PARALLEL BIO-INSPIRED SINGLE SOURCE SHORTEST PATH ALGORITHMS

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

BY

HİLAL ARSLAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

DECEMBER 2017

Approval of the thesis:

PARALLEL BIO-INSPIRED SINGLE SOURCE SHORTEST PATH ALGORITHMS

submitted by **HİLAL ARSLAN** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy in Computer Engineering Department, Middle East Technical University** by,

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

Prof. Dr. Halit Oğuztüzün _____
Head of Department, **Computer Engineering**

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

Examining Committee Members:

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

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

Prof. Dr. Cevdet Aykanat
Computer Engineering Department, Bilkent University _____

Prof. Dr. Bülent Karasözen
Institute of Applied Mathematics, METU _____

Assist. Prof. Dr. Adnan Özsoy
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: HİLAL ARSLAN

Signature :

ABSTRACT

PARALLEL BIO-INSPIRED SINGLE SOURCE SHORTEST PATH ALGORITHMS

ARSLAN, HİLAL

Ph.D., Department of Computer Engineering

Supervisor : Assoc. Prof. Dr. Murat Manguoğlu

December 2017, 102 pages

Real-world shortest path problems that usually dynamically change are challenging and computationally expensive, and earlier studies in the literature require prohibitively long time to obtain the solution. To cope with such problems, we introduce novel bio-inspired parallel algorithms based on *Physarum Solver*. The first proposed algorithm is a fast and efficient parallel *Physarum Solver* with the objective to find the shortest path for static graphs with positive edge weights. Next, we propose a novel fully-dynamic bio-inspired parallel algorithm in order to find the shortest path on dynamically changing graphs with positive edge weights. The proposed dynamic algorithm efficiently computes the shortest path when the edge weights of the graph increases, decreases, or both over time. The proposed algorithms include various improvements and optimizations for *Physarum Solver*. We parallelize the sequential *Physarum Solver* proposed by Tero, Kobayashi and Nakagaki in 2006. Furthermore,

Physarum Solver requires the solution of linear systems whose coefficient matrix is a symmetric M-matrix and we note that this step is the most time consuming step. In the literature, however, there are not any studies that take an advantage of M-matrix property of the coefficient matrix to solve the linear systems efficiently in *Physarum Solver* and they are often solved by using a direct method, which is infeasible for large scale problems. We efficiently solve such linear systems using a parallel iterative solver with a preconditioner based on M-matrix property of the coefficient matrix. Finally, we propose a novel hybrid algorithm in order to compute the shortest path exactly since *Physarum Solver* is not guaranteed to find the exact shortest path. This underlines the accuracy of the hybrid method to compute the shortest path exactly. In the hybrid algorithm, *Physarum Solver* is used in order to detect the edges which cannot form the shortest path tree. The algorithm, first, sparsifies a given graph by removing the edges which cannot form the shortest path tree and then we apply a classical shortest path algorithm, such as *Dijkstra*, or Breadth First Search (if the graph is unweighted) on the sparser graph with positive edge weights. The proposed method is, therefore, a two stage hybrid algorithm. The accuracy and the required solution time by the proposed hybrid method are compared against a state-of-the-art implementation of the *Dijkstra's* algorithm on the original graph as the baseline. The results show that the proposed hybrid method achieves a significant speed improvement compared to the baseline. In order to evaluate the parallel algorithms, we use three different large graph models representing diverse real life applications. The parallel scalability, running time and accuracy of the proposed algorithms are presented and compared against Δ-stepping which is the most representative parallel implementation of *Dijkstra's* algorithm. The proposed algorithms exhibit remarkable parallel speedup with comparable accuracy for sparse large graphs. This underlines the effectiveness of the proposed algorithm to deal with hard real-life problems requiring long running time using classical algorithms.

Keywords: Symmetric M-matrix, Parallel shortest path, Preconditioning, Krylov subspace methods, Δ-stepping, Dynamic graphs

ÖZ

PARALEL BİYOLOJİK TABANLI TEK KAYNAKLI EN KISA YOL ALGORİTMALARI

ARSLAN, HİLAL

Doktora, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi : Doç. Dr. Murat Manguoğlu

Aralık 2017 , 102 sayfa

Gerçek dünyadaki en kısa yol problemleri genellikle dinamik olarak değişip, zorlu ve hesaplama masrafı yüksektir. Literatürdeki çalışmalar, bu problemleri çözmek için çok uzun zaman gerektirir. En kısa yol problemlerini çözmek için, biyolojik yöntemlerden esinlenilmiş *Physarum Çözücü*'ye dayalı paralel algoritmalar sunuyoruz. İlk önerilen algoritma, pozitif kenar ağırlıklı durağan graflarda en kısa yolu bulmak için geliştirilmiş hızlı ve etkin *Physarum Çözücü*'dür. Ardından, pozitif kenar ağırlıklı dinamik olarak değişen graflarda en kısa yolu bulmak için, biyolojik tabanlı dinamik paralel bir algoritma öneriyoruz. Önerilen dinamik algoritma, grafın kenar ağırlıkları zamanla arttığında, azaldığında veya hem artıp hem azaldığında en kısa yolu verimli bir şekilde hesaplar. Önerilen algoritmalar, *Physarum Çözücü* için çeşitli iyileştirmeler ve optimizasyonlar içerir. 2006 yılında Tero, Kobayashi ve Nakagaki tarafından

önerilen *Physarum Çözücü*'yü paralelleştirdik. Ayrıca, *Physarum Çözücü*, katsayılar matrisi simetrik M-matris olan doğrusal denklemlerin çözümünü gerektirir ve bu adım algoritmanın en zaman alan kısmıdır. Fakat literatürdeki çalışmalar, *Physarum Çözücü*'deki bu doğrusal sistemleri çözmek için katsayılar matrisinin M-matris özelliğini ihmal etmektedirler ve genellikle büyük ölçekli problemler için uygulanabilir olmayan bir direk çözücü kullanarak çözmektedirler. Bu çalışmada, bu doğrusal sistemleri, katsayı matrisinin M-matris özelliğine dayanan bir ön koşullayıcılı paralel iteratif çözücü kullanarak verimli bir şekilde çözüyoruz. Son olarak, *Physarum Çözücü*'nün en kısa yolu tam olarak bulması her zaman garanti edilmediğinden, en kısa yolu hesaplamak için yeni bir hibrit algoritma önermekteyiz. Bu method en kısa yolun tam olarak bulunduğunu ön plana çıkartmaktadir. Hibrit algoritmada, en kısa yol ağacını oluşturamayan kenarları saptamak için *Physarum Çözücü* kullanılır. Algoritma, öncelikle, en kısa yol ağacını oluşturamayan kenarları kaldırarak grafı seyrek yapar ve sonra Dijkstra (veya ağırlıksız graflarda Breadth First Search) gibi klasik en kısa yol algoritmaları, pozitif kenar ağırlıklı seyrek grafa uygulanarak en kısa yol bulunur. Önerilen hibrit algoritma bu nedenle iki aşamalıdır. Hibrit algoritma'nın problemi çözme süresi ve dogruluğu, Dijkstra algoritmasının modern uygulamasıyla karşılaştırılmıştır. Sonuçlar, önerilen hibrit yöntemin belirgin bir hız artışı sağladığını göstermektedir. Paralel algoritmaların sonuçlarını değerlendirmek için de, gerçek yaşam uygulamalarını da içeren üç farklı büyük graf modeli kullanıyoruz. Önerilen paralel algoritmaların paralel ölçeklenebilirlik, çalışma süreleri ve doğrulukları, Dijkstra algoritmasının en iyi parallel versiyonu olan Δ-stepping metodu ile karşılaştırılmıştır. Önerilen paralel algoritmalar büyük boyutlu seyrek graflarda önemli paralel hızlanma göstermektedir. Bu da geliştirdiğimiz algoritmaların klasik algoritmalar kullanıldığında uzun çözüm süresi gerektiren büyük ölçekli gerçek dünya problemlerini verimli bir şekilde çözdüğünü göstermektedir.

Anahtar Kelimeler: Simetrik M-matris, Paralel en kısa yol algoritmaları,Ön koşullandırma, Krylov alt uzay metodları, Δ-stepping, Dinamik Çizgeler

To my little daughter Ela

ACKNOWLEDGMENTS

I would like to thank my supervisor Assoc. Prof. Dr. Murat Manguoğlu for his constant support, guidance and friendship. It was a great honor to work with him and our cooperation influenced my academical and world view highly. I am so much thankful for his patience and understanding that he showed at every point that I feel myself in a bottleneck. He always encouraged me for the better, and hence, I believe in that I have done a good job under his guidance.

I am also thankful to my thesis committee members, Prof. Dr. Halit Oğuztüzün, Prof. Dr. Cevdet Aykanat, and Assist. Prof. Dr. Adnan Özsoy for their precious discussions and suggestions about my thesis. My very special thanks to Prof. Dr. Bülent Karasözen for his invaluable guidance, useful suggestions, and comments during this study.

My sincere thanks also goes to my friends, Serdar Çiftçi for being there for me whenever I needed, Özcan Dülger whose colorful personality turned all difficulties into an amusement, Mine Yoldaş who motivated for close friendship, and Merve Asiler who is a perfect roommate. I would also like to thank my other roommates, Engin Deniz Usta and Alper Demir for friendship. Moreover, I want to thank Özlem Erdaş, Aslı and Murat Gençtav, Alperen Eroğlu, Çağrı and Esra Kaya, Adnan Kılıç for their precious friendships. There are more friends that I want to thank Gamze Toybıyık Kaya, Esra Yeniaras, Özgür Ünsal, and F. Gökhan Tuna even though we can not have the opportunity to meet frequently.

I take this opportunity to record, my sincere gratitude to Scientific and Technological Research Council of Turkey (TÜBİTAK) for providing me Ph.D. fellowship (2211).

