals only with such graphs), its rank is $|V| - 1$, and the 0 eigenvalue is associated with the eigenvector $e_0 = (1, 1, \cdots, 1)^T$. For any vector $w$ in the subspace $S$ orthogonal to the eigenvector $e_0$, there are infinitely many solutions to the system $L\lambda = w$. Furthermore, each solution is of the form $\hat{\lambda} + ae_0$, where $\hat{\lambda}$ is a unique vector in $S$, and $a$ can be any real number. In the dynamic load balancing application, we will solve such a system (with $w \in S$), and any of the infinite solutions will be acceptable.

**Two-phase load balancing:** As mentioned earlier, the process of load balancing is divided into two phases: (i) determining how much data should be moved from one process to another and (ii) determining which data should should be sent. Our focus is on the first phase, that is, determining the amount of data to be moved. Once this is determined, algorithms, such as that by Walshaw and Cross [1] can be used to determine the actual data that need to be moved. We next discuss a popular technique to address the problem of the first phase, which is optimal in some sense.

**Hu, Blake, and Emerson's Algorithm:** Hu, Blake, and Emerson [2] proposed an algorithm based on the following idea, shown in algorithm 2.2. They used the Conjugate Gradient method to solve the linear system involving the Laplacian mentioned above, *requiring three global communication operations in each iteration* of the Conjugate Gradient method, which makes the communication cost high. It can be shown [2] that Algorithm 2.2 balances the load, and it is also optimal in the sense that it minimizes the Euclidean norm of the data movement required to balance the load[1].

**Algorithm 2.2:** LoadBalancing()

1. Calculate the mean load $\bar{\mathrm{b}}$
2. Compute the vector w with $\mathrm{w_i} \leftarrow \mathrm{b_i} - \bar{\mathrm{b}}$
3. Solve the system $\mathrm{L}\lambda = \mathrm{w}$
4. Each process i sends $\max\{\lambda_\mathrm{i} - \lambda_\mathrm{j}, 0\}$ load to each neighbor j

[1] In the algorithm described above, even though $L$ is singular, $\lambda_i - \lambda_j$ is unique [2] for a connected graph, and is equal to $\hat{\lambda}_i - \hat{\lambda}_j$.

# 3 Monte Carlo Linear Solvers

In this section, we briefly summarize the idea behind MC linear solvers, which we will use to solve the system in Algorithm 2.2, instead of using the Conjugate Gradient algorithm. As we show in § 4, they potentially reduce the number of global communication operations compared with the deterministic technique, and also offer the potential of hiding much of the computation cost.

## 3.1 Linear Solvers

In order to solve $Ax = b$, $A \in \Re^{n \times n}$ and $x, b \in \Re^n$, the starting point of MC techniques is to split $A$ as

$$A = N - M, \tag{1}$$

and write the following fixed-point iteration, as in deterministic stationary iterative schemes.

$$x^{(m+1)} = N^{-1}Mx^{(m)} + N^{-1}b = Cx^{(m)} + h, \tag{2}$$

where $C = N^{-1}M$ and $h = N^{-1}b$. Then we get

$$x^{(m)} = C^m x^{(0)} + \sum_{i=0}^{m-1} C^i h. \tag{3}$$

The initial vector $x^{(0)}$ is often taken to be $h$ for convenience, yielding

$$x^{(m)} = \sum_{i=0}^{m} C^i h. \tag{4}$$

$x^{(m)}$ converges to the solution as $m \to \infty$ if $\|C\| < 1$.

## 3.2 MC Linear Solvers

MC techniques construct a Markov chain to estimate the sum in (4), with the initial probabilities determined by $h$, and transition probabilities by $C$. Each step of a walk estimates one of the $C^i h$ terms. In this paper, we take walks of $L$ steps, causing a truncation error[2]. We wish to note that there are many different estimators available[3] [3, 4, 5, 6, 7] (with the one we use being similar to that introduced by Wasow [7]).

Conventional MC techniques explicitly use the $C$ matrix, in contrast to most deterministic schemes. Therefore, $N$ is taken to be a diagonal matrix, yielding $C = N^{-1}M$ efficiently. This particular choice is equivalent to the Jacobi method. We recently introduced a non-diagonal splitting, called the *SDI* scheme [8, 9], and developed an efficient MC implementation for the splitting with $N$ chosen as the diagonal and first subdiagonal of $A$, as shown below:

$$N = \begin{pmatrix} a_{11} & & & & & \\ a_{21} & a_{22} & & & & \\ & a_{32} & a_{33} & & & \\ & & & \ddots & & \\ & & & & a_{n,n-1} & a_{nn} \end{pmatrix}. \tag{5}$$

[2] Alternatively, *absorbing* random walks can be used to eliminate the truncation error, at the cost of increased variance, which can even be unbounded, if we are not careful with transition probability choices.

[3] Each of the estimators has a *direct* formulation and an *adjoint* formulation. The former is useful in determining specific components of a solution. We use the adjoint formulation.

