PARALLEL COMPUTATION OF THE DIAGONAL OF THE INVERSE OF A SPARSE MATRIX

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EDONA FASLLIJA

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JULY 2017

Approval of the thesis:

PARALLEL COMPUTATION OF THE DIAGONAL OF THE INVERSE OF A SPARSE MATRIX

submitted by **EDONA FASLLIJA** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. Murat Manguoğlu
Supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. Halit Oğuztüzün
Computer Engineering Department, METU _____

Prof. Dr. Bülent Karasözen
Department of Mathematics, METU _____

Assoc. Prof. Dr. Murat Manguoğlu
Computer Engineering Department, METU _____

Assist. Prof. Dr. Emre Akbaş
Computer Engineering Department, METU _____

Assist. Prof. Dr. Kayhan İmre
Computer Engineering Department, Hacettepe University _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to thiswork.

Name, Last Name: EDONA FASLLIJA

Signature:

ABSTRACT

PARALLEL COMPUTATION OF THE DIAGONAL OF THE INVERSE OF A SPARSE MATRIX

Fasllija, Edona

M.S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Murat Manguoğlu

July 2017, 75 pages

We consider the parallel computation of the diagonal of the inverse of a large sparse matrix. This problem is critical in many applications such as quantum mechanics and uncertainty quantification, where a subset of the entries of the inverse matrix, usually the diagonal, is required. A straightforward approach involves inverting the matrix explicitly and extracting the diagonal of the computed inverse. This approach, however, almost always is too costly for large sparse matrices since the inverse is often dense. In this thesis, we develop a novel parallel algorithm for computing the diagonal of the inverse based on the parallel DS factorization and approximate inverse techniques combined with a special structural dropping strategy step that exploits the peculiar sparsity pattern of the S matrix. We analyze the parallel scalability and performance of the proposed algorithm using sparse matrices from various applications.

Keywords: sparse matrices, inverse, diagonal, SPIKE

ÖZ

SEYREK BİR MATRİSİN TERSİNİN DİAGONALİNİN PARALEL OLARAK HESAPLANMASI

Fasllija, Edona

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Doç. Dr. Murat Manguoğlu

Temmuz 2017, 75 sayfa

Bu tezde büyük seyrek bir matrisin diagonalinin paralel olarak hesaplanmasını ele alıyoruz. Bu problem, bir matrisin tersinin elemanlarının bir alt kümesinin hesaplanmasını gerektiren kuantum mekaniği ve belirsizlik ölçümü gibi birçok uygulama için büyük önem arz etmektedir. Bu problemin en temel ele alış biçimi matrisin tersinin açıkça hesaplanmasının ardından diagonalinin alınmasıdır. Seyrek matrislerin tersinin yoğun olduğunu düşünürsek, bu yöntem bilgisayımsal olarak çok maliyetlidir. Bu tezde seyrek bir matrisinin diagonalini hesaplamak için paralel DS ayrıştırma ve yaklaşık ters hesaplama metodlarını kullanan yeni paralel bir algoritma geliştiriyoruz. Bunun yanında, S matrisin özel yapısından faydalanan yeni yapısal atma stratejisini kullanıyoruz. Geliştirdiğimiz algoritmanın farklı uygulamalardan seyrek matrisler kullanarak paralel ölçeklenebilirliğini ve performansını inceliyoruz.

Anahtar Kelimeler: seyrek matris, ters, diagonal, SPIKE

Dedikuar Eartës. E lindur me dashuri. E rritur me dashuri. E dashur gjithmonë.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my thesis advisor Assoc. Prof. Dr. Murat Manguoğlu for patiently steering me into the right direction during the research and writing of this thesis. Without his valuable support and guidance, this study could not have been successfully conducted.

Secondly, I would like to acknowledge the members of the committee, Prof. Dr. Halit Oğuztüzün, Prof. Dr. Bülent Karasözen, Assist. Prof. Emre Akbaş and Assist. Prof. Kayhan İmre, to whom I am gratefully indebted to for their very valuable comments on this thesis.

My sincere thanks go to my work colleagues at the EU Projects Office of TUBITAK for providing me with continuous support and encouragement throughout my years of study.

Finally, I would like to express my profound gratitude to my family, my parents Astrit and Liljana Fasllija, and my sister Ela Fasllija for supporting me throughout all my studies. Last but not least, my most sincere thanks go to my spouse Ardi Xhelilaj, and our little daughter Earta Xhelilaj, for providing me with unfailing support. This accomplishment would not have been possible without them.

The work of this thesis has been partially led under the support of METU BAP-08-11-2011-128 grant, on the Solution of Linear Systems on Parallel Computers.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ALGORITHMS

ALGORITHMS

LIST OF ABBREVIATIONS

GMR	Global Minimum Iteration Algorithm
GSD	Global Steepest Descent Algorithm
CMR	Column-Based Minimum Iteration Algorithm

CHAPTER 1

INTRODUCTION

The computation of the diagonal of the inverse of a matrix is crucial to many applications, examples of which include quantum mechanics, Uncertainty Quantification, Density Functional Theory, and Dynamic Mean Field theory. In this thesis, we aim to develop a novel algorithm for computing the diagonal of the inverse of a sparse matrix.

Given a large, sparse nonsingular matrix, $A \in \mathbb{C}^{n \times n}$, we are concerned about finding the vector that contains the same entries as the diagonal of the inverse of the matrix A, A^{-1}, which we denote by $diag(A^{-1})$.

Finding $diag(A^{-1})$ is considered challenging and computationally expensive since A^{-1} is usually dense even if A itself is sparse. This makes it impractical to invert the matrix explicitly and then extract the diagonal of the computed inverse.

This study is structured into two parts. In the first part, we start by introducing the problem of computing the diagonal of the inverse of a large sparse matrix. We study various academic and industry applications where the problem of the matrix inverse diagonal computation arises. We next examine a number of existing algorithms and techniques that have been developed for handling this problem. In the second part, we describe the details of the methodology by which the study was conducted. Experimental results and implementation considerations are presented in the second part followed by a summary of the findings and and conclusions drawn by the results of the experiments.

1.1 Applications of Computing the Diagonal of the Inverse of a Matrix

Throughout this study, we focus on the case where only a specific set of entries of the inverse of a sparse matrix, namely the diagonal, is needed. This problem is essential to, among others, the Dynamic Mean-Field Theory, the Density Function Theory in electronic structure calculations, and Uncertainty Quantification, which are also the motivation for the work of this thesis. The problem of finding the diagonal of the inverse mainly arises in the aforementioned applications in the following four different forms:

1. finding the diagonal of a matrix inverse, $diag(inv(A))$,
2. finding the diagonal of a matrix function $f(A)$, $diag(f(A))$,
3. finding the trace of a matrix inverse , $Tr(inv(A))$, or
4. or finding the trace of a matrix function $f(A)$, $Tr(f(A))$.

The following sections go through example applications for $diag(inv(A))$ first, and then give an insight on each of the other forms of the problem.

DMFT

Dynamic Mean Field Theory is a powerful tool in physics for studying quantum many-body systems, and more specifically in the analysis of lattice models of correlated electron systems. The problem of computing the diagonal of the inverse of a matrix arises in DMFT, specifically in quantum mechanical studies of highly correlated particles and the related Non Equilibrium Green's Function approach. The diagonal of a "Green's function", which solves Dyson's equation in a self-consistent way, is required to be computed in DMFT.[12]

The Dyson's equation that needs to be solved repeatedly and for many frenquencies ,ω's, is the following:

$$G(\omega) = \left[(\omega + \mu)I - V - \sum(\omega) + T\right]^{-1},$$

where μ is the chemical potential , V is the trap potential ,$\sum(\omega)$ is the local self-energy , and T is the hopping matrix. [12] In these settings, only the diagonal of $G(\omega)$ is of interest.

Uncertainty Quantification

Another important application where the problem of computing entries of a matrix inverse also arises, is uncertainty quantification. Uncertainty Quantification for Risk Analysis holds a key role for many different applications of Science, Engineering and Business, such as Geology, Portfolio Management, Astrophysics and Signal Processing, etc.[2] The computational challenge faced in these applications consists of analysing very large amounts of data, and answering one of the most important questions in Data analysis for Risk Management , i.e. the degree of confidence in the quality of data. The computation of the main diagonal of the inverse covariance matrices is of extreme importance to Uncertainty quantification for risk analysis, since these entries of the matrix inverse provide a very valuable measure for determining the extend to which one can confide in the quality of data.

In [26], a class of covariance functions, namely the piecewise polynomial covariance functions with compact support, is presented. Compact Support implies that the covariance for points with a distance greater than a certain treshhold between them becomes zero, hence leading to a sparse covariance matrix.

Denoting by $A = [a_{ij}]$ the covariance matrix that is computed by the such a function, by α the compact treshold, and by β the smootheness of the function, we obtain the following:

$$a_{ij} = \begin{cases} (1 - \frac{r}{\alpha})^{\beta} & \text{if } r \leq \alpha \\ 0 & \text{if } r > \alpha \end{cases} \tag{1.1}$$

where r denotes the Euclidean distance between points i and j given as $r(i,j) = \sqrt{i^2 + j^2}$.

Given that the covariance of points that have a distance greater than α betwwen them is equal to 0, the covariance matrix resulting from the expression above is sparse.

1.2 Applications of Computing the Diagonal of a Matrix Function

The problem of computing the diagonal of a matrix function is of paramount importance in Density Functional Theory (DFT). In this context, Density Matrices point to the key properties of the atomic systems. The computation or the approximation of

the diagonals of the density matrices is critical because the diagonal entries of these matrices represent the charge densities of the electronic distribution. There are three main categories of the approaches to this problem,

1. The traditional way of computing the diagonal by calculating the eigenstates associated with the energy levels. These algorithms are computationally expensive and have the disadvantage of scaling cubically with respect to the size of the Hamiltionian matrix of the system.

2. The O(N) methods described in [13] which rely on the decaying property of the density matrices and scale linealry with respect to the numbers of atoms.

3. Another approach that estimates the density matrix by expanding the Fermi-Dirac operator of the Hamiltonian Matrix on a basis of Chebyshev polynomials.

In the context of the third approach, the density matrix denoted by D can be expressed in the following form as

$$D = f(H) \tag{1.2}$$

, where H denotes the abovementioned Hamiltonian matrix, and the function f is the Fermi-Dirac operator for which the following expression holds:

$$f(\epsilon) = \frac{1}{1 + \exp(\frac{\epsilon - \mu}{K_B T})} \tag{1.3}$$

.

Here K_B,μ,T denote the Boltzmann's constant, the chemical potential, and the temperature respectively.

The problem described above eventually evolves into the computation of the diagonal of the density matrix D, otherwise formulated as $diag(f(H))$.

1.3 Applications of Computing the Trace of the Inverse of a Matrix

Applications of the problem of computing the trace of the inverse of a matrix include the Generalized Cross-Validation approach used in Image Restoration. In this context, regularized solutions of least-squares problems are to be solved and a certain regularization parameter ρ is to be estimated. In the Cross-Validation approach to

this problem, the parameter ρ is chosen so that it minimizes the Generalized Cross Validation function given as :

$$GVC = \frac{\|I - A(\rho)g\|_2}{tr(I - A(\rho))}, \tag{1.4}$$

, I being the identity matrix, and A is the $n \times n$ symmetric influence matrix given as

$$A(\rho) = I - D(D^T D + \rho L L^T)^{-1} D^T$$

, D and L being respectively the blurring and the regularization operator. Given that the expression of A includes the inverse of another matrix, $tr(I - A)$ can not be easily computed. Hutchinson in [15] describes an unbiased stochastic estimator for the calculation of the trace $tr(I - A)$. This trace estimator can be used for the approximate minimization of the GVC described, when A is the matrix that is associated with the Laplacian smoothing splines when fitting these splines to very large data sets.

1.4 Applications of Computing the Trace of a matrix function

The computation (or estimation) of the trace of a matrix function is required in several applications in Physics, Machine Learning, and more recently in Statistics, Data Analysis, and Signal Processing. The most widely used techniques are based on stochastic methods, such as stochastic estimations of $tr(f(A))$ for estimating Density of States (also known as spectral density) of a matrix A in Quantum Chemistry, as described in [27].

The main idea of this method is to use the Kernel Polynomial Method (KPM) for the purpose of computing (estimating) the Desity of States , by first expanding the Density of States into Chebyshev Polynomials, and then using statistical methods to estimate the traces that are required in the calculations. Here, the Hermitian Matrix A is rescaled such that its eigenvalues lie in the interval [-1, 1], which is also the interval in which the Chebyshev Polynomials $T_m(A)$ are defined. The coefficients (moments) of the expansion are then given by

$$\mu_m = Tr(T_m(A)) = \int_{-1}^{1} Tm(t)\rho(t)dt \tag{1.5}$$

where $\rho(t)$ denotes the Density of States (DOS). $Tr(C_m(t))$ is estimated by statistical methods for many m values, and these traces are eventually used to compute the density states of ρ.

Density of States are also used in many other applications in Physics, such as Solid-State Physics, where they represents the number of energy levels per unit.

However, the computation of the diagonal problem is more complex and less dealt with in the literature when compared to the problem of estimating the trace.

Existing Algorithms for Computing the Diagonal of Inverse

The algorithms for computing the diagonal of a matrix inverse range from the $\mathcal{O}(n^3)$ naive algorithm that extracts the diagonal after explicitly computing the matrix inverse, to direct techniques based on the standard LU factorization [10, 11, 21, 16] and iterative methods such as stochastic estimations[3] or inverse approximations[8]. Other approaches include probing methods[32], domain-decomposition methods[31], multi-frontal methods[6], heuristic methods[1] that make use of elimination tree, etc. We refer the reader to Chapter 2 for a more detailed description of the existing algorithms for the computation of the diagonal of a matrix inverse.

We, on the other hand, propose an algorithm that uses the parallel DS factorization technique to factorize the input matrix into D and S matrices in the first step, and then constructs the required diagonal of the inverse by using the inverses of these two matrices. The inverse of the S matrix is approximated via approximate inverse techniques that exploit its peculiar sparsity structure. The details of the proposed algorithm and its performance results obtained from various experiments are given in chapter 3 and 4, respectively.

CHAPTER 2

ALGORITHMS FOR COMPUTING THE DIAGONAL OF THE MATRIX INVERSE

The most obvious and straightforward approach is to compute explicitly the inverse of A, and then select its diagonal. This approach would have in the worst case a $\mathcal{O}(n^3)$ complexity , where n is the dimension of the matrix A. Improvements can be made by taking into consideration and taking advantage of the sparsity pattern of the A matrix via reordering techniques. While this approach may be favorable for 2D problems, 3D problems still present a challenge. Hence, there is a clear necessity for an efficient approach for computing the diagonal of the inverse of a sparse matrix. In the past decades, several methods have been developed in order to address this problem. The methods can be mainly categorized as :

1. Direct Methods [10, 16, 19]
2. Iterative Methods [4, 7, 18, 29]
3. Stochastic Methods [2, 3]

The primary purpose of using direct methods for solving linear systems is usually to obtain simpler systems to solve by factorizing the matrix A of a linear system $Ax = b$ into the product of two or more factor matrices and then use those to solve the system. These methods generally have four main stages: the *reordering phase*, *symbolic and numerical factorization*, and *triangular solves* stage.

The *reordering step* prepares the original matrix for the factorization phase that follows by applying a pretreatment to it, either numerical or structural. Structural pretreatments generally consist of reordering the rows and columns of the original matrix. The main aim is to reduce the number of the non-zero elements that show up in the factor matrices but are not present in the original matrix.

The *symbolic and numerical factorization phase* takes the matrix that results from the reordering phase and transforms it into a product of factor matrices. Examples of such factorizations include LU: L being a lower triangular matrix and U the upper triangular one,LDU LDL^T, LL^T , and Cholesky factorizations.

In the *triangular solves phase*, the linear system containing the factor matrices that were computed in the factorization phase is solved. Generally, this is done in two steps: the forward and backward substitution. In the case of LU factorization, the system is solved as

(i) $y = L^{-1}b$

(ii) $x = U^{-1}y$

In case a factorization is applied to the sparse matrix A, A^{-1} is usually dense when compared to the initial matrix or to the factors obtained from a certain factorization. Hence, computing the whole A^{-1} instead of using its factors, would result in a much less efficient method. Methods based on the standard LU factorization are one of the main approaches for computing elements of the inverse of A. They involve solving the equation $AX = I$ after computing the LU factorization of the matrix A, i.e. column-wise computation of the inverse of A. Hence, for computing the diagonal entries, the entire lower triangle of A^{-1} has to be computed.

These methods take as their basic reference the work done by Takahashi, Fagan, and Chin[30]. These authors were the first ones to use directly of the factors L and U instead of using their inverses.

Duff, Erisman, and Reid [10] proposed an algorithm for computing only the elements in the sparsity pattern of $(L\backslash U)^T$. They propose a direct method for finding $diag(B)$, where $B = A^{-1}$ based on the LDU decomposition of A. The LDU factorization is slightly different from the LU factorization, D is the diagonal matrix containing the diagonal elements of U. Denoting $B = U^{-1}D^{-1}L^{-1}$, the following relations due to Takahashi et al.[30] are utilized:

$$B = U^{-1}D^{-1} + B(I - L) \tag{2.1}$$

$$B = D^{-1}L^{-1} + (I - U)B \tag{2.2}$$

These equations enable the computation of entries of B that belong only to the sparsity pattern of the L and U matrices, with no further computation of any other entry of the

matrix. For example, the entries in the upper part of B can be computed by using only the U matrix, since for this part the expression $B(I-L)$, being lower triangular, does not contribute. The equations can then be rewritten for the computations of a single entry as follows:

$$b_{ij} = {d_{ij}}^{-1} - \sum_{k>i}^{n} u_{ik} b_{kj}, \quad for \quad i \leq j \tag{2.3}$$

$$b_{ij} = {d_{ij}}^{-1} - \sum_{k>j}^{n} u_{ik} b_{kj}, \quad for \quad i \geq j \tag{2.4}$$

Then started from the b_{nn} entry, the entries can be computed . The whole matrix can be computed using the equations above recursively. These algorithms can be beneficial for the cases when the LDU factorization of a matrix is not expensive, and does not require significant storage.

Erisman and Tinney [11] improved this algorithm further, by analyzing the special case of computing the diagonal of the inverse, when the A matrix is a symmetric matrix. They propose a method to compute a subset of entries of B based on the theorem given below.

Defining a matrix C to be as :

$$C_{ij} = \begin{cases} 1 & \text{, if } L_{ij} \text{ or } U_{ij} \neq 0 \\ 0 & \text{, otherwise.} \end{cases} \tag{2.5}$$

Hence the sparsity pattern of C is of the form $(L+U)^T$. The following theorem holds:

Any entry b_{ij} *such that* $c_{ji} = 1$ *can be computed as a function of L, U and* $b_{p,q}$ *where* b_{pq} *such that* $c_{qp} = 1, q \geq j, p \geq i$. This implies that the equation below can be constructed :

$$b_{ij} = -\sum_{k=i+1}^{n} u_{ik} b_{kj} \tag{2.6}$$

A different form of this approach is exploited by Lin Lin et. al.[21] They describe an algorithm called *SelInv* which computes selected entries of the inverse of a general sparse matrix A. The method is based on a left-looking supernodal approach that computes the LDL^T factorization of A. They exploit supernodes and block algorithms in order to achieve efficiency and performance. In addition, they propose to use a relative index array for handling indirect addressing.

Li, Ahmed, Darvem and Klimeck developed a computationally efficient algorithm named Fast Inverse using Nested Dissection (**FIND**) that is based on nested dissection [16]. The algorithm was then used to compute the required components of non-equilibrium Green's functions with an application on the simulation of nanoscale devices. This method consists of mainly three steps: (i) modelling an elimination process by using a graph structure, (ii) decomposing it into a tree structure (iii) traversing the tree in an upward and downward fashion to find the diagonal matrix that contains the same entries as the diagonal of A^{-1}, denoted as $diag(A^{-1})$.

A close relative to the abovementioned algorithms is the one proposed by Lin et al. in [19]. It describes a fast sequential algorithm to draw the diagonal or the subdiagonal elements of matrix inverses that are the output of a finite difference discretization process of a Laplacian operator or are extracted from lattice models with a local Hamiltonian. Denoting by n the dimension of the matrix A, it is seen that this approach lowers the complexity from $\mathcal{O}(n^3)$ to $\mathcal{O}(n^{1.5})$ and $\mathcal{O}(n^2)$ for 2D and 3D problems,respectively. . This fast algorithm, similar to *SelInv*, first computes the LDL^T factorization of A, and then uses L and D matrices in order to compute the desired entries of A^{-1}. In the first step of the algorithm, the factorization step, Lin et al perform a bottom up traversion and construct a hierarchy of Schur complements of the interior points for the blocks of the domain by using Block Gaussian elimination. In the second step they extract the diagonal of the inverse matrix in a top-bottom fashion by making use of the hierarchical dependence of matrix inverses.