Last but not the least, I would like to thank my family, especially my parents Ali and Gülten Kılıç, for supporting me unconditionally throughout my life. I would also like thank my sisters Burcu Nihal and Gökçe Zülal, and my father in law Avni Arslan

for being a part of my life. I want to express my sincere gratitude to my beloved husband Güray Arslan, an extremely talented engineer and most importantly a loving husband and a father. Thank you for not only providing me with motivation, but also understanding, support and encouragement at every step of this pursuit. You are the most important factor for my well being and achievements in life. Many thanks go to Ela as well, the best baby a mother could have, for becoming the meaning of my life.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

DSP	Dynamic Shortest Path Problem
PPA	Parallel Physarum Algorithm
dynPPA	Dynamic Physarum Algorithm
CG	Conjugate Gradient
BiCGStab	Biconjugate Gradient Stabilized
RCG	Recycling Conjugate Gradient
SOR	Succesive Over Relaxation
GS	Gauss-Seidel
SAI	Sparse Approximate Inverse
AMG	Algebraic Multigrid
PETSc	Portable, Extensible Toolkit for Scientific Computation

CHAPTER 1

INTRODUCTION

1.1 Motivation

Computing the single source shortest path is a crucial kernel that arises in many applications such as large road networks [34], wireless networks [57], social networks [39], multicast routing [11], and route information protocol [69]. Almost every process that is found in natural systems is parallel and dynamic. Hence, the algorithms that are inspired by biological systems are known to be suitable for computational environments that support parallelism and should be able to adapt themselves for dynamically changing problems. For instance, if we consider real-time traffic information system and we want to find the least cost traffic route from our current position to a destination, both our route and the state of each link are likely to change over time. Therefore, the shortest path needs to be recomputed [21]. Much effort has been spent to solve dynamic shortest path problems [22, 36, 66, 10, 35]. The algorithms in the literature are generally sequential and there are only few parallel algorithms to solve the dynamic shortest path problem. Additionally, when both the percentage of changing edges and the size of a graph become larger, these algorithms are inefficient and the process of determining affected edges gets more time consuming. In this thesis, we consider the problem of computing the shortest paths and we propose bio-inspired algorithms to solve the single source and single target shortest path problems for both static and dynamic graphs. However, these algorithms may not compute the shortest path exactly for every graph since they are based on *Physarum Solver* which is a heuristic. In order to compute exact single source shortest path, we propose a hybrid algorithm based on *Physarum Solver* and *Dijkstra*.

1.2 Problem Statement

In this thesis, novel parallel methods are developed to solve the single source-single target shortest path problem based on *Physarum Solver*, which computes the shortest path approximately. Moreover, we propose a hybrid method to compute the single source shortest path problem exactly based on *Physarum Solver* and *Dijkstra*. Therefore, in this section, we give the definition of the shortest path problem for static and dynamic graphs.

Let $G(V, E, w)$ be a graph where V is the set of vertices, E is the set of edges, and w is a weight function from E to R^+. When a source and a destination vertices are given, the smallest weight path from one vertex to an other is determined in order to find the shortest path between two vertices, which is called as single source-single target shortest path problem. If the shortest path is computed from a source to every other vertices, the problem is called as the single source shortest path problem. In some problems, the edge weights of the graph may change over time. We assume that the edge weights of G are changing over time and so, another graph $G'(V, E, w')$ is obtained such that G' and G have same set of vertices and edges. The main goal is to compute the shortest path on G' efficiently by using the shortest path information obtained from G. Computing the shortest path in dynamically changing graphs is called the Dynamic Shortest Path (DSP) problem. In DSP problems, the edge weights can change in three different ways: edge weights may increase, decrease, and both increase and decrease (mix case). The algorithms that are designed for graphs whose edge weights either strictly increase or decrease are called semi-dynamic. Moreover, the algorithms that permit mixed edge weight changes are called fully-dynamic. Therefore, dynamic algorithms are further divided into two categories, which are semi-dynamic and fully-dynamic. These algorithms compute the shortest path by recomputing only affected vertices and so they save time when the small portion of the edge weights change.

1.3 Contributions

In the literature, there are not any efficient parallel shortest path algorithm which are applicable to dynamically changing graphs and real world graphs. The primary goal in this thesis is to propose efficient parallel shortest path algorithms for both dynamic and static graphs.

The highlights of our contributions are:

- Parallel bio-inspired shortest path algorithm for static graphs (PPA)
- Parallel fully-dynamic shortest path algorithm for dynamically changing graphs (dynPPA)
  - Suitably designed for dynamically changing graphs, that is, the previous iteration feeds completely into the next one
- Hybrid algorithm in order to compute the single source shortest path exactly

1.4 Organization of the Thesis

In Chapter 2, previous studies related to the shortest path problems are summarized. Our proposed methods are based on *Physarum Solver* and we compare our results with a well-known implementation of Dijkstra's algorithm, Δ-stepping. Therefore, we give detailed information about the implementations of these algorithms in this chapter. In Chapter 3, we summarize solution methods for sparse linear systems since our proposed algorithms require solution of these systems. In Chapter 4 and 5, we introduce the proposed parallel bio-inspired shortest path algorithms for static and dynamically changing graphs, respectively. In Chapter 6, we propose a hybrid method based on *Physarum Solver* in order to compute the single source shortest path exactly. In Chapter 7, we present experimental results for large graph models including real-world graphs. Finally, Chapter 8 concludes this thesis.

CHAPTER 2

RELATED WORK

In this chapter, earlier studies in the literature for finding the single shortest path algorithms are summarized. Algorithms solving such problems can be divided into two main groups which are static and dynamic shortest path algorithms.

First, we summarize the shortest path algorithms for static graphs. The Dijkstra's algorithm [27] is one of the most well-known algorithms to solve the single source shortest path problem on graphs which have nonnegative edges. Given a graph $G = (V, E)$ in which V and E are the set of vertices and edges, respectively. The Dijkstra's algorithm has $O(|V|log|V| + |E|)$ time complexity when a priority queue is utilized. The shortest path is obtained by incrementally building up a tree of shortest paths from the source vertex s to all other vertices in Dijkstra's algorithm. Priority queues, which holds the distance from each vertex to the source vertex, are used in the implementation. The pseudocode of the original algorithm is shown in Algorithm 1. The **relax** function decides whether there is a shorter path from s to v, or not. After the vertex u that has the minimum distance in the priority queue is dropped, dist[v] is updated where $w(u, v)$ is the weight of an edge (u, v) and dist[v] is the distance from v to s.

In our experiments we have used Parallel Dijkstra implementation in Parallel Boost Graph Library (Parallel BGL) [41], which is a library including graph algorithms on parallel and distributed computing platforms for large graphs. In Parallel BGL, three versions of Dijkstra's algorithm, which are Δ-stepping, Crauser Dijkstra's, and Eager Dijkstra's, are implemented. In all versions, as an input, Dijkstra's algorithm takes row wise distributed graph (stored by the distributed compressed sparse row format

Algorithm 1 Dijkstra's Algorithm

1: $dist[s] \leftarrow 0$
2: **for all** vertices $v \in V \setminus \{s\}$ **do**
3: dist[v] $\leftarrow \infty$
4: **end for**
5: **for all** edges $(u, v) \in E(G)$ **do**
6: RELAX(u, v, w)
7: **end for**
8: **function** RELAX(u,v,w)
9: **if** (dist[v] $<$ dist[u]+$w(u, v)$)
10: dist[v] = dist[u]+$w(u, v)$
11: **end function**

or adjacency list) that each processor has its own the vertices and all edges which are directed from those vertices. Each version of the Dijkstra's algorithm uses a different heuristic to specify which vertices are dropped out from the priority queue. That is, these variants provide different priority queue implementations. Pseudocode of the Boost Dijkstra algorithm is shown in Algorithm 2 [30]. The algorithm can be considered as a modified version of distributed parallel breadth first search.

Algorithm 2 dijkstra_ shortest_ paths()

1: *//G is the input graph*
2: *//s is the source node*
3: relaxed_heap<Vertex> Q;
4: *//Visitor that updates the priority queue*
5: dijkstra_bfs_visitor bfs_vis(Q);
6: breadth_first_visit (G, s, Q, bfs_vis);

In Algorithm 3, dijkstra_bfs_visitor is given. **tree_edge()** and **gray_target()** are functions which relax the edges for updating the priority queue. **tree_edge()** is called if the target node of an edge has not been seen, but the breath first search spanning tree includes the edge. Furthermore, **gray_target()** is called if the target node of an edge has been seen; however, it is not processed [30]. Next we describe different implementations of the Dijkstra's algorithm.

Algorithm 3 struct dijkstra_bfs_visitor

```
1: function TREE_EDGE(Edge e, Graph &g)
2:     if(dist(source(e,g))+weight(e) < dist(target(e,g)))
3:         dist(target(e,g))=dist(source(e,g))+weigh(e)
4: end function
5: function GRAY_TARGET(Edge e, Graph &g)
6:     if(dist(source(e,g))+weight(e) < dist(target(e,g)))
7:         Q.update(target(e,g))=dist(source(e,g))+weigh(e)
8: end function
```

Eager Dijkstra's algorithm uses a constant lookahead factor λ. In Eager version, every vertex u is removed from the priority queue by each processors if dist(u) $\leq \mu + \lambda$ where μ is the global minimum value. If $\lambda = 0$, Eager Dijkstra's algorithm is equivalent to the naive parallelization of Dijkstra's algorithm. It is noted that the larger λ results in more parallelism, but this may cause a lot of reinsertions in the priority queue. The optimum λ value depends on the graph density, shape, and distribution of the edge weights.

Crauser et al. [25] variant of the Dijkstra's algorithm introduces more precise heuristic in order to remove more vertices from the priority queue without causing reinsertions. Crauser et al. variant of the Dijkstra's algorithm does not include any parameter which will be tuned differently than the Eager version. This algorithm employs two different criteria that determine which vertices are removed from the priority queue. They are the OUT-criterion and IN-criterion. A threshold L based on the outgoing edges is computed such that $L = min\{dist(u) + weight(u,w)|$ u *is queued and* $(u,w) \in E\}$. Each vertex v with $dist(v) \leq L$ is safely removed from the priority queue by the processors. In the IN-criterion, a threshold based on the incoming edges is computed. If the condition $dist(v) - min\{weight(u,v) : (u,v) \in E\} \leq \mu$ (where μ is the global minimum) is satisfied, the vertex v is safely removed from the priority queue.