(We assume $a_{ii} \neq 0,\ 1 \leq i \leq n$, as with the Jacobi method. Otherwise $N^{-1}$ does not exist. This requirement is satisfied by the Laplacian of a connected graph with at least two vertices.)

### 3.3 MC Implementation of Chebyshev Iterations

Deterministic Chebyshev iterations, in essence, improve upon Jacobi iterations. Similarly, an MC implementation of Chebyshev iterations can be easily obtained, as shown below. Let $x^{(i)}$ denote the result of the Jacobi method with $i$ iterations. Chebyshev iterations [10] yield a result that is the linear combination of $x^{(i)}$s of the form:

$$y^{(L)} = \sum_{i=0}^{L} \nu(i, L) x^{(i)}, \tag{6}$$

where $L$ is the largest number of iterations that we wish to perform, and $y^{(L)}$ is the corresponding Chebyshev result. Given bounds on the eigenvalues of the $C$ matrix for the Jacobi method, $\nu(i, L)$ can be chosen to yield a low error in the result. Deterministic Chebyshev iterations are used to improve on the result of the Jacobi method and, in practice, use a three-term recurrence [10] among the $y^{(i)}$s to avoid storing each $x^{(i)}$. There is no direct MC realization of the deterministic scheme that is apparent. However, if we fix the number of iterations at a given number, $L$, then we can rewrite Eqn. (6) as follows.

$$y^{(L)} = \sum_{i=0}^{L} \mu(i, L) C^i h, \tag{7}$$

where

$$\mu(i, L) = \mu(i+1, L) + \nu(i, L),\ 0 \leq i < L, \tag{8}$$

and $\mu(L, L) = \nu(L, L)$. The $\mu(i, L)$s can be precomputed. As mentioned earlier, the $i$th step of a walk in the Jacobi-based MC technique estimates $C^i h$. Instead, we multiply the Jacobi estimate in step $i$ by $\mu(i, L)$, to yield an MC version of Chebyshev iterations.

The time taken for the MC linear solver is $O(NL)$ for $N$ walks, independent of the size of the matrix. However, we incur an additional pre-processing overhead that is proportional to the number of non-zero entries in $A$, in order to set up some data structures that enable constant time sampling using the alias method [11]. The variance of the estimate for each component of the solution decreases proportional to $N^{-0.5}$.

## 4 Load Balancing through an MC Scheme

### 4.1 The Algorithm

We first estimate $\Lambda$ in $L\Lambda = I$, where $I$ is the identity matrix, using an MC linear solver. Each process $i$ estimates column $i$ of $\Lambda$, corresponding to the right hand side being column $i$ of the identity matrix. This can be done with little computational overhead by performing the random walks when waiting for communication in step 2 of algorithm 2.1 to complete. This communication is an inherent part of the parallelization[4]. The processes perform an *Alltoall* global communication

[4]We give some sample figures below, to give an idea of the number of walks that can be performed, in some example situations. The time taken for the communication depends on the amount of data exchanged and on the communication network. Even if the amount of data sent is just one byte, the time taken can vary from a few microseconds on an Infiniband interconnect, to a few milliseconds on an Ethernet interconnect. This permits around $10^2$ steps in our random walks at the lower end, and around $10^5$ at the higher end, with a fast random number

operation on their column of $\Lambda$, after which each process $i$ has row $\Lambda^{(i)}$ of the matrix $\Lambda$. They then perform an *Allgather* operation on their loads $b_i$ to eventually determine the load imbalance vector $w$. The inner product of $\Lambda^{(i)}$ and $w$ yields the $i$ th component of the desired solution $\lambda$ on processor $i$. The remaining steps are as with the deterministic algorithm, and are shown in algorithm 4.1. If the MC estimate is very accurate, then the load is balanced with just two global communication operations, and little computational overhead. However, if the estimate is not accurate, then steps 1 through 6 of algorithm 4.1 need to be repeated. *Since the main overhead is due to the global communication operations, we can judge the effectiveness of different MC linear solvers by the number of repetitions of this step required for sufficiently balanced load.*

**Algorithm 4.1:** BALANCELOAD()

1. Alltoall($\Lambda$)
2. AllGather($b$)
3. Calculate $\bar{b} \leftarrow \frac{1}{n}\sum_{i=1}^{n} b_i$
4. Calculate $w_i \leftarrow b_i - \bar{b}$
5. Calculate $\lambda_{MyRank} \leftarrow \sum_{j=1}^{n} \Lambda_{MyRank,j} w_j$
6. Exchange $\lambda_{MyRank}$ with neighbors
7. Each process i sends load $\max\{\lambda_{\mathrm{i}} - \lambda_{\mathrm{j}}, 0\}$ to each neighbor j

## 4.2 MC Linear Solvers, Applied to Load Balancing

We now mention certain details regarding the use of MC linear solvers in the load balancing application.