In a more recent work[20], Lin describes the parallel version of this algorithm executed on a distributed memory machine. In the parallel algorithm, the elimination tree is used for organizing purposes. More specifically, the data is passed by level by level along the elimination tree by making use of local buffers and relative indices , and hence reducing the synchronization overhead. The fast algorithms decribed above, however, depend on the domain shape and discretization stencil for the Laplacian operator. Hence, the application of the algorithm is restricted.

Many investigations on the inverse of banded and tridiagonal matrices have been conducted. Ran and Huang developed an algorithm for the inversion of a banded matrix by first employing 'twisted' decompositions of matrices[25]. Based on the count of arithmetic operations performed, the method is two times faster than the standard approach of using direct methods based on the LU factorization. The complexity is $\mathcal{O}(nb)$, b being the bandwidth of the matrix A. Bowden developed a method that consists of defining a set of matrix sequences for the calculation the inverse of any

block-tridiagonal matrix efficiently [5]. They use recurrence relations for defining four sequences of matrices and eventually define an expression for the computation of any block(i,j) of matrix B.

Several sparse approximate inverse techniques have been proposed for the approximation of A^{-1}. Approximate inverse methods that are based on Frobenius norm minimization are one subset of these techniques.

The approximate inverses of general sparse matrices are computed by attempting to find a sparse matrix, denoted by M, that minimizes the Frobenius norm of the residual matrix, I - AM,

$$F(M) = \|I - AM\|_F{}^2 \tag{2.7}$$

The objective function $F(M)$ can be decoupled into the sum of the squares of the 2-norms of the individual columns of the residual matrix I - AM[8], as follows:

$$F(M) = \|I - AM\|_F{}^2 = \sum_{j=1}^{n} = \|e_j - Am_j\|_2^2, \tag{2.8}$$

in which e_j is the j-th column of the Identity matrix I, m_j is the j-th column of the matrix M.

Approximate Inverse Algorithms minimize the aforementioned function in two different ways, by minimizing it globally as a function of the sparse matrix M, or alternatively by minimizing the individual functions

$$f(j) = \|e_j - AM_j\|_2^2, \quad j = 1, 2, 3, \ldots, n. \tag{2.9}$$

The Global Minimal Residual Algorithm and the Global Steepest Descent algorithms follow the first approach. They describe global techniques to minimizing the objective function $F(M)$ by treating M as an unknown sparse matrix, while Column-based Minimum Residual algorithm approximates the matrix M by minimizing the individual functions mentioned above.

Another approach for approximate inverse algorithms are the ones based on the Newton-Schulz iterations. These methods are also referred as Hotelling-Bodewig algorithms. The well-known method of Schulz[14] is defined by

$$V_{n+1} = V_n(2I - AV_n), n = 0, 1, 2, ...n \tag{2.10}$$

for the inversion of a matrix $A \in \mathbb{C}_{m\times m}$. This approach has been used combined

with wavelets or hierarchical matrices in order to find the diagonal of V in [18]. Ceriotti et al.[7] also proposed a method for the computation of the Fermi function of the Hamiltionian that combines polynomial expansion techniques with Newton-like iterative approaches.

In both of the subsets of approximate inverse algorithms, a numerical dropping step is usually applied in order to keep the approximated matrix sparse, which may affect the convergence of the algorithms.

In [29] Saad and Sidje show how one can use the iterative Lanczos algorithm together with sparse direct methods for approximating the diagonal of the Fermi-Dirac matrix function. The Lanczos vectors for a Hermitian matrix A can be generated using the following expression:

$$\beta_{i+1}q_{i+1} = Aq_i - \alpha_i q_i - \beta_i q_{i-1}, \tag{2.11}$$

where α_i and β_{i+1} are chosen in such a way that the Lanzcos vector q_{i+1} has a unit norm and is orthogonal to q_i and q_{i-1}. This method applies m steps of the Lanczos algorithm on the Hamiltonian matrix H, usign a unit norm vector q_1 as a starting vector. As a result, the following expression is obtained:

$$HQ_m = Q_m T_m + \beta_{m+1} q_{m+1} {e_m}^T, \tag{2.12}$$

where Q_m is given as $Q_m = [q_1, ..., q_m]$, q_{m+1} denotes the last vector obtained from the Lanczos algorithm and e_m is a vector with a entry equal to 1 at the m-th position and 0 elsewhere, T_m denotes a tridiagonal symmetric matrix given as $T_m = tridiag[\beta_i, \alpha_i, \beta_{i+1}]$, with nonzero entries $\beta_i, \alpha_i, \beta_{i+1}$ at the i-th row.

Taking n to be the iteration number, when $m = n$ then

$$A = Q_n T_n {Q_n}^T \tag{2.13}$$

and

$$A^{-1} = Q_n {T_n}^{-1} {Q_n}^T. \tag{2.14}$$

For $m < n$ the expression above can be approximated as: $A^{-1} \approx Q_m {T_m}^{-1} {Q_m}^T$ and hence the diagonal can be computed via:

$$diag(A^{-1}) \approx diag(Q_m {T_m}^{-1} {Q_m}^T) \tag{2.15}$$

can be used. The resulting algorithm has the disadvantage that it usually requires all of the n iterations to converge an accurate approximation of the diagonal of A^{-1}.

In [32] Tang and Saad propose a probing method for the computation of the diagonal of the inverse of a sparse matrix for the special case where the inverse exhibits a decay property. In generic probing approaches , a subset or all of the entries of an unknown matrix are extracted by using matrix-vector products. These methods prove to be quite effective when the initial matrix A is diagonally dominant and/or positive definite, which leads to A^{-1} being composed of many small entries. This method consists of the following steps : (i) Small entries of A are discarded, hence A is sparsified, (ii) the sparsity pattern of the sparsified A^{-1} is determined, (iii) appropriate probing vectors are determined based on the properties of the inverse using graph coloring arguments, and finally (iv) linear systems are solved either via direct or iterative methods. The diagonal of the inverse of the matrix is extracted from these linear systems.

Stochastic methods can also be used for the estimation of the diagonal of a matrix inverse. In [3] Bekas and Saad extended the idea of using an unbiased stochastic estimator for estimating tr(I-A), using random vectors described by Hutchinson in [15] to find $diag(B)$. They apply matrix-vector products using random test vectors to estimate the diagonal. This solution can achieve a desired accuracy only if the number of vectors used is large enough, hence making the methods more expensive from a computational point of view. Bekas at al. also described an approach that can be thought of as a probing method for banded approximate inverses that makes use of Hadamard vectors in [3]. These vectors are dense and do not exploit the sparsity pattern except for the bandedness.

More recently, Bekas, Curioni and Fedulova [2] retrieve the diagonal entries of the inverse of the covariance matrix by using a minimum bias stochastic estimator. A quite accurate estimate of the diagonal is obtained by means of a few matrix vector products that involve the inverse covariance matrix and specially designed vectors.

Campbell and Davis [6] describe a multifrontal method for computing a set of the entries of the inverse for the case where of the matrix is numerically symmetric. This method also exploits the Takahashi's equations mentioned above. Multifrontal methods are a variant of direct methods, that consist of a multifrontal factorization and a multifrontal solution phase. The method they proposed exploits the elimination tree of the input matrix and Level 3 BLAS matrix-matrix computations.

Amstoy et al. [1] also proposed a method to compute entries of the inverse. Their approach relies on a traditional solution method and the exploitation of the equation $AA^{-1} = I$. They describe a heuristic method that applies a post-ordering step on the elimination tree followed by a partitioning step of the nodes. More specifically, a

particular entry a_{ij} is computed using $(A^{-1}e_j)_i$.

In the case of computing an entry of the diagonal, i.e. for finding ${a_{ii}}^{-1}$:

1. solve $x = L^{-1}e_i$ for x
2. solve $y = U^{-1}x$ for y
3. compute the i-the value ${a_{ii}}^{-1} = {e_i}^T y$

For each diagonal entry ${a_{ii}}^{-1}$ that is to be computed, the following steps are followed:

i. traverse the elimination tree from the node i to the root by accessing only necessary parts of L at each node,

ii. traverse the tree from the root to the node i by accessing only necessary parts of U this time around.

Lastly, domain-decomposition techniques were also investigated for the purpose of computing the diagonal of the inverse of a sparse matrix in [31]. Tang and Saad describe two domain-decomposition methods, both of which can achieve better efficiency and performance if used together with iterative solvers and approximation approaches. The first method is based on the divide-and-conquer concept. It applies recursively the Sherman-Morrison-Woodbury formula. The main idea of the method can be described as follows. An assumption is made on the decomposition of a non-singular and complex symmetric matrix $A \in \mathbb{C}^{n\times n}$ into two matrices, the first being a 2×2 block-diagonal matrix C, and the latter a low-rank matrix $-L$. Denoting the dimension of first block of C and the rank of L by m and q respectively, the following expression holds:

$$\begin{aligned} A &= C - L, \\ C &:= \begin{bmatrix} C_1 & \\ & C_2 \end{bmatrix}, \\ L &:= EE^T \end{aligned}$$

where C_1 and C_2 are the diagonal blocks of C of size $m \times m$ and $(n-m) \times (n-m)$ respectively, and $E \in \mathbb{C}^{n\times q}$, with $0 < q < m < n$. Making the appropriate substituions in the Sherman-Morrison-Woodbury formula, the diagonal of the inverse can be found via

$$diag(A^{-1}) = diag(C^{-1}) + diag(UR^{-1}U^T) \tag{2.16}$$

with $U \in \mathbb{C}^{n\times q}$ equal to $U := C^{-1}E$ and $R \in \mathbb{C}^{q\times q}$ equal to $R := I_q - E^T U$.

The latter method is a standard domain decomposition method where local solves and global correction are used together. After possible row and column permutations, a nonsingular and complex-symmetric matrix A has the form of:

$$C =: \begin{bmatrix} B & F \\ F^T & G \end{bmatrix},$$

where B is a block-diagonal matrix of blocks $B_j \in \mathbb{C}^{n_j \times n_j}$, $G \in \mathbb{C}^{n_G \times n_G}$, and F is composed of the bottom and rightmost blocks $F_j \in \mathbb{C}^{n_j \times n_g}$ with $n_G + n_B = n$ and $n_B := \sum_{j=1}^{p} n_j$.

Inverting both sides of the equation above, the following expression for the computation of the diagonal can be derived:

$$diag(A^{-1}) = \begin{bmatrix} diag(B^{-1}) + diag(HS^{-1}H^T) & \\ & diag(S^{-1}) \end{bmatrix}$$

,

$$H = B^{-1}F$$

and

$$S := G - F^T B^{-1} F$$

Despite the promising potential of the methods described in this section, many limitations exist. Some of these limitations consist of high computational cost, difficult implementation, inaccuracy of the solution, and high memory consumption. Moreover, not much has been done for a parallel solution of the problem. Hence there is a need to explore more novel algorithms for finding the diagonal of the inverse of a sparse matrix in parallel.

CHAPTER 3

COMPUTING THE DIAGONAL OF THE INVERSE USING DS FACTORIZATION

We propose a novel parallel algorithm for computing the diagonal of the inverse of a large sparse matrix. The proposed method is a hybrid method in the sense that it uses both direct and iterative methods that were briefly described in the previous chapter. This method pre-computes a less known parallel factorization instead of the commonly used standard LU factorization: the parallel DS factorization [23] of the matrix, and then uses the D and S matrices to compute the diagonal components of A^{-1} via sparse approximate inverse.

We first give the mathematical formulation of the problem of computing the diagonal of the matrix inverse, and then go through the details for each step of the proposed algorithm.

Proposed Method Formulation

The problem to be solved is formally described below: Suppose $A \in \mathbb{C}^{n \times n}$ is a nonsingular matrix, i.e. its inverse exists. We are interested in computing $diag(A^{-1})$, the diagonal of A^{-1}.

Given that e_i is the i-the vector of the canonical basis, (i.e. all of the entries of e_i are 0, except for the i-th entry, which is 1):

The i-th entry of the diagonal $diag(A^{-1})$ can then be calculated as follows:

$$a_{ii}^{-1} = e_i^T A^{-1} e_i, \tag{3.1}$$

where e_i^T is the transpose of vector e_i. In order to retrieve the complete diagonal of the matrix, n linear systems of the form

$$Ax_i = e_i, \quad i = 1, 2, \ldots, n$$

must be solved. The i-th entry of the diagonal will then be equal to

$$a_{ii}^{-1} = e_i^T x_i.$$

To achieve a less computationally expensive solution, we first compute the DS factorization of the sparse matrix A, where D consists of the the diagonal blocks of the A matrix, A_j, and S is the spike matrix with identity matrices as diagonal blocks, together with spikes of m columns in the adjacent non-diagonal blocks. Then the expression above can be rewritten as:

$$a_{ii}^{-1} = e_i^T S^{-1} D^{-1} e_i$$

where

$$D = \begin{pmatrix} A_1 & & & & & \\ & A_2 & & & & \\ & & A_3 & & & \\ & & & \ddots & & \\ & & & & \ddots & \\ & & & & & A_p \end{pmatrix}$$

.

There are three different ways of parenthesizing this equation. Our approach corresponds to the following parenthesization:

$$a_{ii}^{-1} = e_i^T (S^{-1} (D^{-1} e_i)).$$

Hence the equations to be solved this time around are of the form:

(i) solve $Dy_i = e_i$

(ii) solve $Sx_i = y_i$, or $x_i = S^{-1} y_i$

(iii) $a_{ii}^{-1} = {e_i}^T x_i$

After DS factorization, step (i) involves a linear system with a coefficient matrix D, which involves analysis and factorization. Then, in step (ii), we use an iterative

method to compute the approximate inverse of S. We compute the reverse Cuthill McKee ("RCM") ordering of the input matrices in the analysis phase before proceeding with the DS factorization.

We notice that in step (i), there is a single non-zero entry on the RHS e_i, and in step (ii), only one entry is required to be computed from the solution vector. We take advantage of this observation and the sparsity structure of the matrix for a more efficient solution of the problem.

To compute the DS factorization, the original matrix A, or the reordered matrix Ar, is factorized into the product of two matrices, namely D and S, using the sparse DS factorization which is based on the SPIKE algorithm [24]. The SPIKE algorithm was originally developed as a hybrid solver for banded systems of linear equations. It consists of the preprocessing (partitioning), factorization and post-processing (solution) phases. In this algorithm, the solution to the sparse $AX = F$ system is considered, where A is a banded $n \times n$ matrix of bandwidth $b << n$, and F is a $n \times s$ RHS matrix, s being the number of right-hand-sides.

The banded linear system is partitioned into a block tridiagonal form [24]. The original matrix is partitioned into blocks in the main diagonal which are denoted by A_j,$(j = 1, .., p)$. These banded diagonal blocks are of size $n_j \times n_j$, where n_j is calculated approximately as n/p, and p is the number of blocks.

Besides the diagonal blocks A_j, B_j denotes the first super-diagonal block and C_j matrix the first sub-diagonal blocks that couples the A_j respectively. Each of these blocks is of order $m << n_j$. m is the semibandwidth and is calculated based on the bandwidth b of the original matrix as $\frac{b-1}{2}$ for structurally symmetric matrices.

A can be then expressed in the form of the product of two factor matrices, namely the D and S matrix.

The D matrix is the block-diagonal matrix composed of the diagonal blocks A_j. The S matrix, given by $D^{-1}A$, is the 'spike' matrix, which is a matrix whose block main diagonals are identity matrices of order n_j, together with spikes of m columns in the immediate off-diagonal blocks, hence the name 'Spike'.

These spikes are denoted as W_j and V_j and , and correspond to the right and left spikes of each partition, each of size $n_j \times m$. The next stage of the algorithm is the solution of the system which consists of two steps:

(i) $DG = F$

(ii) $SX = G$

After performing the partitioning, the system in step (i) can be solved in parallel using a number of options. When solving the system in step (ii), a further reduction can be performed by identifying the independent reduced linear system, resulting in a much smaller system of equations $S_r X_r = G_r$ to be solved, from which the original solution X can be restored.

In summary, the SPIKE algorithm consists mainly of the following three stages:

1. factorizing of the diagonal blocks A_j
2. solving the reduced system
3. retrieving the solution

To compute the DS factorization, we use the generalized sparse DS factorization algorithm [22].

The Generalized DS factorization is similar to the original SPIKE algorithm in the sense that it performs the DS factorization and partitioning of the matrix A , but it does not assume a banded structure for the coefficient matrix A. The reordering step is no longer needed in this case.

In the General DS factorization, similar to the banded DS factorization, given a general sparse linear system:

$$Ax = f,$$

the coefficient matrix A is again partitioned into p block rows $A = [A_1, A_2, \ldots A_p]^T$. R is defined as $A = D + R$, where D consists of the p block diagonals of A, and R is the matrix that contains the remaining elements of A. The S matrix is defined as $S = D^{-1}A$

Replacing $A = D + R$ in the equation above we obtain

$$S = D^{-1}A = D^{-1}(D + R) = I + D^{-1}R.$$

The S matrix that results from the equation above is not banded any more and it contains sparse spikes.

Although this is the basis algorithm, there are many alternatives available for each step, giving rise to a ́familyóf solvers. We refer the reader to [24] for a more detailed description of the existing variations of the algorithm. Our proposed approach applies the general DS factorization not to the $AX = F$ system, but to the $AX = I$ system, to solve for X, the inverse of the A matrix and then extracting the diagonal: $diag(X)$ by computing the inverses of the D and S matrices.

In the *solution step*, The solution of the system $AX = I$ reduces into two sub steps:

(i) solve $DG = I$, or $G = D^{-1}I$

(ii) solve $SX = G$, or $X = S^{-1}G$

The inverse of the D matrix can be computed with perfect parallelism, since the inverse of a block diagonal matrix is composed of the independent inverses of the diagonal blocks A_j. The computation of the Spike Matrix S can also done blockwise in order to suit the parallel nature of the factorization.

Next, we iteratively approximate the inverse of the spike matrix S, via appropriate inverse approximation techniques for sparse matrices. Note that, since we approximate the inverse of the S matrix, there is no need for solving a reduced system.

After the sparse DS factorization, the next step involves solving $DG = I$, or expressed in a different form as $G = D^{-1}I$ .The inverse of the block-diagonal matrix D given as $D = diag(A_1, A_2, \ldots A_p)$ can be computed blockwise, in parallel, by solving the following linear system of equations:

$A_j X_j = I_j,$

where j is the partition number, I_j is the identity matrix of the size n_j of the diagonal blocks of A, $\frac{n}{p}$.

For the further computation of a single entry of the diagonal, the equation above can be reduced to

$Dx_i = e_i$, or as individual block linear systems, $A_j x_{j_i} = e_{ij}$

First, we will investigate the effectiveness of various algorithms form computing the sparse approximate inverse of S. Then, we will study the effectiveness of the pro-

posed scheme for computing the diagonal of the inverse of a sparse matrix. The pseudocode that includes the main steps of the proposed algorithm can be found below:

Algorithm 1: Computation of the diagonal of the inverse

- Factorize the A matrix into D and S matrix
- Compute D^{-1}
- Approximate S^{-1}
- Compute the diagonal of A^{-1} using D^{-1} and S^{-1}

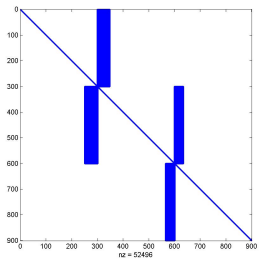
Approximating S^{-1}

In order to determine the strategy for the computation of the inverse of the SPIKE matrix S, first the sparsity pattern of the original S matrix and its inverse were observed.

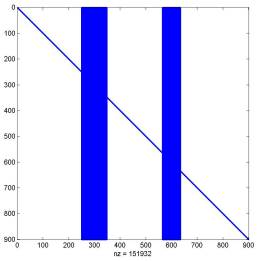
The sparsity pattern of S and S^{-1}

It was already mentioned that the matrix S has $n_j \times n_j$ identity matrix on the main diagonal blocks, with spikes of m columns in the immediate off-diagonal blocks. The inverse of the S matrix, keeps the same identity matrices as diagonal blocks, but the spikes this time, instead of being the order $n_j \times m$, become the order $n \times m$. In the inverse of the S matrix these spikes are extended throughout whole columns. The spy plots for the S and S^{-1} matrices obtained from the gr_30_30 matrix of the Collection of Sparse Matrices of the University of Florida are depicted below.

