IMPROVED PHYSARUM POLYCEPHALUM SHORTEST PATH ALGORITHM WITH PRECONDITIONED ITERATIVE METHODS

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

BY

HAMIDE HANDE KESKINER

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

SEPTEMBER 2015

Approval of the thesis:

IMPROVED PHYSARUM POLYCEPHALUM SHORTEST PATH ALGORITHM WITH PRECONDITIONED ITERATIVE METHODS

submitted by **HAMIDE HANDE KESKINER** 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. Bülent Karasözen
Department of Mathematics, METU ______________

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

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Department, METU ______________

Assoc. Prof. Dr. Burak Aksoylu
Department of Mathematics, TOBB ETU ______________

Assist. Prof. Dr. Ahmet Burak Can
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 this work.

Name, Last Name: HAMIDE HANDE KESKINER

Signature :

ABSTRACT

IMPROVED PHYSARUM POLYCEPHALUM SHORTEST PATH ALGORITHM WITH PRECONDITIONED ITERATIVE METHODS

Keskiner, Hamide Hande
M.S., Department of Computer Engineering
Supervisor : Assoc. Prof. Dr. Murat Manguoğlu

September 2015, 70 pages

Algorithms for finding the shortest path has many applications in Computer Science, or in other areas of science and engineering. Network optimizations, artificial intelligence and robotics are just a few examples where efficient computation of the shortest path is needed. Various algorithms have been proposed to solve this fundamental problem. Physarum Solver is biologically inspired method that deals with this problem. In the end, a sparse linear system needs to be solved at each iteration of the algorithm. Direct and iterative solvers are two main classes of algorithms for solving sparse linear systems. Direct solvers are robust but they could consume a lot memory due to fill-in. In this thesis, Physarum Polycephalum Shortest Path algorithm is improved using preconditioned iterative methods. We study the convergence behavior as well as memory consumption of various solvers and preconditioners. We show that preconditioned iterative solvers are quite robust and requires much less memory and solution time.

Keywords: Physarum Polycephalum, Shortest Path, Preconditioned Iterative Methods

ÖZ

ÖN KOŞULLU YİNELEMELİ YÖNTEM İLE GELİŞTİRİLMİŞ PHYSARUM POLYCEPHALUM EN KISA YOL ALGORİTMASI

Keskiner, Hamide Hande
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Doç. Dr. Murat Manguoğlu

Eylül 2015 , 70 sayfa

En kısa yol problemi algoritmalarının Bilgisayar Bilimi içerisinde veya bilim ve mühendislik alanında pek çok uygulaması bulunmaktadır. Ağ optimizasyonu, yapay zeka ve robotik en kısa yol probleminin uyguluma alanlarına örnektir. Pek çok algoritma bu problemi çözebilmek için öne sürülmüştür. Physarum Çözümü en kısa yol problemini çözebilen biyolojik olarak esinlenilmiş bir yöntemdir. Nihayetinde, algoritma içerisinde her iterasyonda karşımıza çözülmesi gereken seyrek doğrusal sistem çıkmaktadır. Direkt ve yinelemeli çözücüler iki ana seyrek doğrusal sistem çözücüleridir. Direkt Çözücüler dayanıklı olmasına rağmen doldurulmuş hücreler sebebiyle fazla hafıza harcar. Bu tez çalışmasında, Physarum Polycephalum en kısa yol algoritması ön koşullu yinelemeli yöntemler ile geliştirilmektedir. Pek çok doğrusal sistem çözücü ve önkoşulun yakınsama davranışı ve hafıza tüketimi üzerinde çalışılmıştır. Ön küşullu yinelemeli çözücülerin gayet dayanıklı olduğu ve daha az hafızaya ve zamana ihityaç duyduğu gösterilmiştir.

Anahtar Kelimeler: Physarum Polycephalum, En Kısa Yol Problemi, Ön Koşullu Yinelemeli Yöntemler

To my family

ACKNOWLEDGMENTS

I would like to thank my supervisor Assoc. Prof. Dr. Murat Manguoğlu for his constant support and guidance. I would also like to acknowledge Turkish Academy of Sciences Distinguished Young Scientist Award M.M/TUBA-GEBIP/2012-19 for providing the computing platform used in this thesis.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

BiCG	Biconjugate Gradient
BiCGStab	Biconjugate Gradient Stabilized
ICHOL	Incomplete Cholesky
ILU	Incomplete LU
PCG	Preconditioned Conjugate Gradient
QMR	Quasi-Minimal Residual

CHAPTER 1

INTRODUCTION

The shortest path problem in graph theory is to find the minimum length route which connects two vertices. The problem differs according to type of graph that is handled. Graph can be undirected, directed, weighted or their combination or etc. Solution technique of the problem may vary from one type to another according to the application field of the problem.

The shortest path problem has emerged in many fields such as network optimizations [7], artificial intelligence [16] and robotics [9]. There have been many studies to come up with this problem. Dijkstra and Bellman-Ford algorithms can be given as an example of most known solution of the problem. As a matter of fact, development in the theory of the shortest path algorithms is still in progress. After all these studies, many bioinspired algorithms have appeared, for instance, genetic algorithm [1], ant colony algorithm [6] and Physarum Polycephalum algorithm [14].

Physarum Solver algorithm inspired from an amoeba-like organism behavior [11] in a labyrinth while finding shortest path by Tero, Kobayashi and Nakagaki [14]. They built a maze filled by Physarum Polycephalum and placed food sources at two locations. After a while, plasmodium form a path between two food sources with minimum length. Such behavior of this organism is due to its biological structure. Physarum Polycephalum is a unicellular amoeba like organism. It contains a network of tubes which transmits signals and nutrients all over its body. When the food sources were presented to plasmodium, it concentrated to food sources to absorb nutrients and the remaining tubes are the shortest path.

Physarum Solver differs from the classical shortest path algorithms in terms of being an iterative approach to the solution. In execution, it gradually eliminates the paths that are not between start and ending node or that are not the shortest path. Then, it reduces longer paths that connect start and ending node and reinforces the shortest one. Thus, it easily handles problems that have more than one shortest path contrary to some classical shortest path algorithms such as Dijkstra. Also, it is flexible according to classical algorithms because the stopping criteria of the algorithm changes accuracy rate of the solution. In classical algorithms, stopping criteria of the algorithm is not interrupted, it stops whenever the execution ends. On the contrary, Physarum Solver stops the execution according to stopping criteria defined before the execution.

In the Physarum Solver, a sparse linear system has to be solved in every iteration. As far as we know, in the literature, only direct solvers are used and large scale graphs are not studied. In this work we propose and use iterative solvers and study various preconditioning techniques. The reason is that direct solvers consume a lot of memory due to fill-in and great amount of time. In general, even if direct solvers are robust, namely they can solve systems that iterative solvers can not, iterative solvers have good performance on this algorithm.

We analyzed the convergence behavior, time and memory consumptions of different iterative methods and preconditioners. Results are shown where we compare the convergence rate, computation time and memory requirements for each of these methods against direct solver as well as against one another. At the end, we show that preconditioned iterative solvers are quite robust and requires much less memory and computation time.

This thesis is organized as follows. In Chapter 2, detailed explanation about Physarum Polycephalum shortest path algorithm and introduction about solving linear systems are given. In the following chapter, Methods and Motivation are described. In Chapter 4, programming and computing environment, and the experimental results are presented. Finally, in Chapter 5, conclusion and future work is stated.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Physarum Polycephalum Shortest Path Algorithm

The plasmodium of true slime mould Physarum Polycephalum is a unicellular amoeba-like organism. It contains a tube network perform circulation of signals and nutrients through the body. Two food sources were positioned at different locations on the plasmodium was spread over the entire agar surface. Then, starved plasmodium concentrated at the food sources to absorb nutrients and the shape of the plasmodium became a path that connects two food sources in minimum length [11].

Tero, Kobayashi and Nakagaki claim that reason behind the organism behavior is that hydrostatic pressure along the tube [14]. If organism forms the thickest, shortest tubes, it enhances its survival for the following reasons: (1) the body area cover food sources and absorption nutrients is maximized and (2) the communication between the locations of two food sources is at most effective.

According to the experiment performed Tero et al., two rules specify changes in the tubular structure of the plasmodium [12]. The tubes that are not connected to a food source inclined to eliminate and if two or more tubes connect the same food sources, the lengthier tubes inclined to eliminate.

The tube network are formed in a specific direction driven by hydrostatic pressure due to rhythmic contractions. When food sources are placed at the agar surface filled with plasmodium, the oscillations between a food source and the adjacent tube are occurred. The sol in the food sources flows in and out of the tube, exchanges between the two food sources. The sol flows through the tube that is because the pressure oscillation of food sources are different from each other. Thus, it can be said that one food source is the source and the other is the sink of sol flow [14].

2.1.1 Mathematical Model

In this section, Physarum Polycephalum shortest path algorithm mathematical model will be introduced. Mathematical model and physiological backgrounds are detailed in [14, 10]

Let $G = (V, E)$ be a graph where $V = v_0, \ldots, v_n$ is the set of nodes and e_{ij} where $i \neq j$ is the edge between v_i and v_j if it exists. There is a function L from E to R^+ calculated length and assume that $L(e_{ij}) = L(e_{ji})$. G is considered a flow network v_0 to v_n which source s is equal to v_0 and target t is equal to v_n. Assume that there exactly one source and one target. For a path $P = v_{\beta_0} \cdots v_{\beta_k}$ in G, length of the P calculated as follows:

$$L(P) = \sum_{i=0}^{k-1} L(e_{\beta_i \beta_i + 1}).$$

Let τ indicate the time variable. For each node, the variable $p_i(\tau)$ is pressure. For each edge, $D_{ij}(\tau)$ is conductivity, $Q_{ij}(\tau)$ is flux and L_{ij} is length. While p_i, D_{ij} and Q_{ij} are variables changing with time, L_{ij} is a positive constant variable. Should be noted that D_{ij} is nonnegative for each edge.

The flux equation is an affinity of *Ohm's Law* for electric circuits and Q_{ij} is computed in this way:

$$Q_{ij} = \frac{D_{ij}}{L_{ij}}(p_i - p_j) = g_{ij}(p_i - p_j), \tag{2.1}$$

where $g_{ij} = \frac{D_{ij}}{L_{ij}}$ is conductance of the edge e_{ij}. It can be seen in the equation that

$Q_{ij} = -Q_{ji}$ so if we applied the *Kirchhoff's Law* at each node:

$$\sum_{j \neq i} Q_{ij} = \begin{cases} I_0 & \text{if } i = 0, \\ 0 & \text{if } 0 < i < n, \\ -(I_0) & \text{if } i = n, \end{cases} \tag{2.2}$$

where I_0 is the flux from the source node and it is a positive constant.

According to experimental results that tubes with smaller fluxes disappear while tubes with larger fluxes are reinforced [14]. Therefore, conductivity changes in time with regard to the flux Q_{ij}. Thus, the disappearance of the tubes is revealed by the destruction of the conductivity edges. The conductivity can be calculated as follows:

$$\dot{D_{ij}} = |Q_{ij}| - D_{ij}, \tag{2.3}$$

where $\dot{x}$ denotes the derivative $\frac{d_x}{d_\tau}$.

After substitution equation 2.1 in equation 2.2

$$\sum_{j \neq i} g_{ij}(p_i - p_j) = \begin{cases} I_0 & \text{if } i = 0, \\ 0 & \text{if } 1 \leq i \leq n-1, \end{cases} \tag{2.4}$$

where $p_n = 0$ and $g_{ij} > 0$ which give rise to the following linear system of equations:

$$Ap = b, \tag{2.5}$$

where

$$p = (p_0, p_1, \dots, p_{n-1})^t, \quad b = (I_0, 0, \dots, 0)^t, \tag{2.6}$$

and $A = (A_{ij})$ is a square matrix of order n and computed as follows:

$$A_{ij} = \begin{cases} \sum_{l \neq i} g_{il} & \text{if } i = j, \\ -g_{ij} & \text{otherwise} \end{cases} \tag{2.7}$$

where $i, j = 0, \dots, n-1$. It is proved that the coefficient matrix A is symmetric nonsingular M-matrix in [10].

2.1.2 Algorithm

Algorithm 1 is the pseudocode for Physarum Polycephalum shortest path algorithm. It is clear that, it is an iterative solution for the shortest path problem. In every iteration flux Q_{ij}'s are changed so conductivity D_{ij}'s are also changed.

Algorithm 1 Physarum Polycephalum Shortest Path Algorithm

// L is an $n \times n$ matrix, L_{ij} denotes the length between v_i and v_j.
// s is the source node and t is the sink node.
// I_0 is the flux from the s.
$D_{ij} \leftarrow (0, 1](\forall i, j = 1, 2, \ldots, n)$
$Q_{ij} \leftarrow 0(\forall i, j = 1, 2, \ldots, n)$
$A_{ij} \leftarrow 0(\forall i, j = 1, 2, \ldots, n)$
$g_{ij} \leftarrow 0(\forall i, j = 1, 2, \ldots, n)$
$p_i \leftarrow 0(\forall i, j = 1, 2, \ldots, n)$
$b = \left[I_0, 0, 0, \ldots, 0\right]^T$
$count \leftarrow 1$
repeat
    $g_{ij} \leftarrow \frac{D_{ij}}{L_{ij}}(\forall i, j = 1, 2, \ldots, n)$
    $A_{ij} = \begin{cases} \sum_{l \neq i} g_{il} & \text{if } i = j, \\ -g_{ij} & \text{otherwise} \end{cases}$
    solve $Ap = b$ where $(\forall i, j = 1, 2, \ldots, n-1)$
    $p_n \leftarrow 0$
    $Q_{ij} \leftarrow g_{ij} \times (p_i - p_j)$
    $D_{ij} \leftarrow |Q_{ij}|$
    $count \leftarrow count + 1$
until $a\, termination\, criterion\, is\, met$

Execution continues until the termination criteria is met. To break out the execution, conductivity values of nodes which are composed of minimum length path, are monitored to determine that there is no changing in values. If so, iteration halts. Another approach to end execution is limiting the iteration count. When iteration count has reached maximum value, execution ends.

Clearly, for large problems the most time consuming operation of the algorithm is expected to be Step 14 where the sparse linear system is solved. In the literature, there is not much attention paid on various solution strategies, simply a dense or a sparse direct solver is often used [17, 18, 15].

2.2 Linear Algebra Terminology

This section includes definition of some main and most frequent notions in linear algebra used throughout this thesis.

- A matrix is sparse if it is primarily composed of zeros.
- A matrix A is symmetric if it is equal to its transpose $A = A^T$.
- A matrix is positive-definite if $x^T Ax > 0$ for all non-zero vectors $x \in \mathbb{R}^n$.
- A *Krylov subspace of dimension n* composed of a linear combination of the vectors $b, Ab, \ldots, A^{n-1}b$ where A is matrix and b is a vector.
- p_{k-1} and p_k are A-conjugate or conjugate with regard to A if $p_{k-1}Ap_k = 0$.
- A matrix is nonsingular if there is a unique solution given by $x = A^{-1}b$ where $Ax = b$.

2.3 Solving Linear Systems

Consider a linear system.

$$Ax = b$$

where $A \in R^{n \times n}$ is a large and sparse nonsingular coefficient matrix and $x, b \in R^n$ are the solution and right-hand-side vectors.

Sparse linear systems can be solved using direct or iterative methods. In direct solvers, Gaussian Elimination or other factorization techniques are used to reduce the system in a simpler form, then the solution is obtained finite number of arithmetic operations. On the other hand, iterative methods start with an initial guess and try to get more accurate solutions to a linear system at each step.

Direct solvers are widely preferred due to their reliability and being predictable in time and memory usage. Nonetheless, direct solvers usually require a lot of memory for solving large scale problems. Problems appearing from discretization of three dimensional partial differential equations is just one example of such problems where direct solvers usually have difficulty. For these problems, iterative methods could be used due to the fact that they usually require less storage and hence less time to solution. However, iterative solvers are not robust as direct solvers [3]. Iterative solvers without a preconditioner is usually not practical since they require a lot of iterations to converge. Therefore, preconditioners are used for both improving the number of iterations and the robustness.

Preconditioning involves transforming the linear system into another system where the eigenvalue distribution of the coefficient matrix is more favorable (clustered around 1) for iterative solvers to converge. The following shows left preconditioning,

$$M^{-1}Ax = M^{-1}b$$

which has the same solution vector x as the original system. But the coefficient matrix $M^{-1}A$ may be better conditioned than A if M is chosen properly.

There are many options to find preconditioning a matrix M, but it must satisfy a few constraints. First of all, it is expected to improve the convergence of the iterative method. Second, the eigenvalues of $M^{-1}A$ should be clustered more around 1 and M should be nonsingular [13].