**Jacobi and Chebyshev based solvers:** Let $D$ denote the diagonal matrix with $d_{ii} = l_{ii}$. We are essentially solving $L\lambda = w$ (albeit indirectly, in the MC load balancing algorithm). We will instead solve

$$\hat{L}y = \hat{w}, \tag{9}$$

where $\hat{L} = D^{-1/2}LD^{-1/2}$ is the scaled Laplacian of the graph, $y = D^{1/2}\lambda$, and $\hat{w} = D^{-1/2}w$. After determining $y$, we can compute $\lambda$ as $D^{-1/2}y$. $\hat{L}$ has an eigenvalue 0, but all other eigenvalues are in the interval $[\gamma = 1/(2|E|\delta), 2]$ for a connected graph, where $\delta$ is the diameter of the graph [12].

We will modify the Jacobi and Chebyshev iterations so that they are based on the following recurrence:

$$y^{m+1} = \left[I - \frac{\hat{L}}{1+\gamma/2}\right] y^m + \frac{\hat{b}}{1+\gamma/2}. \tag{10}$$

This corresponds to setting $C = I - \hat{L}/(1+\gamma/2)$ in Eqn. (2). The matrix $C$ has one eigenvalue equal to 1, corresponding to the 0 eigenvalue of $L$, and so the Jacobi iteration is not guaranteed to converge for an arbitrary starting vector. However, it will converge when the starting vector $y^0 = \hat{b}/(1+\gamma/2)$, as it is in the MC process, for the following reasons. $\hat{e}_0 = D^{1/2}e_0$ can easily be verified to be an eigenvector corresponding to the eigenvalue 1, and it is unique for this eigenvalue, within a constant factor, for a connected graph. Observing that $w$ is orthogonal to $e_0$, we can show

generator. If the walk length used is ten, and if we load balance every ten iterations, then we can perform between $10^2$ and $10^5$ walks per processor, depending on the communication network. In realistic applications, the amount of data exchanged will be much larger, and so a larger number of walks can be performed. In fact, we may also perform load balancing less frequently, leading to an even greater number of walks.

that $\hat{w}$ is in the subspace $\hat{S}$ orthogonal to $\hat{e}_0$. Consequently, the vectors $y^m$ are always in $\hat{S}$. Now, $\hat{S}$ has a basis of eigenvectors with eigenvalues in the interval $[-(1-\gamma/2)/(1+\gamma/2), (1-\gamma/2)/(1+\gamma/2)]$, from the bounds given above for the non-zero eigenvalues of $\hat{L}$. Since the absolute values of the eigenvalues in $\hat{S}$ are always less than 1, the Jacobi iteration converges. Chebyshev, being a linear combination of the Jacobi iterates, also converges.

**SDI solver:** We first scale the system by pre-multiplying by $D^{-1}$. We show in appendix A that for a connected graph with at least three vertices, there exists an ordering of vertices (which is also easy to find) such that the SDI iteration matrix $C$ is aperiodic, irreducible, and row-wise stochastic. Consequently, it has a single dominant eigenvalue 1, with a unique normalized eigenvector $e_0$, and all other eigenvalues (which may be complex) have absolute values strictly less than one [13]. If we express the initial vector as a linear combination of the eigenvectors, then the component corresponding to $e_0$ does not converge. However, the components corresponding to the other eigenvectors converge. As mentioned in § 2.3, adding a multiple of $e_0$ does not change the solution, since the load sent is the difference in the components of $\lambda$. A practical numerical consideration is roundoff errors from computing the difference of large numbers, if the component corresponding to $e_0$ is very large. This is not a difficulty in practice, because the walk length is of the order of 10.

# 5 Experimental results

### Test Goals

We want to compare the relative effectiveness of the three MC schemes in our load balancing application. The results will demonstrate that the underlying iterative process plays an important role in the effectiveness of MC linear solvers, even when similar estimators are used.

### Test Methodology

We compared the three solvers under conditions which were similar to what we expect in applications with sudden changes in load. We chose process interaction graphs with 121 vertices (processes). Initially, a fixed number of random walks, each with a fixed number of steps, are performed to collect information about the system solved. Then the load is taken as 1 on all processes except one, where the load is assigned the value 200. The linear solve and load balancing step is then repeatedly applied. We plot the load imbalance (defined as $|MaximumLoad - AverageLoad|/AverageLoad$) after each iteration of the load balancing step, and compare the different MC techniques. The experiments are carried out sequentially, since our goal in this paper is to compare the effectiveness of the different underlying iterative processes, rather than to develop parallel software.

### Test Results

Fig. 1 shows the change in load imbalance with number of iterations of the load balancing step. The number of random walks is around 830 per process (corresponding to 100,000 total), and the walk length is 3 steps. We see that the Chebyshev method with the exact eigenvalues performs best, and the Jacobi method the worst. The SDI method is substantially better than the Jacobi method. For a small number of iterations, it is better than the Chebyshev method if eigenvalue bounds (derived in § 4) are used with Chebyshev, instead of exact eigenvalue information. This suggests that for small walk lengths, if good estimates of the extreme eigenvalues are known, then

the Chebyshev method is to be preferred. For example, MC techniques may be used to estimate the largest and the smallest eigenvalues. On the other hand, if the extreme eigenvalues cannot be estimated fast, then the SDI method is preferable.