Figure 3.1: Sparsity Pattern Plots of S and S^{-1} obtained from gr_30_30



(a) Sparsity structure of S



(b) Sparsity structure of reordered S^{-1}

Given that the sparsity structure of the inverse to be computed can be known beforehand, we use sparse approximate inverse algorithms in order to compute the inverse of the spike matrix S. These algorithms have many parameters and in the following section we study various algorithms, as well as their parameters.

Sparse Approximate Inverse

Recall that in general the approximate inverse methods based on the minimization of the Frobenius norm seek to find an approximate inverse for a sparse nonsingular matrix by using iterative procedures. Without loss of generality, we assume the right aproximate inverse.

The expression of the minimization of the residual $I - AM$ was given as :

$$F(M) = \|I - AM\|_F{}^2 = \sum_{j=1}^{n} = \|e_j - Am_j\|_2^2. \tag{3.2}$$

The equivalence expressed in the equation above gives rise to a series of options to be followed. There are basically two options for iterative procedures for approximating the inverse of a general sparse matrix,

1. Treating M as an unknown matrix entirely
2. Treating M as individual columns $m_j, j = 1, 2, \ldots n$

The Global Iteration methods follow a global approach to minimizing 3.2, by treating M as an unknown sparse matrix. The function 3.2 is associated with the following

inner product of matrices:

$$\langle X, Y \rangle = tr(Y^T X)[8]. \tag{3.3}$$

The column-oriented algorithms are based on minimizing the individual objective functions of the expression 3.2, by treating individual columns of M $m_j, j = 0, 1, 2 \ldots n$.

n_i iterations are used to approximately solve the n linear subsystems $Am_j = e_j$ for each column of M.

A series of decent steps are taken in a search direction denoted as G, each of them defining a new iterate M_{new}, as follows:

$$M_{new} = M + \alpha G \tag{3.4}$$

There are options for the selection of the search direction G. The most obvious one is choosing G to be equal to the residual matrix $R = I - AM$. This choice leads to the Global Minimum Residual Iteration Algorithm. Another option for the selection of G is to take it to be the direction of steepest descent, or $G = A^T R$. Taking this direction for G leads to the Global Steepest Descent Algorithm.

Based on various options for the criteria described above, three different algorithms were inspected for the purpose of approximating the inverse of the S matrix. Two of those algorithms were investigated under this global iteration category: the Global Minimum Residual Algorithm and the Global Steepest Descent algorithms. The pseudocodes for these two approaches are given in Algorithm 3 and 4, respectively.

Algorithm 3: Global Minimal Residual Descent Algorithm

Input : initialM, stopTol, dropTol

Output: M

1: Select an initial M
2: Until convergence Do:
3: Compute $C := AM$ and $G := I - C$
4: Compute $\alpha = \frac{tr(G^T AG)}{tr((AG)^T (AG))}$
5: Compute $M := M + \alpha G$
6: Apply numerical dropping to M
7: End Do

We note that Global Minimum Residual Iteration algorithm is guaranteed to converge if A is SPD.

The difference of the Global Steepest Descent algorithm from GMRES is the direction G that is selected: GSD algorithm take G as the direction of the steepest descent, i.e. the direction opposite to the gradient. In this case $G = A^T R$, where R is the residual matrix. Global Steepest Descent algorithm is guaranteed to converge if A is nonsingular.

Algorithm 4: Global Steepest Descent Algorithm

Input : initialM, stopTol, dropTol

Output: M

1: Select an initial M
2: Until convergence Do:
3: Compute $R = I - AM$,and $G := A^T R$
4: Compute $\alpha = \frac{tr(G^T AG)}{tr((AG)^T (AG))}$
5: Compute $M := M + \alpha G$
6: Apply numerical dropping to M
7: End Do

Under the category of Column-based iteration Algorithms, the column-based version of GMR is described by the Column-Based Minimum Residual Iteration Algorithm. The pseudo code is given in Algorithm 5:

Algorithm 5: Column-Based Minimum Residual Iteration Algorithm

Input : initialM, stopTol, dropTol

Output: M

1: Start: set $M = M_0$
2: For each column $j = 1, ...n$ Do:
3: Define $m_j = Me_j$
4: For $i = 1, ..., n_i$ Do:
5: $r_j := e_j - Am_j$
6: $\alpha_j := (r_j, Ar_j)/(Ar_j, Ar_j)$
7: $m_j := m_j + \alpha_j r_j$
8: Apply numerical dropping to m_j
9: End Do
10: End Do

This algorithm is guaranteed to converge if the original matrix A is positive definite.

The Initial Guess

All three of the aforementioned approximate inverse algorithms need an initial guess M_0 in order to start their iterative process. In the column-based case, columns from M_0 are used as initial guesses for the solution of each subproblem. There exist two main options for the selection of M_0 : namely $M_0 = \alpha I$, where I is the Identity matrix of size n, or $M_0 = \alpha A^T$. The coefficient α is chosen in a way that minimizes the spectral radius $\rho(I - \alpha M)$ as follows:

$$\alpha = \frac{tr(AM)}{tr(AM)(AM)^T} \tag{3.5}$$

Note that the second one of choosing the transpose of the original matrix as the initial guess is a more dense option when compared to the Identity matrix as initial guess. Additional analysis on the choice of initial guess is presented in the experimental work section.

The Dropping Strategy

In the previous sections, it was already mentioned that, as the iterative process progresses, the matrix M tends to become more and more dense, hence making the computations more and more expensive. A dropping strategy is needed in order to preserve the computational efficiency of the methods. There are several directions to take as a dropping strategy. Numerical dropping is the most obvious one. A new dropping strategy is developed based on the special structure of the S matrix inverse and explained in more detail in the following section. We first go through the traditional numerical dropping strategy that is performed in the Approximate inverse iterative methods and then explain the new dropping strategies introduced in this thesis, namely the *structural* and *hybrid* dropping.

Numerical Dropping

A numerical dropping step can be performed in two possible places, either the iterate M, or the search direction G. Two criteria can be used for determining a treshold for the elements to be dropped: a dropping tolerance, or a maximum number of fill-ins per column. In the first case, elements of M that are smaller than a tolerance value, which do not contribute much to the eventual approximate inverse, are discarded.

The time at which the numerical dropping is done is another factor to be taken in consideration. Dropping performed after the new iterate M is computed, or before the new iterate M is updated, are some of the alternatives.

If the dropping is applied to the iterate M, the descent property of the algorithm is lost, in other words there is no guarantee that the new iterate will have a smaller Frobenius norm than the previous one. This phenomenon is known to *spoil* the minimization process, since the residual norm increase at a certain point, and affect the quality and convergence of the methods.

In the latter case of performing the numerical dropping step in the search direction G, the descent property of the algorithm is maintained through the iterations. In this case, the fill-in of the matrix M becomes difficult to control.

Structural Dropping

Previously, the structure of the inverse of the Spike matrix S was investigated. Given that the sparsity pattern of the inverse to be approximated can be known beforehand, a new dropping strategy besides the numerical one described above, namely structural dropping strategy was developed. In this dropping strategy, elements that are outside the sparsity pattern of S^{-1} are dropped or discarded. In the case of global iterations where M is treated as an unkown matrix in its entirety, this dropping is performed to the iterate M after it is updated. For the column-oriented case, the columns outside the pattern of the spike columns are not traversed, in other words, no iterative process is applied to these columns.

Hybrid Dropping

A combination of the two abovementioned strategies was also analysed with the approximate inverse algorithms, namely a hybrid dropping strategy. Using this strategy, the Global iteration algorithms, perform two steps of dropping: first entries outside the sparsity pattern are dropped, then the entries smaller than a dropping tolerance value inside the sparsity pattern of S^{-1} are discarded. In the column oriented case, the columns outside the spike pattern are also not traversed, while during the iterations for the columns inside the spike pattern, a numerical dropping step is performed to the vector m_j after it is updated.

Other considerations

The iterative algorithms stop when the residual norm, $\|I - AM\|_F$ in the global case, or $\|ej - Am_j\|_2$ in the column-oriented case, is less than or equal to a certain stopping tolerance value. Another alternative is setting a maximum number of iterations, or both of the above.

In order for the abovementioned algorithms to be efficient, all the storing and operating of matrices and vectors must be perfomed by treating them as sparse matrices. In this sparse implementation, matrix-vector products are much more efficient if the sparse matrix is stored by columns. In this way, not all the columns need to be traversed in order to approximate the inverse. This is especially useful for the Column-Oriented Minimum Residual Algorithm, which approximates each column of the sparse matrix at a time.

Parallelization of the Proposed Method

Both of the strategies selected for each one of the two main phases of the proposed method, the DS factorization and the approximate inversion phase, were chosen by keeping in mind their potential for parallelization.

Most existing parallel solvers of systems of linear equations are based on the LU decomposition, and consequently inherit the limits that the nature of this factorization brings. The DS factorization was originally developed for parallel solvers of banded systems of linear equations. The Spike Algorithm itself relies on the idea of partitioning the matrix in such a way that the communication between processes is necessary only during the solution phase, and each processing element can work on its partition of the matrix independently during the other stages.

The Column-based Approximate Inverse Algorithm is chosen as the best option for the parallelization of the proposed method. Given that it computes the inverse of the matrix S by iterating a certain number of iterations n_i column-by-column, each column's inverse can be calculated independently without the need of any communication.

CHAPTER 4

EXPERIMENTAL WORK

We conduct two main sets of experiments throughout this study. In the first set, we study the approximate inverse algorithms for computing the inverse of S (spike) matrix. We investigate the effect of the various alternatives that we explain in detail in Chapter 3, such as dropping tolerance, initial guess, and dropping strategy has on the number of iterations, time to convergence, and the accuracy of the diagonal computed using the approximated inverse of S. Three algorithms are then compared against each other in order to find on the most appropriate and efficient one. In the second set of experiments, we study the parallel scalability of the proposed algorithm that uses the method that was studied in the first part.

4.1 Experimental Framework

For the first set of experiments, we implemented and tested the Approximate Inverse Algorithms using MATLAB R2012a. We executed the tests on a 4-core Intel(R) Core i7 processor running at 2.4GHz with a total of 8GB RAM memory.

We run the second set of tests on Greyfurt, which is a 64-core computer that has a shared memory architecture with 4 16-core AMD Opteron 6376 processors running at 2.3GHz with a total of 128 GB DDR3 memory. We implemented the parallel algorithm using in C++ with OpenMP directives. We used the Armadillo [28] and SuperLU[17] libraries in order to perform matrix operations.

For the first set of tests, we implemented the Global Minimum Residual Iteration algorithm (GMR), Global Steepest Descent algorithm (GSD), and the Column-based Minimum Residual Iteration algorithm (CMR). We give the pseudocodes for the algorithms in Chapter 3. We test the algorithms with a set of predefined parameters, such as maximum number of iterations, targeted stopping tolerance, dropping strat-

egy, etc. We examine the performance of these algorithms in terms of number of iterations that is required to meet the desired stopping tolerance, the final residual norm defined as the Frobenius norm of (I - AM), error history throughout iterations, the time required, and the sparsity of the resulting matrix inverse. In the latter set, we focus on measuring the time of execution for each of the substeps of the proposed algorithm using 1, 2, 4, 8, 16, 32 and 64 cores.

We use the matrices in Table 4.1. in the aforementioned sets of experiments. The matrices are obtained from the UF Sparse Matrix Collection [9]. The properties of the matrices used are given in the table below. The property of S being positive definite is important for the convergence of some approximate inverse algorithms. Therefore, we choose the test matrices such that the resulting S matrix is positive definite even though A may or may not be positive definite.

Table 4.1: Input Matrices for Approximate Inverse Algorithms

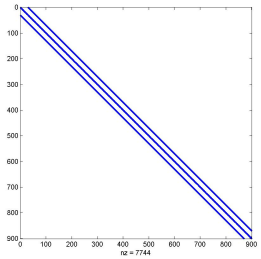
Matrix Name	rows	cols	nonzeros	A posdef	S posdef
bfwb782	782	782	7514	no	yes
bfwb62	62	62	450	no	yes
cage6	93	93	785	no	yes
cage7	340	340	3084	no	yes
dw256B	512	512	2500	no	yes
gr_30_30	900	900	7744	yes	no
mcca	180	180	2659	yes	yes
mesh1e1	48	48	306	yes	yes
mesh1em6	48	48	306	yes	yes
Trefethen_150	150	150	2040	yes	yes

A matrix is positive definite if its eigenvalues are positive.

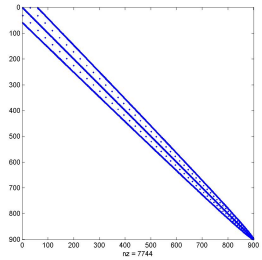
Sparsity Patterns of Some Input Matrices

Now we plot the factor matrices resulting from the DS factorization, namely D, D^{-1}, S and S^{-1} are plotted in order to investigate their sparsity patterns. In order to show the general trend, the plots for *gr_30_30* and *dw256B* matrices partitioned into $p = 3$ and $p = 2$ partitions respectively are given in Figures 4.1a - 4.1e and 4.2a -4.2e, respectively. As it can be seen, the sparsity of S^{-1} follows the sparsity pattern of S. The spikes in S are just extended in S^{-1} but the nonzero columns remain the same.

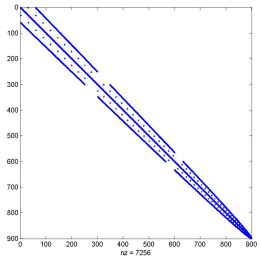
Figure 4.1: Sparsity Pattern Plots of gr_30_30



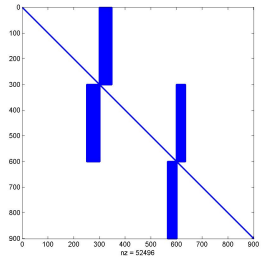
(a) Sparsity structure of A



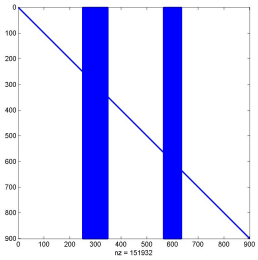
(b) Sparsity structure of reordered A



(c) Sparsity structure of D

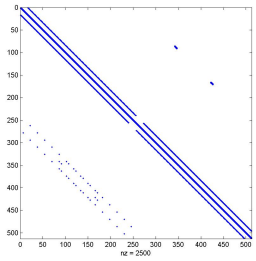


(d) Sparsity structure of S

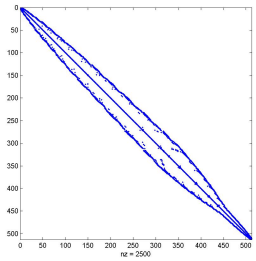


(e) Sparsity structure of the S^{-1}

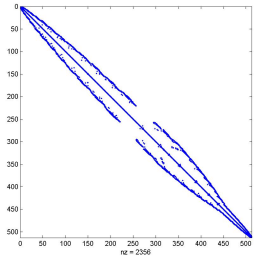
Figure 4.2: Sparsity Pattern Plots of dw256B



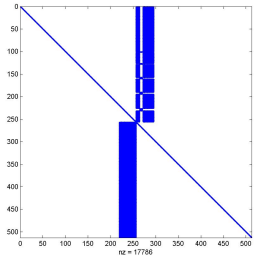
(a) Sparsity structure of A



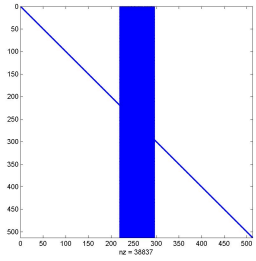
(b) Sparsity structure of reordered A



(c) Sparsity structure of D



(d) Sparsity structure of S



(e) Sparsity structure of S^{-1}

4.2 Approximate Inverse Algorithms Experimental Results

In this section we experiment with the S matrices that arise from the matrices given in Table 4.1.

4.2.1 Effect of Stopping Tolerance on Number of Iterations and Time

GMR, GSD, and CMR algorithms were tested with three different targeted stopping tolerance as condition for convergence, namely $10^{-5}, 10^{-10}$, and 10^{-15}. The other parameters are set as follows: The maximum number of iterations is set to 200, the

initial guess is taken as a zero matrix, and a numerical dropping strategy, with a drop tolerance of 10^{-4} is used.

Unlike GMR and GSD, the CMR algorithm iterates for a certain number of iterations on each column. In order to have comparable metrics for the number of iterations, the total number of iterations performed for all of the input matrix columns was divided by the number of columns to obtain an average number of iterations. The number of iterations for various stopping tolerances for the three algorithms are shown in Figures 4.3, 4.4, and 4.5, respectively:

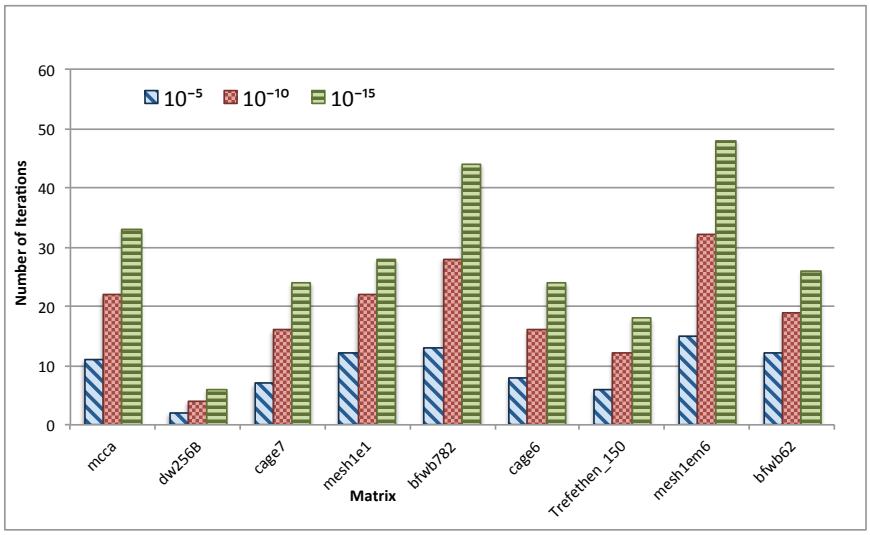


Figure 4.3: Number of Iterations for stopping tolerances for GMR

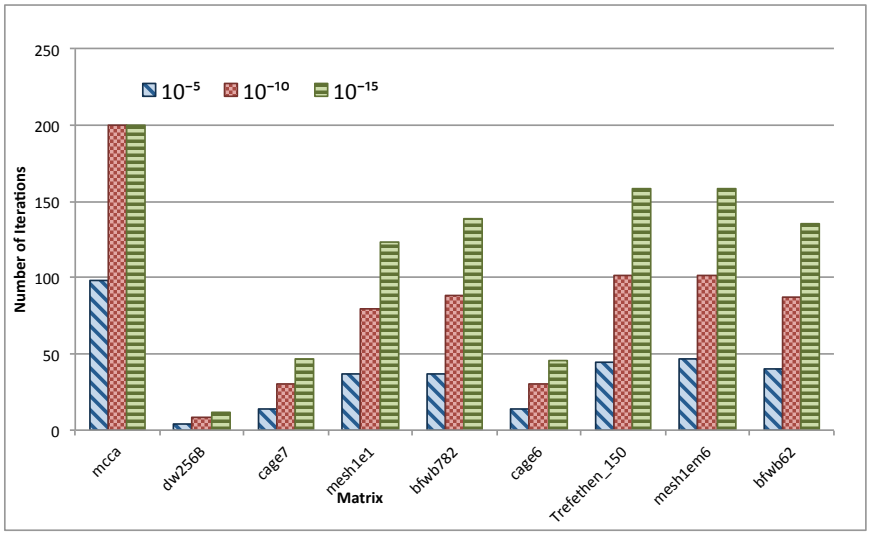


Figure 4.4: Number of Iterations for stopping tolerances for GSD

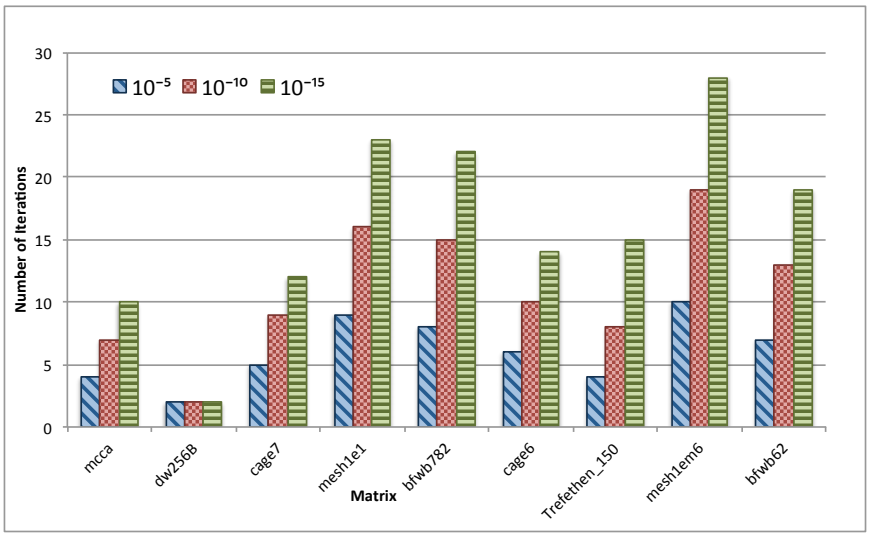


Figure 4.5: Number of iterations for stopping tolerances for CMR

As expected, in all of the matrices tested, the number of iterations increased with decreasing stopping tolerance for all of the algorithms. GSD could not converge within the maximum number of iterations for *mcca* for the stopping tolerance of 10^{-10} and 10^{-15}. The results of the table were also plotted under a different point of view. The three algorithms were compared in terms of iterations they require to reach the desired error tolerance. In Figure 4.6, 4.7, and 4.8, the required number of iterations for a stopping tolerance of 10^{-5}, 10^{-10} and 10^{-15} are given respectively.