2.4 Gaussian Elimination

Gaussian elimination is a method of solving linear systems by eliminating unknowns and reducing the problem into systems of equations where the coefficient matrices are lower and upper triangular. In other words, if we reduce a linear system $Ax = b$ into an equivalent system $Ux = g$. U is upper triangular matrix. Hence, the system can be solved using back-substitution in which we start from the lowest unknown and sweep upwards. To reduce system in the triangular form $Ux = g$, some row operations are performed.

CHAPTER 3

METHODS AND MOTIVATION

In previous section, background information about solving linear systems and preconditioning techniques are given. In this chapter, linear solvers and preconditioning methods which are used in implementation are explained.

3.1 Direct Solver

Direct solver is the solver of choice in the literature for Physarum Polycephalum shortest path algorithm. And hence, we use it as a basis comparison for our study in this thesis. Since the implementation were performed on Matlab, *mldivide* operation were used as direct solver. *mldivide* algorithm first determines which direct method to use based on the coefficient matrix type. Due to coefficient matrix in Physarum Solver is symmetric positive definite matrix, *mldivide* selects Cholesky Factorization. This operation in Matlab is using the Cholesky factorization implementation CHOLMOD [4] which is a part of SuitSparse package.

3.1.1 Cholesky Factorization

When coefficient matrix is symmetric and positive definite, the system can be solved using the Cholesky Factorization. The idea of the Cholesky Factorization is that reducing coefficient matrix A into product of lower triangular matrix L and its transpose such that

$$A = LL^T$$

so that solving linear system $Ax = b$ turn into solving two triangular systems

$$Ly = b \text{ and } L^T x = y.$$

- The system $Ly = b$, written as follows

$$Ly = \begin{pmatrix} l_{1,1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ l_{n,1} & \cdots & l_{n,n} \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

  (y_i) where $1 \leq i \leq n$ values are calculated as follows

$$\begin{aligned} l_{11}y_1 = b_1 \quad &\Rightarrow \quad y_1 = (l_{11})^{-1} b_1 \\ l_{21}y_1 + l_{22}y_2 = b_2 \quad &\Rightarrow \quad y_2 = {l_{22}}^{-1}(b_2 - l_{21}y_1) \\ &\vdots \\ \sum_{j=1}^{n} l_{nj}y_j = b_n \quad &\Rightarrow \quad y_n = {l_{nn}}^{-1}\left(b_n - \sum_{j=1}^{n-1} l_{nj}y_j\right) \end{aligned}$$

- Similarly the system $L^T x = y$, written as follows

$$L^T x = \begin{pmatrix} l_{1,1} & \cdots & l_{1,n} \\ \vdots & \ddots & \vdots \\ 0 & \cdots & l_{n,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

  (x_i) where $1 \leq i \leq n$ values are calculated using backward substitution, as follows

$$\begin{aligned} l_{nn}x_n = y_n \quad &\Rightarrow \quad x_n = (l_{nn})^{-1} y_n \\ &\vdots \\ \sum_{n-1}^{1} l_{j1}x_j = y_1 \quad &\Rightarrow \quad x_1 = {l_{11}}^{-1}\left(y_1 - \sum_{j=2}^{n} l_{j1}x_j\right) \end{aligned}$$

Algorithm for computing lower triangular matrix L is described below.

$$a_{ij} = (LL^T)_{ij} = \sum_{i=1}^{n} l_{ik}l_{jk} = \sum_{i=1}^{min(i,j)} l_{ik}l_{kj}, \ 1 \leq i, j \leq n$$

$l_{pq} = 0$ if $1 \le p < q \le n$ because L is lower triangular matrix. Because of A being symmetric, the upper triangular part of A identities must satisfy $i \le j$, and the entries l_{ij} simply like that

$$a_{ij} = \sum_{k=1}^{i} l_{ik} l_{jk},\ 1 \le i, j \le n$$

All entries of L can be computed by reading the columns of A in increasing order.

1. for $i = 1$, the first column of L is

$$\begin{aligned} a_{11} = l_{11} l_{11} \quad &\Rightarrow \quad l_{11} = \sqrt{a_{11}} \\ a_{12} = l_{11} l_{21} \quad &\Rightarrow \quad l_{21} = {l_{11}}^{-1} a_{12} \\ &\vdots \\ a_{1n} = l_{11} l_{n1} \quad &\Rightarrow \quad l_{n1} = {l_{11}}^{-1} a_{1n} \end{aligned}$$

2. for $i \ge 1$, while computing the $(i-1)$ columns of L assumed to be already computed. Then, i^{th} column of A is given below:

$$\begin{aligned} a_{ii} = \sum_{k=1}^{i} l_{ik} l_{ik} \quad &\Rightarrow \quad l_{ii} = \left(a_{ii} - \sum_{k=1}^{i-1} {l_{ik}}^{2} \right)^{\frac{1}{2}} \\ a_{ii+1} = \sum_{k=1}^{i} l_{ik} l_{i+1k} \quad &\Rightarrow \quad l_{i+1i} = {l_{ii}}^{-1} \left(a_{ii+1} - \sum_{k=1}^{i-1} l_{ik} l_{i+1k} \right) \\ &\vdots \\ a_{in} = \sum_{k=1}^{n} l_{ik} l_{nk} \quad &\Rightarrow \quad l_{ni} = {l_{ii}}^{-1} \left(a_{in} - \sum_{k=1}^{i-1} l_{ik} l_{nk} \right) \end{aligned}$$

3.2 Preconditioned Iterative Methods

Iterative methods can be expressed as

$$x^k = B x^{k-1} + c$$

where $k = 1, 2, \ldots$ is the iteration count. If a method has variables like B and c that does not change in every iteration, it is stationary method. Otherwise, it is a nonstationary method. Jacobi and Gauss-Seidel are examples of stationary methods, and Krylov Subspace family methods are examples of nonstationary methods [2].

Krylov Subspace methods can handle large general sparse matrices. Besides their performance can be enhanced using preconditioner techniques. As a consequence of this, Preconditioned Conjugate Gradient, Biconjugate Gradient Stabilized and Quasi Minimal Residual which are Krylov Subspace methods were used in Physarum Solver implementation. These methods are introduced and explained in this section.

3.2.1 Preconditioned Conjugate Gradient

Preconditioned Conjugate Gradient (PCG) method solves sparse linear systems whose coefficient matrix is symmetric and positive definite and there is a preconditioner M that is also symmetric and positive definite.

PCG starts with an initial guess of the solution, initial residual and initial search direction and at each iteration the iterates and the residuals are updated using the search directions. The residuals generated at each steps are orthogonal to Krylov subspace defined by b and A. Update scalars are used to ensure orthogonality conditions. These conditions diminishes the distance to true solution.

The iterates $x^{(i)}$ calculated in every iteration as follows

$$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)} \tag{3.1}$$

where α_i is update scalar and $p^{(i)}$ is search direction vector. The residuals $r^{(i)} = b - Ax^{(i)}$ are updated like that

$$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)} \text{ where } q^{(i)} = Ap^{(i)}. \tag{3.2}$$

The update scalar α_i is computed as

$$\alpha_i = \frac{{r^{(i-1)}}^T r^{(i-1)}}{{p^{(i)}}^T A p^{(i)}}. \tag{3.3}$$

The search direction $p^{(i)}$ is computed in every iteration using the residuals

$$p^{(i)} = r^{(i)} + \beta_{i-1} p^{(i-1)} \tag{3.4}$$

where $\beta_i = \frac{{r^{(i)}}^T r^{(i)}}{{r^{(i-1)}}^T r^{(i-1)}}$ ensures that $p^{(i)}$ and $Ap^{(i-1)}$ are orthogonal. Also, β_i make sure that $r^{(i)}$ and $r^{(i-1)}$ are orthogonal.

Algorithm 2 is the pseudocode for the Preconditioned Conjugate Gradient method [2]. If we choose $M = I$ then we can obtain unpreconditioned version of the Conjugate Gradient Algorithm.

Algorithm 2 Preconditioned Conjugate Gradient Method

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
for i = 1,2,... **do**
  solve $Mz^{(i-1)} = r^{(i-1)}$
  $\rho_{i-1} = {r^{(i-1)}}^T z^{(i-1)}$
  if $i = 1$ **then**
    $p^{(1)} = z^{(0)}$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$
  end if
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/{p^{(i)}}^T q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence; continue if necessary
end for

3.2.2 Biconjugate Gradient Stabilized Method

The Biconjugate Gradient (BiCG) method is applicable for nonsymmetrical systems in addition to symmetrical systems. BiCG method uses two mutual orthogonal subspaces to ensure the orthogonality of the residuals.

The update process in each iteration is augmented with respect to Preconditioned Conjugate Gradient method. Processes are similar but based on A^T instead of A. So residuals are updated as

$$r^{(i)} = r^{(i-1)} - \alpha_i A p^{(i)}, \quad \hat{r}^{(i)} = \hat{r}^{(i-1)} - \alpha_i A^T \hat{p}^{(i)},$$

and search directions are updated as

$$p^{(i)} = r^{(i-1)} - \beta_{i-1} p^{(i-1)}, \quad \hat{p}^{(i)} = \hat{r}^{(i-1)} - \beta_{i-1} \hat{p}^{(i-1)}.$$

The update scalars

$$\alpha_i = \frac{\hat{r}^{(i-1)^T} r^{(i-1)}}{\hat{p}^{(i)^T} A p^{(i)}}, \quad \beta_i = \frac{\hat{r}^{(i)^T} r^{(i)}}{\hat{r}^{(i-1)^T} r^{(i-1)}}$$

ensure the bi-orthogonality condition

$$\hat{r}^{(i)^T} r^{(j)} = p^{\hat{(i)}^T} A p^{(j)} = 0 \quad \text{if } i \neq j.$$

Although BiCG consumes less time while constructing basis vectors and less data stroge, several variants of BiCG have been proposed to improve its convergence. The Biconjugate Gradient Stabilized is one of these variants.

The Biconjugate Gradient Stabilized method differs from BiCG in minimizing residual vector which leads to significantly smoother convergence behavior. Additionally, BiCGStab handles the irregular convergence patterns that may emerge squaring the residual polynomial [2].

Algorithm 3 is the pseudocode for the Preconditioned Biconjugate Gradient Stabilized method [2]. M is the preconditioner.

Algorithm 3 Biconjugate Gradient Stabilized Method

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
Choose $\check{r}$ (for example, $\check{r} = r^{(0)}$)
for i = 1,2,... **do**
    $\rho_{i-1} = \check{r}^T r^{(i-1)}$
    if $\rho_{i-1} = 0$ **then method fails**
    end if
    if $i = 1$ **then**
        $p^{(i)} = r^{(i-1)}$
    else
        $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
        $p^{(i)} = r^{(i-1)} + \beta_{i-1}(p^{(i-1)} - \omega_{i-1} v^{i-1})$
    end if
    solve $M\check{p} = p^{(i)}$
    $v^{(i)} = A\check{p}$
    $\alpha_i = \rho_{i-1}/\check{r}^T v^{(i)}$
    $s = r^{(i-1)} - \alpha_i v^{(i)}$
    check norm of s; if small enough: set $x^{(i)} = x(i) + \alpha_i \hat{p}$ and stop
    solve $M\hat{s} = s$
    $t = A\hat{s}$
    $\omega_i = t^T s / t^T t$
    $x^i = x^{(i-1)} + \alpha_i \hat{p} + \omega_i \hat{s}$
    $r^i = s - \omega_i t$
    check for convergence; continue if necessary
    for continuation it is necessary that $\omega_i \neq 0$
end for

BiCGStab algorithm has two checkpoints per iteration for the stopping criterion. The method may converge at the first test on the norm of s then the following update could be numerically unreliable. In addition, stopping in the first half of the iteration saves a few unnecessary operations.

3.2.3 Quasi Minimal Residual

The Quasi-Minimal Residual algorithm proposed by Freund and Nactigal [8] to solve nonsingular, non-Hermitian linear systems. CG-type methods for example Biconjugate Gradient method shows a rather irregular convergence behavior moreover breakdowns may arise. QMR intends to handle these problems.

Algorithm 4 is the pseudocode for the Preconditioned Quasi Minimal Residual method [2]. M_1 and M_2 used as a preconditioner.

Algorithm 4 Quasi Minimal Residual Method

Compute $r^{(0)} = b - Ax^{(0)}$ for some initial guess $x^{(0)}$
$\check{v}^{(1)} = r^{(0)}$; **solve** $M_1 y = \check{v}^{(1)}; \rho_1 = \|y\|_2$
Choose $\check{\omega}^{(1)}$, (for example $\check{\omega}^{(1)} = r^{(0)}$)
solve $M_2^t z = \check{\omega}^{(1)}; \xi_1 = \|z\|_2$
$\gamma_0 = 1; \eta_0 = -1$
for i = 1,2,... **do**
    if $\rho_i = 0$ or $\xi_i = 0$ **then method fails**
    end if
    $v^{(i)} = \check{v}^{(i)}/\rho_i; y = y/\rho_i$
    $w^{(i)} = \check{w}^{(i)}/\xi_i; z = z/\xi_i$
    $\delta_i = z^T y$;
    if $\delta_i = 0$ **then method fails**
    end if
    solve $M_2 \check{(y)} = y$
    solve $M_1^T \check{z} = z$
    if $i = 1$ **then**
        $p^{(1)} = \check{y}; q^{(1)} = \check{z}$
    else
        $p^{(i)} = \check{y} - (\xi_i \delta_i / \epsilon_{i-1}) p^{(i-1)}$
        $q^{(i)} = \check{z} - (\rho_i \delta_i / \epsilon_{i-1}) q^{(i-1)}$
    end if

$\check{p} = Ap^{(i)}$

$\epsilon_i = {q^{(i)}}^T \check{p}$;

if $\epsilon_i = 0$ **then method fails**

end if

$\beta_i = \epsilon_i / \delta_i$;

if $\beta_i = 0$ **then method fails**

end if

$\check{v}^{(i+1)} = \check{p} - \beta_i v^{(i)}$

solve $M_1 y = \check{v}^{(i+1)}$

$\rho_{i+1} = \|y\|_2$

$\check{\omega}^{(i+1)} = A^T q^{(i)} - \beta_i \omega^{(i)}$

solve $M_2^T z = \check{\omega}^{(i+1)}$

$\xi_{i+1} = \|z\|_2$

$\theta_i = \rho_{i+1}/(\gamma_{i-1}|\beta_i|); \gamma_i = 1/\sqrt{1+\theta_i^2}$;

if $\gamma_i = 0$ **then method fails**

end if

$\eta_i = -\eta_{i-1}\rho_i {\gamma_i}^2/(\beta_i \gamma_{i-1}^2)$

if $i = 0$ **then**

  $d^{(1)} = \eta_1 p^{(1)}; s^{(1)} = \eta_1 \check{p}$

else

  $d^{(i)} = \eta_i p(i) + (\theta_{i-1}\gamma_i)^2 d^{(i-1)}$

  $s^{(i)} = \eta_i \check{p} + (\theta_{i-1}\gamma_i)^2 s(i-1)$

end if

$x^{(i)} = x^{(i-1)} + d^{(i)}$

$r^{(i)} = r^{(i-1)} - s^{(i)}$

check for convergence; continue if necessary

end for

3.3 Preconditioning Techniques

Preconditioning is essential for the effective use of iterative solvers. While improving spectral properties of coefficient matrix, the linear system is transformed into a more favorable condition. Hence a good preconditioner should meet some requirements:

- The preconditioner should reduce the number of iterations.
- Constructing and applying the preconditioner should not be expensive computationally or in terms of storage.

3.3.1 Diagonal (Jacobi) Preconditioner

Diagonal preconditioner generated by getting diagonal entries of coefficient matrices and putting these entries into preconditioner matrix diagonal.

$$m_{ij} = \begin{cases} a_{ij} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

Memory usage of diagonal preconditioner is very small (O(n)) and construction is low cost. If, however, the diagonal contains zeros the diagonal preconditioner is singular.

3.3.2 Incomplete LU Factorization

Consider a sparse matrix A. Incomplete LU (ILU) factorization computes lower and upper triangular matrices L and U which satisfies the same nonzero structure of A lower and upper parts and also $LU \approx A$.

In general, ILU factorizations can be obtained by performing Gaussian elimination and dropping some predetermined nondiagonal elements [13]. To determine which elements are dropped a zero pattern set P chosen, such that

$$P \subset \{(i, j) \,|\, i \neq j;\ 1 \leq i, j \leq n\}. \tag{3.5}$$

The Incomplete LU factorization with no fill-in, ILU(0) in this context, takes the zero pattern A as P. After ILU(0), L and U has the exact non-zero structure of A, and if we calculate LU, there could be extra diagonal elements in the product. These extra diagonal elements are called *fill-in elements* [13]. If all these fill-in elements are discarded, this factorization type is called ILU(0) factorization.