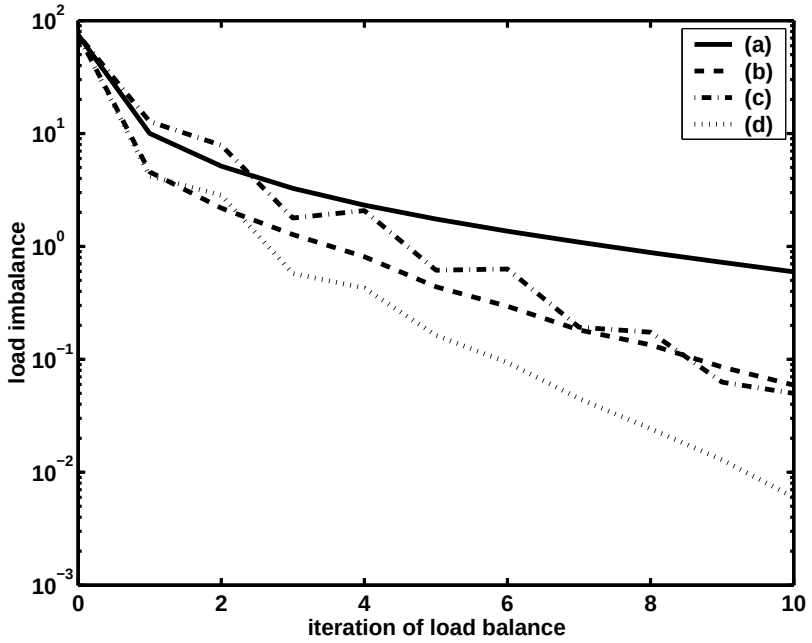


Figure 1: Plot of load imbalance against number of iterations of the load balancing step for a $11 \times 11$ 2-D torus. The underlying iterative schemes are: (a) Jacobi, (b) SDI, (c) Chebyshev with eigenvalue bounds, and (d) Chebyshev with exact eigenvalues. On the average, each process starts around 830 walks, with walk length 3.

Fig. 2 repeats the above experiment, but with walk length of 10. Here, the SDI scheme is much better than the others, and the Chebyshev solution does not converge. The Chebyshev solution with eigenvalue bounds performs so poorly, that it is not shown in the plot.

We conjecture that the reason for the MC version of Chebyshev not converging, even though the deterministic one converges, is as follows. The Chebyshev coefficients are very large and alternate in sign. In the exact deterministic computation, these large numbers cancel each other, resulting in a small number. However, larger variance in the MC estimates may lead to large errors, since we multiply by large coefficients. In fact, experiments with a smaller $3 \times 3$ torus, which has much smaller coefficients, showed that the MC solution converged, giving support to the above argument. In Fig. 3, we present a plot similar to Fig. 2, but with around 8300 walks per process. The variance should be lower here, and the Chebyshev results better, if our conjecture were right. This is indeed the case, as seen from the plots. The Chebyshev results with eigenvalue bounds are still poor, though much better than in the previous case.

It is also interesting to observe the effect of the number of walks on the relative performance of SDI and Jacobi based MC methods. This is shown in Fig. 4. The Jacobi method is better than SDI when the number of walks is very small. The reason for this is that the variance of the SDI method appears to be larger than that for the Jacobi method in this case. So the errors are larger. When the number of walks is larger, the error from the underlying iterative process dominates, and the SDI method performs better, as seen from Fig. 2.

Experiments with different graphs showed that the observations above apply to other graphs