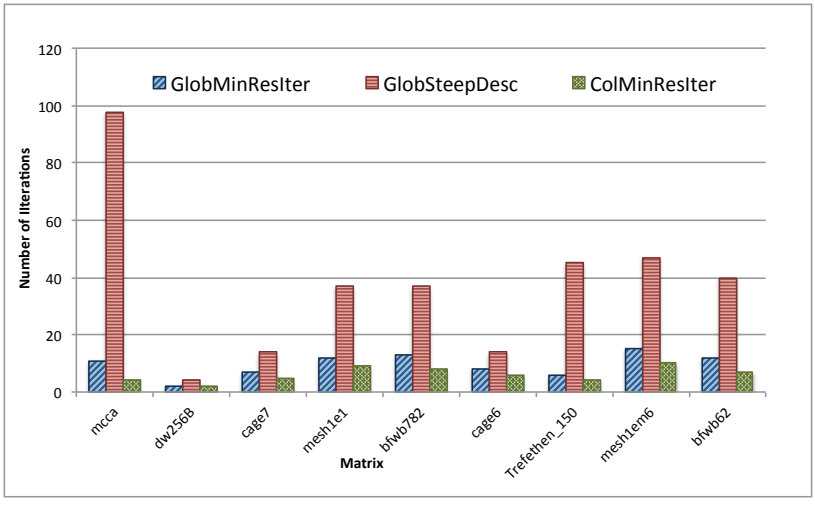


Figure 4.6: Number of iterations for GMR, GSD, CMR for 10^{-5} stopping tolerance

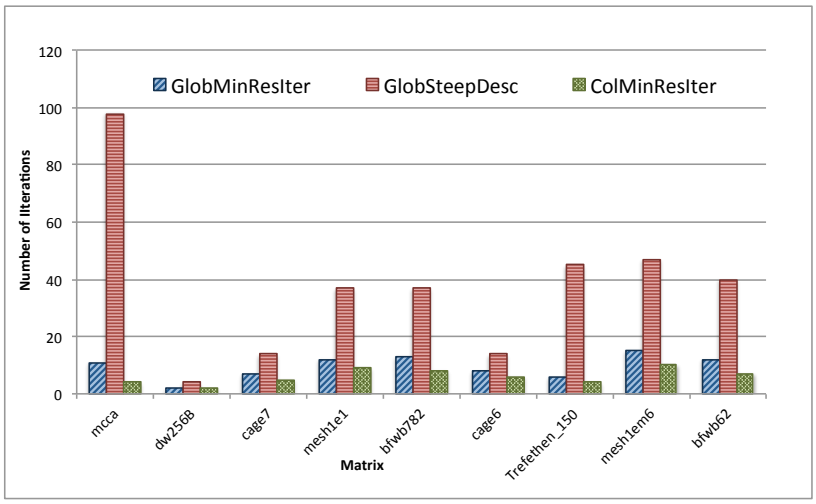


Figure 4.7: Number of Iterations for GMR, GSD, CMR for 10^{-10} stopping tolerance

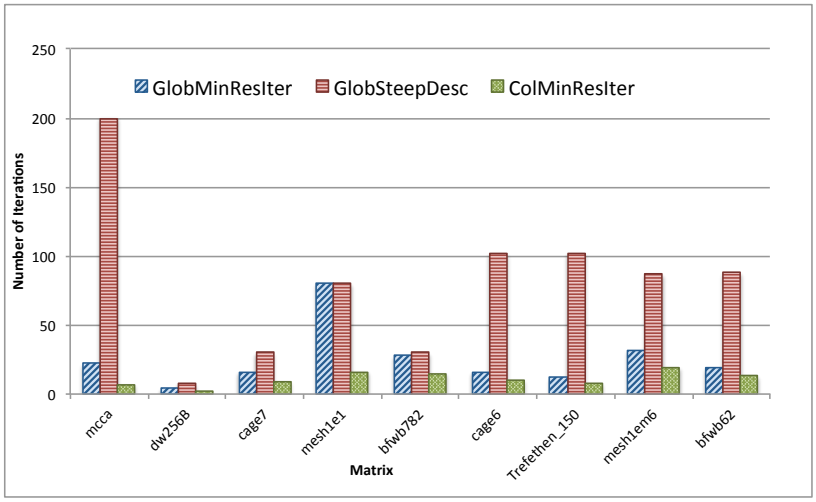


Figure 4.8: Number of Iterations for GMR, GSD, CMR for 10^{-15} stopping tolerance

The number of iterations until convergence was the highest for the GSD algorithm, followed by GMR which had a slight difference in terms of number of iterations when compared to CMR. For the *mesh1e1* matrix, for the 10^{-15} stopping tolerance case, both the GMR and GSD algorithms required an equal number of iterations. The time elapsed until convergence for GMR, GSD, and CMR are given in Figures 4.9, 4.10, and 4.11, respectively. We measure the wall-clock time using the tic and toc functions of MATLAB. For the cases when the time measured was smaller that 1/10 of a second, the algorithms are run in a loop and an average is reported.

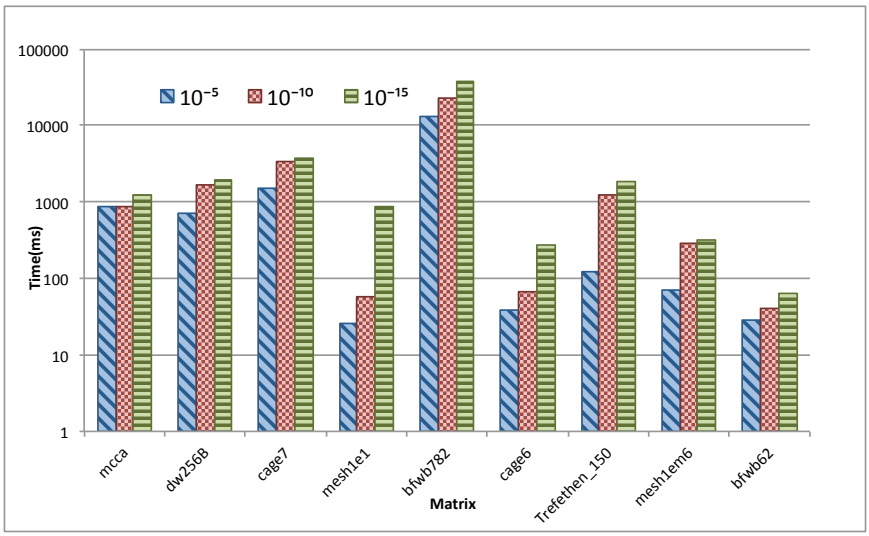


Figure 4.9: Time for stopping tolerances for GMR

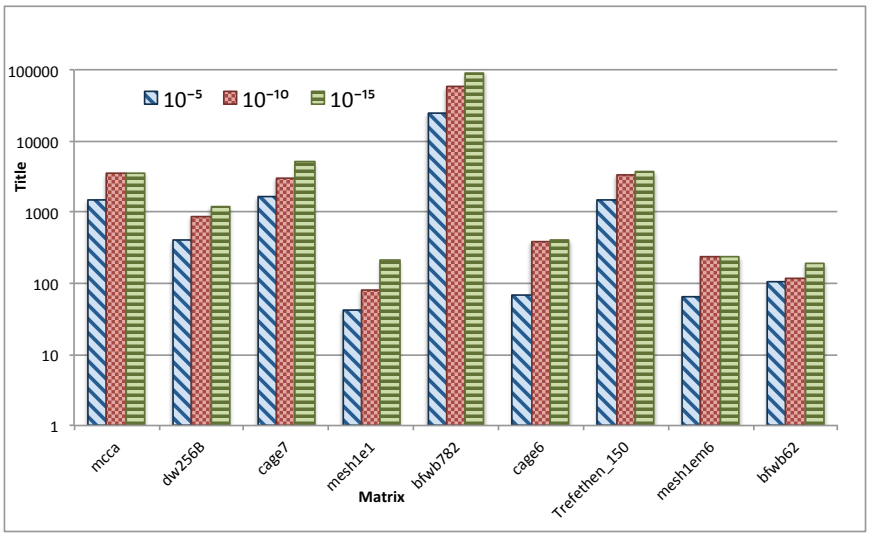


Figure 4.10: Time for stopping tolerances for GSD

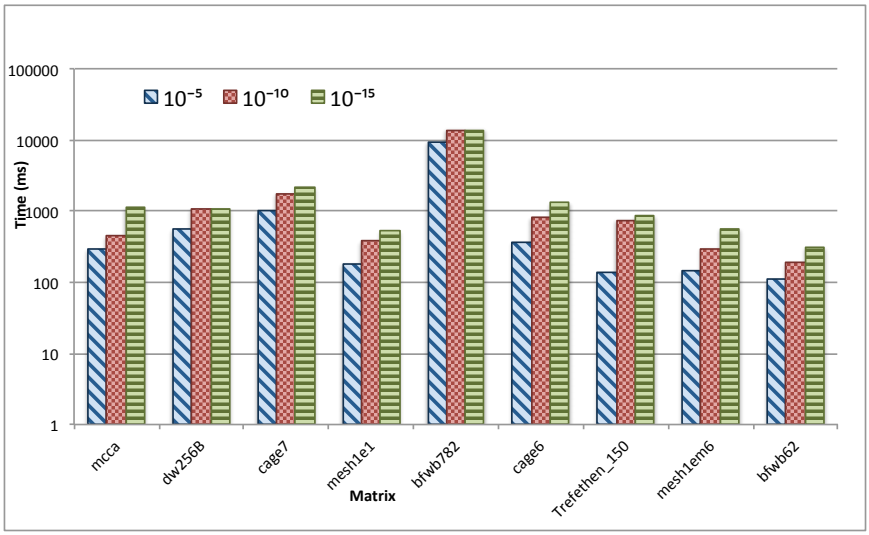


Figure 4.11: Time for stopping tolerances for CMR

As seen in the figures, the elapsed wall-clock time closely follows the number of iterations: time elapsed until convergence also increases with decreasing stopping tolerance, i.e. increased accuracy for the S^{-1} approximation.

In Figures 4.12, 4.13, and 4.14 we give the elapsed wall-clock time for three algorithms using a stopping tolerance of 10^{-5}, 10^{-10}, and 10^{-15}, respectively.

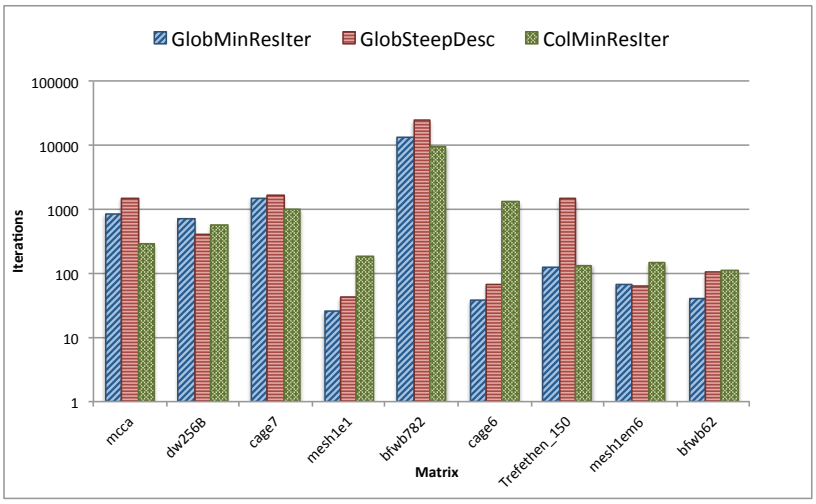


Figure 4.12: Time for GMR, GSD, and CMR for stopping tolerance 10^{-5}

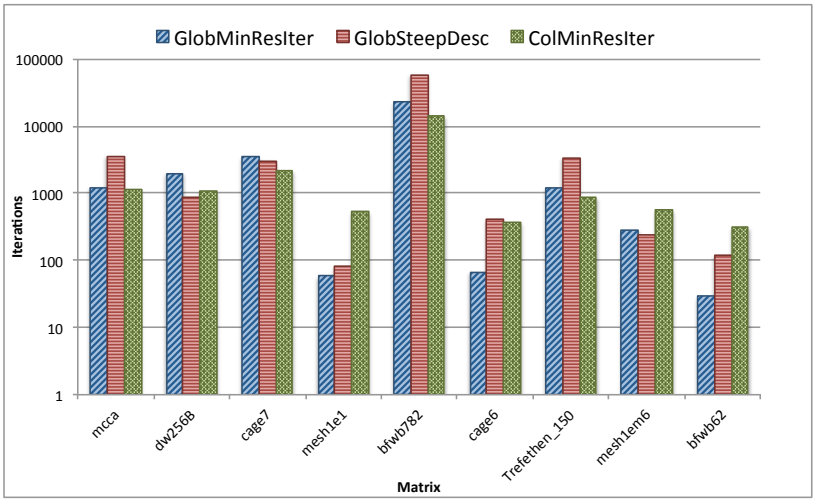


Figure 4.13: Time for GMR, GSD, and CMR for stopping tolerance 10^{-10}

We observe that, regardless of the stopping tolerance, for three of the input matrices, namely *mcca*, *bfwb782*, and *Trefethen_150*, GSD is the slowest, followed by GMR and CMR. For *cage7* and *mesh1em6* the time elapsed is close to each other for all three algorithms. CMR is the most time consuming algorithm for *mesh1e1*, *cage6* and *bfwb62*.

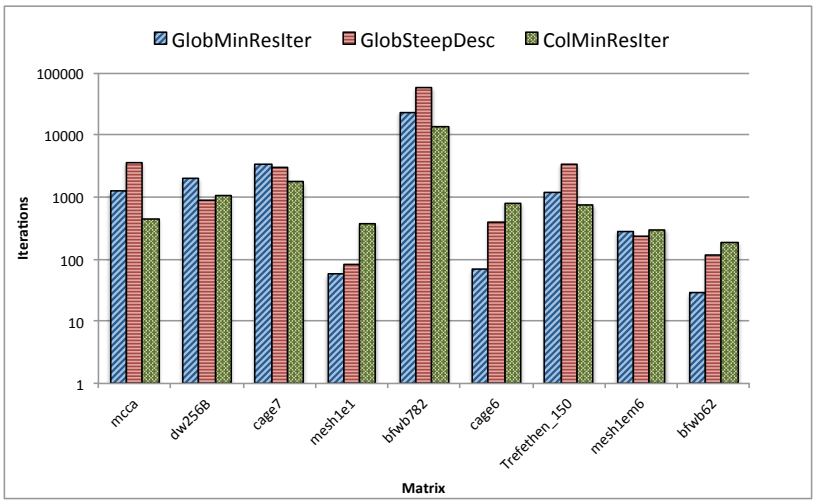


Figure 4.14: Time for GMR, GSD, and CMR for stopping tolerance 10^{-15}

In order to examine how the residual norm, defined as $\|I - AM\|_F$, changes with the iterations for each algorithm, the following residual norm history graphics are plotted under the following assumptions: stopping tolerance 10^{-10}, maximum number of iterations 200, initial guess $M_0 = \alpha I$, and numerical dropping strategy. Since the Column-based Minimum Residual Iteration Algorithm computes the approximate inverse column-by-column, which is quite different from the other two global Approximate Inverse Algorithms, in order to compare the residual norm of all of the three algorithms, its residual norm per iteration was calculated by taking the average residual norm of its columns per iteration.