Algorithm 5 is the pseudocode for ILU(0) [13].

Algorithm 5 ILU(0)

for $i = 1, 2, \ldots, n$ **do**
  for $k = 1, \ldots, i-1$ and for $(i,k) \in NZ(A)$ **do**
    Compute $a_{ik} \leftarrow \frac{a_{ik}}{a_{kk}}$
    for $j = k+1, \ldots, n$ and for $(i,j) \in NZ(A)$ **do**
      Compute $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$
    end for
  end for
end for

To increase efficiency and reliability of Incomplete LU factorization, some fill-in elements may need to be allowed. ILU(p) represents the Incomplete LU factorization with level of fill is p. The zero pattern of the product of L and U obtained from ILU(0) taken as the P for ILU(1).

There are strategies for accepting or discarding fill-in such as level of fill. A level of fill is applied to each element that is processed by Gaussian elimination, and the dropping will be performed according to the value of the level of fill.

The initial level of fill of each element of a sparse matrix A is defined by

$$lev_{ij} = \begin{cases} 0 & \text{if } a_{ij} \neq 0 \text{ ,or } i = j \\ \infty & \text{otherwise} \end{cases}$$

The level of fill is updated in line 6 of Algorithm 6 as follows

$$lev_{ij} = min\{lev_{ij}, lev_{ik} + lev_{kj} + 1\}. \tag{3.6}$$

Equations given to update level of fill value of each element demonstrated that the level of fill of an element will never increases during the execution. Non-zero elements in the original matrix A has the 0 level of fill in entire execution. It can be expressed that in ILU(p), all fill-in elements whose level of fill are not greater than p are kept [13].

Algorithm 6 is the pseudocode of the Incomplete LU factorization with level of fill is p [13].

Algorithm 6 ILU(p)

```
For all nonzero elements a_ij define lev(a_ij) = 0
for i = 2, ..., n do
    for k = 1, ..., i − 1 and for lev(a_ik) ≤ p do
        Compute a_ik ← a_ik / a_kk
        Compute a_i* ← a_i* − a_ik a_k*
        Update the levels of fill of the nonzero a_ij 's using Equation 3.6
        Replace any element in row i with lev(a_ij) > p by zero
    end for
end for
```

3.3.3 Incomplete Cholesky Factorization

Incomplete Cholesky (ICHOL) Factorization is another important preconditioning technique yet it is applicable to only the symmetric positive definite matrices. Preconditioned Conjugate Gradient can use ICHOL as preconditioning techniques because the method is also suitable for spd matrices.

Basically, the idea of ICHOL factorization similar to ILU factorization except that finding upper triangular factor of A. It is enough to finding lower triangular factor of A and the product of L and its transpose L^T is the Cholesky factorization of A. Likewise ILU, LL^T is much less sparse than A because of fill-in.

The incomplete factorization may have zeros in the same positions as A, if we set the non-zero set as A's non-zero structure. This leads to Incomplete Cholesky factorization with no fill-in. ICHOL(0) algorithm given as pseudocode in Algorithm 7.

Algorithm 7 ICHOL(0)

$l_{11} = \sqrt{a_{11}}$
for i = $2, \ldots, n$ **do**
  for j = $1, \ldots, i-1$ **do**
    if $a_{ij} = 0$ **then**
      $l_{ij} = 0$
    else $l_{ij} = \dfrac{\left(a_{ij} - \displaystyle\sum_{k=1}^{j-1} l_{ik} l_{jk}\right)}{l_{ij}}$
    end if
    $l_{ij} = \left(a_{ii} - \displaystyle\sum_{k=1}^{i-1} {l_{ik}}^2\right)^{\frac{1}{2}}$
  end for
end for

Allowing some fill-in entries may enhances the reliability of factorization. In this case, non-zero entries are included when they are larger than the determined threshold parameter.

CHAPTER 4

NUMERICAL EXPERIMENTS

4.1 Computing and Programming Environment

All the numerical experiments were performed on Greyfurt which contains 4 x 1400MHz AMD Opteron(tm) Processor 6376 16-cores CPUs and 126GB memory. The operating system is Debian GNU/Linux 7.8 and computer architecture is `x86_x64`.

The algorithm was implemented using MATLAB R2013a 64-bit. All the test cases were executed on this platform.

4.2 Experiments and Results

The matrices used in the simulations were obtained from University of Florida Sparse Matrix Collection [5]. All the test matrices are square matrix and arising in various application areas. Their sizes change in the range 956×956 between 99617×99617 and their number of non zero entries per row change in the range $2,6$ and $398,8$. Detailed information about matrices are given in the Appendix A. Since the number of test matrices are large (total 70), we present the best and worst cases, average and median of the results.

The algorithm is tested using preconditioned iterative methods which are PCG, QMR and BiCGStab and direct solver. Although the matrix is symmetric and positive definite, we have experimented with other general iterative schemes other than CG as the matrix is updated in each iteration which could potentially loose some symmetry or positive definiteness. Preconditioners which are used by preconditioned iterative methods are as follows;

1. No-Preconditioner
2. Diagonal
3. ICHOL/ILU no fill-in
4. ICHOL/ILU with drop tolerance

While QMR and BiCGStab use Incomplete LU factorization, PCG uses Incomplete Cholesky factorization as Preconditioner.

In the experiments outer tolerance, inner tolerance and drop tolerance are used as variables.

Outer Tolerance : Outer tolerance mentioned in the previous chapters is the condition to break out the execution. Outer tolerance affects the iteration count directly. As outer tolerance increases the iteration time decreases; however, the iteration is essential point of the algorithm. In every iteration, the algorithm converges further to the exact solution. On the other hand, the execution time increases with iteration count. If the outer tolerance is too large, the solution could be far from the exact solution. On the other hand, if it is too small, we could be doing work that is not really needed since the solution is already the exact solution. So for that reason, we should find an optimum outer tolerance which finds the solution with reasonable amount of iteration count.

Different outer tolerances in the range of $0,00001$ and 1 are tested and $0,001$ is chosen as the outer tolerance. In most test cases, the shortest path is found and the iteration count is not more than adequate.

Inner Tolerance : Iterative solvers uses a stopping tolerance to terminate the iteration for solving the linear system. If the tolerance is greater than the necessary, iteration stops and the solution may not be accurate enough. When the solution of the linear system is not accurate enough, Physarum

Polycephalum algorithm could fail. Because of that, the inner tolerance should be chosen carefully.

After experimenting with various inner tolerances, $0,01$ which is produces less errors, chosen as the inner tolerance (see Appendix B.3 for details).

Drop Tolerance : Drop tolerance is a parameter used in incomplete CHOL/LU factorization. In the runs we experimented with three drop tolerances, these are: $0,001$, $0,01$ and $0,1$.

Simulation results will be analyzed in three aspects; speed, convergence and memory usage. Preconditioned iterative methods are compared to the direct solver. Effect of the preconditioner on the performance studied separately. All methods are compared within themselves, using different preconditioners and against each other.

Since large number of experiments were performed with many parameters, only a summary of the results are presented here. For the details of these results the reader is referred to Appendices B.1 and B.2. Plots composed of maximum, median, average and minimum speedup and memory optimization values are presented.

Iterative methods are examined for all preconditioners to find out which preconditioner gives better performance. "Preconditioner vs. Speedup" and "Preconditioner vs. Memory Optimization" graphs for all three methods PCG, QMR and BiCGStab are given later in this section. In the following sections, we also study the robustness of various solvers and preconditioners.

4.2.1 Speedup and Convergence

In this section, results are evaluated in speedup and convergence perspective. The speedup is a metric to measure how fast is an algorithm with respect to another algorithm. We compute the speedup with respect to the direct solver.

An iterative methods start with an initial guess and improve the initial guess. Sometimes iterative methods fail to find the solution of linear system. In addition, computation of a preconditioner may fail especially when the preconditioner is in a factorized form. In addition to the failure that are related to solving the sparse linear systems, Physarum Solver may end up with a wrong path. Overall, the failure rate of preconditioned iterative methods is found by adding number of execution failures and number of wrong path calculations.

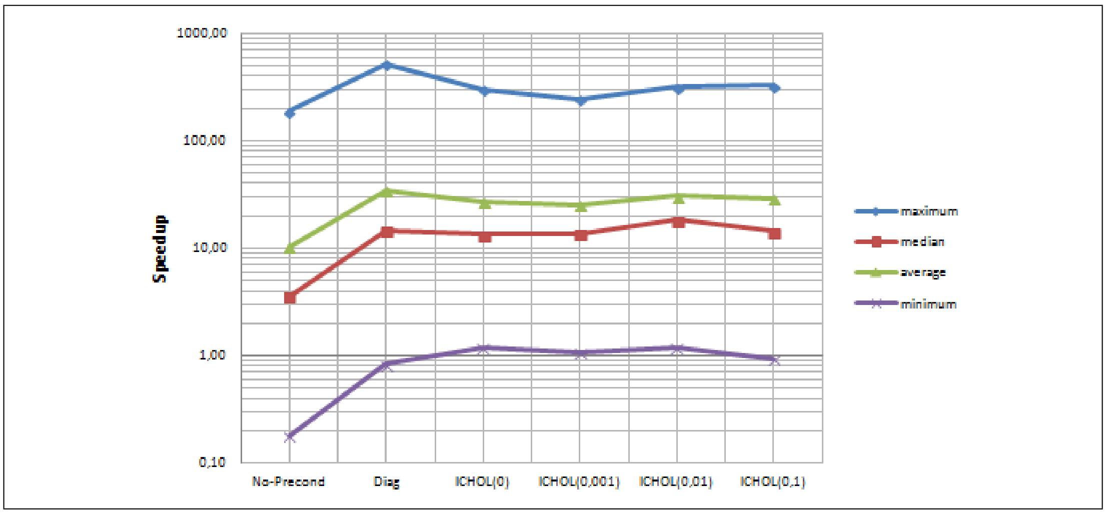

Figure 4.1: PCG speedup for various preconditioners compared to the direct solver

Table 4.1: The sequential running time on solution using the direct solver for corresponding PCG speedup values

-	No-Precond (s)	Diag (s)	IC(0) (s)	IC(0,001) (s)	IC(0,01) (s)	IC(0,1) (s)
maximum	2236,21	2236,21	2236,21	2236,21	2236,21	2236,21
median	295,10	381,95	87,50	693,15	613,50	4572,70
average	751,57	751,57	751,57	751,57	751,57	751,57
minimum	20,59	0,56	0,56	0,56	0,56	20,59

In Figure 4.1, PCG speedup values compared to the direct solver for different preconditioners are given. The sequential running time of the direct solver is given on Table 4.1 in seconds. Because maximum, median and minimum speedup values for each preconditioner are from different test matrices, the direct solver time consumption of corresponding speedup values not the same.

In all test cases, the diagonal preconditioner has the best maximum speedup value. It is roughly $523,9$ times faster than the direct solver in the best case and $34,7$ times better on average. The minimum, however, speedup is slightly worse than the other preconditioners. The reason behind such surprising performance is that the matrix may be well conditioned or the fill-in with the direct solver is too much for some matrices. Although the diagonal preconditioner is the best on average, other preconditioners also have significant speedup values, all of them being greater than 10. On the other hand, the minimum speedup value of all preconditioner cases are around 1. It can be said that the performance of PCG method is better than the direct solver method, in the worst case, it spends time as much as the direct solver.

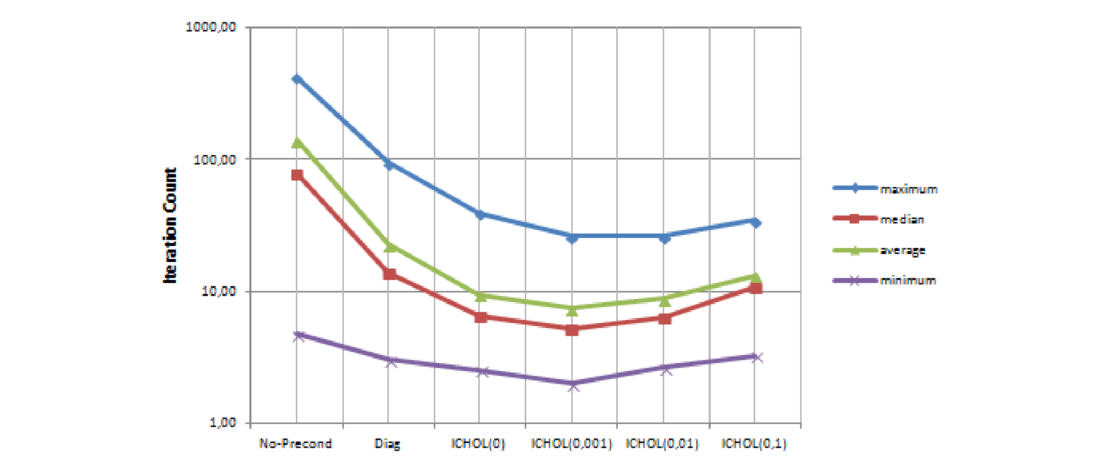

Figure 4.2: PCG average iteration count for various preconditioners

To analyze the effect of preconditioners on iterative solvers convergence rate, average iteration counts are calculated by dividing sum of the iteration counts that PCG made in every iterations of Physarum Solver to outer iteration count. Figure 4.2 express the average iteration count of PCG for various preconditioners. As expected, iteration count of PCG decreases when preconditioner is used. On average, PCG requires roughly 90% less iterations if a preconditioner is used. Despite the fact that the diagonal preconditioner has better time performance than others, the incomplete Cholesky factorization preconditioners increases convergence rate of PCG more than the diagonal preconditioner. Because of that, deciding the best preconditioner all of the metrics should be considered.

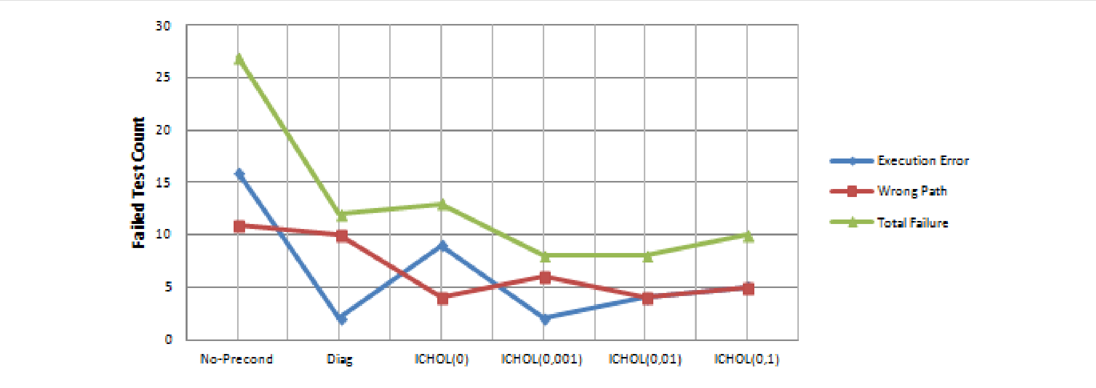

Figure 4.3: Number of failures of PCG using various preconditioners

In figure 4.3, the number of failures of PCG using different preconditioners is presented. Execution error, wrong path and the total failure are indicated with different lines. Based on the figure, PCG with No-Preconditioner there are more failures than with a preconditioners. Other preconditioners have lower number of failures yet the incomplete Cholesky with $0,001$ and $0,01$ drop tolerance are slightly better than others. As expected we observe that using a preconditioner improves the robustness of the Physarum Solver and the incomplete factorization with smaller dropping has better success rate than ILU with no fill-in.