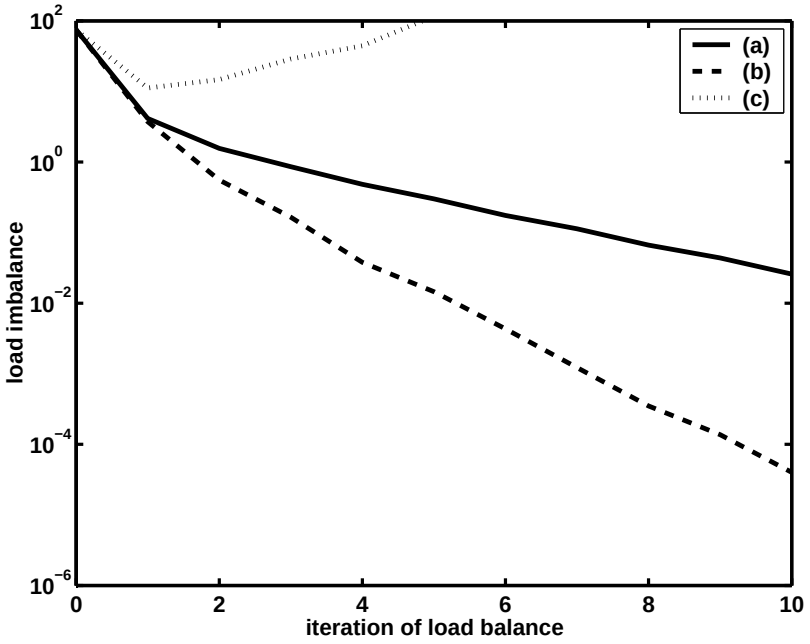


Figure 2: Plot of load imbalance against number of iterations of the load balancing step for a $11 \times 11$ 2-D torus. The underlying iterative schemes are: (a) Jacobi, (b) SDI, (c) Chebyshev with exact eigenvalues. On the average, each process starts around 830 walks, with walk length 10.

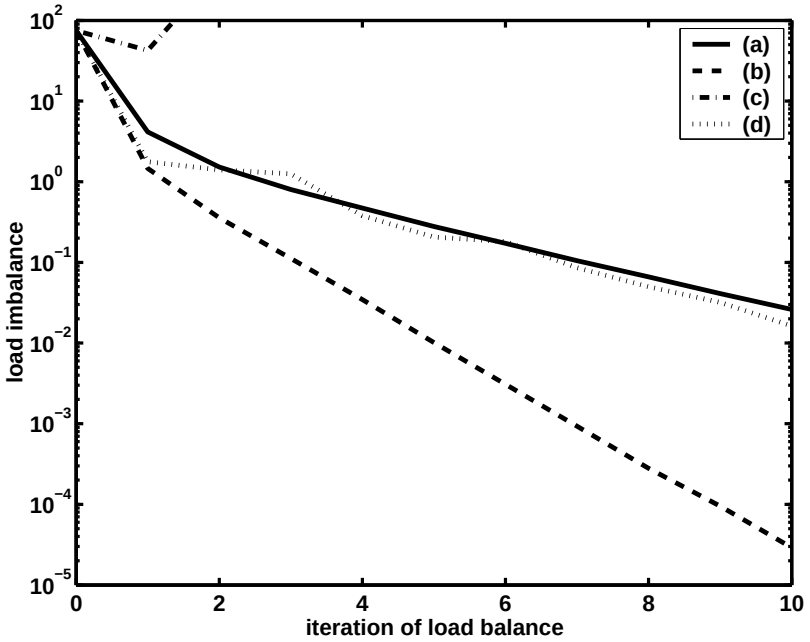


Figure 3: Plot of load imbalance against number of iterations of the load balancing step for a $11 \times 11$ 2-D torus. The underlying iterative schemes are: (a) Jacobi, (b) SDI, (c) Chebyshev with eigenvalue bounds, and (d) Chebyshev with exact eigenvalues. On the average, each process starts around 8300 walks, with walk length 10.

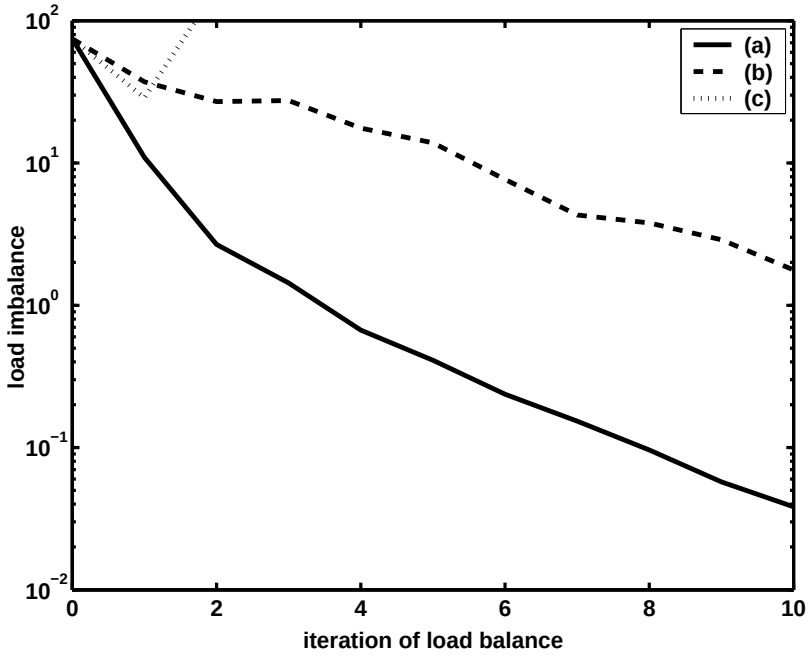


Figure 4: Plot of load imbalance against number of iterations of the load balancing step for a $11 \times 11$ 2-D torus. The underlying iterative schemes are: (a) Jacobi, (b) SDI, (c) Chebyshev with exact eigenvalues. On the average, each process starts around 83 walks, with walk length 10.

too. For example, Fig. 5 and Fig. 6 show results with walk length 10 for a 121 process ring and a 121 process linear array respectively. As in the previous cases with walk length 10 and number of walks around 830 per process, SDI performs better than Jacobi, while Chebyshev does not converge. Results with smaller walks lengths and number of walks too follow the same trend as with the torus. Additionally, we can observe that the convergence rate is low for the ring and the linear array. This has been observed in deterministic schemes too, and follows from the fact that large eigenvalues of the iteration matrices for these two graphs are close to 1 in absolute value.