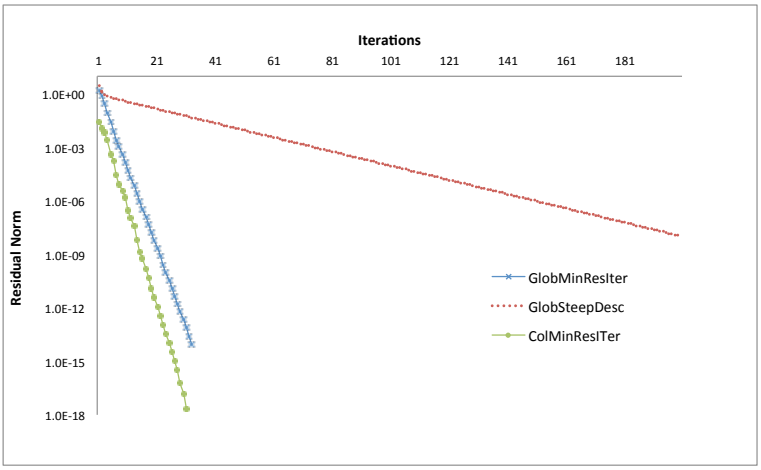


Figure 4.15: Error History Through Iterations for mcca

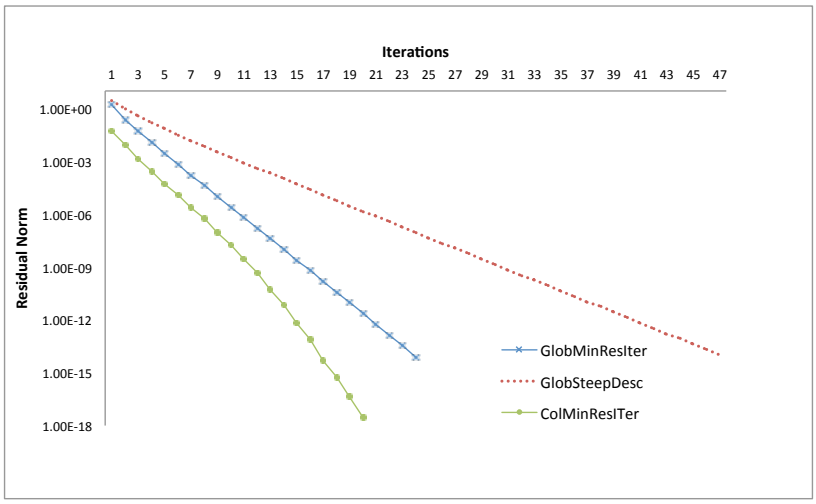


Figure 4.16: Error History Through Iterations for cage7

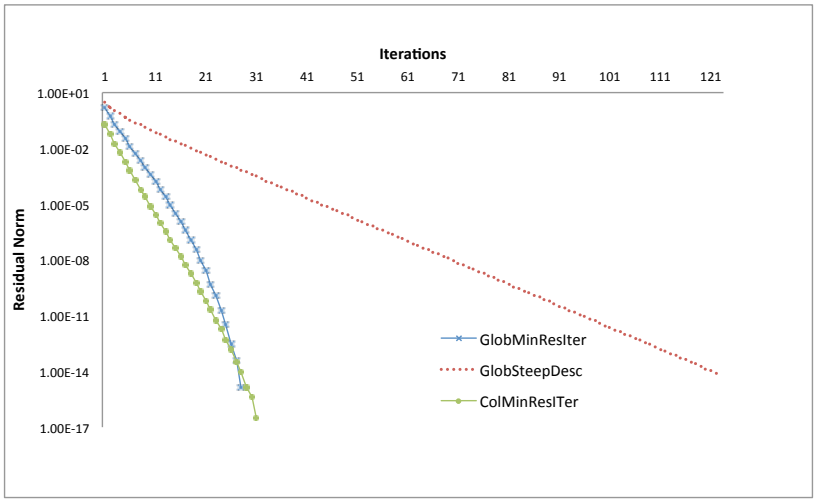


Figure 4.17: Error History Through Iterations for mesh1e1

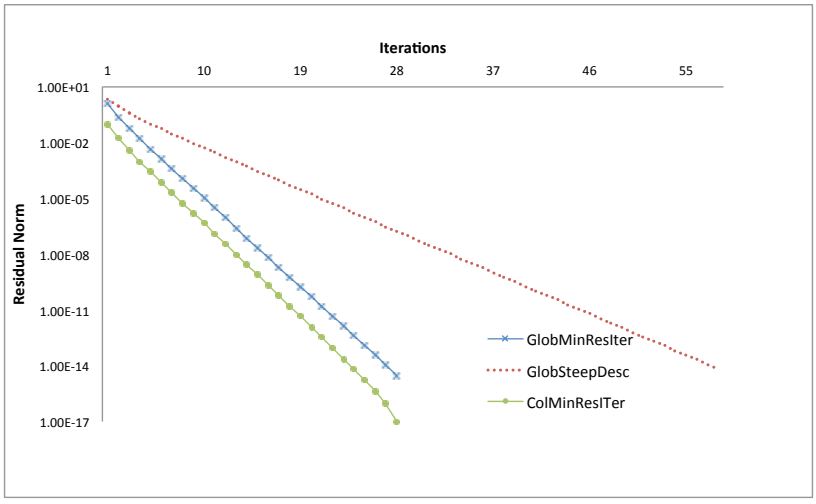


Figure 4.18: Error History Through Iterations for cage6

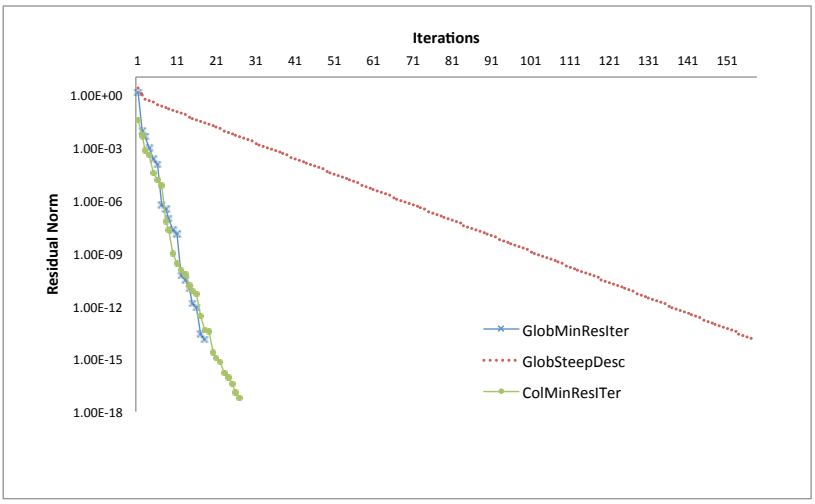


Figure 4.19: Error History Through Iterations for Trefethen150

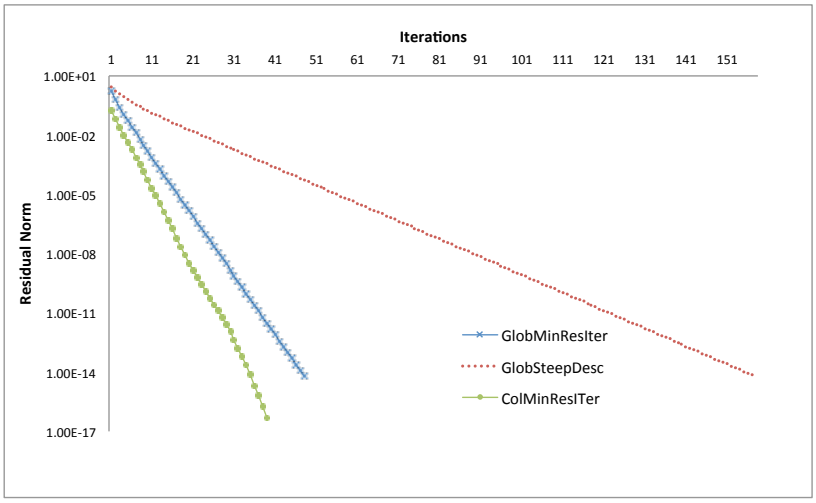


Figure 4.20: Error History Through Iterations for mesh1em6

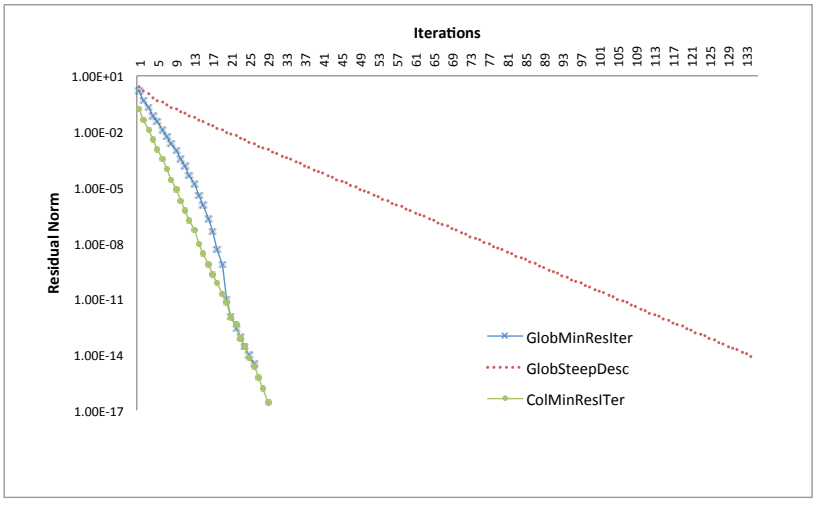


Figure 4.21: Error History Through Iterations for bfwb62

What can be observed from the residual norm history plots, is that GSD reaches the desired tolerance for the longest time, while CMR follows the same pattern as GMR, but reaches a more accurate tolerance for the same number of iterations, when compared to GMR. In Figures 4.15 - 4.20 the residual history plots using all three methods are given for mcca, cage7, mesh1e1, cage6, Trefethen150, mesh1em6 and bfwb62, respectively. For all problems, GSD requires the largest number of iterations to reach the desired tolerance, while CMR and GMR require fewer iterations. We note that, for each method, the cost per iteration is different.

4.2.2 Effect of Initial Guess on Number of Iterations and Time

We have studied the effect of the initial guess for M_0 using three different initial guess matrices, namely $M_0 = \alpha I$ and $M_0 = \alpha A^T$, and the zero matrix.

The scale factor α is chosen to minimize the norm of $I - AM_0$. Thus, the initial guess is of the form $M_0 = \alpha G$ where G could be either the identity matrix or or the transpose of the A matrix, A^T. The optimal α can be computed using

$$\alpha = \frac{tr(AG)}{tr(AG(AG)^T)}.$$

The tests were performed using a stopping tolerance of 10^{-10}, maximum number of iterations 200, and numerical dropping strategy. The table containing the detailed numerical results can be found in the appendix. Here we summarize the results.

In Figure 4.21, 4.22, and 4.23 the required number of iterations to approximate the inverse of the S matrix are given for GMR, GSD, and CMR, respectively.

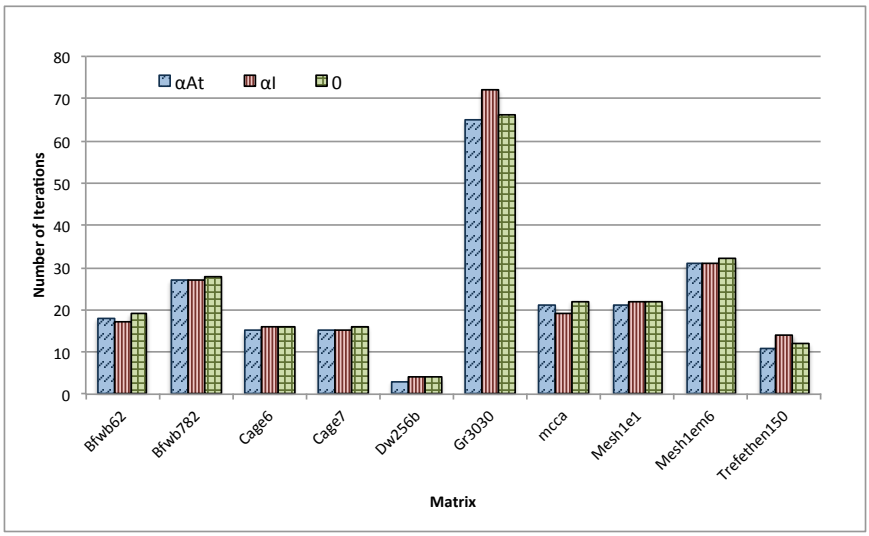


Figure 4.22: Iteration Numbers for initial guesses for GMR

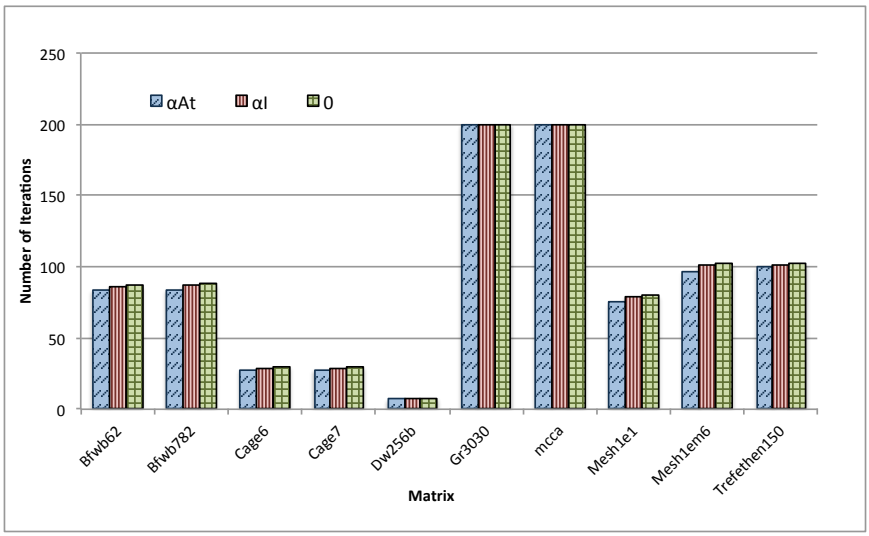


Figure 4.23: Iteration Numbers for initial guesses for GSD

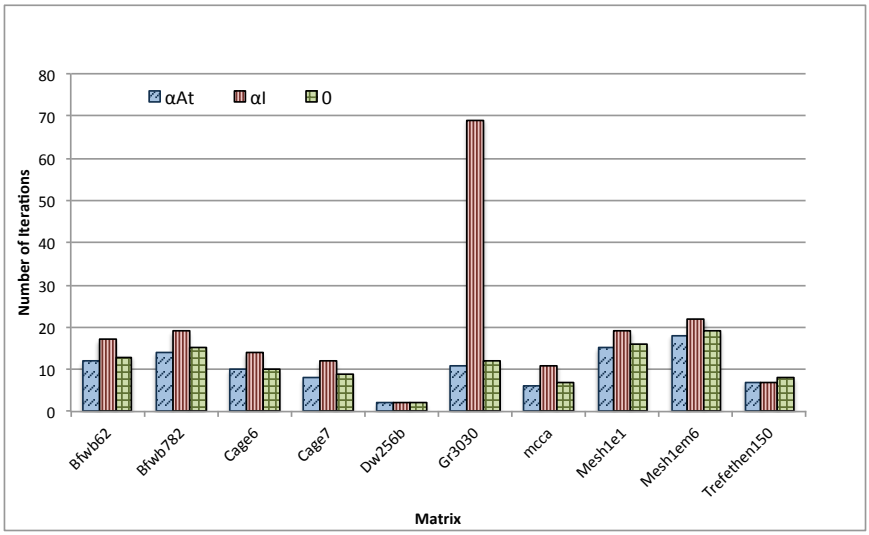


Figure 4.24: Iteration Numbers for initial guesses for CMR

In terms of number of iterations, the initial guess has little or no effect for both GMR and GSD. The only exception is the *gr_30_30* matrix with GMR, where the number of iterations is larger for αI initial guess when compared to the other initial guesses. For CMR, the number of iterations are also close to each other, with a slightly larger number of iterations for αI for all of the matrices. The difference is substantial for *gr_30_30*.

In Figures 4.33, 4.34, and 4.35, wall-clock times are given for GMR, GSD, and CMR, respectively.

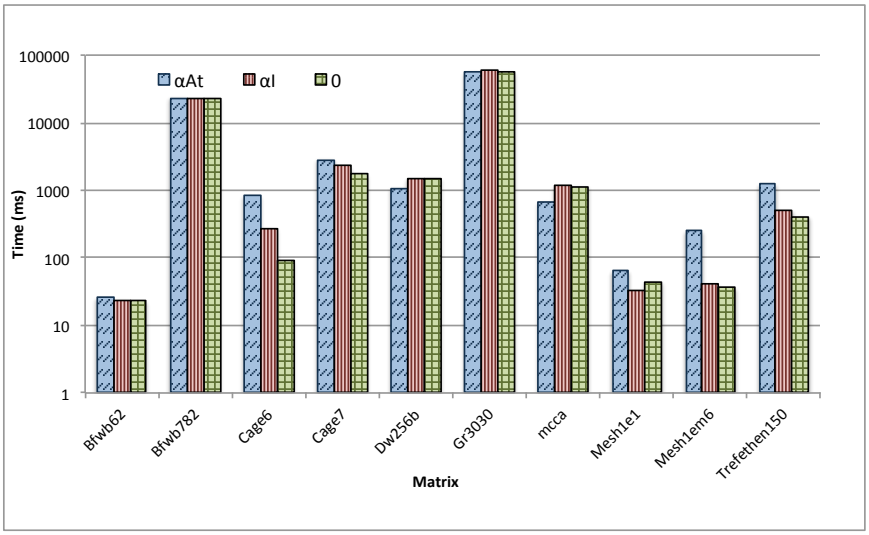


Figure 4.25: Time for initial guesses for GMR

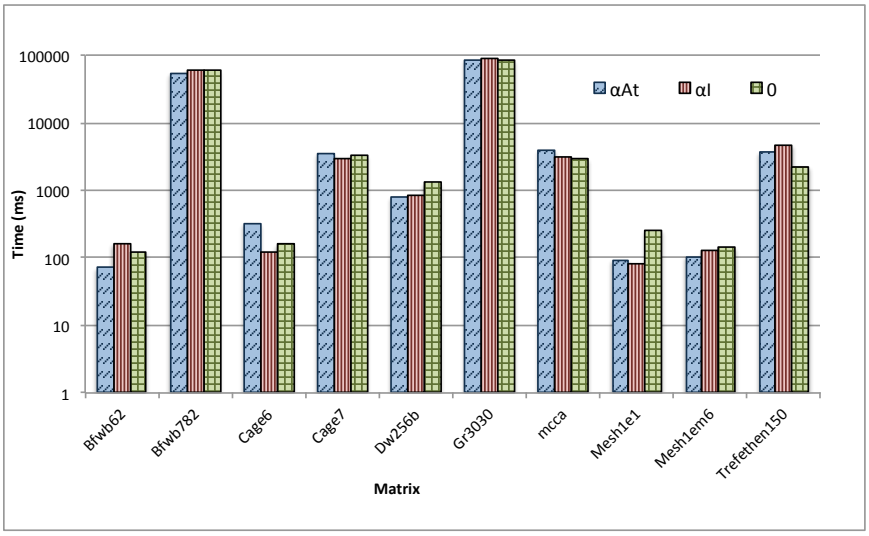


Figure 4.26: Time for initial guesses for GSD

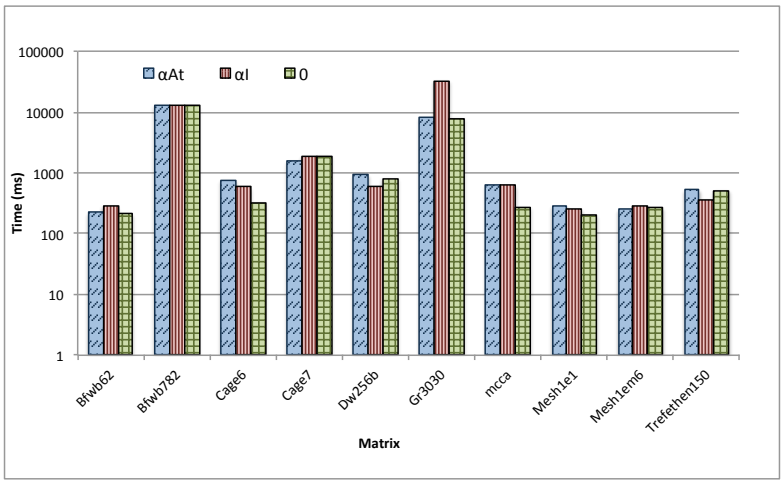


Figure 4.27: Time for initial guesses for CMR

The initial guess choice has little or no effect on the time elapsed for 5 of the input matrices for GMR, namely *bfwb62*, *bfwb782*,*dw256B*, *gr_30_30*, and *mcca*. For the remaining matrices, the αA^T initial guess requires a longer time to meet the convergence criteria, followed closely by αI and the 0 matrix initial guess. For CMR, the time elapsed is close for initial guesses and for all of the matrices, except for *gr_30_30*, where the time required to converge for the αI initial guess is considerably longer than the other initial guesses. We also plot the data by fixing the initial guess and changing the algorithm. Figures 4.28, 4.29, and 4.30 depict the number of iterations and Figures 4.31, 4.32 and 4.33 the time until convergence for initial guess αA^T, αI and 0 matrix, respectively.