Δ-stepping is known to be the best version of the Dijkstra's algorithm [52] and hence, in our studies, we use Δ-stepping as the baseline algorithm. Δ-stepping is given in detail in the following section.

2.1 Δ-stepping

Δ-stepping proposed by Meyer and Sanders [52] solves the single source shortest path problem efficiently for large sparse graphs. It can be implemented in parallel or sequentially, and achieves significant speedup on parallel computers. The average sequential running time of Δ-stepping algorithm is $O(n + m + dL)$ where d is the maximum vertex degree, L is the maximum shortest path length, n is the number of vertices, and m is the number of edges. In parallel implementations, the time complexity for PRAM model is $O(dLlogn + log^2 n)$ on average [52].

Δ-stepping can be considered as a generalization of Dijkstra's and Bellman-Ford algorithms. The degree of the parallelism and processing time depend on Δ-parameter. when $\Delta = 1$, the algorithm turns into Dijkstra's algorithm. On the other hand, when Δ is infinity, the algorithm turns into Bellman-Ford algorithm.

In Δ-stepping algorithm, a tentative distance is assigned for each node as in Dijkstra's algorithm. This distance represents the weight from the node to the source vertex. The main difference between Dijkstra's and Δ-stepping algorithms is that Dijkstra's algorithm removes one vertex from a priority queue at a time; on the other hand, Δ-stepping groups the vertices in buckets storing the tentative distance in range of size Δ and uses bucket based priority queue. That is, i^{th} bucket stores the nodes whose tentative distance within $[i\Delta, (i + 1)\Delta]$. Δ-stepping removes all vertices from the smallest bucket in parallel and relaxes all *light* edges (whose weight is less than Δ) of these nodes (relax function is given in Algorithm 5). In this step, there may be reinsertion into the current bucket if their tentative distance is improved. Moreover, new nodes may be added into the current bucket. Next, the remaining *heavy* edges (whose weight is greater than Δ) are relaxed once when a bucket is empty [52] after relaxations. Δ parameter should be chosen carefully. If Δ is too small, then the number of reinsertions is large. Otherwise, if it is too large, the degree of parallelism is small. The pseudocode of the algorithm is given in Algorithm 4. B represents bucket arrays in the queue. The nodes are sorted by using the buckets which denote priority ranges of size Δ [23]. All nodes are deleted from the bucket and light edges are relaxed, so new nodes may be added to the current bucket in each iteration of the algorithm (the inner while loop, lines 10-17 in Algorithm 4). When a bucket is empty, all heavy

Algorithm 4 Δ-stepping Algorithm [23]

Input: $G(V, E)$ and weights, $w : E \to R^+$

Output: $dist(u)$, $u \in V$, the shortest path length from source to u

1: **for** $u \in V$ **do**
2: $heavy(u) \leftarrow \{(u, v) \in E : w(u, v) > \Delta\}$
3: $light(u) \leftarrow \{(u, v) \in E : w(u, v) \leq \Delta\}$
4: $dist(u) \leftarrow \infty$
5: **end for**
6: relax(source,0)
7: $i \leftarrow 0$
8: **while** B is not $\emptyset$ **do**
9: $S \leftarrow \emptyset$
10: **while** B[i] is not $\emptyset$ **do**
11: $Req \leftarrow \{(v, dist(u) + w(u, v)) : u \in B[i] \wedge (u, v) \in light(u)\}$
12: $S \leftarrow S \cup B[i]$
13: $B[i] \leftarrow \emptyset$
14: **for** $(u, x) \in Req$ **do**
15: relax(u,x)
16: **end for**
17: **end while**
18: $Req \leftarrow \{(v, dist(u) + w(u, v)) : u \in S \wedge (u, v) \in heavy(u)\}$
19: **for** $(u, x) \in Req$ **do**
20: relax(u,x)
21: **end for**
22: **if** u == target **then**
23: break
24: **end if**
25: $i \leftarrow i + 1$
26: **end while**

Algorithm 5 relax(u,x) function in Δ-stepping algorithm

1: **if** $x < dist(u)$ **then**
2: B[$\lfloor dist(u)/\Delta \rfloor$] $\leftarrow$ B[$\lfloor dist(v)/\Delta \rfloor$] $\setminus \{u\}$
3: B[$\lfloor x/\Delta \rfloor$] $\leftarrow$ B[$\lfloor x/\Delta \rfloor$] $\setminus \{u\}$
4: $dist(u) \leftarrow x$
5: **end if**

edges are relaxed at once (lines 18-21 in Algorithm 4) in each iteration. Moreover, while deletion and edge relaxation can be done in parallel, individual relaxation can be done in atomically in constant time.

There are several algorithms to solve the single source and single target shortest path problem such as bidirectional search [63], and the algorithms in [45, 74, 82, 38]. We recall that Δ-stepping is a single source shortest path algorithm and computes the shortest path between the source node to all other nodes. In order to compute the single source-single target shortest path using Δ-stepping, we straightforward modify Δ-stepping algorithm to terminate when no paths shorter than the shortest path from source to target existed by permitting to make early-termination for the source-target shortest paths (see steps 22-24 in Algorithm 4).

When edge weights are negative, Bellman-Ford algorithm [13], which is another well-known single source shortest path algorithm, is used instead of the Dijkstra's algorithm. It allows negative edge weights and can detect negative cycles in a graph. It has $O(|E||V|)$ time complexity and more expensive than Dijkstra's algorithm. Chakaravarthy et.al. [19] introduce a novel algorithm to solve the shortest path problem by combining Bellman-Ford and Dijkstra's algorithms. They do not only apply some pruning techniques to reduce communication, but also propose load balancing techniques to cope with higher degree vertices. They obtain promising results on real world and scale free graphs.

In some applications like distributed computing [8, 61], computational geometry [6] and others, it may be sufficient to compute approximate shortest path instead of exact shortest path. In this case, the shortest path is computed with some errors. Gubichev et al. [42] computed approximate shortest paths with high accuracy over large real world social and biological networks. Yuster [81], Elkin [33] and Sen [75] have

studied on approximate shortest path algorithms. However, their methods are mainly for solving all pairs shortest path problems.

In the studies mentioned above, static graphs are used and if the graph changes over time, they solve this problem by recomputing the shortest path information whenever the graph changes. They, however, consume relatively more time when the number of edges that are changed is small. Therefore, dynamic algorithms have been emerged in order to avoid unnecessary computation involving unchanged edges [21].

King [47] proposed the first fully-dynamic algorithm to solve all pairs shortest path problems on directed graphs with positive edge weights. Then, Demetrescu and Italiano [26] introduced the first fully-dynamic algorithm on directed graphs with real edge weights by improving King's algorithm. Narvaez et al. [55] introduced a new dynamic algorithm that makes use of the previously computed shortest path information in the graph to overcome single edge weigh change although the algorithms mentioned above can accept single edge weight changes [86]. To update changes more efficiently, Narvaez [54] proposed another algorithm called as *BallString* [54] which is a semi-dynamic algorithm. They compared their results with existing results and concluded that *BallString* is the best performing algorithm when the changes are small.

Chan and Yang [21] proposed *DynDijkstra* algorithm which is a semi-dynamic version of *Dijkstra*. Ramalingam and Reps [67] proposed the fully dynamic version of the algorithm which is called as *Dynamic*SWSF-FP. Later, it has been optimized by Chan and Yang [21] and the resulting algorithm is called as *MFP*. They concluded that these algorithms should be used instead of static Dijkstra's algorithm if percentage of edges changed is smaller than a threshold. *FMN* [37, 36] and *RR* [66] are other fully-dynamic algorithms. Firigioni et al. [35] compared the performance of these algorithms. They showed that *RR* has a better performance than *FMN* due to its efficient cache usage. Djidjev et al. [28] described the first parallel algorithm for solving the dynamic shortest path problems in a planar directed graph. Chabini and Ganugapati [17] reported the design, implementation, and computational testing for parallel algorithms to solve many-to-many shortest path problems in dynamic networks. They achieved satisfactory speedups with respect to the existing sequential algorithms.

The algorithms mentioned above have some important shortcomings. First of all, most of algorithms mentioned above are sequential and there are only few parallel algorithms to solve the DSP problem. Additionally, when both the percentage of changed edges and the size of a graph become larger, these algorithms are inefficient and the process of determining affected edges gets more complicated especially in the *mix case*. Moreover, these methods implement different algorithms depending on whether increase or decrease of the edge weights. For instance, *DynDijkstra* runs *DynDijkInc* method to cope with increased edge weights and runs *DynDijkDec* to overcome decreased edge weights [86]. Because of these shortcoming, new algorithms are needed for solving the DSP problems.

Although all of these algorithms can accurately find the shortest path, they are computationally expensive and are not much amenable to parallelism [18]. Therefore, many bio-inspired algorithms have emerged, such as genetic [5], ant colony [29] and Physarum Solver [78].

Physarum Solver [78] is a popular bio-inspired method to solve the shortest path problem efficiently. This method is inspired by the behaviour of a single-celled amoeba like organism called as Physarum polycephalum which is able to find the shortest path in a labyrinth. The mathematical model of Physarum polycephalum, which exhibits behaviour of path finding in a labyrinth, is described by Tero et al. [79]. Then, Miyaji and Ohnishi [53] mathematically proved that Physarum can solve the shortest path problem on Riemann surface. Recently, Becchetti et al. [12] proved that the mathematical model of Physarum computes the shortest path approximately. Our proposed methods are based on Physarum Solver, therefore we give a detailed description of this model in the following section.

2.2 Physarum Solver

Physarum Solver [78] is developed based on physiological observations of an amoeba-like organism. It is capable of finding path in a maze which can be considered as a graph. Tero et al. [79] put food resources at two different locations of a maze which is filled to Physarum Polycephalum. Then, plasmodium of Physarum Polycephalum

constitutes a path which has the minimum length between two resources thanks to its biological structures. The body of the plasmodium of Physarum Polycephalum includes a network of tubes which circulates to nutrients and chemical signals and it finds the shortest path by concentrating to food resources to feed on the nutrients. This experiments are repeated for different mazes and for all experiments, the organism can be able to find the shortest path. Modelling the behaviour of Phsarum Polycephalum has been developed, namely Physarum Solver [79].