In summary, the above results indicate that the SDI scheme is much better than the Jacobi method, provided the number of walks is not too small. The Chebyshev method requires care in its use, and is effective when the walk length is relatively small or when the number of walks are very large. Furthermore, an accurate estimate of the extreme eigenvalues is useful there.

# 6 Conclusions

We have compared MC schemes based on two new iterative processes (new for MC linear solver implementations), with the conventional Jacobi-based MC implementation. All the three schemes use essentially the same MC estimator. However, there are substantial differences in their performance in practical applications involving solving linear systems for dynamically load balancing parallel computations.

The SDI scheme consistently outperforms the Jacobi based scheme if the number of walks is not too small, while the Chebyshev based scheme outperforms Jacobi when its coefficients do not become very large.

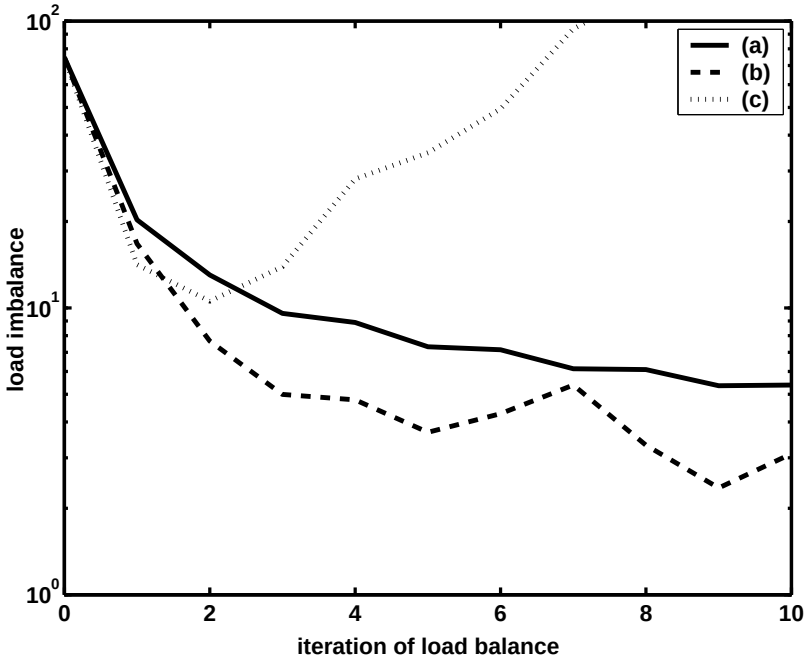


Figure 5: Plot of load imbalance against number of iterations of the load balancing step for a 121 node ring. The underlying iterative schemes are: (a) Jacobi, (b) SDI, (c) Chebyshev with exact eigenvalues. On the average, each process starts around 830 walks, with walk length 10.

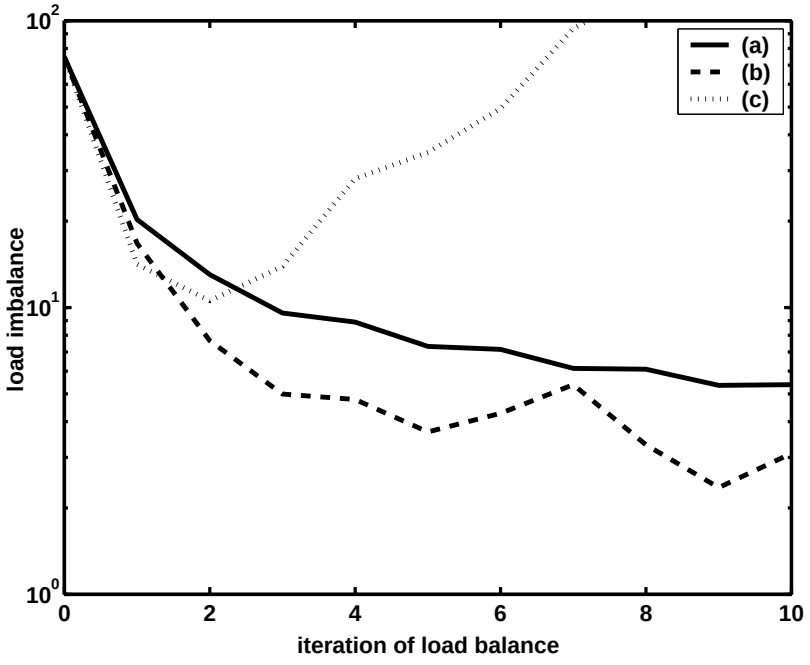


Figure 6: Plot of load imbalance against number of iterations of the load balancing step for a 121 node linear array. The underlying iterative schemes are: (a) Jacobi, (b) SDI, (c) Chebyshev with exact eigenvalues. On the average, each process starts around 830 walks, with walk length 10.