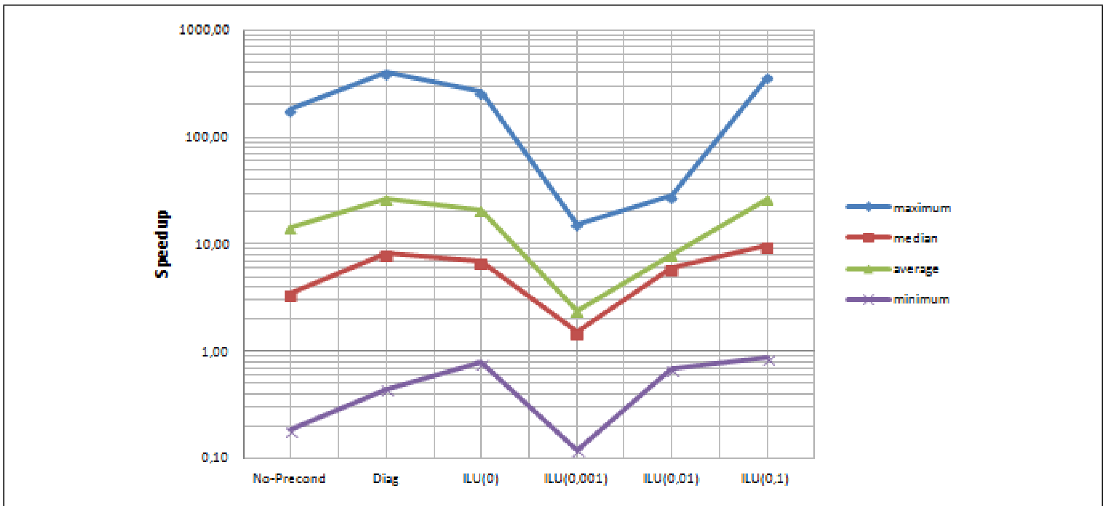

Figure 4.4: QMR speedup for various preconditioners compared to the direct solver

Table 4.2: The sequential running time on solution using the direct solver for corresponding QMR speedup values

-	No-Precond (s)	Diag (s)	ILU(0) (s)	ILU(0,001) (s)	ILU(0,01) (s)	ILU(0,1) (s)
maximum	2236,21	2236,21	2236,21	1021,61	596,55	2236,21
median	658,30	346,80	4436,40	1121,20	214,20	354,65
average	774,90	774,90	774,90	774,90	774,90	774,90
minimum	51,46	0,08	0,08	215,87	215,87	11,07

Figure 4.4 is preconditioner versus speedup graphs for QMR and Table 4.2 is the sequential running time of the direct solver. While the diagonal preconditioner has the maximum speedup value, the incomplete LU factorization with $0,001$ drop tolerance has the minimum speedup value. Apart from ILU$(0,001)$, QMR method with or without preconditioner has better time performance over direct solver. It can be inferred from the Figure 4.13, that the diagonal and the ILU$(0,1)$ preconditioners are a little better than others in average. Considering median line on graph, half of the test cases have speedup values over 1. So that, QMR method spends less computational time than the direct solver in general.

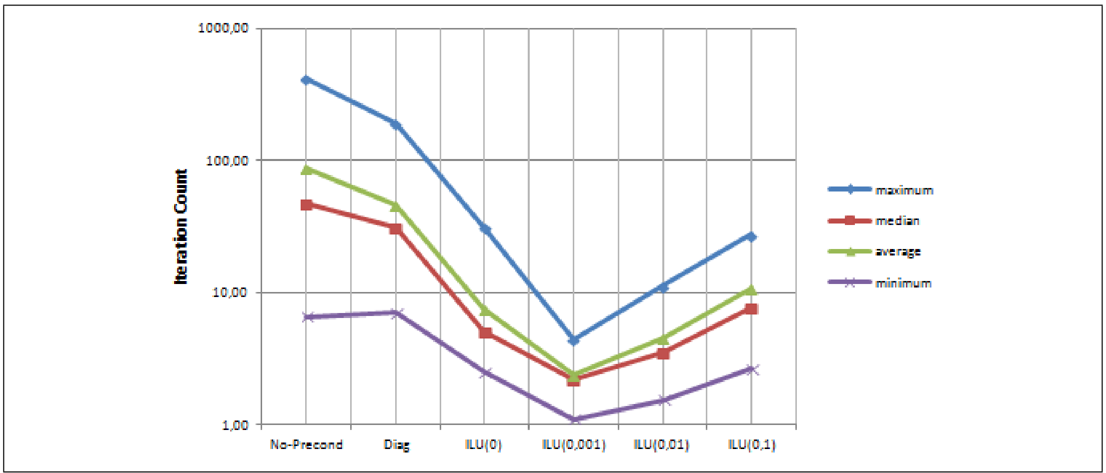

Figure 4.5: QMR average iteration count for various preconditioners

QMR average iteration counts are illustrated at Figure 4.5. Likewise PCG, using preconditioning techniques increases the convergence rate of QMR and the diagonal preconditioner is also not as good as the others in terms of the convergence rate.

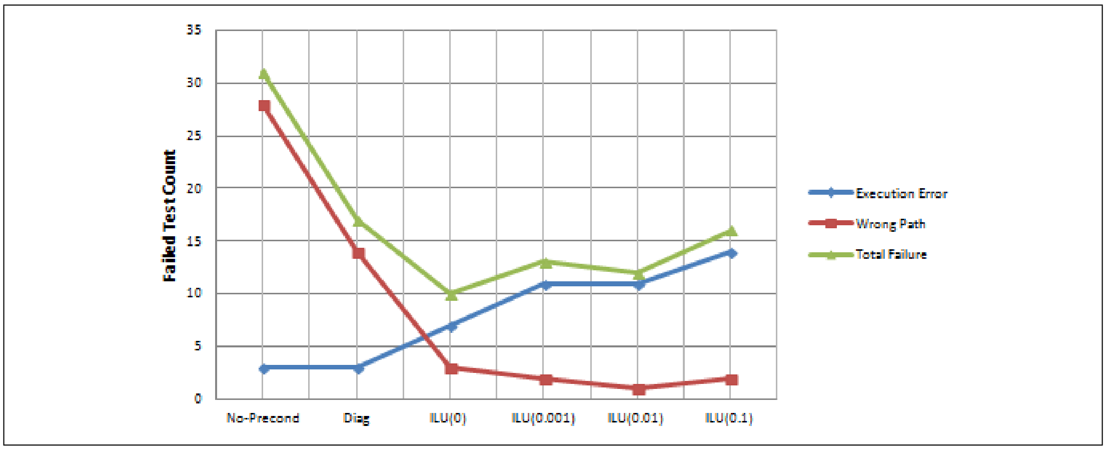

Figure 4.6: Number of failures of QMR using various preconditioners

QMR failure rates are illustrated at in the Figure 4.6, Similar to PCG method, QMR method fails most when preconditioner is not used. The diagonal and ILU$(0,1)$ produces almost the same total error and ILU(0) has better success rate. The execution errors for preconditioners generated by factorization have the majority of total failure values. On the other hand, no-preconditioner and the diagonal preconditioner failed to find correct path in most case. It can be inferred from the results, execution errors are emerged while constructing the preconditioner.

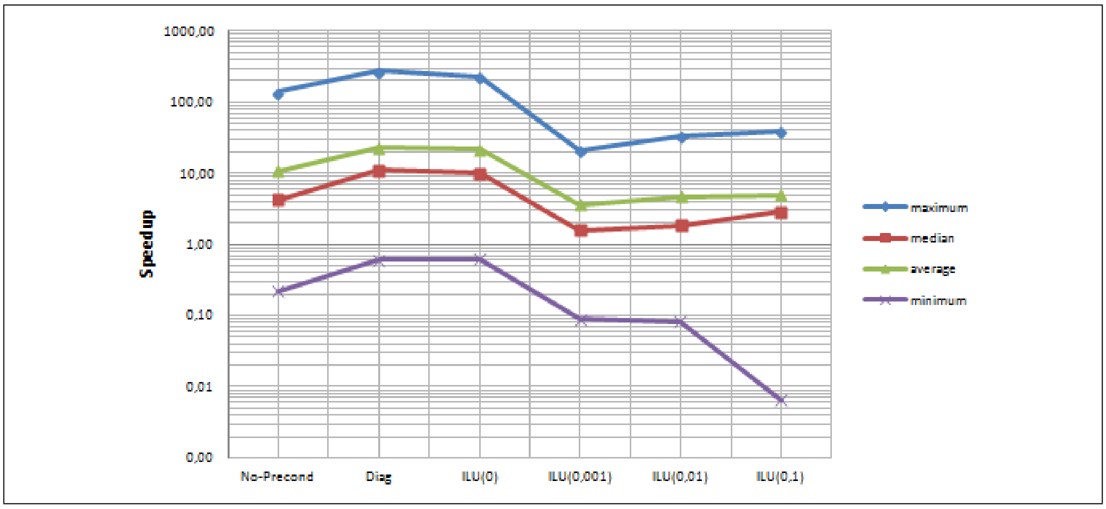

Figure 4.7: BiCGStab speedup for various preconditioners compared to the direct solver

Table 4.3: The sequential running time on solution using the direct solver for corresponding BiCGStab speedup values

-	No-Precond	Diag	ILU(0)	ILU(0,001)	ILU(0,01)	ILU(0,1)
maximum	2236,21	2236,21	2236,21	2236,21	42,02	2236,21
median	432,75	431,40	636,55	613,50	915,75	431,40
average	760,07	760,07	760,07	760,07	760,07	760,07
minimum	51,46	0,56	0,56	51,46	51,46	6,19

Figure 4.7 shows the speedup for BiCGStab and Table 4.3 is the sequential running time of the direct solver. On average, the diagonal and the incomplete LU factorization with no fill-in preconditioners time performance preferable to others. No-Preconditioner displays also good time performance on average. Like QMR, the minimum results of BiCGStab are below 1 and the median values are above 1. As a result, BiCGStab method gives better time performance than the direct solver by a majority.

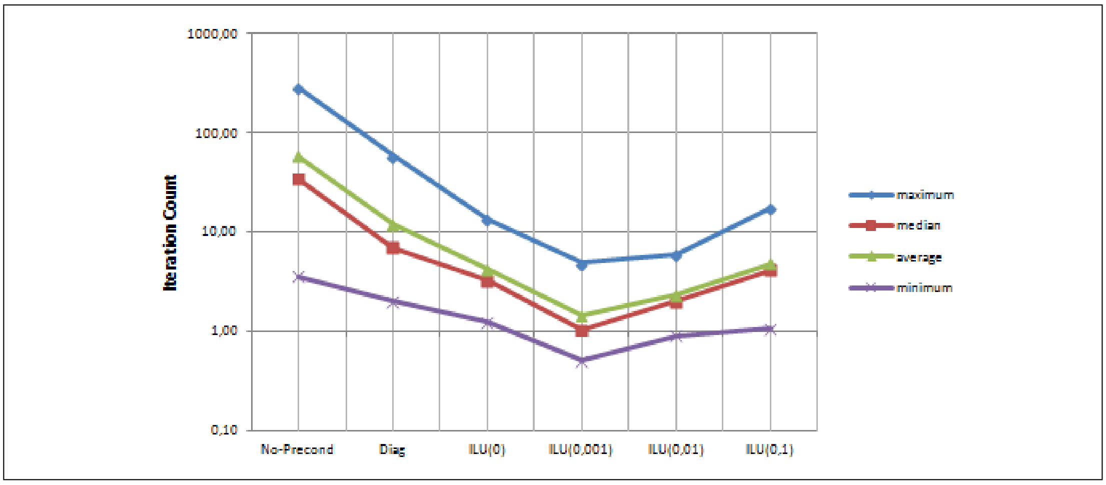

Figure 4.8: BiCGStab average iteration count for various preconditioners

In Figure 4.8 average iteration count of BiCGStab are presented. As expected, using preconditioning techniques decreases the iteration count of the method. Similarly PCG and QMR, the diagonal preconditioner has better time performance compared to other preconditioners but it gives significantly more iteration count than others.

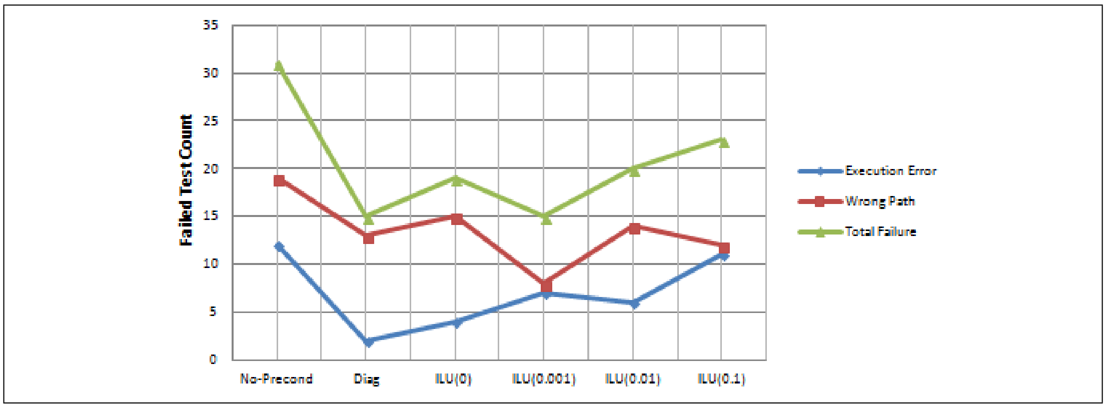

Figure 4.9: Number of failures of BiCGStab using various preconditioners

Figure 4.9 illustrates BiCGStab preconditioner failures . As in other methods, BiCGStab failed most when no preconditioner is used.e. Among all preconditioners, the diagonal and ILU$(0, 001)$ preconditioners have better success rate.

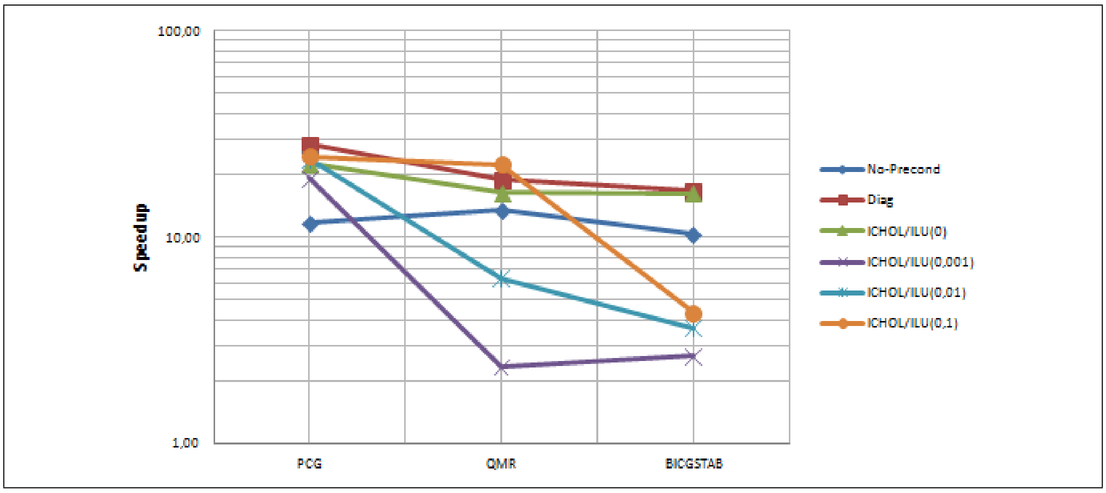

Figure 4.10: Iterative Solvers speedup for various preconditioners compared to the direct solver

Finally, all methods are analyzed with respect to each other. Figure 4.10 is the method versus speedup graph. Each line represents a preconditioner and markers on it are average speedup values of iterative solvers. Although, QMR method is slightly better than PCG method when preconditioner is not used, PCG method has better time performance of all other preconditioning techniques as expected. It is clearly seen that the average speedup values of all methods for various preconditioners are above 1. Thus, we can conclude that iterative solution of those linear systems should be preferred over direct solvers. Additionally, it can be inferred that the diagonal and the incomplete factorization with no fill-in are relatively better than other preconditioning techniques since they have better speedup values among preconditioning techniques and they display similar time performance for all methods.

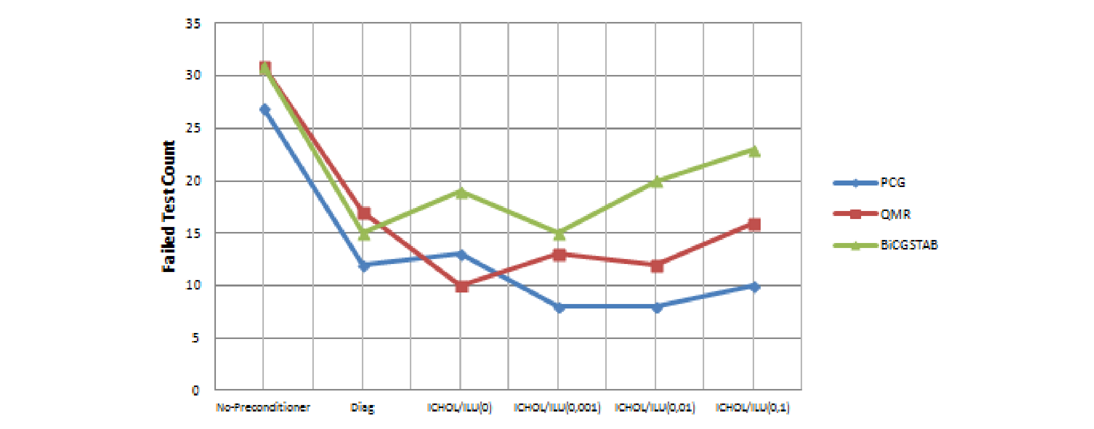

Figure 4.11: Number of failures of Iterative Solvers using various preconditioners

In Figure 4.11, PCG achieves the least number of failures almost all preconditioner cases. On the contrary, BiCGStab brings out the most number of failures of almost all preconditioner cases.