Physarum Solver is an iterative solver and it differs from other well known methods with the following capabilities. The paths which do not start at the starting or end at the ending vertices and longer paths are gradually eliminated by Physarum Solver. Moreover, it is capable of finding more than one shorter paths contrary to the other shortest path algorithms. Furthermore, Physarum Solver is a flexible algorithm when compared to the classical shortest path algorithms in the sense that it ends the execution whenever the desired accuracy is obtained. A proof of an approximation of the shortest path by using the iterative method and the analysis of the running time are presented in [12].

Physarum Solver requires solution of sparse linear system of equations and Miyaji and Ohnishi [53] showed that the coefficient matrix is a symmetric M-matrix. Although they can be solved via direct or iterative techniques, it is well known that the preconditioned iterative methods are more suitable for such systems.

2.2.1 Mathematical Model

Let $G(V, E, w)$ be a graph where V is the set vertices, E is the set of edges, and w is a function from E to R^+. Assume that G is a connected undirected graph. E_{ij} represents the edge connecting from vertex V_i to vertex V_j. The variable Q_{ij} is used to state the flux though E_{ij} from V_i to V_j. On the assumption that the tube has the Poiseuille flow approximately, Equation 2.1 is obtained.

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

where D_{ij} is the conductivity of the tube E_{ij}, L_{ij} is the weight of the edge E_{ij}, and p_i is the pressure at vertex V_i.

On the assumption of zero capacity of each vertex except the starting and ending vertices, if the *Kirchoff's* law [53] is applied, Equation 2.2 is obtained.

$$\sum_{i \neq j} Q_{ij} = \begin{cases} I_0 & \text{if } i = source \\ -I_0 & \text{if } i = target \\ 0 & otherwise \end{cases} \tag{2.2}$$

Tero et al. [79] prove that while the tubes which have larger fluxes are supported, tubes which have smaller fluxes are eliminated. Equation 2.3 called as *adaptation equation* describes the change in the conductivity D_{ij} with respect to time.

$$\frac{d}{dt} D_{ij} = f(|Q_{ij}|) - D_{ij}. \tag{2.3}$$

The Poisson's equation is obtained by using Equation 2.1 and 2.2 to compute the pressure of each node in Equation 2.4.

$$\sum_{i} \frac{Q_{ij}}{L_{ij}} (p_i - p_j) = \begin{cases} I_0 & \text{if } j = source \\ -I_0 & \text{if } j = target \\ 0 & otherwise \end{cases} \tag{2.4}$$

where $p_n = 0$ and Equation 2.4 can also be written in the matrix from

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

where p=$(p_1, p_2, ..., p_n)^T$ is the unknown, b is right hand side vector with zeros except +1 and -1 on the source and target vertices, respectively, and $A = (A_{ij})$ is a symmetric M-matrix [53] given by Equation 2.6.

$$A_{ij} = \begin{cases} \sum_{l \neq i} T_{il} & \text{if } i = j \\ -T_{ij} & otherwise \end{cases} \tag{2.6}$$

where $T_{ij} = D_{ij}/L_{ij}$. Therefore, all p_i's can be computed by solving Equation 2.5. Although there exists more complicated models, we assume a simpler $f(|Q_{ij}|) =$

$|Q_{ij}|$ as in [79]. We use Equation 2.7 to discretize the conductivity values in Equation 2.3

$$\frac{D_{ij}^{n+1} - D_{ij}^{n}}{\Delta t} = |Q_{ij}^{n}| - D_{ij}^{n+1} \tag{2.7}$$

where Δt indicates the duration of one discrete time step (or iteration) and D_{ij}^{n+1} represents the conductivity of the edge E_{ij} in $(n+1)^{th}$ nonlinear iteration (i.e. time step). Note that here we use the term "nonlinear iteration" to distinguish it from the iterations to solve the linear systems. For more details, see [53, 79].

2.2.2 Applications of Physarum Algorithm

Physarum related algorithms have various application areas. Liu et al. used Physarum algorithm to solve Steiner tree problem [50] in networks and called the resulting algorithm as Physarum optimization. In order to accelerate the convergence of the Physarum algorithm, they used edge-cutting scheme and feedback-adjusting scheme. The fundamental idea of the edge-cutting scheme is to delete the edge whose conductivity is small enough [50]. The main disadvantage of this scheme is that the deleted edges cause to fluctuate quickly of the nearby vertices. Thus, the difference between $p_i - p_j$ changes in a larger amplitude. In this scheme, it is important how to determine whether the conductivity of an edge is small enough. To do this, they used a threshold τ. If the conductivity of an edge (D_{ij}) is smaller than this threshold, then the corresponding edge (e_{ij}) will be deleted. They experiment that the number of remaining edges decreases quickly at first, however, after a while the number of remaining edges decreases slightly [50]. Therefore, after several iterations, deleting an edge is difficult by using a fix threshold. To handle this problem, a dynamic threshold is used to delete the edges. This threshold increases when the number of iterations increases. In feedback-adjusting scheme, they used positive feedback between the flux and conductivity shown in Equation 2.8

$$D_{ij}^{t+1} = \kappa_{ij}^{t} D_{ij}^{t} \tag{2.8}$$

where $\kappa_{ij}^{t} = 1 + \sigma \frac{|\sum_{k=1}^{g}(p_{ik}^{t} - p_{jk}^{t})|}{L_{ij}} - \rho c_{ij}$, σ and ρ control the flux weights and cost c_{ij}, respectively.

Zhang et al. [85] described an other Physarum related bio-inspired algorithm in the

transportation networks for identification of critical components. In their model, they used a flow value which can be calculated by the following equality:

$$L_{ij} = L_{ij} + c_{ij}(f_{ij}) \tag{2.9}$$

where c_{ij} is a function presenting the relationship between the flow and the cost of the edge (i, j). The Physarum model for directed network can be given as in the following equation:

$$\sum_{i} (\frac{D_{ij}}{L_{ij}} + \frac{D_{ji}}{L_{ji}})(p_i - p_j) = \begin{cases} +f_{ij} & \text{if } j = source \\ -f_{ij} & \text{if } j = target \\ 0 & otherwise \end{cases} \tag{2.10}$$

where f_{ij} represents the flow on the edge (i, j). Their proposed algorithm is given in Algorithm 6. In this algorithm, when the termination criterion is satisfied, it is accepted that the network achieves an equilibrium state and they compute the total cost of the network by using the following equation

$$TC = \sum_{i,j=1,2,...,N} L_{ij} f_{ij} \tag{2.11}$$

where L_{ij} is the weight associated with the edge (i, j) and f_{ij} shows the flow on the edge (i, j).

In order to identify the critical links in the transportation network, they used the following strategy. To remove the edge g from a network G, first, the importance of this edge is measured by using following equation

$$l(g) = |TC(G) - TC(G - g) - L_g f_g| \tag{2.12}$$

where $G - g$ is the network which does not include the edge g, L_g and f_g show edge weight and the flow of the edge g, respectively. They noticed that an edge is more important is its removal has significant effect on the reconstruction of equilibrium state in the transportation network. Therefore, by using this strategy, they identified all important edges in the network and removed the edges entering and existing that edge.

Zhang et al. [90] combined Physarum model with ant colony approach and they called resulting algorithm as *PMACO*, which is more efficient than original ant colony algo-

Algorithm 6 Physarum inspired algorithm in transportation network [85]

//N is the size of network
L_{ij} is edge weight between vertex i and vertex j
s is the starting vertex and t is the ending vertex
$D_{ij} \leftarrow 1$ where $L(i,j) \neq 0$
$Q_{ij} \leftarrow 0$
count $\leftarrow 1$
while a termination criterion is met **do**
    Calculate pressures by using Equation 2.10
    $Q_{ij} = \frac{D_{ij}}{L_{ij}}(p_i - p_j)$
    $D_{ij} = Q_{ij} + D_{ij}$
    if $Q_{ij} < 0$ **then**
        $Q_{ij} = 0$
    end if
    //Update the cost of each edge weights
    for i=1:N **do**
        for j=1:N **do**
            if $Q_{ij} \neq 0$ **then**
                $L_{ij} = L_{ij} + c_{ij}(f_{ij})$
                $L_{ji} = L_{ij}$
            end if
        end for
    end for
    count $\leftarrow$ count+1
end while

rithm. By using Physarum, the efficiency and robustness of the ant colony algorithm is improved. Zhang et al. [87] proposed another Physarum related algorithm to solve 0-1 knapsack problem.

Liang et al. [49] proposed a new genetic algorithm based on the modified Physarum network. In their hybrid algorithm, to improve the crossover operator in the traditional genetic algorithms, Physarum network model is used. By ignoring the parts related to the genetic algorithm, we summarizes only their modified Physarum network model. In this model, they used a source node s and the destination nodes (DE) to present the starting and ending points of the flux, respectively. By using Kirchhoff's law, the flux at the node s is equals to the total flux of output at all nodes in DE. Therefore, Equation 2.13 is obtained

$$\sum_i Q_{ij} = \begin{cases} I_0(N-1) & \text{if } j = source \\ -I_0 & \text{if } j \in DE \\ 0 & otherwise \end{cases} \tag{2.13}$$

Therefore, Physarum spanning tree is obtained. The linear equations (see Equation 2.5) in Physarum model is generally solved by the Gaussian elimination to find the pressure values (p_i). In their modified Physarum algorithm, they used an approximate solution to the linear system shown in Equation 2.14

$$p_i^{t+1} = \begin{cases} \frac{I_0 + \sum_j \frac{D_{ij}}{L_{ij}} p_j^t}{\sum_j \frac{D_{ij}}{L_{ij}}} & \text{if } v_i \text{ is a target node} \\ 0 & \text{if } v_i \text{ is a source node} \\ \frac{\sum_j \frac{D_{ij}}{L_{ij}} p_j^t}{\sum_j \frac{D_{ij}}{L_{ij}}} & otherwise \end{cases} \tag{2.14}$$

However, they reported that the approximate solution sometimes leads to nonconvergence of the algorithm. In order to provide convergence, they use a parameter λ which controls the fluctuations of pressures and conductivities. Therefore, the following approximation is obtained

$$p_i^{t+1} = \begin{cases} \frac{I_0 + \sum_j \frac{D_{ij}}{L_{ij}} p_j^t}{\sum_j \frac{D_{ij}}{L_{ij}}} \lambda + (1-\lambda) p_i^t & \text{if } v_i \text{ is a target node} \\ 0 & \text{if } v_i \text{ is a source node} \\ \frac{\sum_j \frac{D_{ij}}{L_{ij}} p_j^t}{\sum_j \frac{D_{ij}}{L_{ij}}} \lambda + (1-\lambda) p_i^t & otherwise \end{cases} \tag{2.15}$$

Moreover, they used the following adapted equation in their study

$$D_{ij}^{t+1} = \frac{D_{ij}^{t} + Q_{ij}^{t}}{k} \tag{2.16}$$

where k is a parameter relative to the amplitude of conductivities in the Physarum network. The pseudocode of the modified Physarum network is given in Algorithm 7.