## Acknowledgment

A.S. wishes to express his gratitude to Sri S. S. Baba for his help, in particular with the convergence proofs for the SDI scheme.

## A Appendix: SDI Iteration Matrix Properties

We will scale $L$ by $D^{-1}$, so that the equation to be solved becomes $\hat{L}\lambda = \hat{w}$, where $\hat{L} = D^{-1}L$ and $\hat{w} = D^{-1}w$. Note that the diagonal entries of $\hat{L}$ are all 1. We split $\hat{L}$ as $N - M$ where $N$ is the diagonal (which is all ones) and subdiagonal of $\hat{L}$ as mentioned earlier. This leads to the stationary iteration: $x^{(m+1)} = Cx^{(m)} + h$, with $C = N^{-1}M$ and $h = x^0 = N^{-1}\hat{w}$. We first show that the matrix $C$ is row-wise stochastic.

**Lemma 1:** If at least one of the subdiagonal entries of $N$ is non-zero, then the SDI iteration matrix $C = N^{-1}M$ is row-wise stochastic for a connected graph with at least two vertices.

**Proof:** All entries in $M$ are clearly non-negative, from the definition of the Laplacian (even after the scaling). The entries of $N^{-1}$ are given by [8, 9]:

$$N_{ij}^{-1} = \begin{cases} 0 & \text{if } i < j \\ \frac{1}{\hat{l}_{ii}} & \text{if i = j} \\ \frac{(-1)^{i-j}}{\hat{l}_{jj}} \Pi_{k=j+1}^{i} \frac{\hat{l}_{k,k-1}}{\hat{l}_{kk}} & \text{otherwise} \end{cases} \tag{11}$$

Each $\hat{l}_{k,k-1}$ is negative, if it is non-zero, and each $\hat{l}_{kk}$ is positive and equal to 1 after the scaling. Consequently, from Eqn. 11, we see that all entries of $N^{-1}$ are non-negative. Therefore all entries of $C = N^{-1}M$ too are non-negative.

We can now show $C$ is row-wise stochastic by also showing that its row sums are 1. That is, we wish to show that $Ce_0 = e_0$. This is equivalent to showing that $Me_0 = Ne_0$. Note that $(Me_0)_i = \sum_{j \neq i-1,i} -\hat{l}_{ij}$ and $(Ne_0)_i = 1 + \hat{l}_{i,i-1}$ (where we define $\hat{l}_{1,0} := 0$ for notational convenience). Since $\sum_{j \neq i} \hat{l}_{ij} = -\hat{l}_{ii} = -1$, $Me_0 = 1 + \hat{l}_{i,i-1} = Ne_0$ and so $Ce_0 = e_0$. Q.E.D.

Consider a connected graph with at least three vertices. Let us call a vertex with only one edge as a *leaf*. A connected graph with at least three vertices has at least one non-leaf vertex. Let us number the vertices so that all non-leaf vertices are numbered before any leaf. Furthermore, let us number at least one pair of neighboring vertices consecutively (that is, as $i$ and $i + 1$). This can easily be accomplished. Under this ordering, we next show that the SDI iteration matrix $C$ is irreducible.

**Lemma 2:** Under the numbering of vertices given above, the SDI iteration matrix $C = N^{-1}M$ is irreducible for a connected graph with at least three vertices.

**Proof:** Under the given numbering, at least one pair of neighbors are numbered consecutively, and so at least one subdiagonal entry of $N$ is non-zero. From lemma 1, $C$ is row-wise stochastic. Consequently, we can view $C$ as the transition probability matrix of a Markov Chain[5], with $c_{ij}$ denoting the transition probability of a move from state $i$ to state $j$.

Since the diagonal entries of $N^{-1}$ are non-zero, a transition from state $i$ to state $j$ is always permitted if $m_{ij}$ is non-zero. Note that $m_{ij}$ is non-zero when $l_{ij}$ is non-zero, except for $j = i - 1$ or $i$. If $m_{i,i-1}$ too were non-zero when $l_{i,i-1}$ was non-zero, then $C$ would clearly be an irreducible matrix, since the graph is connected, and so a random walk could eventually transition from any state $i$ to any state $j$ (possibly taking several steps). Though $m_{i,i-1}$ are zero, we show that a

[5]This statement should be viewed independently of the Markov chain used in our Monte Carlo technique. $C$ is *not* the transition probability matrix for the MC technique.

transition from state $i$ to state $i-1$ can still be accomplished (in at most $n-2$ steps) when $l_{i,i-1}$ is non-zero. This will show that $C$ is irreducible.