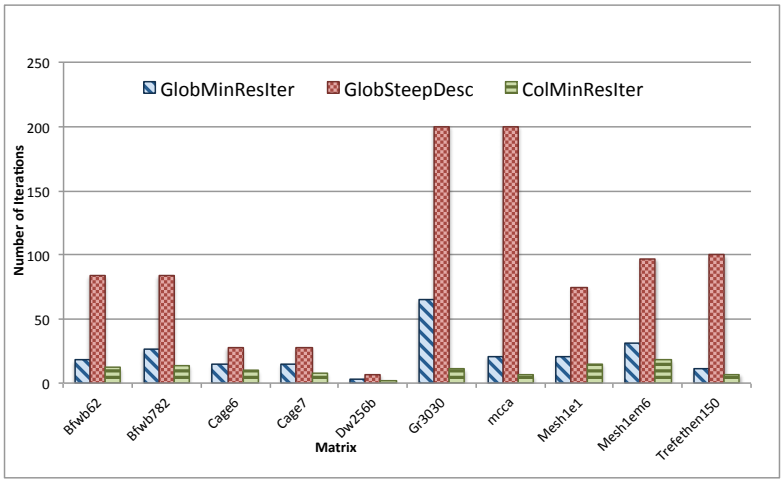


Figure 4.28: Iteration No. for GMR , GSD, CMR - $M_0 = \alpha A^T$

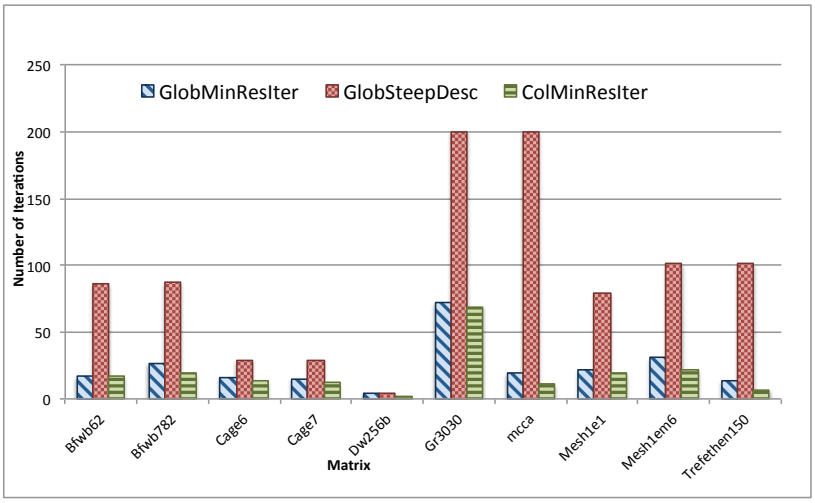


Figure 4.29: Iteration No. for GMR , GSD, CMR - $M_0 = \alpha I$

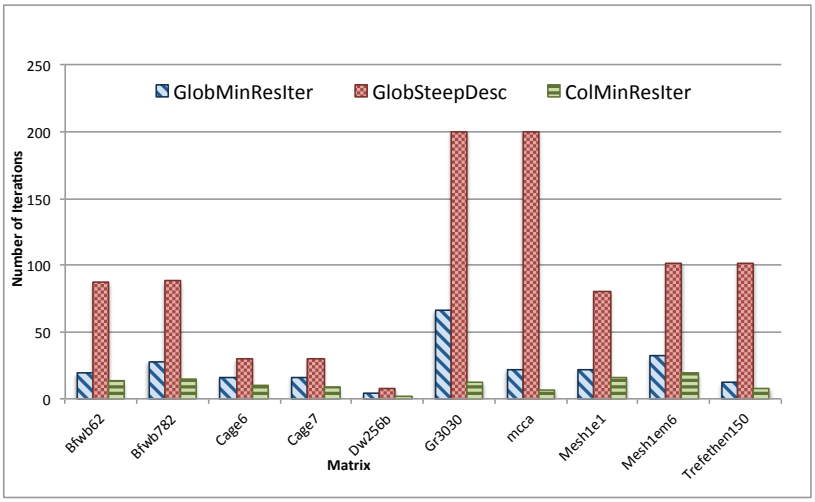


Figure 4.30: Iteration No. for GMR , GSD, CMR - $M_0 = 0$

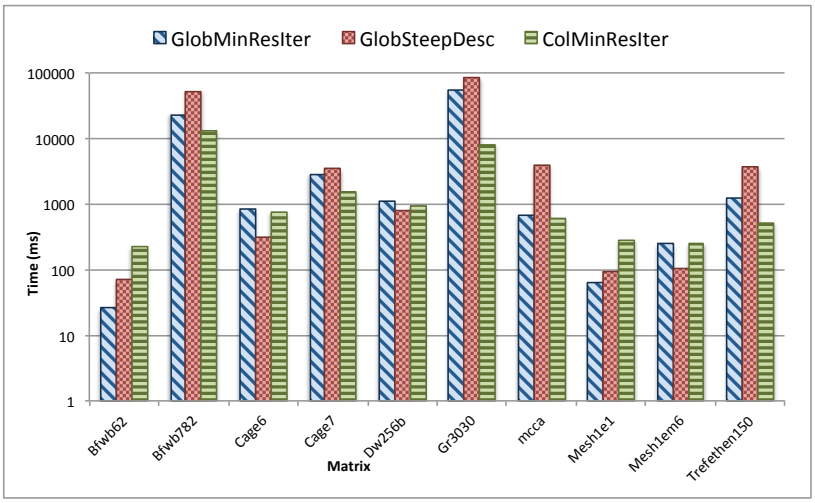


Figure 4.31: Time for GMR , GSD, CMR - $M_0 = \alpha A^T$

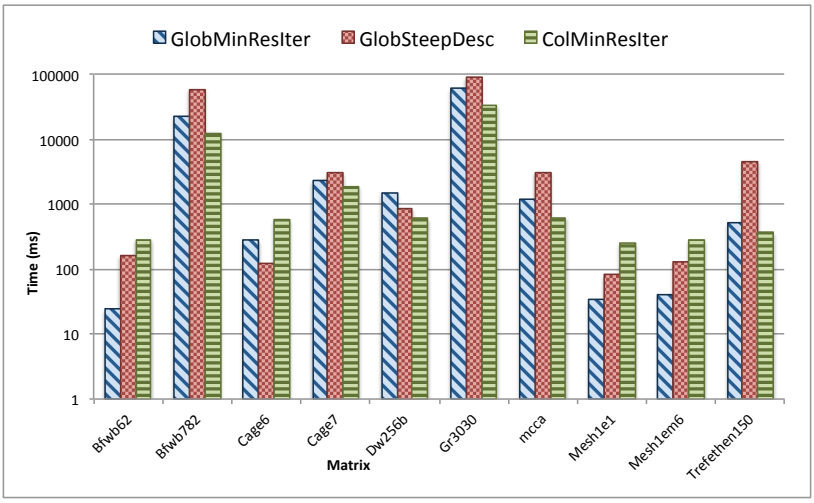


Figure 4.32: Time for GMR , GSD, CMR - $M_0 = \alpha I$

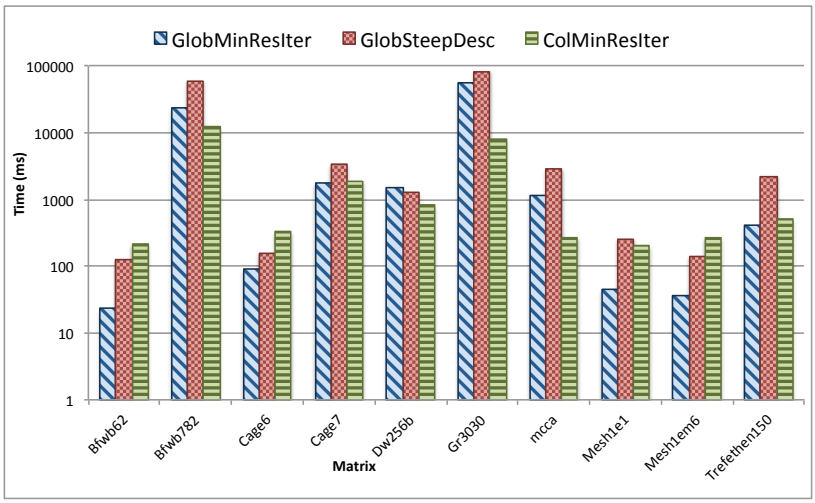


Figure 4.33: Time for GMR , GSD, CMR- $M_0 = 0$

4.2.3 Effect of Dropping Strategy on Number of Iterations and Time

The three dropping strategies , namely the numerical, structural, and the hybrid dropping strategies that were proposed in this thesis and described in detail in the previous chapter were studied in terms of their effect on the performance of the Approximate Inverse Algorithms for computing the inverse of the S matrix.

We observe that the dropping strategy had no or very little effect on the iteration number elapsed until the algorithms converged. As an example, the number of iterations for the three different dropping strategies for the Global Minimum Residual iteration algorithm are given in Figure 4.34:

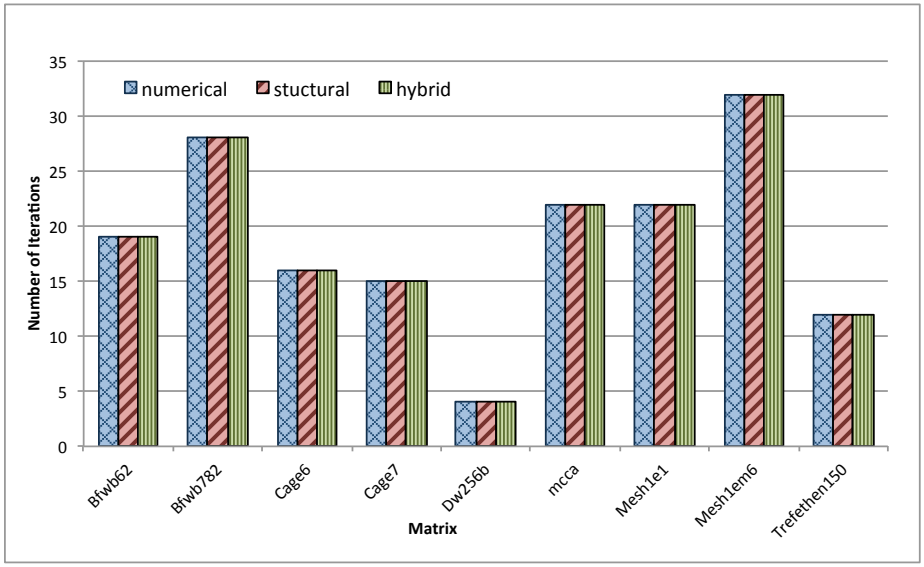


Figure 4.34: Number of Iterations for dropping strategies for GMR

Although the number of iterations is not affected, the dropping strategy is expected to have an effect on the computational time, given that it affects the cost per iteration of the algorithms. In Figure 4.35, 4.36, and 4.37 wall-clock times are given for GMR, GSD, and CMR, respectively. The structural dropping strategy is seen to perform better for CMR, followed by numerical dropping. The hybrid dropping strategy was the most computationally expensive dropping strategy for all of the matrices.

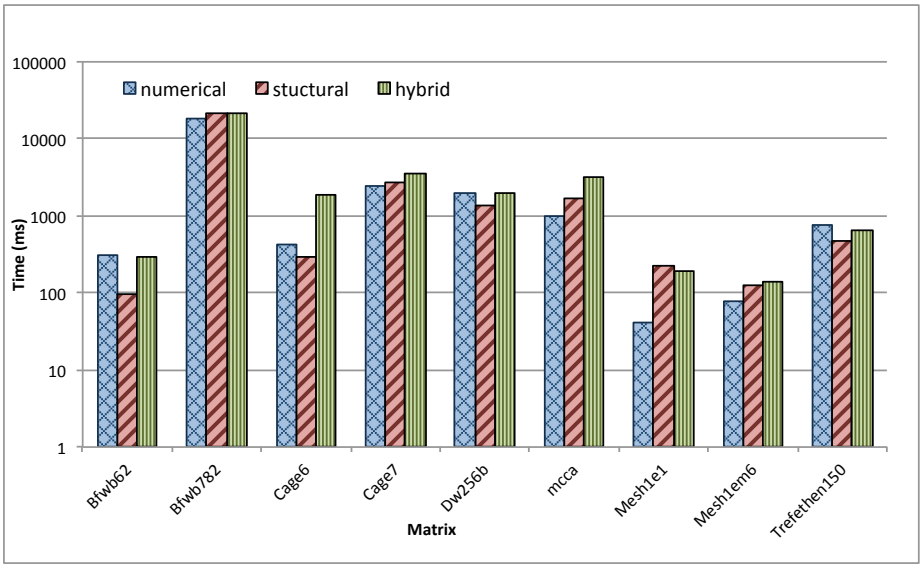


Figure 4.35: Time for dropping strategies GMR

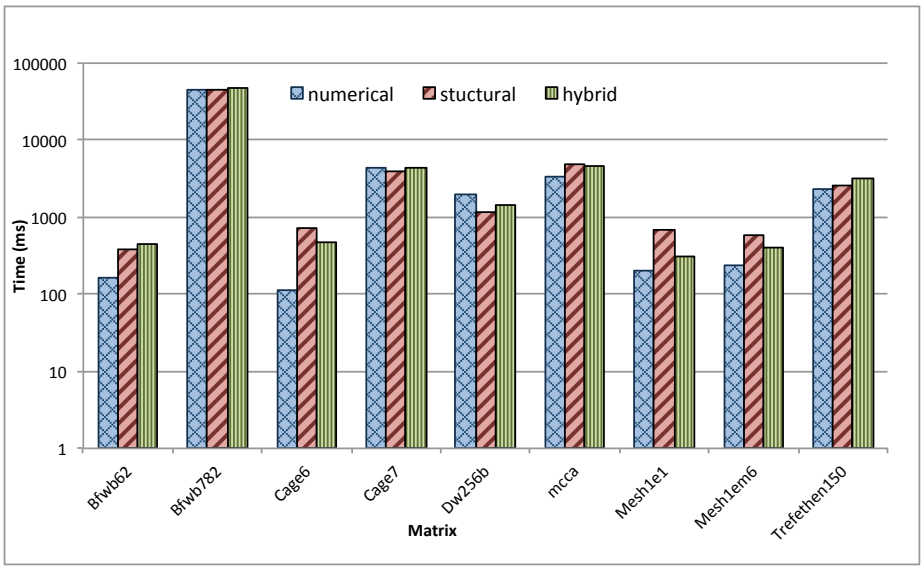


Figure 4.36: Time for dropping strategies GSD

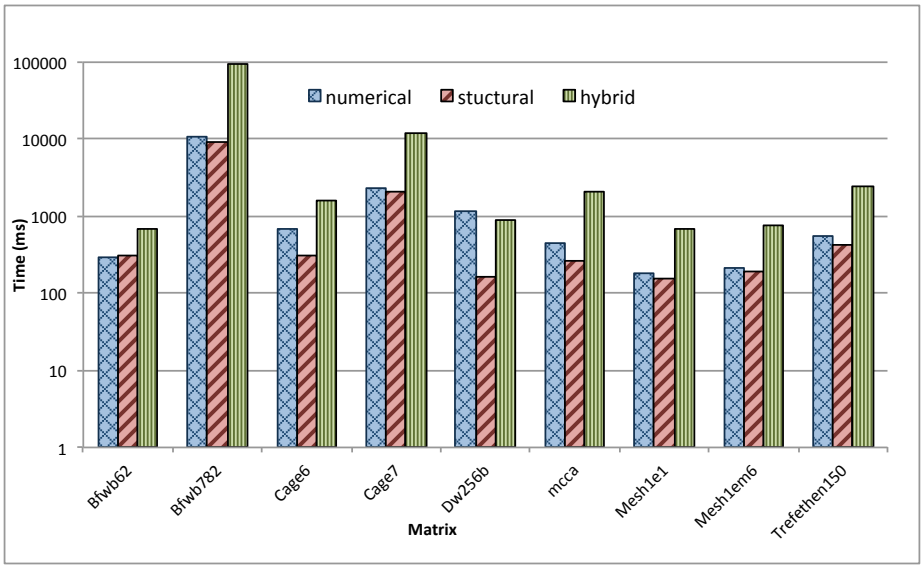


Figure 4.37: Time for dropping strategies CMR

Now, we fix the dropping strategy and provide the required number of iterations and time for all the three methods. In figure 4.38, 4.39, and 4.40, the required time is given using numerical dropping, structural dropping, and hybrid dropping, respectively. For numerical dropping, GSD requires the longest time to converge for $bfwb782$, $cage7$, $mcca$, and $Trefethen_150$ followed by GMR, and CMR. GMR is faster than CMR and then GSD for the $mesh1e1$ and $mesh1em6$ matrices, while GSD requires a shortest amount of time for $bfwb62$, $cage7$ and $dw256B$, where GMR and CMR require longer time to converge.

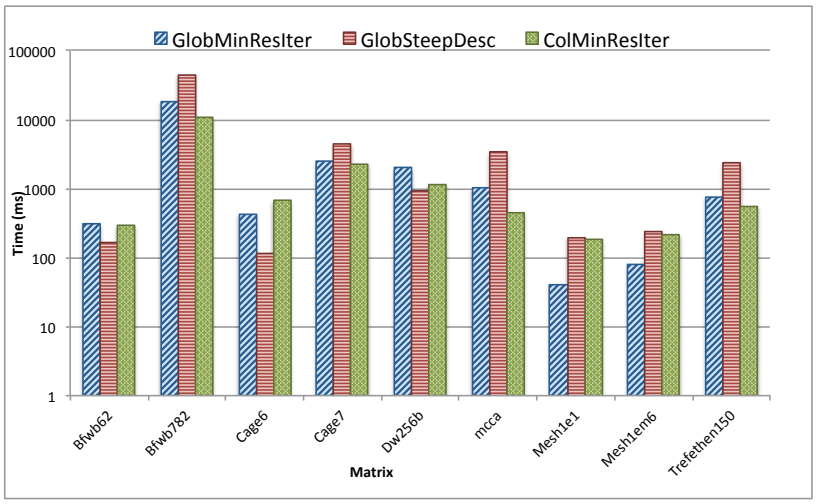


Figure 4.38: Time for numerical dropping for GMR, GSD,CMR

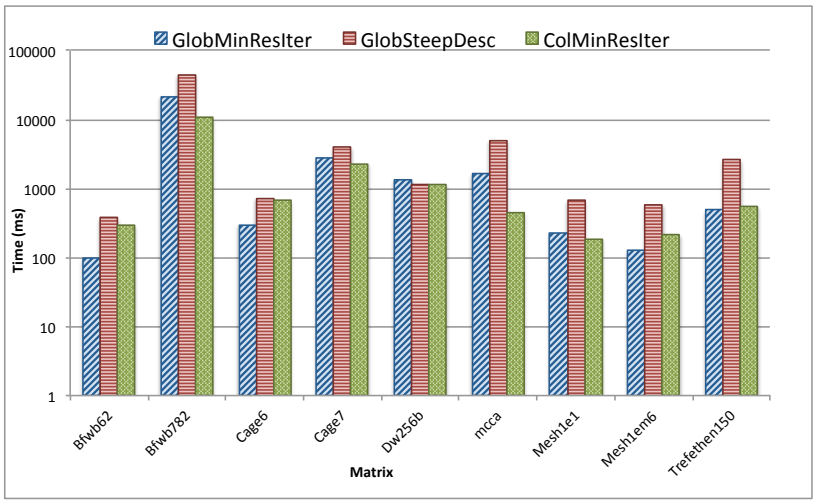


Figure 4.39: Time for structural dropping for GMR, GSD,CMR

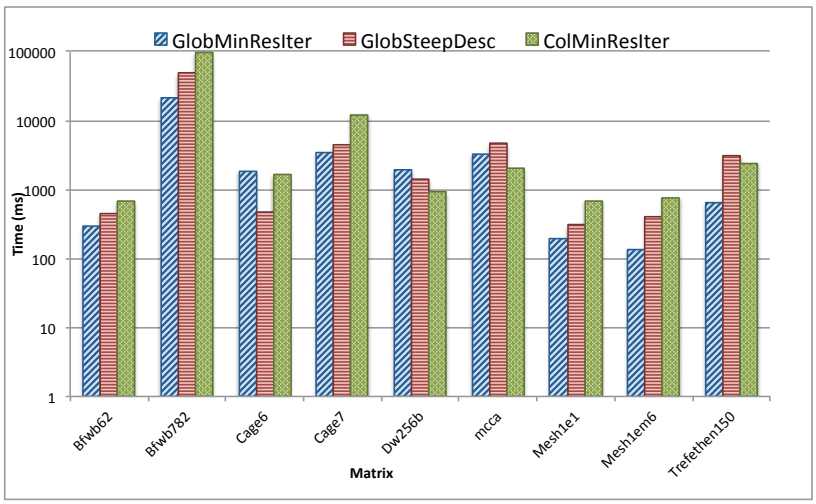


Figure 4.40: Time for hybrid dropping for GMR, GSD,CMR

The time required for convergence slightly differ for the structural dropping strategy. All of the three algorithms took similar times to converge for $dw256B$. GSD requires the longer time to converge for all of the other input matrices. GMR performed better than CMR for $bfwb62$, $cage6$,$Trefethen_150$ and $mesh1em6$, while CMR converged for a shorter time compared to GMR for $bfwb782$, $cage7$, $mcca$ and $mesh1e1$.

In the hybrid dropping case, time to convergence was the shortest for GMR, followed by GSD and then CMR for the following matrices: $bfwb62$, $bfwb782$, $cage7$, $mesh1e1$ and $mesh1em6$. For $cage6$, GSD converged for the shortest time, followed by close times for GMR and CMR. GSD required the longest time to converge for $mcca$ and $Trefethen_150$,

4.3 The Parallel Results using the Proposed Algorithm

In the earlier sections, we have studied the performance of part of the proposed algorithm, namely computing the approximate inverse of the S matrix. Based on the results, we use the Column-based Minimum Iteration Algorithm, with structural dropping and a stopping tolerance of 10^{-10}. The algorithm described in this thesis is implemented in its sequential version using C++ and the Armadillo library for matrix operations. Next, OpenMP directives were added in order to introduce parallel sections. We use another set of matrices obtained from the UF Sparse Matrix Collection listed in Table 4.2 which contains slightly larger matrices.

Table 4.2: Input Matrices for Parallel Tests

Matrix Name	rows	cols	nonzeros	A posdef	S posdef
circuit_1	2,624	2,624	35,823	no	yes
ex10	2,410	2,410	54,840	yes	yes
saylr4	3,564	3,564	22,316	no	yes
sherman4	1,104	1,104	3,786	no	yes
orsirr_1	1,030	1,030	6,858	no	yes
crystk01	4,875	4,875	315,891	no	yes
s1rmq4m1	5,489	5,489	262,411	yes	yes
bcsstk16	4,884	4,884	290,378	yes	yes

4.3.1 Comparison of the Straightforward algorithm to the Proposed Algorithm

The straightforward algorithm that computes the full inverse of the matrix sequentially using a sparse direct solver (SuperLU) and then extracts its diagonal is used as a baseline. The results are shown in Table 4.3. The speedup is calculated as $Speedup = \frac{Time_{Straightforward}}{Time_{Parallel}}$ and the time is given in milliseconds.