Algorithm 7 The modified Physarum model [49]

1: Initializing with $D_{ij} = (0, 1]$ and $p_i = 0$

2: **while** the terminal condition is not satisfied **do**

3: Computing p_i using

$$p_i^{t+1} = \begin{cases} \frac{I_0 + \sum_j \frac{D_{ij}}{L_{ij}} p_j^t}{\sum_j \frac{D_{ij}}{L_{ij}}} \lambda + (1-\lambda) p_i^t & \text{if } \ v_i \ is \ a \ target \ node \\ 0 & \text{if } \ v_i \ is \ a \ source \ node \\ \frac{\sum_j \frac{D_{ij}}{L_{ij}} p_j^t}{\sum_j \frac{D_{ij}}{L_{ij}}} \lambda + (1-\lambda) p_i^t & otherwise \end{cases} \tag{2.17}$$

4: Update D_{ij} using

$$D_{ij}^{t+1} = \frac{D_{ij}^{t} + Q_{ij}^{t}}{k} \tag{2.18}$$

5: **end while**

One disadvantage of this method is that if the parameters (k and λ) are not chosen carefully, the scalability of the algorithm will not be better. They run the model for different λ and k values on small size random graphs. They showed that the modified Physarum model is more sensitive to λ rather than k, while the higher k means that the algorithm converge in a few number of iterations, the higher λ means that the algorithm converge in a larger number of iterations.

We note that in the earlier studies, they use only small scale random graphs (such as the graphs only 50 vertices) and when large graphs are used (typically the case for real world applications), the algorithm becomes inefficient. Moreover, the parameters they experimented may change with respect to the graph structure and parameter analysis step is too much time consuming if large graphs are used.

Xu and Jiang [80] proposed a Physarum related model to solve the profit based stochastic user equilibrium problem. They modified the original Physarum model to find the shortest path in traffic direction networks. In the original Physarum model,

there is one source and one target node and they extended the model with accepting the multiple source and multiple target nodes. Furthermore, their modified Physarum model can find the shortest path on directed networks. Now, we give the details of their modified Physarum model. They modified the conductivity equation defined by Equation 2.1 into the following:

$$Q_{ij} = \begin{cases} \frac{D_{ij}}{L_{ij}}(p_i - p_j) & \text{if} \quad \frac{D_{ij}}{L_{ij}}(p_i - p_j) > 0 \\ 0 & otherwise \end{cases} \tag{2.19}$$

Let O present the set of source nodes (origin nodes) and D be the set of destination nodes. The linear equation (Equation 2.2) accepting multiple source and target nodes turns into the following equation

$$\sum_i (\frac{D_{ij}}{L_{ij}} + \frac{D_{ji}}{L_{ji}})(p_i - p_j) = \begin{cases} -I_o & \text{if} \quad \forall o \in O \\ +I_d & \text{if} \quad \forall d \in D \\ 0 & otherwise \end{cases} \tag{2.20}$$

Further applications of Physarum related algorithms can be found in [88, 84].

When compared to the existing methods, the main advantages of Physarum algorithm is its adaptivity in network design. Zhang et al. [86] uses this property to solve the DSP problems. In their study, Physarum algorithm is modified for directed graphs (see Algorithm 8). They used four random graphs whose sizes change from 500 to 2,000 and their edge weights change over time. They gave experimental results by evaluating graph size, ratio of updated edges, and ratio of changed weight. They presented that while the ratio of updated edges has little impact on the performance of the algorithm, the ratio of changed weight has a larger effect on the performance.

Algorithm 8 Physarum Algorithm for directed graphs [86]

1: $D_{ij} \leftarrow 1$ where $L(i,j) \neq 0$

2: $Q \leftarrow 0$

3: $p \leftarrow 0$

4: **while** the terminal condition is not met **do**

5: $p_t \leftarrow 0$

6: Computing the pressures using the following equation

$$\sum_i (\frac{D_{ij}}{L_{ij}} + \frac{D_{ji}}{L_{ji}})(p_i - p_j) = \begin{cases} -1 & \text{if} \quad j = source \\ +1 & \text{if} \quad j = target \\ 0 & otherwise \end{cases} \tag{2.21}$$

7: $Q_{ij} \leftarrow D_{ij} \times (p_i - p_j)$

8: **if** $L_{ij} \neq \inf$ and $p_i < p_j$ **then**

9: $Q_{ij} = 0$

10: **end if**

11: $D \leftarrow Q + D$

12: **end while**

CHAPTER 3

SOLVING SPARSE LINEAR SYSTEMS WHOSE COEFFICIENT MATRIX IS A SYMMETRIC M-MATRIX

Solving sparse linear systems whose coefficient matrix is a symmetric M-matrix plays a major role in *Physarum Solver*. In the sequential solution of those systems, it is known that Gauss-Seidel method has a favorable rate of convergence for solving such systems. In this thesis, we further investigate the parallel scalability of the solution of those linear systems by using state-of-the art methods. Therefore, in this chapter, we give an overview of solution methods of general sparse linear systems and recalls important results.

Plemmon [62] gave 40 different properties of the M-matrices and we give some of the important properties below. First we give the definition of an M-matrix.

A is an $n \times n$ square matrix. If A can be written as $\alpha I - B$ where B is nonnegative matrix and $\rho(B) \leq \alpha$, then A is called as an ***M-matrix***. Some important properties of the M-matrices are listed as follows [62]:

1. M-matrices have positive principal minors
2. For each M-matrix A, $A + D$ is nonsingular where D is a diagonal matrices with positive entries
3. For each $x \neq 0, \quad x^T ADx > 0$ where D is a nonnegative diagonal matrix
4. All real eigenvalue of an M-matrix is positive
5. Let A be an M-matrix. $A = LU$ where L and U are lower and upper triangular matrices with positive diagonals

6. For an M-matrix A, A^{-1} exists and $A^{-1} \geq 0$. That is, A is inverse-positive

7. For an M-matrix A, A is monotone. That is,

$$Ax \geq 0 \Longrightarrow x \geq 0 \quad for \ \ all \quad x \in R^n \tag{3.1}$$

8. For an M-matrix A, A can be split into two parts such that

$$A = M - N, \quad M^{-1} \geq 0, \quad N \geq 0 \tag{3.2}$$

   where $\rho(M^{-1}N) < 1$

9. For any M-matrix A,

$$AD + DA^T \tag{3.3}$$

   is positive definite where D is a positive diagonal matrix

10. For each M-matrix A and positive semidefinite matrix P, PA has a positive diagonal element

11. The real part of the eigenvalues of an M-matrix is positive

12. For any M-matrix A,

$$AW + WA^T \tag{3.4}$$

   is positive definite where W is a symmetric positive definite matrix

13. For each M-matrix A, A is *semipositive*. That is, $Ax > 0$ for any $x > 0$

14. For any M-matrix A, the row summation of AD matrix is positive where D is a positive diagonal matrix

15. For any M-matrix A, A has positive diagonal elements, and there exist a positive diagonal matrix D such that AD is strictly diagonal dominant

Next, we summarize solution methods for general linear systems. A linear system of m equations with n unknowns can be formed as

$$\textstyle\sum_{j=1}^{n} a_{ij}x_j = b_i \qquad \text{for} \quad i = 1, 2, 3, ..., m$$

where a_{ij} is the coefficient of the linear systems, x_j is unknowns, and b_i is the right hand side. In this thesis, we interest the problems where $n = m$, that is, the number of unknowns is equal to the number of equations. We can write these equations compactly as follows

$$Ax = b, \tag{3.5}$$

where A is large and sparse coefficient matrix of order n, b is the right hand side, and x is the unknown vector. However, in order to find the solution, computing A^{-1} is computationally expensive and often numerically unstable. There are various techniques for solving sparse linear system of equations. These techniques can be investigated in two categories which are direct and iterative techniques. In these techniques, time to obtain the solution is a major factor because the size of the problem is usually large. Another important factor for solving linear system of equations is accuracy.

Direct methods are used in many problems in which robustness is the main issue [14]. In general, a factorization followed by solution via the factors is considered to be a direct method. There a number of factorizations such LU, QR , WZ and many others. However, they are not preferred for solving large linear system of equations because of the considering memory requirements fill-in and arithmetic operation counts. Iterative methods are preferred for solving large linear system of equations because they use less memory and operation counts than direct methods, especially when an approximate solution of relatively low accuracy is sought [14]. They do not show as much robustness as direct methods, therefore they often fail in some applications and preconditioning is needed in order to improve robustness and solution time. While direct methods try to compute solution of linear systems in a finite number of operations, iterative methods start with an initial guess for the solution and improve the initial guess in a finite sequence. Accuracy in iterative methods mostly depends on the number of iterations.