Let us consider the situation when $l_{i,i-1}$ is non-zero for some $i$. Let $i-k, i-k+1, \cdots, i$ be a sequence of consecutive states such that edges $\{s, s+1\}$ are in $E$ for each $s \in \{i-k, \cdots, i-1\}$, and $\{i-k-1, i-k\}$ is not in $E$. In the situation considered, $k$ exists, is at least 1, and at most $i-1$. Note that transitions from state $s$ to state $s+1$ are preserved in $C$, since the transition from a lower numbered state to a higher numbered state is not removed from the original graph. We will consider the two possible cases for $k$.

**Case 1** $k=1$: Since $l_{i,i-1}$, and consequently $\hat{l}_{i,i-1}$, is non-zero, $N^{-1}_{i,i-1}$ is non-zero from Eqn. 11. So $c_{ir}$ are non-zero for all $r$ that are neighbors of $i-1$ (obtained directly from the matrix multiplication of $N^{-1}$ and $M$, and from the fact $i-2$ is not a neighbor of $i-1$, since $k=i-1$). From the numbering given above, $i-1$ is a vertex with at least two neighbors, and so at least one of those, say $t$, is different from $i$. Since $t \neq i$, $t$ still retains its transition to $i-1$. Consequently, in two steps, one can transition from $i$ to $i-1$.

**Case 2** $k \geq 2$: Here, we note that $\hat{l}_{s+1,s}$ are non-zero, and so $N^{-1}_{i,s}$ are non-zero from Eqn. 11. In particular, $N^{-1}_{i,i-k}$ is non-zero. Since $m_{i-k,i-k+1}$ is also non-zero, $c_{i,i-k+1}$ is non-zero. So a transition from $i$ to $i-k+1$ is available in $C$. From $i-k+1$, state $i-1$ can be reached in at most $k-2$ steps. So we can reach state $i-1$ from state $i$ in at most $k-1 \leq n-2$ steps.

Thus we have shown that $C$ is irreducible. Q.E.D.

**Lemma 3:** Under the numbering of vertices given above, the SDI iteration matrix $C = N^{-1}M$ is aperiodic for a connected graph with at least three vertices.

**Proof:** From lemma 1 and lemma 2, we know that $C$ is an irreducible stochastic matrix. If we show that at least one state in $C$ has a transition to itself, then this is sufficient to establish that $C$ is also aperiodic [13].

From our construction, there is at least one pair of neighbors, say $i$ and $i-1$, that are numbered consecutively. It can be seen from the analysis of case 1 of lemma 2 that in $C$, $i$ has a transition to all neighbors of $i-1$, except possibly $i-2$. Consecutively, it has a transition to itself. Q.E.D.

**Theorem 1:** Under the numbering of vertices given above, the SDI iteration matrix $C = N^{-1}M$ is an irreducible, aperiodic, stochastic matrix for a connected graph with at least three vertices.

**Proof:** This follows from lemmas 1, 2, and 3.

## References


[1] C. Walshaw, M. Cross, Dynamic mesh partitioning and load-balancing for parallel computational mechanics codes, in: B. H. V. Topping (Ed.), Computational Mechanics Using High Performance Computing, Saxe-Coburg Publications, Edinburgh, 1999.

[2] Y. F. Hu, R. J. Blake, D. R. Emerson, An optimal migration algorithm for dynamic load balancing, Concurrency: Practice and Experience 10 (1998) 467–483.

[3] J. H. Curtiss, Monte Carlo methods for the iteration of linear operators, Journal of Mathematical Physics 32 (1954) 209–232.

[4] J. H. Curtiss, A theoretical comparison of the efficiencies of two classical methods and a Monte Carlo method for computing one component of the solution of a set of linear algebraic equations, in: Symposium on Monte Carlo methods, University of Florida, 1954, 191–233, John Wiley and Sons, New York, 1956.

[5] G. E. Forsythe, R. A. Leibler, Matrix inversion by a Monte Carlo method, Mathematical Tables and Other Aids to Computation 4 (1950) 127–127.

[6] J. H. Halton, Sequential Monte Carlo, Proceedings of the Cambridge Philosophical Society 58 part 1 (1962) 57–78.

[7] W. Wasow, A note on the inversion of matrices by random walks, Mathematical Tables and Other Aids to Computation 6 (1952) 78–78.

[8] V. A. Aggawal, Improving Monte Carlo linear solvers through better iterative processes, Master's thesis, Florida State University (2004).

[9] A. Srinivasan, Improved Monte Carlo linear solvers through non-diagonal splitting, Tech. Rep. TR-030203, Department of Computer Science, Florida State University, (2003).

[10] G. H. Golub, C. F. V. Loan, Matrix Computations, The Johns Hopkins University Press, Baltimore, 1996.

[11] D. E. Knuth, The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, Third edition, Addison-Wesley, Reading, Massachusetts, 1998.

[12] F. R. K. Chung, Spectral graph theory, American Mathematical Society, Providence, Rhode Island, 1997.

[13] T. H. Haveliwala, S. D. Kamvar, The second eigenvalue of the google matrix, Tech. rep., Stanford University (2003).