4.2.2 Memory Usage

Memory usage of direct solver and iterative solver differs from each other. In this thesis, Cholesky Solver were used as direct solver and it consumes much more memory because of fill-in. In addition to this, there have been memory usage differences between all preconditioning techniques. In this section, memory consumptions of methods are analyzed compared to the direct solver.

In order to simplify our analysis, memory optimization values were computed by dividing non-zero entry count of the direct solver to non-zero entry count of the iterative solver. Non-zero entry count of the direct solver is equal to summation of lower triangular matrix $nnz(L)$, in which nnz represents the non-zero entry count of a matrix, and its transpose $nnz(L^T)$. Same calculations were made for iterative solvers.

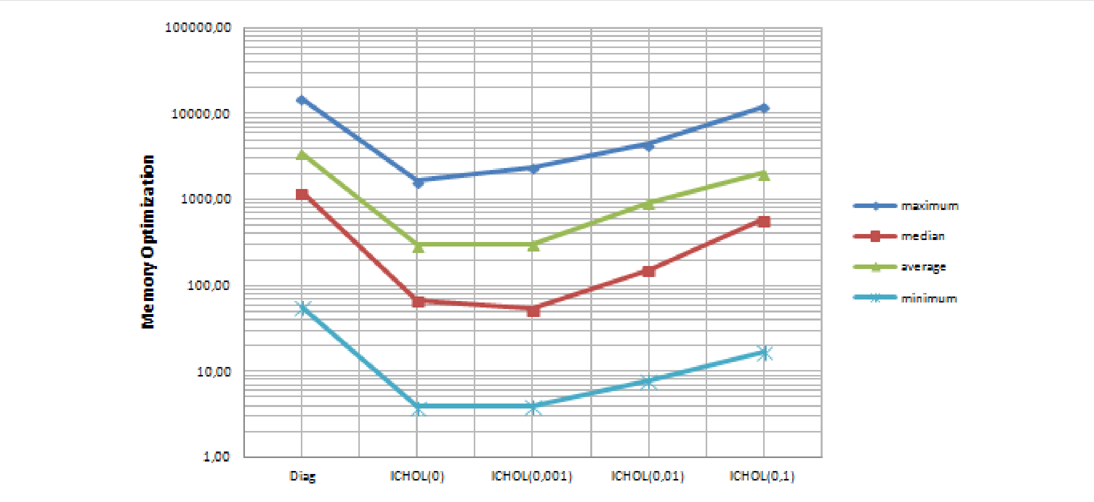

Figure 4.12: PCG memory improvement for various preconditioners compared to the direct solver

Figure 4.12 is PCG memory improvement plot. The figure clearly shows that PCG method memory requirement is less than the direct solver. Namely, the smallest improvement greater than 1. As expected, the diagonal preconditioner has better memory improvement than other preconditioning techniques. Although the diagonal preconditioner is the best, other preconditioners have also significant memory improvement. Besides, it can be said that, higher drop tolerance increases the memory improvement of the factorized preconditioners.

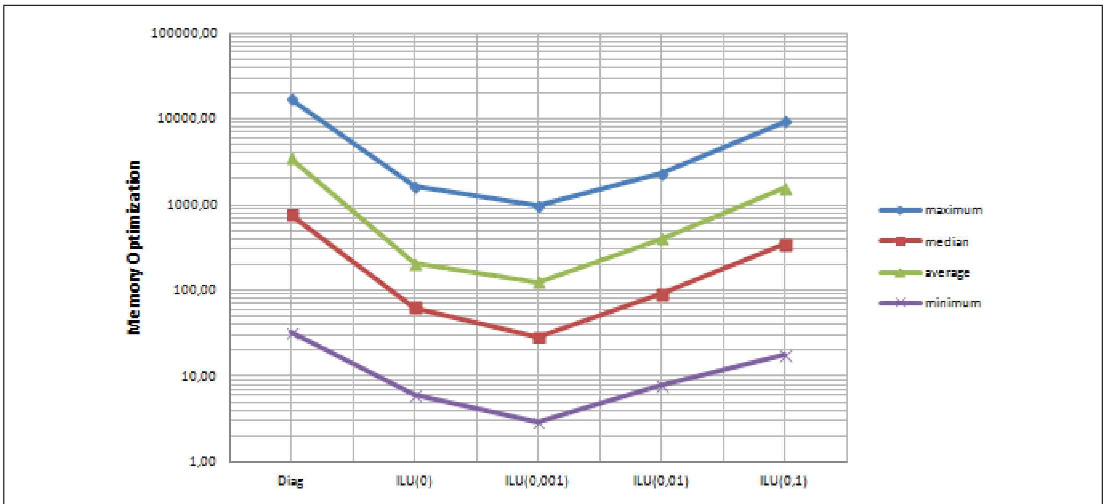

Figure 4.13: QMR memory improvement for various preconditioners compared to the direct solver

Figure 4.13 is the memory improvement for QMR. Memory usage of QMR method is as good as PCG method. While the diagonal preconditioner has the best memory improvement, the incomplete LU factorization with $0,001$ drop tolerance case has the worst memory requirement. It can be inferred from Figure 4.13, that the diagonal and ILU$(0,1)$ preconditioners have better memory improvements than others on average.

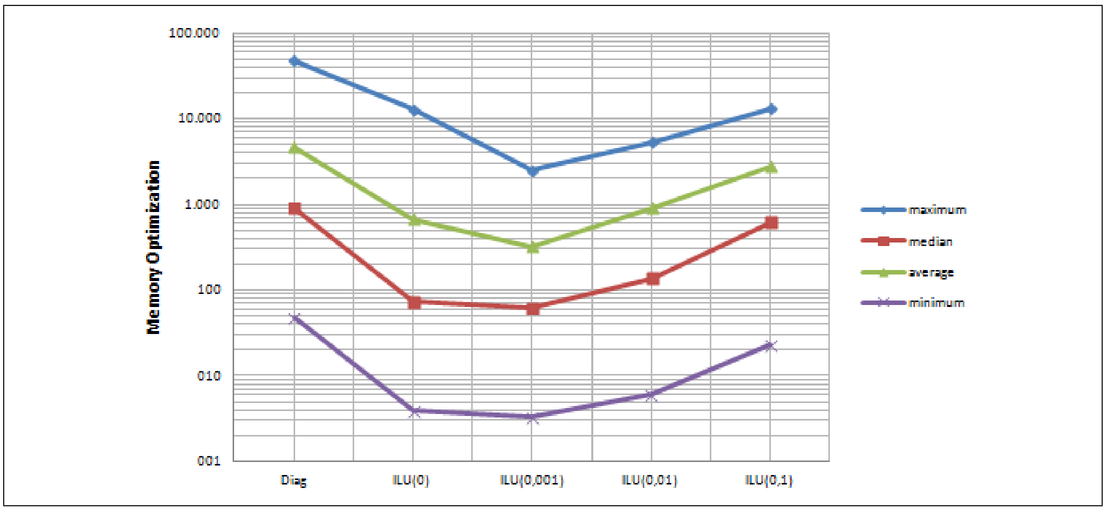

Figure 4.14: BiCGStab memory improvement for various preconditioners compared to the direct solver

BiCGStab memory improvement is given in Figure 4.14. All preconditioners improve the memory usage compared to the direct solver. Even though ILU$(0,1)$ are close to the diagonal preconditioner, the diagonal preconditioner is well ahead among others.

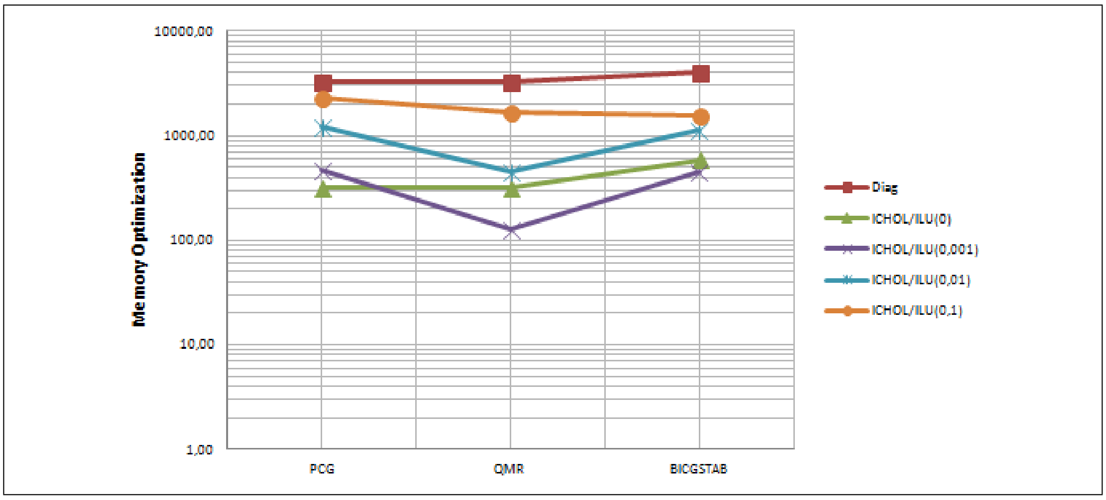

Figure 4.15: Iterative Solvers memory improvement for various preconditioners compared to the direct solver

Figure 4.15 shows the summary of memory improvement for each iterative method of preconditioning technique. The figure emphasizes that the memory usage of an iterative solver is much less than memory usage of the direct solver. It can be said that, the diagonal preconditioner has better memory improvement than other preconditioning techniques. ILU$(0, 1)$ has also quite well memory improvement, just behind the diagonal preconditioner.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Physarum Solver provides an unconventional solution to the shortest path problem. It approaches to the solution gradually changing the conductivity value of nodes in each iteration. A sparse linear system that needs to be solved in each iteration can be solved using either direct or preconditioned iterative solvers.

The main idea behind the direct solver is reducing the coefficient matrix A into easily invertible matrices. Direct solvers are widely used especially where reliability is the fundamental concern. However, as problem size increases, efficiency of direct solvers decreases in terms of time and memory requirement. In this case, Iterative solvers may be a better option due to fewer memory requirement and generally less time consumption. Nevertheless, they are not as robust as direct solvers but by means of preconditioning techniques, the robustness of iterative solvers can be improved.

In this thesis, we present the improvement of preconditioned iterative solvers on Physarum Polycephalum shortest path algorithm compared to the direct solver. While Preconditioned Conjugate Gradient, Quasi Minimal Residual and Bi-conjugate Gradient Stabilized methods were used as iterative method, the incomplete LU and Cholesky factorization and the diagonal preconditioners were used as preconditioning techniques. Experiments were evaluated in time consumption, convergence rate and memory requirement.

It has been observed that, preconditioned iterative solvers make improvement in time and memory consumption with respect to the direct solver. Indeed, preconditioned iterative solvers spend as much time as direct solver even in the worst case yet in general time consumption of preconditioned iterative solvers is less than the direct solver. Besides, memory consumption of iterative solver is much better than memory consumption of the direct solver in the worst case. Although, some of the test cases end with execution failure or wrong solution, the failures can be reduced with preconditioning techniques.

Even though the matrix is symmetric and positive definite, we have experimented QMR and BiCGStab method beside PCG. And still the expected result did not change, PCG method performed better speedup, memory and success rate than other iterative schemes. In addition to that, the diagonal preconditioner has better speedup and memory optimization values than other preconditioning techniques however the success rate of the factorized preconditioners are slightly better than the diagonal preconditioner.

In the future, Physarum Solver will be enhanced using parallelization techniques. Based upon to be held sequential computations, incomplete LU and Cholesky factorizations have limits of parallelization. Diagonal preconditioner is parallel, however it may not reduce the number of iterations as much as one would expect and hence other alternative parallel preconditioning schemes could be explored.

REFERENCES

[1] C. W. Ahn and R. S. Ramakrishna. A genetic algorithm for shortest path routing problem and the sizing of populations. *Trans. Evol. Comp*, 6(6):566–579, December 2002.

[2] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[3] M. Benzi. Preconditioning techniques for large linear systems: A survey. *Journal of Computational Physics*, 182(2):418 – 477, 2002.

[4] T. A. Davis. User guide for cholmod: a sparse cholesky factorization and modification package, 2009.

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

[6] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *Trans. Evol. Comp*, 1(1):53–66, April 1997.

[7] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[8] R. W. Freund and N. M. Nachtigal. Qmr: a quasi-minimal residual method for non-hermitian linear systems. *Numerische Mathematik*, 60(1):315–339, 1991.

[9] J. Lee and W. Yu. A coarse-to-fine approach for fast path finding for mobile robots. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IROS'09, pages 5414–5419, Piscataway, NJ, USA, 2009. IEEE Press.

[10] T. Miyaji and I. Ohnishi. Physarum can solve the shortest path problem on riemannian surface mathematically rigorously. *International Journal of Pure and Applied Mathematics*, 47(3):353–369, 2008.

[11] T. Nakagaki, H. Yamada, and Á. Tóth. Intelligence: Maze-solving by an amoeboid organism. *Nature*, 407(6803):470–470, 2000.

[12] T. Nakagaki, H. Yamada, and Á. Tóth. Path finding by tube morphogenesis in an amoeboid organism. *Biophysical Chemistry*, 92(1–2):47 – 52, 2001.

[13] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[14] A. Tero, R. Kobayashi, and T. Nakagaki. A mathematical model for adaptive transport network in path finding by true slime mold. *Journal of Theoretical Biology*, 244(4):553 – 564, 2007.

[15] H. Wang, X. Lu, X. Zhang, Q. Wang, and Y. Deng. A bio-inspired method for the constrained shortest path problem. *The Scientific World Journal*, 2014:271280, 2014.

[16] K. C. Wang and A. Botea. Tractable multi-agent path planning on grid maps. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 1870–1875, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[17] S. Watanabe, A. Tero, A. Takamatsu, and T. Nakagaki. Traffic optimization in railroad networks using an algorithm mimicking an amoeba-like organism, physarum plasmodium. *Biosystems*, 105(3):225 – 232, 2011.

[18] X. Zhang, Q Wang, A. Adamatzky, F.T.S Chan, S. Mahadevan, and Y. Deng. An improved physarum polycephalum algorithm for the shortest path problem. *The Scientific World Journal*, 2014:9, 2014.

APPENDIX A

TEST MATRICES

The explanation about table contents are as follows:

Matrix Name : Matrix name in university of florida matrix collection.

Rows: Number of rows.

Non-Zeros: Number of non-zero entry.

nnz/n : Number of non-zero entry per row.

Kind: Matrix kind.