Table 4.3: Solution Time (ms) and Speedup for the proposed algorithm compared to the straightforward algorithm

Matrix Name	Number of Partitions	Straightforward	Parallel	Speedup
circuit_1	3	578,0	55,6	10,4
ex10	2	273,1	31,7	8,6
saylr4	2	1744,7	231,8	7,5
sherman4	2	38,2	8,8	4,3
orsirr_1	2	39,5	10,9	3,6
cryst01	3	1721,8	593,1	2,9
s1rmq4m1	3	3325,6	734,8	4,5
bcsstk16	2	2576,1	147,0	17,5

The results in this table were plotted in Figure 4.41. For better visibility, the time axis was logscaled.

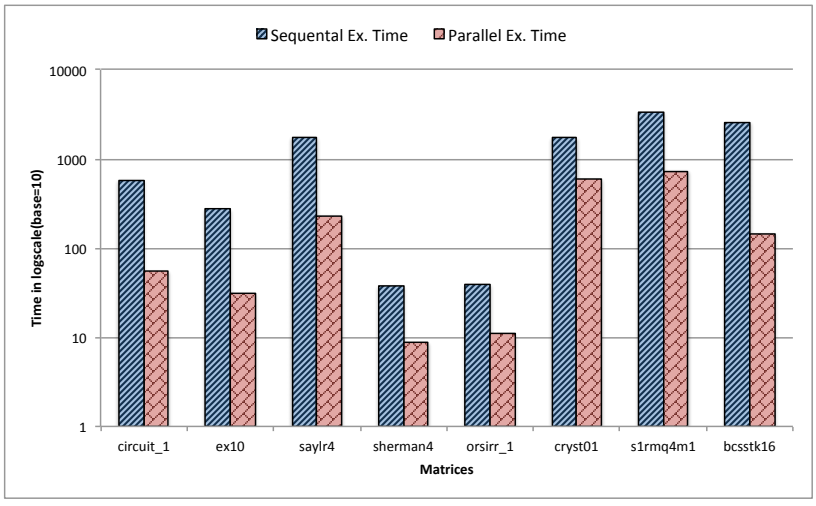


Figure 4.41: Execution time(ms) for the Straightfroward algorithm vs Proposed Algorithm

As it can be seen from the speedup column, considerable improvements are achieved from the proposed method executed in parallel. The proposed algorithm run in parallel can be up to 17 times faster compared to the straightforward approach.

4.3.2 The parallel scalability

In this section, we study the parallel scalability of the proposed algorithm. We use the Column-based Minimum Iteration Algorithm for approximating the inverse of the S matrix. The parameters for CMR are structural dropping strategy, 10^{-10} stopping tolerance and αI initial guess.

The algorithm was run with different numbers of threads in order to analyse its scalability. First the total duration of the algorithm through different number of threads was plotted for each duration in the following chart:

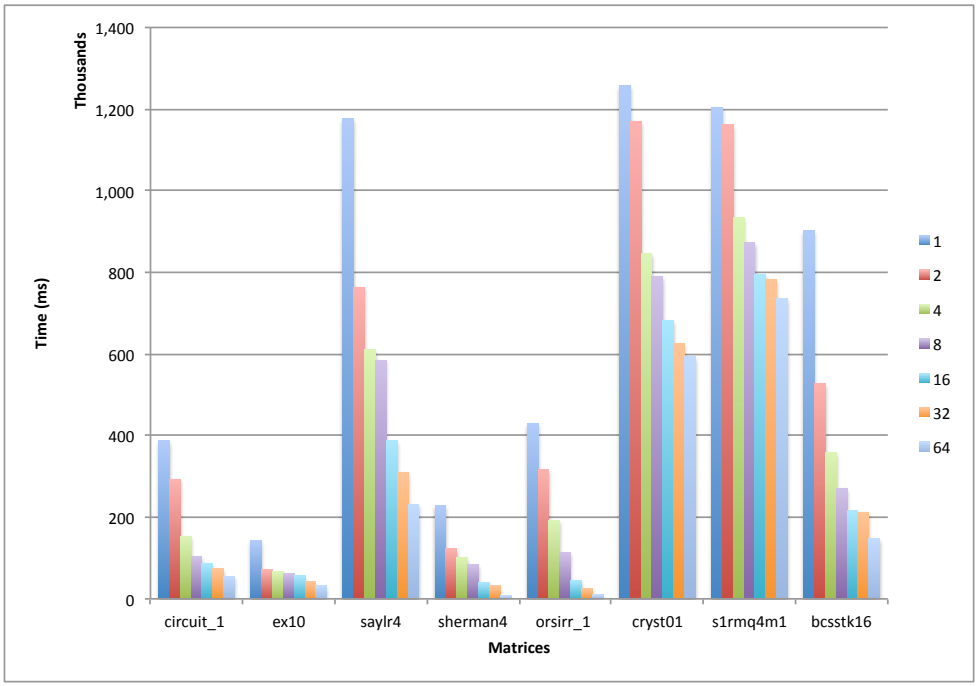


Figure 4.42: Execution Time

In figure 4.52, the parallel speedup with respect to the sequential running time of the algorithm is given. The speedup for $orsirr\_1$ is the closest to the linear speedup, followed by $sherman4$. $circuit\_1$, $bcsstk16$, $saylr4$ and $ex10$, which scale similarly to each other, but not as good as the first two matrices. Finally, $cryst01$ and $s1rmq4m1$ show a slower speedup when compared to the sequential running time of the other matrices.

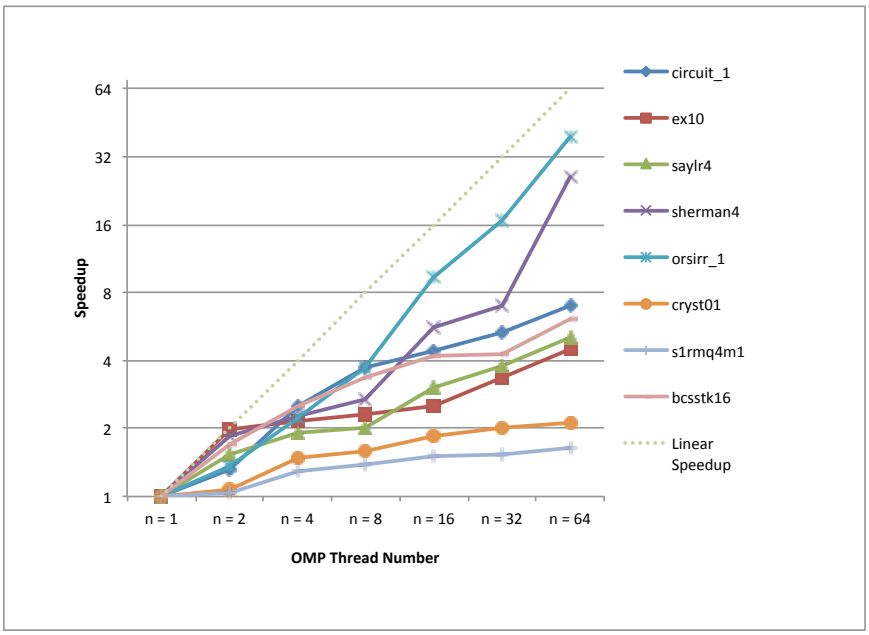


Figure 4.43: Parallel running time as the number of threads change.

In Figures 4.54 and 4.53, detailed breakdowns of the time spent in each phase of the proposed algorithm as the number of threads increase are given for the bcsstk16 and saylr4 matrices, respectively.

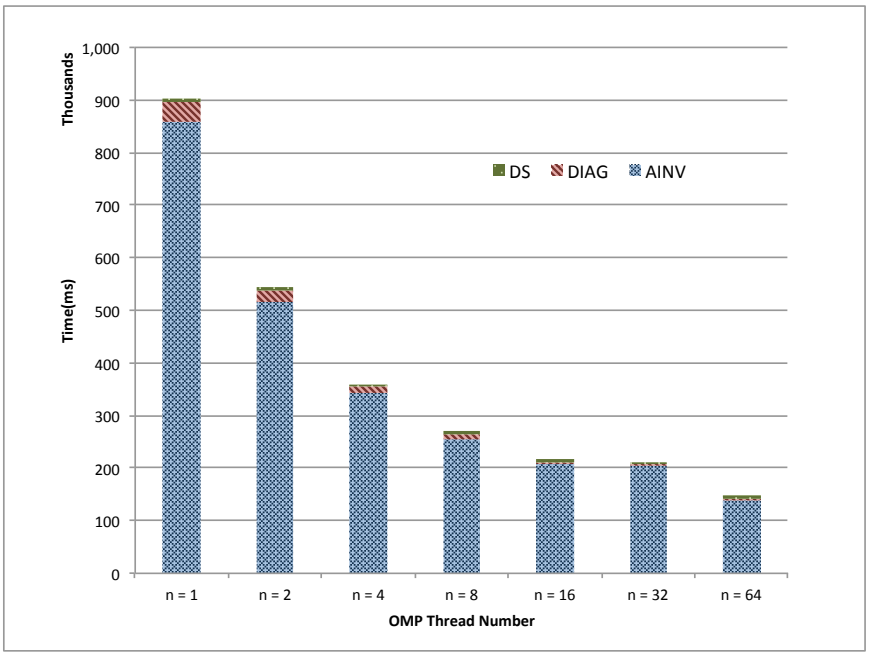


Figure 4.44: Duration of the three phases of the algorithm for bcsstk16

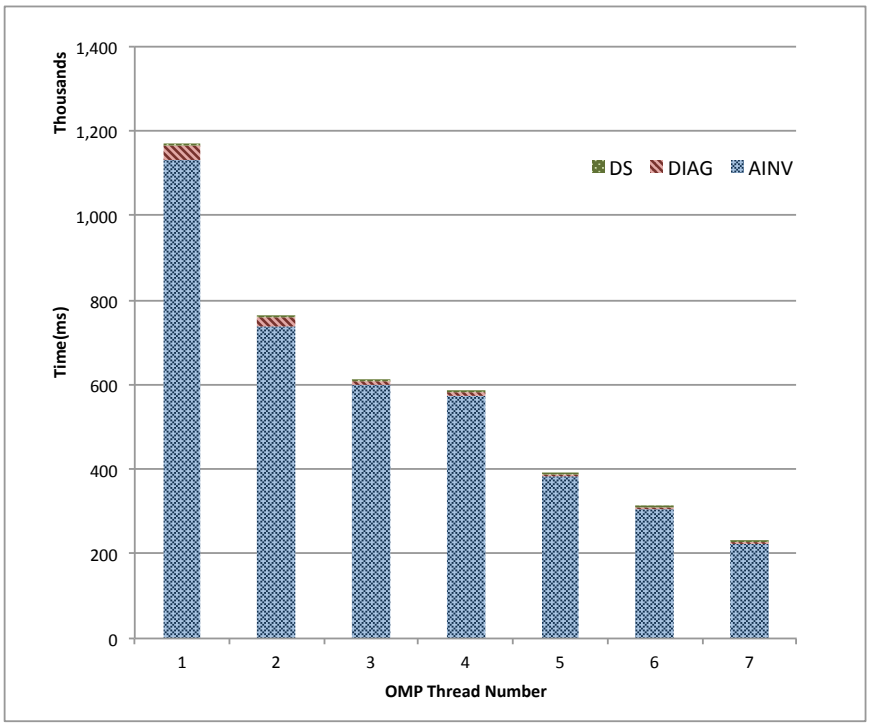


Figure 4.45: Duration of the three phases of the algorithm for saylr4

The most time-consuming phase of the proposed method is the approximate inverse calculation of the S matrix, followed by the calculation of the diagonal, and then the factorization phase. The factorization phase scalability is bound by the number of partitions, since we assign one partition per processing element. The reason we kept the partition number generally low is explained with the effect that it has on the sparsity pattern of the resulting matrix S. Increasing the partition number does decrease the execution time of the DS factorization with increasing number of threads, but it also increases the number of spikes in the S matrix, hence increasing the number of columns that have to be traversed by the Column-Based Minimum Iteration Algorithms.

CHAPTER 5

CONCLUSIONS

We developed a parallel algorithm for the computation of the diagonal of the inverse of a sparse matrix. The main motivation for this work were the various science and engineering applications in which this problem arises, such as Dynamic Mean Field Theory, Uncertainty Quantification, and Density Function Theory.

The existing solutions to this problem ranged from the straghtforward solution of full inversion of the sparse matrix to extract its diagonal, to direct methods based on the standard LU factorization and triangular solves, and stochastic or iterative approximations. However, most of these solutions were limited by the application that motivated them. Moreover, we noticed that there was very little work done in finding a parallel solution to this problem.

Our main objective was to develop a parallel algorithm for an efficient computation of the diagonal of the sparse matrix inverse. The proposed method uses elements of both direct and iterative methods. It first factorizes the sparse matrix into the D and S matrices by applying the generalized sparse DS factorization. Next, it uses approximate inverse algorithms for the inversion of the S matrix, which has a peculiar sparsity structure. During this iterative phase of the proposed method, a new dropping strategy based on the structure of the S matrix, namely structural dropping, is applied.

Various experiments were performed in order to analyse the performance of the proposed method and the different parameters that affected its performance. We first studied in detail the approximate inverse algorithms and how three different parameters, namely the stopping tolerance, the initial guess, and the dropping strategy affected their performance. Based on the results of this first set of experiments and other observations, we decided to use the Column-Based Minimum Iteration algorithm with Structural dropping and αI intial guess for the second step of the parallel algorithm. The parallel algorithm was implemented and tested agains the straightfor-

ward method. Next, the parallel scalability of the proposed method was studied and the speedup against the sequentail running time of the algorithm were reported. We noticed that the proposed algorithm perfomed significantly better when compared to the straightforward algorithm, and the algorithm scales quite good with the increasing number of threads. The Approximate Inverse computation is noticed to be the most time-consuming step of the algorithm. Moreover, the parallelization of the DS factorization step is bound to the number of partitions; even though the DS factorization time decreases when increasing the number of partitions and executing the method in parallel, the increased the number of non-zero entries on the resulting S matrix lead to a siginifant increase in the second step of the proposed method.

The main advantages of the proposed algorithm are that it is applicable to general matrices and not necessarily of a particular application, and that it is based on the Generalized DS factorization and Approximate Inverse Algorithms, both of which are suitable for parallelization.

Comparison of the proposed algorithm against other existing algorithms that find the diagonal of the inverse of a sparse matrix is a remaining issue due to the lack of available and non-application specific implementations of these methods.

REFERENCES

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, Y. Robert, F.-H. Rouet, and B. Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. *SIAM journal on Scientific Computing*, 34(4):A1975–A1999, 2012.

[2] C. Bekas, A. Curioni, and I. Fedulova. Low cost high performance uncertainty quantification. In *Proceedings of the 2Nd Workshop on High Performance Computational Finance*, WHPCF '09, pages 8:1–8:8, New York, NY, USA, 2009. ACM.

[3] C. Bekas, E. Kokiopoulou, and Y. Saad. An estimator for the diagonal of a matrix. *Applied numerical mathematics*, 57(11):1214–1229, 2007.

[4] M. Benzi and G. H. Golub. Bounds for the entries of matrix functions with applications to preconditioning. *BIT Numerical Mathematics*, 39(3):417–438, 1999.

[5] K. Bowden. A direct solution to the block tridiagonal matrix inversion problem. *International Journal Of General System*, 15(3):185–198, 1989.

[6] Y. E. Campbell and T. A. Davis. Computing the sparse inverse subset: an inverse multifrontal approach. *University of Florida, Gainesville, FL, Tech. Rep. TR-95-021*, 1995.

[7] M. Ceriotti, T. D. Kühne, and M. Parrinello. An efficient and accurate decomposition of the fermi operator. *The Journal of chemical physics*, 129(2):024707, 2008.

[8] E. Chow and Y. Saad. Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal on Scientific Computing*, 19(3):995–1023, 1998.

[9] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[10] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., New York, NY, USA, 1986.

[11] A. M. Erisman and W. F. Tinney. On computing certain elements of the inverse of a sparse matrix. *Commun. ACM*, 18(3):177–179, Mar. 1975.

[12] J. K. Freericks. *Transport in multilayered nanostructures: the dynamical mean-field theory approach*. World Scientific, 2006.

[13] S. Goedecker. Linear scaling electronic structure methods. *Reviews of Modern Physics*, 71(4):1085, 1999.

[14] A. S. Householder. *The theory of matrices in numerical analysis*. Courier Corporation, 2013.

[15] M. F. Hutchinson. A stochastic estimator of the trace of the influence matrix for laplacian smoothing splines. *Communications in Statistics-Simulation and Computation*, 19(2):433–450, 1990.

[16] S. Li, S. Ahmed, G. Klimeck, and E. Darve. Computing entries of the inverse of a sparse matrix using the find algorithm. *J. Comput. Phys.*, 227(22):9408–9427, Nov. 2008.

[17] X. S. Li. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.*, 31(3):302–325, September 2005.

[18] L. Lin, J. Lu, R. Car, and E. Weinan. Multipole representation of the fermi operator with application to the electronic structure analysis of metallic systems. *Physical Review B*, 79(11):115133, 2009.

[19] L. Lin, J. Lu, L. Ying, R. Car, E. Weinan, et al. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. *Communications in Mathematical Sciences*, 7(3):755–777, 2009.

[20] L. Lin, C. Yang, J. Lu, and L. Ying. A fast parallel algorithm for selected inversion of structured sparse matrices with application to 2d electronic structure calculations. *SIAM Journal on Scientific Computing*, 33(3):1329–1351, 2011.

[21] L. Lin, C. Yang, J. C. Meza, J. Lu, L. Ying, and W. E. Selinv—an algorithm for selected inversion of a sparse symmetric matrix. *ACM Trans. Math. Softw.*, 37(4):40:1–40:19, Feb. 2011.

[22] M. Manguoglu. A general sparse sparse linear system solver and its application in openfoam. *Partnership for Advanced Computing in Europe*, 2012.

[23] M. Manguoglu. Parallel solution of sparse linear systems. In *High-Performance Scientific Computing*, pages 171–184. Springer, 2012.

[24] E. Polizzi and A. Sameh. Spike: A parallel environment for solving banded linear systems. *Computers & Fluids*, 36(1):113–120, 2007.

[25] R.-S. Ran and T.-Z. Huang. An inversion algorithm for a banded matrix. *Computers & Mathematics with Applications*, 58(9):1699–1710, 2009.

[26] C. E. Rasmussen and C. K. Williams. Gaussian processes for machine learning. 2006. *The MIT Press, Cambridge, MA, USA*, 38:715–719, 2006.

[27] H. Röder, R. Silver, D. Drabold, and J. J. Dong. Kernel polynomial method for a nonorthogonal electronic-structure calculation of amorphous diamond. *Physical Review B*, 55(23):15382, 1997.

[28] C. Sanderson and R. Curtin. Armadillo: a template-based c++ library for linear algebra. *Journal of Open Source Software*, 1(2):26–32, 2016.

[29] R. B. Sidje and Y. Saad. Rational approximation to the fermi–dirac function with applications in density functional theory. *Numerical Algorithms*, 56(3):455–479, 2011.

[30] K. Takahashi. Formation of sparse bus impedance matrix and its application to short circuit study. In *Proc. PICA Conference, June, 1973*, 1973.

[31] J. M. Tang and Y. Saad. Domain-decomposition-type methods for computing the diagonal of a matrix inverse. *SIAM Journal on Scientific Computing*, 33(5):2823–2847, 2011.

[32] J. M. Tang and Y. Saad. A probing method for computing the diagonal of a matrix inverse. *Numerical Linear Algebra with Applications*, 19(3):485–501, 2012.