3.1 Direct Methods

Finding the solution of a general sparse linear systems is a computationally expensive problem. However, some special cases are relatively easier to solve. For instance, If the coefficient matrix A is just a diagonal matrix ($A = D$), the solution can be given

by the following equation

$$x_i = \frac{b_i}{D_{ii}}. \tag{3.6}$$

An other example is that if the coefficient matrix is an orthogonal matrix, the solution can be given as follow:

$$x = Q^T b. \tag{3.7}$$

Additionally, if the coefficient matrix is a lower triangular (or upper triangular), the solution can be obtained by computing each x_i successively in Equation 3.8 for the lower triangular case (and similarly for the upper triangular case):

$$x_i = \frac{b_i - \sum_{i=1}^{i-1} L_{i,j} x_j}{L_{i,i}}, \quad i = 1, 2, ..., n \tag{3.8}$$

The direct methods are based on decomposing or factorizing the coefficient matrix A into the product of matrices that are in a more favorable form. There are various direct methods. Most widely used direct methos is the sparse Gaussian elimination and the resulting LU factorization. It requires the factorization of A as shown in Equation 3.9

$$A = LU \tag{3.9}$$

where L is a lower triangular matrix and its diagonal consists of 1's and U is an upper triangular matrix. In order to find the solution, first, we solve the lower triangular system shown in Equation 3.10 and the unknown vector y is found.

$$Ly = b \tag{3.10}$$

Second, we solve the upper triangular (i.e. the forward substitution) system and so the solution vector x that satisfies $Ax = b$ can be obtained from Equation 3.11

$$Ux = y. \tag{3.11}$$

This process, that is, computing the unknowns from an upper triangular system is called the backward substitution. Pseudocode of LU factorization is shown in Algorithm 9. The six permutations of the indices i, j, and k produce six different organizations of LU factorization and we call these "*ijk*" forms, and depending on the computing platform different forms could have advantages [58].

Algorithm 9 LU Factorization (*kij* form)

```
1: for k=1:n-1 do
2:     for i=k+1:n do
3:         A(i,k) = A(i,j)/A(k,k)
4:     end for
5:     for i=k+1:n do
6:         for j=k+1:n do
7:             A(i,j) = A(i,j) − A(i,k)A(k,j)
8:         end for
9:     end for
10: end for
```

When the coefficient matrix A has special properties, using variants of LU factorization improves the performance. For instance, if the coefficient matrix is symmetric, the LDL^T decomposition can be used. Additionally, if the coefficient matrix is symmetric positive definite, the LL^T decomposition also known as Cholesky factorization is used. The other well known factorization methods are *UL*, *QR*, *RQ*, *WZ*, and *SVD* [73, 68, 64].

3.2 Iterative Methods

The main idea of the iterative methods is that passing from one iteration to the next one by gradually improving the approximate solution vector. To solve the linear systems shown in Equation 3.5 by using iterative methods, $\{x_k\}$ sequence vectors is constructed such that

$$\lim_{k\to\infty} x_k,$$

where x represents the real solution given by Equation 3.5 and x_k is the k^{th} approximation of the exact solution. We note that when $k = 0$, x_0 is called as the initial guess. The construction of an iterative method begins with a splitting of A in Equation 3.5. Therefore, A is written as $A = M - N$ where $det(M) \neq 0$ and M is easily

inverted such that Equation 3.5 is equivalent to $x = Tx + d$ where $T = M^{-1}N$ and $d = M^{-1}b$.

Convergence of iterative methods depends on properties of coefficient matrix such as, distribution of eigenvalues, and hence it depends on spectral radius, and condition number. Spectral radius of coefficient matrix is the largest eigenvalue of the coefficient matrix. Condition number is the ratio of the largest eigenvalue and the smallest eigenvalue of the coefficient matrix. If the condition number is much more larger than the unity, then eigenvalues of coefficient matrix are widespread and problem is ill-conditioned. On the other hand, if the condition number is close or equal to one, then eigenvalues are tightly together and problem is well-conditioned. Therefore, knowing distribution of eigenvalues says a lot about the convergence. Gershgorin theorem gives a bound on eigenvalues without finding them. According to theorem, eigenvalues lies in a disk. To be more clear, each eigenvalue satisfies at least one of the inequalities shown in equation 3.12.

$$| \lambda_i - a_{ii}| \leq r_i \tag{3.12}$$

where $r_i = \sum_{j=1, j\neq i}^{n} |a_{ij}|$ and $A = [a_{ij}]$, $i = 1, 2, .., n$.

However, there are some drawbacks of iterative methods. Classical preconditioned iterative methods may not be robust and often this can be improved with using better preconditioning strategies.

3.2.1 Projection Methods

The projection methods aim to find the approximate solution of the linear systems in a subspace of R^n. Let A be n-dimensional matrix, and K and L be two m dimensional subspace of R^n. K represents search space and L represents the subspace of constraints [71]. A projection technique aims to find an approximate solution $\tilde{x} \in K$ such that $b - A\tilde{x} \perp L$. The projection methods can be classified into two groups which are orthogonal and oblique. While in orthogonal projections, the subspace L is equal to K, in oblique projections, the subspace L is different from K, in fact, they

are unrelated to each other. When the initial guess x_0 is provided, the problem turns into the following one:

$$\text{Find} \quad \tilde{x} \in x_0 + K \quad \text{such that} \quad b - A\tilde{x} \perp L.$$

Approximate solution can be stated as

$$\tilde{x} = x_0 + \delta, \quad \delta \in K \tag{3.13}$$

$$(r_0 - A\delta, \omega) = 0, \quad \forall \omega \in L. \tag{3.14}$$

Next, we represent this problem using matrices. The columns of these matrices are constituted as the basis vectors of these subspaces. Assume that V and W are the matrix representations of K and L, respectively. If approximate solution can be formed as:

$$\tilde{x} = x_0 + Vy, \tag{3.15}$$

then, if the orthogonality condition is applied, the following equations can be obtained for vector y:

$$W^T AVy = W^T(b - Ax_0). \tag{3.16}$$

When we assume that $W^T AV$ is nonsingular, the following expression is obtained for the approximate solution $\tilde{x}$,

$$\tilde{x} = x_0 + V(W^T AV)^{-1} W^T(b - Ax_0). \tag{3.17}$$

A prototype projection algorithm is summarized by Algorithm 10.

We note that $W^T AV$ may be singular. Anyone of the following conditions guarantees a nonsingular of $W^T AV$:

- A is positive definite and $L = K$.
- A is nonsingular and $L = AK$.

Then, $W^T AV$ is nonsingular for any basis V and W of K and L, respectively. Further information can be found in [71].

Algorithm 10 Prototype Projection Algorithm [71]

1: **while** until convergence **do**
2: Determine K and L subspaces
3: Select V and W for K and L, respectively
4: $r := b - Ax$
5: $y := (W^T AV)^{-1} r$
6: $x := x + Vy$
7: **end while**

3.2.2 Krylov Subspace Methods

Krylov subspace methods based on the projection methods where K_m is an m-dimensional Krylov subspace are defined as following:

$$K_m(A, r_0) = span\{r_0, Ar_0, A^2 r_0, ..., A^{m-1} r_0\}, \tag{3.18}$$

where $r_0 = b - Ax_0$. By choosing different subspaces of L_m and using preconditioning, the different versions of Krylov subspace can be obtained. Krylov subspaces are classified into four categories roughly:

1. *Ritz-Galerkin:* Build x_k such that $b - Ax_k \perp K_m(A, r_0)$.

2. *Residual norm minimization:* Choose x_m such that $||b - Ax_m||_2$ is minimized over $K_m(A, r_0)$.

3. *Petrov-Galerkin:* Choose x_m such that $b - Ax_m \perp L_m$.

4. *Error norm minimization:* Choose x_m such that $||x_m - x||_2$ is minimized on $A^T K_m(A^T, r_0)$.

Conjugate Gradient [65], the Generalized Minimum Residual Method (GMRES) [72], and Bi-Conjugate Gradient Stabilized (BiCGSTAB) [9] are examples of the Krylov subspace methods.

3.2.3 Recycling Krylov Subspace Methods

Some applications require to find the solution of a sequence of the linear systems shown in Equation 3.19

$$A^{(i)}x^{(i)} = b^{(i)} \qquad for \qquad i = 1, 2, 3, ... \tag{3.19}$$

where the coefficient matrix $A^{(i)} \in R^{n\times n}$ and the right hand side $b^{(i)} \in R^n$ change from one iteration to the next one. Although the change between two consecutive iterations could be small, that is $A^{(i-1)} \approx A^{(i)}$ and $b^{(i-1)} \approx b^{(i)}$, the cumulative change is significant. In order to solve such systems, we reuse the information which is computed previous iterations and so, the convergence behaviour of Krylov subspace method is improved. This is the main idea of recycling in the Krylov subspace methods. That is, the Krylov space which is generated from the previous iterations is used to solve the next linear systems, and it is expected that the method can converge in a reasonable number of iterations. It is important that the significant convergence improvement is obtained with a relatively small recycle space [60].

3.2.4 Arnoldi's Method

Arnoldi's method [7], which is an orthogonal projection method, generates a suitable orthogonal basis for the Krylov subspace K_m. The algorithm is first defined to find the eigenvalues of large sparse matrix and then it is extended to find the solution of sparse linear systems. A version of the method is given in Algorithm 11.

If Algorithm 11 does not stop before the m^{th} step, $v_1, v_2, ..., v_m$ vectors construct an orthonormal basis of the Krylov subspace.

3.2.5 The Symmetric Lanczos Method

The symmetric Lanczos method can be viewed as the special case of the Arnoldi's method when the matrix is symmetric. If the Arnoldi's method is applied to the symmetric matrix, the method generates the following coefficients h_{ij},

$$h_{ij} = 0 \quad for \quad 1 \leq i < j - 1, \tag{3.20}$$

Algorithm 11 Arnoldi's Method [71]

1: Select a vector v_1 such that $||v_1||_2 = 1$
2: **for** j=1,2,...,m **do**
3: Compute $h_{ij} = (Av_j, v_i)$ for $i = 1, 2, ..., j$
4: Compute $w_j := Av_j - \sum_{i=1}^{j} h_{ij} v_i$
5: $h_{j+1,j} = ||w_j||_2$
6: **if** $h_{j+1,j} = 0$ **then**
7: break
8: **end if**
9: $v_{j+1} = w_j / h_{j+1,j}$
10: **end for**

$$h_{j,j+1} = h_{j+1,j}, \qquad j = 1, 2, ..., n. \tag{3.21}$$

This means that the Arnold's process results in a matrix H_m which is tridiagonal and symmetric. The Lanczos algorithm is given in Algorithm 12 where $\alpha_j = h_{jj}$ and $\beta_j = h_{j-1,j}$.