Table A.1: Test Matrices (1/2)

Matrix Name	Rows	Non-Zeros	nnz/n	Kind
3dtube	45.330	3.213.618	70,89	computational fluid dynamics
a2nnsnsl	80.016	347.222	4,34	optimization
a5esindl	60.008	255.004	4,25	optimization
appu	14.000	1.853.104	132,36	directed weighted random graph
aug2dc	30.200	80.000	2,65	2D/3D
bbmat	38.744	1.771.722	45,73	computational fluid dynamics
bcsstk31	35.588	1.181.416	33,20	structural
big_dual	30.269	89.858	2,97	2D/3D
brack2	62.631	733.118	11,71	2D/3D
c-61	43.618	310.016	7,11	optimization
cca	49.152	139.264	2,83	undirected graph sequence
ccc	49.152	147.456	3,00	theoretical/quantum chemistry
Chebyshev4	68.121	5.377.761	78,94	structural
chem_master1	40.401	201.201	4,98	2D/3D
conf5_4-8x8-20	49.152	1.916.928	39,00	undirected graph sequence
cont-201	80.595	438.795	5,44	optimization
crankseg_1	25.804	5.186.900	201,01	structural
crankseg_2	63.838	14.148.858	221,64	structural
delaunay_n15	32.768	196.548	6,00	undirected graph
delaunay_n16	65.536	393.150	6,00	undirected graph
Dubcova2	65.025	1.030.225	15,84	2D/3D
e40r0100	17.281	553.562	32,03	2D/3D
fe_rotor	99.617	1.324.862	13,30	undirected graph
fe_sphere	16.386	98.304	6,00	undirected graph
fe_tooth	78.136	905.182	11,58	undirected graph
fem_filter	74.062	1.731.206	23,38	electromagnetics
G31	2.000	39.980	19,99	undirected weighted random graph
G56	5.000	24.996	5,00	undirected weighted random graph
gupta1	31.802	2.164.210	68,05	optimization
gupta2	62.064	4.248.286	68,45	optimization
helm3d01	32.226	428.444	13,29	2D/3D
kim1	38.415	933.195	24,29	2D/3D
L	956	3.640	3,81	2D/3D
L-9	17.983	71.192	3,96	2D/3D
mip1	66.463	10.352.819	155,77	optimization

Table A.2: Test Matrices (2/2)

Matrix Name	Rows	Non-Zeros	nnz/n	Kind
msc23052	23.052	1.142.686	49,57	structural
ncvxqp3	75.000	499.964	6,67	optimization
nd12k	36.000	14.220.946	395,03	2D/3D
nd24k	72.000	28.715.634	398,83	2D/3D
net100	29.920	2.033.200	67,95	optimization
net125	36.720	2.577.200	70,19	optimization
net150	43.520	3.121.200	71,72	optimization
pct20stif	52.329	2.698.463	51,57	structural
pdb1HYS	36.417	4.344.765	119,31	weighted undirected graph
pesa	11.738	79.566	6,78	directed weighted graph
pkustk03	63.336	3.130.416	49,43	structural
pkustk05	37.164	2.205.144	59,34	structural
pkustk06	43.164	2.571.768	59,58	structural
pkustk09	33.960	1.583.640	46,63	structural
pkustk11	87.804	5.217.912	59,43	structural
pkustk12	94.653	7.512.317	79,37	structural
pkustk13	94.893	6.616.827	69,73	structural
raefsky3	21.200	1.488.768	70,22	computational fluid dynamics
rajat08	19.362	83.443	4,31	structural
rma10	46.835	2.329.092	49,73	computational fluid dynamics
ship_001	34.920	3.896.496	111,58	structural
shock-9	36.476	142.580	3,91	2D/3D
sme3Da	12.504	874.887	69,97	structural
sme3Dc	42.930	3.148.656	73,34	structural
sparsine	50.000	1.548.988	30,98	structural
t60k	60.005	178.880	2,98	undirected graph
Trefethen_2000	2.000	41.906	20,95	combinatorial
Trefethen_20000	20.000	554.466	27,72	combinatorial
TSOPF_FS_b162_c4	40.798	2.398.220	58,78	power network
tsyl201	20.685	2.454.957	118,68	structural
wathen100	30.401	471.601	15,51	random 2D/3D
wathen120	36.441	565.761	15,53	random 2D/3D
whitaker3_dual	19.190	57.162	2,98	2D/3D
wing	62.032	243.088	3,92	undirected graph
Zd_Jac3	22.835	1.915.726	83,89	chemical process simulation

APPENDIX B

RESULTS

B.1 Speedup Results

The explanation about table contents are as follows:

Matrix Name : Matrix name in university of florida matrix collection.

No-Precond : Speeedup values of PCG\QMR\BiCGStab method when preconditioner is not used.

Diag : Speedup values of PCG\QMR\BiCGStab method while using diagonal preconditioner.

IC\ILU(0) : Speedup values of PCG\QMR\BiCGStab method while using Incomplete Cholesky\LU factorization with no fill-in.

IC\ILU(0,001) : Speedup values of PCG\QMR\BiCGStab method while using Incomplete Cholesky\LU factorization with 0,001 drop tolerance.

IC\ILU(0,01) : Speedup values of PCG\QMR\BiCGStab method while using Incomplete Cholesky\LU factorization with 0,01 drop tolerance.

IC\ILU(0,1) : Speedup values of PCG\QMR\BiCGStab method while using Incomplete Cholesky\LU factorization with 0,1 drop tolerance.

Direct Solver (s) : The sequantial running time of the direct solver in seconds.

Table B.1: PCG Speedup for various preconditioners compared to the direct solver and the direct solver time consumption in seconds (1/2)

Matrix Name	No-Precond	Diag	IC(0)	IC(0,001)	IC(0,01)	IC(0,1)	Direct Solver (s)
3dtube	1,52	34,53	33,43	22,64	33,08	34,21	493,01
a2nnsnsl	wrong	0,39	0,53	0,56	0,55	0,44	17,83
a5esindl	wrong	1,13	0,11	0,49	0,38	0,57	1,31
appu	8,92	31,04	36,47	43,19	44,24	38,95	477,70
aug2dc	wrong	0,68	0,96	1,72	1,32	0,78	20,36
bbmat	44,18	err	err	err	err	err	wrong
bcsstk31	wrong	11,72	err	12,46	13,23	8,11	815,50
big_dual	0,57	1,46	2,10	2,22	2,36	2,03	6,19
brack2	2,70	19,56	19,55	15,60	21,16	12,37	215,87
c-61	err	2,33	err	3,35	4,03	3,38	43,38
cca	7,83	51,94	62,94	42,78	54,63	51,79	165,73
ccc	14,50	59,03	68,43	73,56	80,42	72,21	429,10
Chebyshev4	err	2,84	0,47	0,86	2,75	2,94	399,32
chem_master1	err	0,06	0,02	0,54	0,83	0,63	4,82
conf5_4-8x8-20	26,68	82,49	78,58	84,64	95,98	109,27	1087,29
cont-201	err	3,21	3,71	6,34	7,02	5,52	318,40
crankseg_1	err	wrong	1,46	1,60	1,62	1,61	310,20
crankseg_2	err	wrong	wrong	wrong	err	err	396,16
delaunay_n15	wrong	1,75	2,12	2,59	2,51	2,01	51,46
delaunay_n16	1,83	4,81	4,97	4,52	4,82	3,50	22,45
Dubcova2	err	wrong	0,87	0,88	0,91	0,83	125,17
e40r0100	0,98	1,01	1,50	2,03	2,05	1,73	24,12
fe_rotor	3,33	24,14	23,66	21,49	26,00	21,26	1335,16
fe_sphere	0,18	9,14	27,80	17,80	22,02	0,94	20,59
fe_tooth	2,40	25,48	24,69	25,82	27,48	23,31	617,60
fem_filter	err	wrong	err	wrong	err	err	456,50
G31	13,45	44,73	32,52	26,06	33,60	33,81	10,99
G56	15,88	47,95	40,83	36,98	46,11	40,81	15,07

Table B.2: PCG Speedup for various preconditioners compared to the direct solver and the direct solver time consumption in seconds (2/2)

Matrix Name	No-Precond	Diag	IC(0)	IC(0,001)	IC(0,01)	IC(0,1)	Direct Solver (s)
gupta1	3,99	25,36	2,25	26,98	29,23	5,40	571,83
gupta2	3,70	15,47	2,14	13,11	54,63	53,89	2857,74
helm3d01	3,82	5,12	14,21	7,87	8,28	7,96	100,91
kim1	2,81	6,38	19,75	19,87	21,85	21,72	1021,61
L	0,23	1,02	1,28	1,37	1,38	1,22	0,08
L-9	0,91	6,86	12,00	11,46	12,70	9,42	14,47
mip1	wrong	wrong	wrong	wrong	wrong	wrong	wrong
msc23052	err	0,82	err	0,82	0,84	1,02	15,97
ncvxqp3	5,68	6,80	12,63	13,85	14,49	12,96	74,07
nd12k	7,82	8,85	7,25	9,45	9,63	9,80	3777,71
nd24k	9,96	13,51	10,19	13,51	13,74	14,24	8510,20
net100	11,28	43,59	23,34	18,08	23,23	23,08	596,55
net125	10,83	38,80	22,36	16,33	24,92	25,75	720,83
net150	7,91	28,00	15,62	12,14	17,55	18,30	519,35
pct20stif	3,31	9,32	7,65	6,45	7,96	8,40	192,39
pdb1HYS	wrong	1,53	3,10	2,29	2,25	2,16	222,18
pesa	0,36	0,83	1,19	1,08	1,20	1,02	0,56
pkustk03	3,64	10,19	7,50	8,43	8,58	9,27	227,58
pkustk05	1,94	24,95	8,82	13,71	10,55	14,14	1312,23
pkustk06	3,54	15,15	12,12	12,24	12,79	15,40	362,61
pkustk09	1,64	13,62	9,55	10,10	11,10	10,53	200,80
pkustk11	4,68	12,01	8,38	10,17	10,94	11,65	637,90
pkustk12	0,51	6,98	6,59	5,98	6,79	6,84	375,43
pkustk13	4,68	23,82	16,46	15,74	18,88	21,30	707,73
raefsky3	err	2,24	4,95	3,02	3,79	2,79	89,02
rajat08	1,49	5,10	5,97	5,60	7,23	6,67	4,41
rma10	err	wrong	wrong	wrong	wrong	wrong	1368,71
ship_001	err	wrong	err	2,12	3,11	err	113,51
shock-9	err	2,22	3,34	3,93	4,06	2,83	30,79
sme3Da	err	err	err	err	err	wrong	wrong
sme3Dc	wrong	wrong	err	wrong	wrong	wrong	wrong
sparsine	14,57	30,93	33,02	30,89	32,18	23,79	532,53
t60k	wrong	0,56	0,58	1,06	0,39	0,33	31,36
Trefethen_2000	32,98	116,05	92,12	70,83	86,42	87,00	42,02
Trefethen_20000	188,86	523,92	298,61	244,61	316,70	329,14	2236,21
TSOPF_FS_b162_c4	wrong	wrong	wrong	2,37	2,39	err	126,64
tsyl201	2,25	11,71	11,97	13,41	5,18	14,63	635,15
wathen100	err	0,92	2,08	1,78	1,96	1,75	183,87
wathen120	wrong	0,84	2,60	1,82	1,86	1,52	152,29
whitaker3_dual	0,19	0,93	1,45	2,02	1,83	1,50	11,07
wing	3,20	14,06	18,50	20,23	24,89	19,02	401,30
Zd_Jac3	err	wrong	err	wrong	wrong	wrong	929,70

Table B.3: QMR Speedup for various preconditioners compared to the direct solver and the direct solver time consumption in seconds (1/2)

Matrix Name	No-Precond	Diag	ILU(0)	ILU(0,001)	ILU(0,01)	ILU(0,1)	Direct Solver (s)
3dtube	1,52	34,53	33,43	22,64	33,08	34,21	493,01
a2nnsnsl	wrong	0,39	0,53	0,56	0,55	0,44	17,83
a5esindl	wrong	1,13	0,11	0,49	0,38	0,57	1,31
appu	8,92	31,04	36,47	43,19	44,24	38,95	477,70
aug2dc	wrong	0,68	0,96	1,72	1,32	0,78	20,36
bbmat	44,18	err	err	err	err	err	wrong
bcsstk31	wrong	11,72	err	12,46	13,23	8,11	815,50
big_dual	0,57	1,46	2,10	2,22	2,36	2,03	6,19
brack2	2,70	19,56	19,55	15,60	21,16	12,37	215,87
c-61	err	2,33	err	3,35	4,03	3,38	43,38
cca	7,83	51,94	62,94	42,78	54,63	51,79	165,73
ccc	14,50	59,03	68,43	73,56	80,42	72,21	429,10
Chebyshev4	err	2,84	0,47	0,86	2,75	2,94	399,32
chem_master1	err	0,06	0,02	0,54	0,83	0,63	4,82
conf5_4-8x8-20	26,68	82,49	78,58	84,64	95,98	109,27	1087,29
cont-201	err	3,21	3,71	6,34	7,02	5,52	318,40
crankseg_1	err	wrong	1,46	1,60	1,62	1,61	310,20
crankseg_2	err	wrong	wrong	wrong	err	err	396,16
delaunay_n15	wrong	1,75	2,12	2,59	2,51	2,01	51,46
delaunay_n16	1,83	4,81	4,97	4,52	4,82	3,50	22,45
Dubcova2	err	wrong	0,87	0,88	0,91	0,83	125,17
e40r0100	0,98	1,01	1,50	2,03	2,05	1,73	24,12
fe_rotor	3,33	24,14	23,66	21,49	26,00	21,26	1335,16
fe_sphere	0,18	9,14	27,80	17,80	22,02	0,94	20,59
fe_tooth	2,40	25,48	24,69	25,82	27,48	23,31	617,60
fem_filter	err	wrong	err	wrong	err	err	456,50
G31	13,45	44,73	32,52	26,06	33,60	33,81	10,99
G56	15,88	47,95	40,83	36,98	46,11	40,81	15,07
gupta1	3,99	25,36	2,25	26,98	29,23	5,40	571,83
gupta2	3,70	15,47	2,14	13,11	54,63	53,89	2857,74
helm3d01	3,82	5,12	14,21	7,87	8,28	7,96	100,91
kim1	2,81	6,38	19,75	19,87	21,85	21,72	1021,61
L	0,23	1,02	1,28	1,37	1,38	1,22	0,08
L-9	0,91	6,86	12,00	11,46	12,70	9,42	14,47
mip1	wrong	wrong	wrong	wrong	wrong	wrong	wrong

Table B.4: QMR Speedup for various preconditioners compared to the direct solver and the direct solver time consumption in seconds (2/2)

Matrix Name	No-Precond	Diag	ILU(0)	ILU(0,001)	ILU(0,01)	ILU(0,1)	Direct Solver (s)
msc23052	err	0,82	err	0,82	0,84	1,02	15,97
ncvxqp3	5,68	6,80	12,63	13,85	14,49	12,96	74,07
nd12k	7,82	8,85	7,25	9,45	9,63	9,80	3777,71
nd24k	9,96	13,51	10,19	13,51	13,74	14,24	8510,20
net100	11,28	43,59	23,34	18,08	23,23	23,08	596,55
net125	10,83	38,80	22,36	16,33	24,92	25,75	720,83
net150	7,91	28,00	15,62	12,14	17,55	18,30	519,35
pct20stif	3,31	9,32	7,65	6,45	7,96	8,40	192,39
pdb1HYS	wrong	1,53	3,10	2,29	2,25	2,16	222,18
pesa	0,36	0,83	1,19	1,08	1,20	1,02	0,56
pkustk03	3,64	10,19	7,50	8,43	8,58	9,27	227,58
pkustk05	1,94	24,95	8,82	13,71	10,55	14,14	1312,23
pkustk06	3,54	15,15	12,12	12,24	12,79	15,40	362,61
pkustk09	1,64	13,62	9,55	10,10	11,10	10,53	200,80
pkustk11	4,68	12,01	8,38	10,17	10,94	11,65	637,90
pkustk12	0,51	6,98	6,59	5,98	6,79	6,84	375,43
pkustk13	4,68	23,82	16,46	15,74	18,88	21,30	707,73
raefsky3	err	2,24	4,95	3,02	3,79	2,79	89,02
rajat08	1,49	5,10	5,97	5,60	7,23	6,67	4,41
rma10	err	wrong	wrong	wrong	wrong	wrong	1368,71
ship_001	err	wrong	err	2,12	3,11	err	113,51
shock-9	err	2,22	3,34	3,93	4,06	2,83	30,79
sme3Da	err	err	err	err	err	wrong	wrong
sme3Dc	wrong	wrong	err	wrong	wrong	wrong	wrong
sparsine	14,57	30,93	33,02	30,89	32,18	23,79	532,53
t60k	wrong	0,56	0,58	1,06	0,39	0,33	31,36
Trefethen_2000	32,98	116,05	92,12	70,83	86,42	87,00	42,02
Trefethen_20000	188,86	523,92	298,61	244,61	316,70	329,14	2236,21
TSOPF_FS_b162_c4	wrong	wrong	wrong	2,37	2,39	err	126,64
tsyl201	2,25	11,71	11,97	13,41	5,18	14,63	635,15
wathen100	err	0,92	2,08	1,78	1,96	1,75	183,87
wathen120	wrong	0,84	2,60	1,82	1,86	1,52	152,29
whitaker3_dual	0,19	0,93	1,45	2,02	1,83	1,50	11,07
wing	3,20	14,06	18,50	20,23	24,89	19,02	401,30
Zd_Jac3	err	wrong	err	wrong	wrong	wrong	929,70

Table B.5: BiCGStab Speedup for various preconditioners compared to the direct solver and the direct solver time consumption in seconds (1/2)