APPENDIX A

A.1 Stopping Tolerance Tests Results

Table A.1: Effect of Stopping Tolerance

matrix	algorithm	errtol	#iter	error	time	diff
mcca	GlobMinResIter	1.e-05	11	5.49E-05	855.6	5.76E-06
		1.e-10	22	7.66E-10	870.9	9.01E-11
		1.e-15	33	9.49E-15	1231.4	4.72E-15
	GlobSteepDesc	1.e-05	98	1.24E-05	1503	1.09E-09
		1.e-10	200	1.07E-08	3481.6	9.44E-14
		1.e-15	200	1.07E-08	3600.5	9.44E-14
	ColMinResIter	1.e-05	4	1.15E-08	291.9	9.42E-07
		1.e-10	7	1.37E-13	449.9	7.42E-12
		1.e-15	10	3.30E-19	1112.4	3.07E-15
dw256B	GlobMinResIter	1.e-05	2	1.99E-04	700.5	2.15E+00
		1.e-10	4	7.46E-10	1712	6.57E-06
		1.e-15	6	2.06E-15	1997	2.90E-10
	GlobSteepDesc	1.e-05	4	3.40E-05	404.8	1.05E+00
		1.e-10	8	5.08E-10	882.2	9.63E-06
		1.e-15	12	7.99E-15	1202	1.40E-10
	ColMinResIter	1.e-05	2	2.39E-08	560	1.37E-01
		1.e-10	2	1.03E-13	1072	2.43E-06
		1.e-15	2	3.50E-19	1074	7.84E-11
cage7	GlobMinResIter	1.e-05	7	1.67E-04	1522.1	5.37E+00
		1.e-10	16	6.13E-10	3476	1.24E-05
		1.e-15	24	7.65E-15	3656	1.58E-10
	GlobSteepDesc	1.e-05	14	1.01E-04	1668.7	6.99E+00
		1.e-10	30	1.39E-09	3051	4.51E-05

Table A.1 Continued

		1.e-15	47	1.05E-14	5048	6.82E-10
	ColMinResIter	1.e-05	5	1.42E-06	996	5.04E-01
		1.e-10	9	3.67E-12	1762	3.03E-06
		1.e-15	12	2.22E-19	2166	1.18E-10
mesh1e1	GlobMinResIter	1.e-05	12	5.99E-05	25.9	9.19E-02
		1.e-10	22	4.95E-10	58.1	3.64E-07
		1.e-15	28	1.31E-15	889.6	7.35E-12
	GlobSteepDesc	1.e-05	37	6.11E-05	42.7	5.84E-02
		1.e-10	80	6.26E-10	81.7	8.35E-07
		1.e-15	123	6.43E-15	208.6	1.19E-11
	ColMinResIter	1.e-05	9	6.19E-08	184.5	4.22E-02
		1.e-10	16	8.97E-14	379.2	3.69E-07
		1.e-15	23	1.87E-19	525.9	5.92E-12
bfwb782	GlobMinResIter	1.e-05	13	1.81E-04	13.2	204.884
		1.e-10	28	2.60E-09	23.2	1.82E+00
		1.e-15	44	1.92E-14	37	1.67E-05
	GlobSteepDesc	1.e-05	37	8.88E-06	24.5	257.955
		1.e-10	88	8.86E-11	57.2	2.57E+00
		1.e-15	139	1.63E-15	91.2	4.84E-05
	ColMinResIter	1.e-05	8	9.17E-07	9.4	12.212
		1.e-10	15	7.81E-12	13.9	1.52E-01
		1.e-15	22	1.02E-16	13.9	6.92E-06
cage6	GlobMinResIter	1.e-05	8	3.43E-05	37.8	2.69E+00
		1.e-10	16	5.82E-10	68	3.29E-05
		1.e-15	24	3.05E-15	269.4	3.91E-10
	GlobSteepDesc	1.e-05	14	9.18E-05	67.1	4.16E+00
		1.e-10	30	5.76E-10	388.6	4.11E-05
		1.e-15	46	6.79E-15	405.5	4.52E-10
	ColMinResIter	1.e-05	6	1.09E-07	372.2	4.18E-01
		1.e-10	10	1.55E-12	812.5	4.07E-06
		1.e-15	14	8.28E-18	1313	1.02E-10
Trefethen_150	GlobMinResIter	1.e-05	6	1.10E-04	123.6	5.93E-02
		1.e-10	12	5.36E-11	1210.4	1.06E-06
		1.e-15	18	1.16E-14	1838.5	1.58E-11
	GlobSteepDesc	1.e-05	45	1.06E-04	1463.3	1.24E-01

Table A.1 Continued

		1.e-10	102	1.02E-09	3365.3	1.18E-06
		1.e-15	158	1.21E-14	3719.8	6.65E-10
	ColMinResIter	1.e-05	4	8.26E-09	136.6	1.02E-02
		1.e-10	8	1.33E-16	749.4	1.15E-07
		1.e-15	15	1.27E-21	865.6	3.54E-12
mesh1em6	GlobMinResIter	1.e-05	15	4.59E-05	69.5	6.28E-02
		1.e-10	32	3.71E-10	281.7	6.38E-07
		1.e-15	48	6.06E-15	325.9	1.30E-11
	GlobSteepDesc	1.e-05	47	6.50E-05	64.6	1.20E-01
		1.e-10	102	6.82E-10	234.3	1.51E-06
		1.e-15	158	5.82E-15	235.4	2.38E-11
	ColMinResIter	1.e-05	10	3.70E-07	145.8	4.60E-02
		1.e-10	19	3.65E-17	300.9	2.47E-07
		1.e-15	28	6.60E-14	563.7	1.14E-11
bfwb62	GlobMinResIter	1.e-05	12	3.91E-05	28.9	0.4616
		1.e-10	19	7.02E-10	41.2	8.57E-03
		1.e-15	26	3.11E-15	64.9	7.03E-07
	GlobSteepDesc	1.e-05	40	7.08E-05	104.4	10.383
		1.e-10	87	7.73E-10	120	9.14E-02
		1.e-15	135	6.63E-15	192.9	1.93E-06
	ColMinResIter	1.e-05	7	3.28E-06	109.6	0.3203
		1.e-10	13	1.56E-09	191.7	3.06E-02
		1.e-15	19	8.83E-13	310.4	4.91E-07

A.2 Initial Guess Tests Results

Table A.2: Effect of Initial Guess.

matrix	algorithm	Initial guess	#iter	Sinv sparsity	time	error
Bfwb62	GlobMinResIter	αA^T	18	565	25.9	7.02E-06
		αI	17	753	24	2.20E-06
		0	19	565	23	7.02E-06
	GlobSteepDesc	αA^T	84	616	71.2	6.37E-06
		αI	86	719	161.6	7.73E-06
		0	87	745	124.8	7.73E-06
	ColMinResIter	αA^T	12	566	222.7	9.72E-10
		αI	17	764	285.1	6.30E-06
		0	13	565	212.3	1.56E-05
Bfwb782	GlobMinResIter	αA^T	27	14016	23496.3	2.43E-05
		αI	27	20979	22899	2.50E-05
		0	28	14016	23140.5	2.43E-05
	GlobSteepDesc	αA^T	84	21250	53053.7	2.62E-05
		αI	87	19556	59144.50	2.61E-05
		0	88	19738	59766.7	2.61E-05
	ColMinResIter	αA^T	14	14387	13213.5	2.61E-05
		αI	19	21061	12595.6	2.61E-05
		0	15	14437	12599.9	2.61E-05
Cage6	GlobMinResIter	αA^T	15	1121	835.8	5.82E-06
		αI	16	1209	274.1	3.24E-06
		0	16	1121	89.2	5.82E-06
	GlobSteepDesc	αA^T	28	1148	322.4	5.53E-06
		αI	29	1388	123.5	5.76E-06
		0	30	1543	157.9	5.76E-06
	ColMinResIter	αA^T	10	1119	761.6	4.56E-08
		αI	14	1204	585.2	6.68E-09
		0	10	1118	328.2	1.55E-08
Cage7	GlobMinResIter	αA^T	15	3225	2861.4	6.13E-06
		αI	15	3279	2298	1.06E-05
		0	16	3225	1730	6.13E-06
	GlobSteepDesc	αA^T	28	3267	3477.2	1.54E-05
		αI	29	3389	3.013.5	1.39E-05
		0	30	4033	3.282.2	1.39E-05

Table A.2 Continued

	ColMinResIter	αA^T	8	3229	1552.6	2.82E-07
		αI	12	3283	1890.5	1.74E-08
		0	9	3228	1893.1	3.67E-08
Dw256b	GlobMinResIter	αA^T	3	571	1085.3	7.46E-06
		αI	4	571	1499.8	5.67E-07
		0	4	571	1505.8	7.46E-06
	GlobSteepDesc	αA^T	7	571	789.1	3.62E-06
		αI	7	571	865.8	5.08E-06
		0	8	626	1299.1	5.08E-06
	ColMinResIter	αA^T	2	571	931	1.69E-09
		αI	2	571	600.6	1.82E-10
		0	2	571	808.1	1.03E-09
mcca	GlobMinResIter	αA^T	21	1172	670.7	7.66E-06
		αI	19	2736	1165.5	9.44E-06
		0	22	1172	1141.5	7.66E-06
	GlobSteepDesc	αA^T	200	853	3902.8	7.80E-05
		αI	200	1845	3091	9.81E-05
		0	200	1965	2.902	1.07E-04
	ColMinResIter	αA^T	6	1204	621.6	2.26E-07
		αI	11	3052	619.7	1.76E-09
		0	7	1192	273.8	1.37E-09
Mesh1e1	GlobMinResIter	αA^T	21	416	64.3	6.13E-06
		αI	22	509	33.7	1.06E-05
		0	22	416	44.8	6.13E-06
	GlobSteepDesc	αA^T	75	395	92.6	1.54E-05
		αI	79	497	81.3	1.39E-05
		0	80	510	257.5	1.39E-05
	ColMinResIter	αA^T	15	420	286.9	2.82E-07
		αI	19	518	251.5	1.74E-08
		0	16	418	201.6	3.67E-08
Mesh1em6	GlobMinResIter	αA^T	31	467	249.2	3.71E-06
		αI	31	617	41	5.62E-06
		0	32	467	36.4	3.71E-06
	GlobSteepDesc	αA^T	97	434	104.6	6.00E-06
		αI	101	556	131.2	6.82E-06

Table A.2 Continued

		0	102	570	143.2	6.82E-06
	ColMinResIter	αA^T	18	468	251.8	4.15E-07
		αI	22	614	283.3	
		0	19	470	266.2	
Trefethen150	GlobMinResIter	αA^T	11	350	1223.2	5.36E-07
		αI	14	431	507.4	3.20E-06
		0	12	350	410.9	5.36E-07
	GlobSteepDesc	αA^T	100	335	3637.6	1.01E-05
		αI	101	494	4628.4	1.02E-05
		0	102	508	2234.9	1.02E-05
	ColMinResIter	αA^T	7	349	524.7	1.20E-15
		αI	7	484	368	2.84E-11
		0	8	354	512.7	1.33E-12

A.3 Dropping Strategy Test Results

Table A.3: Effect of Dropping Strategy.

matrix	algorithm	Drop strat	#iter	Sinv sparsity	time	error
Bfwb62	GlobMinResIter	numerical	19	565	305.3	7.02E-10
		stuctural	19	1388	98.1	7.02E-10
		hybrid	19	565	297.5	7.02E-10
	GlobSteepDesc	num	87	745	163.7	7.73E-10
		stuct	87	1388	377.7	7.73E-10
		hybrid	87	642	450.1	7.73E-10
	ColMinResIter	num	13	565	295.9	1.56E-09
		stuct	13	1388	306.5	1.31E-13
		hybrid	13	1016	670.1	1.59E-09
Bfwb782	GlobMinResIter	num	28	14016	18686.8	7.02E-10
		stuct	28	212902	21615.7	7.02E-10
		hybrid	28	14016	21166	7.02E-10
	GlobSteepDesc	num	88	19738	45175.2	7.73E-10
		stuct	88	212902	44132.8	7.73E-10
		hybrid	88	19194	48736.1	7.73E-10
	ColMinResIter	num	15	14437	10582.4	1.56E-09
		stuct	14	212902	9222.9	1.31E-13
		hybrid	14	26137	94842.8	1.59E-09
Cage6	GlobMinResIter	num	16	1121	427.9	5.52E-10
		stuct	16	5827	300.5	5.52E-10
		hybrid	16	1121	1878.1	5.52E-10
	GlobSteepDesc	num	30	1543	113.3	7.76E-10
		stuct	30	5889	728.1	7.76E-10
		hybrid	30	1336	482.6	7.76E-10
	ColMinResIter	num	10	1118	668.2	3.43E-12
		stuct	10	5827	309.2	2.34E-12
		hybrid	10	2011	1641.60	4.26E-12
Cage7	GlobMinResIter	num	15	3279	2489.4	6.13E-10
		stuct	15	70174	2765.6	6.13E-10
		hybrid	15	50611	3480.6	6.13E-10
	GlobSteepDesc	num	29	3389	4357.5	1.39E-09
		stuct	29	70174	3961.4	1.39E-09

Table A.3 Continued

		hybrid	28	52285	4464	1.39E-09
	ColMinResIter	num	8	70174	2288.1	3.67E-12
		stuct	8	70174	2048.6	4.48E-12
		hybrid	8	69918	12212.4	4.38E-12
Dw256b	GlobMinResIter	num	4	571	2000	7.46E-10
		stuct	4	17886	1367.5	7.46E-10
		hybrid	4	571	1942.3	7.46E-10
	GlobSteepDesc	num	8	626	953.1	5.08E-10
		stuct	8	17886	1161.6	5.08E-10
		hybrid	8	579	1437	5.08E-10
	ColMinResIter	num	2	571	1155.3	1.03E-13
		stuct	1	17886	161.1	1.23E-13
		hybrid	1	632	908.7	1.23E-13
mcca	GlobMinResIter	num	22	1172	1007	7.66E-10
		stuct	22	9561	1646.8	7.66E-10
		hybrid	22	4264	3191.3	7.66E-10
	GlobSteepDesc	num	200	1965	3362.9	1.07E-08
		stuct	200	9561	4944.7	1.07E-08
		hybrid	200	4418	4613.8	1.07E-08
	ColMinResIter	num	7	9561	455.9	1.37E-13
		stuct	6	9561	264.5	1.63E-14
		hybrid	6	6669	2058.6	1.82E-13
Mesh1e1	GlobMinResIter	num	22	416	41.5	4.95E-10
		stuct	22	1928	222.2	4.95E-10
		hybrid	22	416	190.7	4.95E-10
	GlobSteepDesc	num	80	510	198.1	6.26E-10
		stuct	80	1928	673.3	6.26E-10
		hybrid	80	465	306.7	6.26E-10
	ColMinResIter	num	16	418	187	8.97E-14
		stuct	16	1928	157.7	7.30E-14
		hybrid	16	822	693.7	8.46E-14
Mesh1em6	GlobMinResIter	num	32	467	78.8	3.71E-10
		stuct	32	1928	126.7	3.71E-10
		hybrid	32	467	137.9	3.71E-10
	GlobSteepDesc	num	102	570	234.4	6.82E-10

Table A.3 Continued

		stuct	102	1928	579.4	6.82E-10
		hybrid	102	506	410.9	6.82E-10
	ColMinResIter	num	19	470	215.6	3.65E-17
		stuct	19	1928	192	2.56E-17
		hybrid	19	883	773	2.73E-17
Trefethen150	GlobMinResIter	num	12	350	768.8	5.36E-11
		stuct	12	22500	484.8	5.36E-11
		hybrid	12	20266	652.4	5.36E-11
	GlobSteepDesc	num	102	508	2338.2	1.02E-09
		stuct	102	22500	2601	1.02E-09
		hybrid	102	15088	3152.8	1.02E-09
	ColMinResIter	num	8	354	558.8	1.33E-16
		stuct	8	22500	421.5	1.33E-16
		hybrid	8	22004	2.412	1.33E-16

A.4 Diagonal of Inverse Computation using DS Factorization

```
 define ARMADONTUSEWRAPPER

 include  stdio.h
 include  stdlib.h
 include  cstdlib
 include  iostream
 include  ctime
 include  armadillo
 include  omp.h

/  Number of threads used  /
 define NRTHREADS 1


using namespace std;
using namespace arma;




int main(int argc, char  argv)

    mat Ar,Aii,D,S,R,Dinv,Sinv,M,Aj,St,I,Inj;
    int i,n,nj,p,j,ind1, ind2=0,itercount,k;
    uvec nnzind;
    uvec ind;
    int tid = 0;
    int numthreads=0;
    double nrj,Derror,DSerror,Serror;
    vec ej,Avj,mj,rj,pj,ei,temp,directdiag,mydiag;
    int ni,nnz;
    double diagentry;
    double totalitercount,totalerror, alpha ,aj,tol;

    double starttime, stoptime;
```

```
double facttime,invtime, diagtime, totaltime;

char  matrixName = argv[1];
int partition = atoi( argv[2]);

Ar.load(matrixName);
SpMat double  Asp = spmat(Ar);

n = Ar.nrows;  // matrix size
p = partition;              //partition
nj = n/p;       // block size

ind1=ind2=0;

//FACTORIZATION


D = zeros(n,n);
R = zeros(n,n);
S = zeros(n,n);
Dinv = zeros(n,n);
I = eye(n,n);
Inj = eye(nj,nj);
Aj = zeros(nj,nj);


spmat Ds(n,n);
spmat Ss(n,n);
spmat Rs(n,n);
spmat Dinvs(n,n);
spmat Sinvs(n,n);
spmat Ms(n,n);
spmat Ajs(nj,nj);
spmat Sts(n,n);

spmat Is = speye spmat (n,n);
spmat Ij = speye spmat (nj,n);
```

```
    spmat Injs = speye spmat (nj,nj);

    ej = zeros(n);
    Avj = zeros(n);
    mj = zeros(n);
    rj = zeros(n);
    pj = zeros(n);
    temp = zeros(n);
    mydiag=zeros(n);

    spmat ejs(n,1);
    spmat Avjs(n,1);
    spmat mjs(n,1);
    spmat rjs(n,1);
    spmat pjs(n,1);
    spmat mydiags(n,1);

    ei = zeros(n);
    Aii = zeros(1,1);

    aj = 0;
    totalerror = 0;
    totalitercount = 0;
    itercount = 0;
    ni=200;

    tol= 0.00000001;



    ompsetnumthreads(NRTHREADS);
pragma omp parallel shared(Asp,Ds,Dinvs,Ss,Sts,Rs,n,p,mydiag,alpha,tol)

        private(Ms,k,ind,i,j,tid,rjs,pjs,ejs,aj,nrj,ind1,ind2,Ajs,Aii,Sinvs)
```

```
ifdef OPENMP
        numthreads = ompgetnumthreads();
endif

ifdef OPENMP
        tid = ompgetthreadnum();
endif

        if (tid == 0)
            starttime = ompgetwtime();

pragma omp for
        for (j = 0; j  p; j++)

            ind1 = j nj + 1  1;
            ind2 = (j + 1)  nj    1;
            //find Ajs
            Ajs=Asp(span(ind1,ind2),span(ind1,ind2));
            Ds(span(ind1,ind2),span(ind1,ind2))=Ajs;
            Dinvs.submat(ind1,ind1,ind2,ind2) = spsolve(Ajs,Inj);


        //compute R
        Rs = Asp    Ds;

        Ss = Is + Dinvs Rs;

        if (tid == 0)
            stoptime = ompgetwtime();
            facttime = stoptime    starttime;


        //COLMINRESITER

        if (tid == 0)
            starttime = ompgetwtime();
```

```
        Sts = trans(Ss);
        alpha = trace(Ss) / trace(Sts    Ss);
        Ms = alpha    Is;

pragma omp for schedule(static,10) firstprivate(mjs)

        for (j = 0; j    n; j++)

            ejs = Is.col(j);
            mjs = Ms    ejs;
            rjs = ejs    Ss    mjs;
            pjs = Ss    rjs;


            for(i=0;i  ni;i++)

                aj = dot(rjs,pjs) / dot(pjs,pjs);
                mjs = mjs + aj  rjs;
                rjs = rjs    aj  pjs;
                pjs = Ss    rjs;

                nrj = norm(rjs,"fro");

                if (nrj    tol)
                    break;



        Ms.col(j) = mjs;

    Sinvs = Ms;
    if (tid == 0)
        stoptime = ompgetwtime();
        invtime = stoptime    starttime;

    //DIAGONAL
    if (tid == 0)
```

```
        starttime = ompgetwtime();

pragma omp for
    for(i=0;i n;i++)
        Aii = Sinvs.row(i) Dinvs.col(i);
        mydiag(i) = Aii.at(0,0);

    if (tid == 0)
        stoptime = ompgetwtime();
        diagtime = stoptime   starttime;
        totaltime = facttime + invtime + diagtime;

        Dinv = spsolve(Ds,I);
        Sinv = spsolve(Ss,I);

        Derror = norm(Dinv   Dinvs,"fro");
        DSerror = norm(Asp   Ds Ss,"fro");
        Serror = norm(Sinv   Sinvs,"fro");
        mat Aspinv = spsolve(Asp,I);
        totalerror = norm(Aspinv.diag()   mydiag,"fro");


return 0;
```