Algorithm 12 Symmetric Lanczos Algorithm [71]

1: Select v_1 such that $||v_1||_2 = 1$
2: Set $\beta_1 = 0$ and $v_0 = 0$
3: **for** j=1,2,...,m **do**
4: $w_j := Av_j - \beta_j v_{j-1}$
5: $\alpha_j := (w_j, v_j)$
6: $\beta_{j+1} := ||w_j||_2$
7: **if** $\beta_{j+1} = 0$ **then**
8: break
9: **end if**
10: $v_{j+1} := w_j / \beta_{j+1}$
11: **end for**

The algorithm guarantees, at least in exact arithmetic, that the vectors v_i for $i = 1, 2, ..., m$ are orthogonal.

3.2.6 Conjugate Gradient Method

The Conjugate Gradient is an iterative method for finding a solution for large sparse linear systems whose coefficient matrix is symmetric and positive definite. There are two ways to derive the conjugate gradient method. One is as a Krylov subspace solver based on the Lanczos method (for the derivation see [59]). The other way is to view it as an enery minimization method which we will describe briefly. In order to solve the linear system of equations (shown in Equation 3.5), Conjugate Gradient algorithm minimizes the following quadratic function:

$$f(x) = \frac{1}{2}x^T Ax - x^T b. \tag{3.22}$$

A search direction d_i which is linearly independent and orthogonal to all previous search directions is determined at each iteration of Conjugate Gradient algorithm. In the orthogonality for all $j < i$, the following equation holds:

$$d_i^T A d_j = 0.$$

After choosing suitable direction vector d_k, x_{k+1} is updated as $x_k + \alpha_k d_k$ where α_k is a minimization value of the quadratic function $f(x_{k+1})$ at each iteration. As a result, a sequence $x_0, x_1, ...x_k$ is generated and $x_k \rightarrow x$ as $k \rightarrow \infty$ where x is the real solution of Equation 3.5. We note that it is expected that Conjugate Gradient algorithm finds the optimum solution in n iterations theoretically since the whole search space is covered by the directions $d_0, d_1, ..., d_{n-1}$. Pseudocode of Conjugate Gradient algorithm is given in Algorithm 13.

We note that the value of r_k is good indicator to break the for loop. If r_k is less than a threshold, the program halts. . If A is not positive definite α may become too large (Line 5) and the algorithm fails.

3.3 Preconditioning Techniques

Major drawback of iterative methods is their lack of robustness when compared to direct methods. Note that robust solvers are guaranteed to find the solution of a non-singular system after a finite number of arithmetic operations [51]. Convergence of

Algorithm 13 Conjugate Gradient Algorithm [71]

1: Determine x_0
2: $r_0 = b - Ax_0$
3: $d_0 = r_0$
4: **for** k=1,2,3,... **do**
5: $\alpha_k = \frac{(r_k, r_k)}{(d_k, Ad_k)}$
6: $x_{k+1} = x_k + \alpha_k d_k$
7: $r_{k+1} = r_k - \alpha_k A d_k$
8: $\beta_k = \frac{(r_{k+1}, r_{k+1})}{(r_k, r_k)}$
9: $d_{k+1} = r_{k+1} + \beta_k d_k$
10: **end for**

iterative methods usually depends on the eigenvalues of the iteration matrix. To improve convergence rate of iterative methods, preconditioning is often used.

The objective is to have a preconditioned system with a more favorable eigenvalue distribution than the original one and also solving linear systems involving the preconditioner is "easier". For parallel computing "easier" can translate to "more parallel (scalable)" preconditioner. The first step of the preconditioning is to define a preconditioning matrix M. A linear system in Equation 3.5 can be converted into a new linear system given in Equation 3.23

$$M^{-1}Ax = M^{-1}b \tag{3.23}$$

such that this system has the same solution of the system given by Equation 3.5.

The preconditioned system satisfies the following properties

- It should be nonsingular
- The preconditioned matrix ($M^{-1}A$) has a more favorable eigenvalue distribution which means the eigenvalues form one or maybe a few clusters.

Then $M^{-1}A$ approximates the identity matrix, so condition number of $M^{-1}A$ is smaller than the condition number of A. It would be expected that the preconditioned system shown in Equation 3.23 should converge faster. Alternatively, the

preconditioning may also be applied from the right

$$AM^{-1}y = b, \qquad x = M^{-1}y. \tag{3.24}$$

In the above equation, we note that in order to compute x by using $x = M^{-1}y$, the linear system of equations, $Mx = y$ is solved.

Moreover, the preconditioning can be applied from both left and right at the same time called as split preconditioning,

$$M_L^{-1}AM_R^{-1}y = M_L^{-1}b, \quad x = M_R^{-1}y \tag{3.25}$$

where the preconditioner $M = M_L M_R$.

Next we presents some preconditioning methods that we use in our proposed methods and for a more detailed discussion on other preonditioners please see [16].

3.3.1 Jacobi (Diagonal) Preconditioner

Jacobi preconditioner is the simplest method and it is computed by taking the diagonal entries of the coefficient matrix $A = [a_{ij}]$ and shown in Equation 3.26.

$$m_{ij} = \begin{cases} a_{ij} & \text{for } i = j \\ 0 & \text{otherwise} \end{cases} \tag{3.26}$$

We note that if diagonal entries of the coefficient matrix contain zeros, the jacobi preconditioner is singular.

If we decompose of the coefficient A matrix into three parts shown in Equation 3.27.

$$A = D + E + F \tag{3.27}$$

where D is the diagonal of A, E is the strictly lower part of A, and F is the strictly upper part of A. Therefore, the Jacobi iteration can be written in vector form as

$$x_{k+1} = -D^{-1}(E + F)x_k + D^{-1}b. \tag{3.28}$$

where the Jacobi preconditioner $M_J = D$.

3.3.2 Gauss-Seidel

Gauss-Seidel iteration [71] determines i^{th} component of the the current approximate solution by annihilating the i^{th} component of the residual and given by Equation 3.29.

$$x_i^{(k+1)} = -\frac{1}{a_{ii}} \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \frac{1}{a_{ii}} \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} + \frac{b_i}{a_{ii}} \tag{3.29}$$

where b_i are the i^{th} component of the right hand side and $i = 1, 2, ..., n$.

If we write the above equation in the matrix form, we obtain Equation 3.30

$$X_{k+1} = -(D+E)^{-1} F x_k + (D+E)^{-1} b \tag{3.30}$$

where the Gauss-Seidel preconditioner $M_{GS} = (D+E)$. In case of the coefficient matrix is symmetric, the symmetric Gauss-Seidel can be formed as:

$$M_{SGS} = (D+E)D^{-1}(D+F). \tag{3.31}$$

3.3.3 Successive Over Relaxation

Overrelaxation based splitting [71] can be defined in Equation 3.32

$$\omega A = (D + \omega E) - (-\omega F + (1-\omega)D) \tag{3.32}$$

and the corresponding Succesive Over Relaxation (SOR) method [71] can be presented in Equation 3.33

$$(D + \omega E)x_{k+1} = [-\omega F + (1-\omega)D]x_k + \omega b \tag{3.33}$$

where $\omega \in R$ is a relaxation parameter and the SOR preconditioner $M_{SOR} = \frac{1}{\omega}(D - \omega E)$. The SOR method converges if and only if

$$0 < \omega < 2. \tag{3.34}$$

The optimum ω value is given [71]:

$$w_{opt} = \frac{2}{1 + \sqrt{1 - \rho(B)^2}} \tag{3.35}$$

where $\rho(B)$ is the spectral radius of the Jacobi iteration matrix. Similarly, a symmetric SOR (SSOR) method is defined in Equation 3.36 and 3.37.

$$(D + \omega E)x_{k+1/2} = [-\omega F + (1 - \omega)D]x_k + \omega b \tag{3.36}$$

$$(D + \omega F)x_{k+1} = [-\omega E + (1 - \omega)D]x_{k+1/2} + \omega b \tag{3.37}$$

These equations can be shown as the following recurrence relation:

$$x_{k+1} = G_\omega x_k + f_\omega \tag{3.38}$$

where

$G_\omega = (D + \omega F)^{-1}(-\omega E + (1 - \omega)D) \times (D + \omega E)^{-1}(-\omega F + (1 - \omega)D)$

$f_\omega = \omega(D + \omega F)^{-1}(I + [-\omega E + (1 - \omega)D](D + \omega E)^{-1})b.$

Therefore, the symmetric SOR preconditioner can be defined as:

$$M_{SSOR} = \omega(2 - \omega)(D + \omega E)D^{-1}(D + \omega F). \tag{3.39}$$

We note that in case of $\omega = 1$, the SOR method turns into the Gauss-Seidel method.

3.3.4 The Algebraic Multigrid

Algebraic Multigrid (AMG) is a popular purely matrix based hierarchical approach [76]. The method separates the error into two parts, which are referred as rough and smooth. The rough components are decreased in size on the original (fine) grid with using some iterative methods such as Gauss-Seidel or succesive over relaxation. Therefore, the smooth components are prevalent in the error. Next, the error is interpolated back to the fine grid and is used to correct the fine-grid approximation on a larger mesh size (coarser graph). Algorithm 14 [40] presents the pseudocode of one iteration of the AMG. We note that I_h^H shows the transfer function from a fine grid (h) to the next coarser grid (H) and I_H^h from a coarse grid to the next finer grid.

In the algebraic multigrid [77, 44], a "grid hierarchy" and inter-grid transfer functions

Algorithm 14 $MGM(x^h, b^h, A_h)$ [40]

if on coarsestgrid **then**
    solve $A_h x^h = b^h$
else smooth $A_h x^h = b^h$
    $r^h = b^h - A_h x^h$
    $r^H = I_h^H r^h$
    $c^H = 0$
    apply $MGM(c^H, r^H, A_H)$
    $x^h = x^h + I_H^h c^H$
    smooth $A_h x^h = b^h$
end if

are obtained from the original matrix A. The pseudocode of the AMG method is given in Algorithm 15.