Matrix Name	No-Precond	Diag	ILU(0)	ILU(0,001)	ILU(0,01)	ILU(0,1)	Direct Solver (s)
3dtube	wrong	4,91	0,41	0,17	wrong	3,13	493,01
a2nnsnsl	wrong	0,19	0,30	0,01	0,01	0,01	17,83
a5esindl	wrong	wrong	0,38	0,01	0,01	0,01	1,31
appu	13,42	16,71	22,70	17,20	15,53	12,81	477,70
aug2dc	wrong	err	0,17	wrong	wrong	err	20,36
bbmat	err	wrong	wrong	wrong	wrong	err	wrong
bcsstk31	wrong	7,34	err	err	err	err	815,50
big_dual	0,48	0,75	0,81	0,11	0,14	0,01	6,19
brack2	2,70	6,35	5,77	0,59	0,89	1,12	215,87
c-61	err	1,80	err	0,54	0,61	err	43,38
cca	5,59	9,77	11,46	0,34	0,51	0,72	165,73
ccc	16,99	35,52	61,93	0,48	0,81	0,92	429,10
Chebyshev4	err	2,84	0,42	1,29	1,38	1,42	399,32
chem_master1	wrong	0,01	wrong	wrong	wrong	wrong	4,82
conf5_4-8x8-20	22,78	22,09	39,61	4,08	6,91	8,27	1087,29
cont-201	wrong	3,34	3,94	0,92	wrong	wrong	318,40
crankseg_1	err	1,13	1,17	1,28	1,38	1,35	310,20
crankseg_2	wrong	wrong	wrong	wrong	wrong	wrong	396,16
delaunay_n15	0,22	1,16	1,65	0,09	0,08	0,10	51,46
delaunay_n16	3,95	5,93	6,21	0,16	0,27	0,39	22,45
Dubcova2	wrong	err	0,58	wrong	err	wrong	125,17
e40r0100	0,58	0,77	0,66	err	0,97	err	24,12
fe_rotor	wrong	22,92	27,03	0,61	0,83	wrong	1335,16
fe_sphere	wrong	0,50	wrong	0,07	wrong	err	20,59
fe_tooth	wrong	1,75	1,30	0,40	0,48	wrong	617,60
fem_filter	err	wrong	1,69	err	err	1,49	456,50
G31	14,17	35,88	30,27	10,45	11,08	12,08	10,99
G56	14,55	33,65	33,95	4,83	5,62	5,15	15,07
gupta1	2,87	13,98	3,22	4,69	0,79	3,61	571,83
gupta2	wrong	14,55	5,99	wrong	wrong	3,79	2857,74
helm3d01	5,11	3,97	10,93	0,90	1,08	1,27	100,91
kim1	4,19	9,92	17,14	2,38	4,53	7,39	1021,61
L	0,43	1,10	1,19	0,56	0,62	1,03	0,08
L-9	0,98	3,52	wrong	0,54	wrong	wrong	14,47
mip1	wrong	wrong	wrong	wrong	wrong	wrong	wrong

Table B.6: BiCGStab Speedup for various preconditioners compared to the direct solver and the direct solver time consumption in seconds (2/2)

Matrix Name	No-Precond	Diag	ILU(0)	ILU(0,001)	ILU(0,01)	ILU(0,1)	Direct Solver (s)
msc23052	err	0,48	0,29	0,23	0,34	err	15,97
ncvxqp3	5,48	10,39	6,46	0,41	0,44	0,45	74,07
nd12k	2,86	4,65	1,30	4,74	5,08	5,19	3777,71
nd24k	8,51	13,83	8,76	3,88	4,51	5,63	8510,20
net100	12,77	33,68	24,21	4,35	4,60	5,05	596,55
net125	10,18	20,22	18,04	3,43	2,94	3,61	720,83
net150	6,34	14,30	11,50	1,65	1,82	2,24	519,35
pct20stif	3,25	8,46	7,29	0,94	1,26	1,70	192,39
pdb1HYS	2,02	1,18	wrong	2,00	1,73	1,71	222,18
pesa	0,59	0,60	0,62	0,17	0,22	0,27	0,56
pkustk03	4,28	11,07	10,20	1,31	1,58	2,64	227,58
pkustk05	3,45	11,80	7,48	1,72	1,88	1,62	1312,23
pkustk06	3,34	8,90	8,18	2,02	2,72	3,64	362,61
pkustk09	1,78	6,43	7,14	1,47	1,72	1,59	200,80
pkustk11	4,20	11,68	10,17	1,38	2,27	3,50	637,90
pkustk12	0,69	1,62	5,60	0,35	0,22	0,34	375,43
pkustk13	5,36	23,42	20,15	1,54	3,19	3,96	707,73
raefsky3	err	1,28	2,84	3,15	wrong	1,97	89,02
rajat08	1,16	1,71	1,33	0,34	0,51	0,50	4,41
rma10	err	wrong	wrong	wrong	wrong	wrong	1368,71
ship_001	err	wrong	err	err	err	err	113,51
shock-9	wrong	0,67	wrong	0,26	wrong	wrong	30,79
sme3Da	err	wrong	err	err	err	err	wrong
sme3Dc	err	wrong	wrong	err	err	err	wrong
sparsine	12,45	26,56	21,54	3,63	5,70	7,82	532,53
t60k	wrong	0,36	0,48	0,10	0,11	0,11	31,36
Trefethen_2000	32,84	103,24	92,15	20,47	33,73	22,84	42,02
Trefethen_20000	139,44	274,95	228,22	20,70	33,53	38,68	2236,21
TSOPF_FS_b162_c4	err	wrong	wrong	err	wrong	wrong	126,64
tsyl201	2,68	11,58	9,96	2,94	3,74	3,23	635,15
wathen100	wrong	wrong	wrong	0,81	0,75	0,58	183,87
wathen120	wrong	wrong	wrong	0,46	0,44	0,34	152,29
whitaker3_dual	0,17	0,67	wrong	0,22	0,24	err	11,07
wing	2,60	7,90	9,59	0,55	0,62	wrong	401,30
Zd_Jac3	wrong	wrong	wrong	3,70	2,91	3,69	929,70

B.2 Memory Optimization Results

The explanation about table contents are as follows:

Memory Optimization : The ratio of the average non-zero entry count of coefficient matrix of the direct solver to average non-zero entry count of coefficient matrix of the preconditioned iterative method.

Matrix Name : Matrix name in university of florida matrix collection.

Diag : Memory optimization of PCG\QMR\BiCGStab method while using the diagonal preconditioner.

IC\ILU(0) : Memory optimization of PCG\QMR\BiCGStab method while using Incomplete Cholesky\LU factorization with no fill-in.

IC\ILU(0,001) : Memory optimization of PCG\QMR\BiCGStab method using Incomplete Cholesky\LU factorization with 0,001 drop tolerance.

IC\ILU(0,01) :Memory optimization of PCG\QMR\BiCGStab method using Incomplete Cholesky\LU factorization with 0,01 drop tolerance.

IC\ILU(0,1) : Memory optimization of PCG\QMR\BiCGStab method using Incomplete Cholesky\LU factorization with 0,1 drop tolerance.

nnz(Direct Solver) : Average non-zero entry count of coefficient matrix of the direct solver.

Table B.7: PCG memory optimization for various preconditioners compared to the direct solver and the direct solver non-zero entry count

Matrix Name	Diag	IC(0)	IC(0,001)	IC(0,01)	IC(0,1)	nnz(Direct Solver)
3dtube	2020,44	56,80	39,92	88,51	1929,30	183173104
appu	5429,92	96,60	590,95	1572,53	2920,46	83957398
big_dual	115,42	39,31	22,05	31,10	44,14	827421
brack2	6274,89	869,94	408,26	1000,05	3901,56	190442780
cca	2660,94	1480,78	275,11	493,35	853,60	184700925
ccc	477,09	273,08	109,29	159,37	211,16	21403380
conf5_4-8x8-20	12376,59	366,72	273,04	952,62	3689,62	240242071
delaunay_n16	417,43	94,10	37,01	63,03	152,20	5889878
e40r0100	82,17	9,97	28,92	33,64	47,86	429439
fe_rotor	5838,30	956,20	442,13	1229,70	3337,39	408622940
fe_sphere	176,99	44,25	24,11	38,14	68,30	5800320
fe_tooth	7060,18	1144,48	634,90	1409,92	3467,12	401173304
G31	686,92	70,38	46,84	148,63	355,08	2004428
G56	694,48	216,59	88,00	189,04	337,39	3425846
gupta1	8257,49	472,22	1651,05	4457,88	6086,50	502385604
gupta2	8487,67	244,37	2384,00	4216,78	4838,12	999338458
helm3d01	13416,03	1232,58	912,98	4341,36	8275,51	101639828
kim1	242,72	21,48	73,50	99,51	123,12	9680640
L	55,68	13,48	5,71	8,58	16,79	43987
L-9	309,96	50,87	28,73	50,62	74,65	5007075
ncvxqp3	11104,19	1660,94	1401,92	1874,97	3033,90	87634304
nd12k	8573,44	51,62	594,71	2176,38	4754,35	182219929
nd24k	13994,42	83,08	821,34	3586,60	8070,33	612312013
net100	1615,48	65,54	34,08	1170,96	1360,52	97736785
net125	3372,77	129,12	66,13	2537,76	2720,10	158361476
net150	3768,70	152,90	79,50	2731,69	3173,89	228006085
pct20stif	1290,35	57,89	45,15	127,43	1387,80	34547802
pesa	1153,36	166,48	52,32	99,89	309,26	1121063
pkustk03	188,33	6,25	3,87	10,75	184,21	2812025
pkustk05	209,56	8,97	6,37	17,82	214,07	13889878
pkustk06	251,51	8,35	6,11	15,94	250,25	6717142
pkustk09	141,68	5,51	4,07	10,03	136,11	6259262
pkustk11	360,36	11,30	8,56	24,09	356,04	11041500
pkustk12	168,68	3,86	4,78	7,86	134,54	21659076
pkustk13	450,95	11,02	10,07	32,41	361,74	16803296
rajat08	246,95	57,20	29,07	62,45	97,06	1321410
sparsine	15192,09	1030,25	588,96	2105,03	12096,46	46897983
Trefethen_2000	712,54	59,19	40,01	110,44	317,35	2274414
Trefethen_20000	6590,69	467,89	358,84	995,81	3429,81	148185145
tsyl201	2451,35	40,74	54,58	231,97	2301,27	86299833
whitaker3_dual	137,39	58,85	34,33	47,44	64,86	1292195
wing	1804,78	711,02	343,81	504,56	815,79	57913670

Table B.8: QMR memory optimization for various preconditioners compared to the direct solver and the direct solver non-zero entry count

Matrix Name	Diag	ILU(0)	ILU(0,001)	ILU(0,01)	ILU(0,1)	nnz(Direct Solver)
appu	5469,18	101,30	338,72	932,41	2406,62	83957398
big_dual	146,94	43,63	11,52	28,28	45,45	827421
brack2	6804,20	965,04	235,55	806,98	2586,31	190442780
conf5_4-8x8-20	7541,97	314,82	167,51	623,54	3368,65	240242071
delaunay_n15	147,21	40,82	16,12	23,88	52,56	5913330
delaunay_n16	651,25	127,90	23,30	64,23	150,61	5889878
G31	704,05	70,38	29,61	99,69	226,62	2004428
G56	787,19	216,69	45,43	142,75	285,13	3425846
gupta1	8167,28	234,63	555,16	1449,46	3958,22	502385604
kim1	271,23	21,76	33,26	81,31	123,47	9680640
L	31,92	14,06	5,47	7,85	17,65	43987
L-9	174,28	60,63	12,17	35,13	67,36	5007075
ncvxqp3	9316,85	1612,91	961,52	229,08	2378,20	87634304
nd12k	9782,57	51,73	309,65	1452,82	3644,62	182219929
nd24k	16832,39	82,57	472,62	2292,63	8244,41	612312013
net100	1935,34	65,54	23,27	382,73	768,03	97736785
net125	3653,34	129,12	47,89	703,75	1518,11	158361476
net150	4514,88	152,90	54,28	892,85	1791,71	228006085
pct20stif	1794,41	59,60	27,39	85,96	986,35	34547802
pesa	839,12	216,84	27,97	86,49	240,16	1121063
pkustk03	176,42	7,34	2,92	8,62	141,61	2812025
pkustk05	257,91	9,06	4,62	12,17	200,25	13889878
pkustk06	330,15	8,85	4,76	15,23	226,02	6717142
pkustk09	182,14	6,15	3,94	8,36	152,45	6259262
pkustk11	487,98	13,31	6,49	19,85	367,94	11041500
pkustk13	729,25	35,55	6,88	28,83	756,33	16803296
rajat08	376,15	77,19	9,47	33,03	88,76	1321410
sparsine	15727,02	1184,02	253,62	1199,10	9281,22	46897983
Trefethen_2000	645,96	59,21	29,02	94,74	314,97	2274414
Trefethen_20000	7837,58	468,23	206,53	865,56	3104,20	148185145
tsyl201	2342,62	40,79	33,30	123,00	2267,65	86299833
whitaker3_dual	172,50	60,89	19,82	38,83	35,80	1292195

Table B.9: BiCGStab memory optimization for various preconditioners compared to the direct solver and the direct solver non-zero entry count

Matrix Name	Diag	ILU(0)	ILU(0,001)	ILU(0,01)	ILU(0,1)	nnz(Direct Solver)
appu	5861,72	101,66	453,33	1617,36	7567,14	83957398
big_dual	114,66	43,55	26,16	26,55	43,21	827421
brack2	6384,06	952,22	638,74	916,72	2802,11	190442780
cca	2159,71	12759,99	411,52	1080,96	1271,59	184700925
ccc	479,03	181,63	144,97	139,94	269,15	21403380
conf5_4-8x8-20	46639,89	473,57	413,94	838,56	5141,51	240242071
delaunay_n15	136,59	42,75	19,35	30,11	59,85	5913330
delaunay_n16	523,08	126,21	86,67	62,07	407,18	5889878
G31	704,30	70,38	36,40	129,18	286,02	2004428
G56	739,13	216,54	301,73	164,08	317,47	3425846
gupta1	8245,43	233,94	1814,24	5345,72	5019,74	502385604
helm3d01	14709,09	1393,57	809,50	2559,36	6126,20	101639828
kim1	256,33	21,94	53,06	131,95	395,50	9680640
L	47,40	13,73	6,24	9,02	22,84	43987
ncvxqp3	4901,80	3593,93	2488,62	1769,28	2631,82	87634304
nd12k	8909,64	52,58	479,37	1874,65	4226,17	182219929
nd24k	15630,97	81,42	859,19	3297,58	9642,25	612312013
net100	1973,60	67,87	28,77	1458,58	5572,22	97736785
net125	3802,56	128,29	57,01	3013,48	10643,29	158361476
net150	4604,14	158,32	67,11	3402,67	12999,21	228006085
pct20stif	1473,82	60,00	93,36	100,59	1174,34	34547802
pesa	1058,61	177,38	36,52	91,75	250,97	1121063
pkustk03	194,03	7,17	3,26	8,20	126,36	2812025
pkustk05	242,48	9,63	5,25	11,81	195,48	13889878
pkustk06	251,50	8,72	5,26	14,85	193,67	6717142
pkustk09	138,51	5,65	4,16	8,00	119,42	6259262
pkustk11	402,95	12,81	7,13	19,60	284,38	11041500
pkustk12	305,94	4,03	4,48	6,09	83,65	21659076
pkustk13	596,12	14,46	32,63	24,33	508,70	16803296
rajat08	273,92	77,47	21,86	213,30	529,84	1321410
sparsine	16323,70	1217,72	509,47	1647,57	10646,53	46897983
Trefethen_2000	656,97	59,11	91,97	92,50	691,10	2274414
Trefethen_20000	6872,83	465,60	932,03	773,49	2700,56	148185145
tsyl201	2384,57	53,72	45,25	133,08	2238,18	86299833

B.3 Results for PCG on pkustk13

The explanation about table contents are as follows:

Inner Tolerance : Tolerance is used by iterative solvers to terminate the iteration for solving the linear system.

Preconditioner : Preconditioner kind is used in the iterative methods.

Wall Clock Time (s) : The execution time of the method in seconds.

Outer Iteration : Physarum Solver iteration count.

Avg. Inner Iteration : Average number of iteration that iterative solver makes to find linear system solution in each Physarum Solver iteration.

Distance : The shortest path length.

Table B.10: Results when outer tolerance is 1E-5. The direct solver time consumption is 695,9 s, outer iteration count is 59 and distance is 10

Inner Tolerance	Preconditioner	Wall Clock Time (s)	Outer Iteration	Avg. Inner Iteration	Distance
1,00E-03	No-Preconditioner	256,68	151	297,70	10
1,00E-03	Diag	141,77	151	7,56	10
1,00E-03	IC(0)	96,17	65	7,98	10
1,00E-03	IC(0,001)	84,32	59	5,15	10
1,00E-03	IC(0,01)	83,16	59	8,56	10
1,00E-03	IC(0,1)	69,27	59	14,92	10
1,00E-02	No-Preconditioner	143,26	151	89,23	10
1,00E-02	Diag	117,73	151	5,87	10
1,00E-02	IC(0)	76,63	78	5,27	10
1,00E-02	IC(0,001)	63,91	59	3,44	10
1,00E-02	IC(0,01)	73,42	59	5,20	10
1,00E-02	IC(0,1)	64,72	60	9,27	10
1,00E-01	No-Preconditioner	17,96	16	18,19	2
1,00E-01	Diag	14,16	17	2,00	2
1,00E-01	IC(0)	112,90	144	3,04	10
1,00E-01	IC(0,001)	53,82	63	2,14	10
1,00E-01	IC(0,01)	61,73	84	3,23	10
1,00E-01	IC(0,1)	err	err	err	err

Table B.11: Results when outer tolerance is 1E-4. The direct solver time consumption is 633,1 s, outer iteration count is 44 and distance is 10

Inner Tolerance	Preconditioner	Wall Clock Time (s)	Outer Iteration	Avg. Inner Iteration	Distance
1,00E-03	No-Preconditioner	239,81	151	297,70	10
1,00E-03	Diag	51,96	39	14,92	10
1,00E-03	IC(0)	63,53	45	8,42	10
1,00E-03	IC(0,001)	59,97	44	5,55	10
1,00E-03	IC(0,01)	60,18	45	9,36	10
1,00E-03	IC(0,1)	58,84	45	16,13	10
1,00E-02	No-Preconditioner	151,03	151	89,23	10
1,00E-02	Diag	112,52	135	5,98	10
1,00E-02	IC(0)	54,27	45	5,62	10
1,00E-02	IC(0,001)	51,82	45	3,58	10
1,00E-02	IC(0,01)	47,08	44	5,61	10
1,00E-02	IC(0,1)	43,99	46	9,65	10
1,00E-01	No-Preconditioner	13,59	16	18,19	2
1,00E-01	Diag	9,68	13	2,00	2
1,00E-01	IC(0)	41,01	50	3,12	10
1,00E-01	IC(0,001)	40,35	48	2,19	10
1,00E-01	IC(0,01)	59,81	79	3,24	10
1,00E-01	IC(0,1)	10,33	13	1,08	2

Table B.12: Results when outer tolerance is 1E-3. The direct solver time consumption is 707,7 s, outer iteration count is 31 and distance is 10

Inner Tolerance	Preconditioner	Wall Clock Time (s)	Outer Iteration	Avg. Inner Iteration	Distance
1,00E-03	No-Preconditioner	155,71	74	320,57	10
1,00E-03	Diag	45,02	32	16,91	10
1,00E-03	IC(0)	52,42	31	9,06	10
1,00E-03	IC(0,001)	52,20	31	6,19	10
1,00E-03	IC(0,01)	48,68	30	11,03	10
1,00E-03	IC(0,1)	45,18	31	18,42	10
1,00E-02	No-Preconditioner	151,13	151	89,23	10
1,00E-02	Diag	29,71	27	9,89	10
1,00E-02	IC(0)	42,99	31	5,90	10
1,00E-02	IC(0,001)	44,97	31	3,84	10
1,00E-02	IC(0,01)	37,49	30	6,37	10
1,00E-02	IC(0,1)	33,23	32	10,38	10
1,00E-01	No-Preconditioner	14,76	16	18,19	2
1,00E-01	Diag	8,54	10	2,00	2
1,00E-01	IC(0)	29,55	35	3,17	10
1,00E-01	IC(0,001)	31,61	34	2,26	10
1,00E-01	IC(0,01)	31,13	40	3,50	10
1,00E-01	IC(0,1)	8,05	10	1,10	2

Table B.13: Results when outer tolerance is 1E-2. The direct solver time consumption is 643,1 s, outer iteration count is 19 and distance is 10

Inner Tolerance	Preconditioner	Wall Clock Time (s)	Outer Iteration	Avg. Inner Iteration	Distance
1,00E-03	No-Preconditioner	108,45	19	382,47	10
1,00E-03	Diag	38,82	19	22,05	10
1,00E-03	IC(0)	40,91	19	10,37	10
1,00E-03	IC(0,001)	39,66	19	7,58	10
1,00E-03	IC(0,01)	41,08	19	13,42	10
1,00E-03	IC(0,1)	35,78	19	22,05	10
1,00E-02	No-Preconditioner	43,70	34	92,85	10
1,00E-02	Diag	24,40	18	12,28	10
1,00E-02	IC(0)	33,45	19	6,47	10
1,00E-02	IC(0,001)	32,69	19	4,37	10
1,00E-02	IC(0,01)	28,79	19	7,74	10
1,00E-02	IC(0,1)	26,21	19	12,00	10
1,00E-01	No-Preconditioner	14,82	16	18,19	2
1,00E-01	Diag	6,38	7	2,00	2
1,00E-01	IC(0)	19,98	22	3,27	10
1,00E-01	IC(0,001)	20,53	20	2,45	10
1,00E-01	IC(0,01)	20,26	25	3,80	10
1,00E-01	IC(0,1)	6,05	7	1,14	2

Table B.14: Results when outer tolerance is 1E-1. The direct solver time consumption is 530,6 s, outer iteration count is 9 and distance is 10

Inner Tolerance	Preconditioner	Wall Clock Time (s)	Outer Iteration	Avg. Inner Iteration	Distance
1,00E-03	No-Preconditioner	92,42	9	393,00	10
1,00E-03	Diag	27,57	9	31,78	10
1,00E-03	IC(0)	32,23	9	14,11	10
1,00E-03	IC(0,001)	34,46	9	10,89	10
1,00E-03	IC(0,01)	31,43	9	20,56	10
1,00E-03	IC(0,1)	27,00	9	31,67	10
1,00E-02	No-Preconditioner	24,09	9	94,22	10
1,00E-02	Diag	16,12	9	16,11	10
1,00E-02	IC(0)	22,94	8	8,63	10
1,00E-02	IC(0,001)	22,50	9	5,89	10
1,00E-02	IC(0,01)	18,81	8	12,25	10
1,00E-02	IC(0,1)	16,16	8	17,00	10
1,00E-01	No-Preconditioner	8,27	8	4,13	3
1,00E-01	Diag	4,04	4	2,00	2
1,00E-01	IC(0)	7,23	7	3,86	10
1,00E-01	IC(0,001)	10,76	8	3,13	11
1,00E-01	IC(0,01)	8,27	8	5,50	11
1,00E-01	IC(0,1)	5,06	5	1,20	2

Table B.15: Results when outer tolerance is 1. The direct solver time consumption is 88,4 s, outer iteration count is 1 and distance is 14

Inner Tolerance	Preconditioner	Wall Clock Time (s)	Outer Iteration	Avg. Inner Iteration	Distance
1,00E-03	No-Preconditioner	5,01	1	114	14
1,00E-03	Diag	3,70	1	63	14
1,00E-03	IC(0)	5,31	1	25	14
1,00E-03	IC(0,001)	4,41	1	22	14
1,00E-03	IC(0,01)	7,15	1	52	14
1,00E-03	IC(0,1)	6,45	1	63	14
1,00E-02	No-Preconditioner	3,58	1	58	14
1,00E-02	Diag	2,96	1	29	15
1,00E-02	IC(0)	3,35	1	13	14
1,00E-02	IC(0,001)	4,32	1	10	19
1,00E-02	IC(0,01)	4,39	1	26	19
1,00E-02	IC(0,1)	3,39	1	29	15
1,00E-01	No-Preconditioner	2,09	1	4	3
1,00E-01	Diag	2,31	1	2	2
1,00E-01	IC(0)	2,40	1	1	3
1,00E-01	IC(0,001)	3,12	1	1	14
1,00E-01	IC(0,01)	2,22	1	2	3
1,00E-01	IC(0,1)	2,19	1	2	2

APPENDIX C

MATLAB CODES

C.1 PhysarumSolver

```
function [ path] = PhysarumSolver(p_OuterTolerance, p_InnerTolerance, p_droptol,
// p_OuterTolerance : Tolerance to end execution.
// p_InnerTolerance : Iterative Solver tolerance.
// p_droptol : Iterative Solvers drop tolerance.
// fileName : Input matrix name.
// outFile : Output file that results are going to write.
// p_SolverType : Linear system solver type.

//the test matrix is loaded and adjusted to an undirected graph.
load(fileName);
L = triu(abs( Problem.A),1) +(triu(abs( Problem.A),1) );

//initally conductivity values of each node assigned as 1.
size = length(L);
[i,j] = find(L);
v = ones(length(i), 1);
D = sparse(i,j,v);

//to check the difference between flux values, the previous flux values are
//stored in this matrix.
prevQ = sparse(size,size);

// right hand side vector of the linear equation. It is conducted from the
// Kirchhoffs Law and it does not change during the execution. b = [1, 0, 0, ...
```

```
b1 = [1];
b = padarray(b1, size-1, post);
b(size) = -1;
b = sparse(b);

//During execution the coefficient matrix may lose its structure
//and there could be errrors. checkP is used to control this kind of errors.
checkP = sparse(zeros(size, 1));

//Initialization of parameters.
iterationNum = 0;
avgIter = 0;
avgInnerTime = 0;
OuterTolerance = str2double(p_OuterTolerance);
InnerTolerance = str2double(p_InnerTolerance);
droptol = str2double(p_droptol);
SolverType = str2double(p_SolverType);

//start timer.
tStart = tic;
while true
  //D_ij / L_ij
  g = D.*spfun(@(x) 1./x, L);

  // if i = j then A_ij = row-sum(g_i)
  // otherwise A_ij = -g_ij
  A = -1*g +diag(abs(sum((-1*g) ,2)));

  //solve linear system Ap = b.
  [p, iter, innerTime] = SolveP(A,b, InnerTolerance,droptol, SolverType);
  avgIter = avgIter +iter;
  avgInnerTime = avgInnerTime +innerTime;

  //check pressure values (p vector) are calculated correctly.
 if(isequal(isnan(p),checkP) == 0)
    fprintf(fileID, 6sn, F2);
     break;
```

```
 end

 // Q_ij = g_ij *(p_i -p_j)
 [i,j,v] = find(g);
Q = sparse(i,j, v.*(p(i) -p(j)));

 // D_ij = abs(Q_ij)
 D = abs(Q);

//Termination criteria.
 //check whether the flux values are still changing or not.
 NormMatrix = Q -sparse(prevQ);
 normValue = norm(NormMatrix,fro)./norm(Q, fro);
 iterationNum = iterationNum +1;
 if( normValue = OuterTolerance)
    break
 end

//Termination criteria.
//check the outer iteration count has reached maximum iteration count.
 if(iterationNum  150)
    break
 end

 //store the flux values for next iteration.
 prevQ = Q;
end
//end timer.
tElapsed = toc(tStart);
InnerAvgIteration = avgIter/iterationNum;


//find the path that connects source and sink node traversing flux from starting
[, dim] = max(Q, [], 2);
path = [1];
i = 1;
while true
```

```
    path(end +1) = dim(i);
      if(dim(i) == 1)
        break;
    end
if(dim(i) == size)
    break;
end
    i = dim(i);
end

//calculate the length of the path.
dist = 0;
for i = 2:length(path)
    dist = dist +L(path(i-1), path(i));
end

end
```

C.2 SolveP

```
function [p, iter, innerTime] = SolveP( A, b, InnerTolerance, droptol, SolverType)
//---------Input Parameters-------------------//
// A : coefficient matrix
// b : right hand side vector
// InnerTolerance : tolerance that are used in iterative solver.
// droptol : iterative solvers droptol
// SolverType : to decide which solver is going to used.
// -0 Direct solver
//  - 1 PCG
//   - 2 QMR
//   - 3 BiCGStab
//---------Output Parameters-------------------//
// p : pressure values.
// iter : iteration count of iterative solvers make.
// innerTime : Time spent while solving linear system iteratively.
```

```

// Some of the values are going to decrease during the execution and they are re
// This could cause computational errors because it breaks down the positive def

 ind = find(sum(abs(A))  1e-10);
 p = zeros(length(A), 1);

 if(SolverType == 0) //DirectSolver
     p(ind) = A(ind, ind)b(ind);
     iter = 0;
     innerTime = 0;
 elseif(SolverType == 1) //PCG
    [p, iter, innerTime ] = SolvePpcg(A,b, InnerTolerance,droptol);
 elseif(SolverType == 2) //QMR
    [p, iter, innerTime ] = SolvePqmr(A,b, InnerTolerance,droptol);
 elseif(SolverType == 3) //BiCGStab
     [p, iter, innerTime ] = SolvePqmr(A,b, InnerTolerance,droptol);
 end

end
```

C.3 SolvePpcg

```
function [p, iter, tElapsed] = SolvePpcg( A, b, tolerance2, droptol)

v = sum(abs(A));
 ind = find(v  1e-10);
 Anew = A(ind, ind);
 bnew = b(ind);

 p = zeros(length(A), 1);
 if(droptol == -2) // No-Preconditoner
    tStart = tic;
     [p(ind),,,iter] = pcg(Anew, bnew, tolerance2, 500);
      tElapsed = toc(tStart);
```

```
 elseif (droptol == -1) // Diagonal
    tStart = tic;
    M = diag(diag(Anew));
    [p(ind),,,iter] = pcg(Anew, bnew, tolerance2, 500, M);
     tElapsed = toc(tStart);
 elseif (droptol == 0) // No Fill-in
    tStart = tic;
   opts.type = nofill;
   [L] = ichol(Anew, opts);
   [p(ind),,,iter] = pcg(Anew, bnew, tolerance2, 500, L, L);
    tElapsed = toc(tStart);
 else // Incomplete factorization with drop tolerance.
    tStart = tic;
    opts.type = ict;
    opts.droptol = droptol;
    alpha = 0.1;
    opts.diagcomp = alpha;
    [L] = ichol(Anew, opts);
   [p(ind),,,iter] = pcg(Anew, bnew, tolerance2, 500, L, L);
    tElapsed = toc(tStart);
 end

end
```

C.4 SolvePqmr

```
function [ p, iter, tElapsed ] = SolvePqmr(A, b, tolerance2, droptol )

v = sum(abs(A));
 ind = find(v  1e-10);

Anew = A(ind, ind);
 bnew = b(ind);

 p = zeros(length(A), 1);
```

```

 if(droptol == -2) // No-Preconditoner
     tStart = tic;
     [p(ind),,,iter] = qmr(Anew, bnew, tolerance2, 500);
      tElapsed = toc(tStart);
 elseif (droptol == -1) // Diagonal
    tStart = tic;
    M = diag(diag(Anew));
    [p(ind),,,iter] = qmr(Anew, bnew, tolerance2, 500, M);
       tElapsed = toc(tStart);
 elseif (droptol == 0) // No Fill-in
   tStart = tic;
   setup.type = nofill;
   [L, U] = ilu(Anew, setup);
   [p(ind),,,iter] = qmr(Anew, bnew, tolerance2, 500, L, U);
     tElapsed = toc(tStart);
 else // Incomplete factorization with drop tolerance.
    tStart = tic;
   setup.type = ilutp;
    setup.droptol = droptol;
    [L, U] = ilu(Anew, setup);
   [p(ind),,,iter] = qmr(Anew, bnew, tolerance2, 500, L, U);
     tElapsed = toc(tStart);
 end
end
```

C.5 SolvePbicgstab

```
function [ p, iter, tElapsed ] = SolvePbicgstab( A, b, tolerance2, droptol )

v = sum(abs(A));
 ind = find(v  1e-10);

Anew = A(ind, ind);
 bnew = b(ind);
```

```
p = zeros(length(A), 1);
if(droptol == -2) // No-Preconditoner
   tStart = tic;
    [p(ind),,,iter] = bicgstab(Anew, bnew, tolerance2, 500);
    tElapsed = toc(tStart);
elseif (droptol == -1) // Diagonal
  tStart = tic;
   M = diag(diag(Anew));
   [p(ind),,,iter] = bicgstab(Anew, bnew, tolerance2, 500, M);
   tElapsed = toc(tStart);
elseif (droptol == 0) // No Fill-in
  tStart = tic;
   setup.type = nofill;
   [L, U] = ilu(Anew, setup);
   [p(ind),,,iter] = bicgstab(Anew, bnew, tolerance2, 500, L, U);
   tElapsed = toc(tStart);
else // Incomplete factorization with drop tolerance.
  tStart = tic;
   setup.type = crout;
   setup.droptol = droptol;
   [L, U] = ilu(Anew, setup);
   [p(ind),,,iter] = bicgstab(Anew, bnew, tolerance2, 500, L, U);
   tElapsed = toc(tStart);
end

end
```