Algorithm 15 The AMG algorithm [40]

on each level set $I_h^H = (I_H^h)^T$
recursively set $A_H = I_h^H A_h I_H^h$
for i=0,1,2,... **do**
    apply $MGM(x^h, b^h)$
    convergence check
end for

3.3.5 Sparse Approximate Inverse

Preconditioning techniques based on sparse approximate inverses [83, 15, 48] became popular in recent years, mainly because its amenability to parallelism. The main idea of sparse approximate inverse is to compute a sparse M which approximates the inverse of A. Without loss of generality, we assume right preconditioning, the following minimization problems is solved.

$$min||I - AM||_F \tag{3.40}$$

where $||.||_F$ shows frobenius norm of a matrix. The main advantage of this approach is that it is highly parallel since

$$||I - AM||_F{}^2 = \sum_{i=1}^{n} ||e_j - Am_j||_2{}^2 \tag{3.41}$$

where e_j shows j^{th} column vector of identity matrix that are independent of each other and m_j shows j^{th} column vector of M. Therefore, the problem is reduced to linear least square problems. If no further constrains are applied the resulting inverse is exact and dense[33]. However, that is not preferred in practice therefore, we impose further constrains on the sparsity. There are many choices about the sparsity for example one can assume A^{-1} and A has the same sparsity structure, or adaptively increase the level of fill-in during the runtime or even assume some problem specific predetermined structure in A^{-1}. We refer the reader to [24, 71] for a more detailed overview of the approximate inverse preconditioners.

CHAPTER 4

A PARALLEL SHORTEST PATH ALGORITHM FOR STATIC GRAPHS

In this chapter, a fast and efficient parallel *Physarum Solver* is proposed for static graphs. The proposed algorithm requires solution of the linear systems whose coefficient matrix is a symmetric M-matrix, and uses an effective parallel iterative linear solver with a parallel preconditioner for M-matrices. The parallel scalability, solution time and accuracy of the proposed algorithm are evaluated and compared to a state-of-the-art parallel implementation of Δ-stepping shortest path algorithm in the Parallel Boost Graph library. The dataset to compare the algorithms contains graphs corresponding to both synthetic and real world applications. Now, we give the details of the algorithm.

4.1 The Proposed Parallel Algorithm

In this section, we present a scalable parallel bio-inspired shortest path algorithm based on *Physarum Solver*, which is called Parallel Physarum Algorithm (PPA). Portable, Extensible Toolkit for Scientific Computation (PETSc) [3], which is a widely used state-of-the-art library, is used for implementing PPA. PETSc also provides efficient parallel implementations of many subroutines such as preconditioners and linear solvers including various Krylov subspace methods.

Before we explain the proposed algorithm (given in Algorithm 16), the data structures used in PPA are explained. Graphs are represented as matrices which are stored in the coordinate format. The proposed algorithm requires solution of the linear sys-

tems and PETSc supports a variety of vector and sparse matrix data structures, which are suitable for various solvers, in parallel (MPI-based). The coefficient matrix of the linear systems (A matrix, step 9 in Algorithm 16) and b vector (right hand side of the linear system, step 10 in Algorithm 16) are stored the MATMPIAIJ storage format required by sparse linear solvers and preconditioners in PETSc. The following data partitioning scheme is used in PPA. A, Q, D, and T matrices in Algorithm 16 have the same sparsity structure and they are partitioned conformably by block rows. For instance, the matrix A is partitioned as in the following

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_t \end{pmatrix}$$

where t is the total number of processors and each process k owns the block row $A_k = [A_{ij}]_k$. Each block row has a dimension of $m_k \times n$ where

$$n = \sum_{k=1}^{t} m_k$$

in which n is the number of vertices of the graph.

Now, we explain the steps of the proposed algorithm in detail. First, the source and the target nodes are set. Next, each process k initializes own conductivity values ($[D_{ij}]_k$) to 1 and own flux values ($[Q_{ij}]_k$) to zero in steps 2 and 3, respectively. Then, each process sets associated part of the A and T matrices to be zero. In the *while* loop (steps 7-15), there is a solution of linear systems whose coefficient matrix (A) is a symmetric M-matrix in step 9. This step is the most challenging and time consuming step of the algorithm. However, in the literature, M-matrix property of the linear systems are not used and they are sequentially solved by using direct solvers. This is not feasible for large problems. We solve these linear systems in parallel by taking consideration into M-matrix property and explain details in the following subsection. Next, Q and D matrices are updated for each *while* loop iteration. The parallelism is achieved by partitioning the data, performing updates (lines 8, 9, 12, and 13) and solving the linear system (line 10) in parallel.

We emphasize that steps 8, 9, and 13 are embarrassingly parallel. That is, after the

Algorithm 16 The Proposed Parallel Algorithm

// L is an nxn matrix, L_{ij} denotes the distance between v_i and v_j

// n is the size of the graph

// t is the number of processors and k is the processor ID

1: **s** $\leftarrow$ the **source** node and **t** $\leftarrow$ the **target** node

2: $[D_{ij}]_k \leftarrow 1$ where $L(i,j) \neq 0$

3: $[Q_{ij}]_k \leftarrow 0$

4: $[A_{ij}]_k \leftarrow 0$

5: $[T_{ij}]_k \leftarrow 0$

6: $iter \leftarrow 1$

7: **while** $until\ a\ termination\ criterion\ is\ met$ **do**

8: $T_k \leftarrow \frac{D_k}{L_k}$

9: $$[A_{ij}]_k = \begin{cases} \sum_{l \neq i} [T_{il}]_k & \text{if } i = j \\ -[T_{ij}]_k & \text{otherwise} \end{cases}$$

10: **solve** $Ap = b$ (Equation 2.5) **in parallel**

11: $p_n \leftarrow 0$

12: $[Q_{ij}]_k \leftarrow [T_{ij}]_k$ x $(p_i - p_j)$

13: $[D_{ij}]_k \leftarrow \frac{1}{2}(|[Q_{ij}]_k| + [D_{ij}]_k)$

14: $iter \leftarrow iter + 1$

15: **end while**

partitioning, they do not require any communication. However, in step 10, the linear systems are solved in parallel and require some communication since each processor has own part. Similarly, in step 12, calculating the difference, $p_i - p_j$, requires a collective communication operation which we implemented using an *MPI_Allgatherv()* operation since each processor i initially only has p_i. The outer *while* loop iterations continue until one of the following termination criteria is met. First one is to check whether the Frobenius norm of the change in the conductivity values (D matrix) is less than or equal to a threshold between two consecutive outer iterations. Second, when the iteration count has reached a maximum value, execution breaks the while loop.

4.1.1 Parallel iterative solution of the linear systems

In this part, we explain some details on the parallel solution of the sparse linear systems of equations in step 10. The coefficient matrix (A) of the linear systems in step 10 has important properties. Miyaji and Ohnishi [53] showed that A (step 9 in Algorithm 16) is a symmetric M-matrix. Hadjidimos [43] shows that when left preconditioners such as Jacobi and Gauss-Seidel (type) methods applied to the systems whose coefficient matrix is an M-matrix, the system convergences in a few number of iterations. Neumann [56] mentions that the Gauss-Seidel method has a favourable rate of convergence compared to the the Jacobi method. The study of Rheinboldt [70] includes some convergence proofs that Gauss-Seidel method convergences to the unique solution when the coefficient matrix is an M-matrix.

In the proposed algorithm, parallel Conjugate Gradient (CG) is used to solve the linear systems iteratively where the coefficient matrix is a sparse symmetric M matrix (Step 9 in Algorithm 16) with symmetric Gauss-Seidel (GS) preconditioning.

Next, we explain implementation of the solution of the linear systems. PETSc KSP solver is used to solve the linear systems efficiently by following the steps in below:

1. The solver is created by using *KSPCreate()*.
2. The preconditioner is created by using *PCCreate()*.
3. For solving the sparse linear systems using preconditioned Conjugate method, *KSPSetType()* is set as *KSPCG*.
4. For using Gauss-Seidel preconditioner, *PCSetType()* is set as *PCSOR*.
5. For solving the sparse linear system, *KSPSolve()* is executed.

Eisenstat trick [32], which can be applied to any Krylov subspace method, is used to make the matrix-vector multiplication in the preconditioning step more efficient.

CHAPTER 5

A PARALLEL SHORTEST PATH ALGORITHM FOR DYNAMIC GRAPHS

In this chapter, we introduce a fully-dynamic bio-inspired parallel algorithm based on *Physarum solver* to find the shortest path from a single source to a single target on dynamically changing graphs with the positive edge weights. Our contributions are multiple folds towards obtaining an efficient parallel fully dynamic bio-inspired shortest path algorithm. Firstly, at each iteration of *Physarum solver*, a sparse linear system of equations needs to be solved, which is the most time consuming and challenging step of the algorithm especially when the problem size is large. We propose a parallel preconditioned iterative method for solving those sparse linear systems as in the static case. The proposed preconditioner is specifically designed based on the properties of the coefficient matrix of those linear systems and the effectiveness of the proposed preconditioner is compared against other state-of-the-art preconditioners on dynamic graphs. Secondly, the proposed algorithm is designed to be suitable for dynamically changing graphs since it uses the information arising in earlier iterations. Finally, while the earlier studies use only small scale random graphs on sequential computing platforms, the proposed algorithm is evaluated using three different large graph models representing diverse real life applications on a parallel multicore cluster. The parallel scalability as well as the effect of changing the edge weights to the time to obtain the solution are evaluated for each graph model, separately and compared against Δ-stepping which is the most representative parallel implementation of Dijkstra's algorithm. Next, we give the details of the proposed method for dynamic graphs.

5.1 The Proposed Fully Dynamic Parallel Algorithm

In this section, a parallel fully dynamic bio-inspired shortest path algorithm, which we call as dynamic parallel Physarum algorithm (dynPPA) is proposed based on *Physarum solver* [78]. We recall that although *Physarum solver* was presented by Tero, Kobayashi, and Nakagaki in 2006, there are only sequential variations of the algorithm. Now, we describe the steps of the proposed algorithm. In Algorithm 17, first, each process k initializes own conductivity values ($[D_{ij}]_k$) to 1 and own flux values ($[Q_{ij}]_k$) to 0. Next, in step 11, each process k updates corresponding edge weights $[L_{ij}]_k$ of the original graph, so the graph is dynamically updated in each the *for loop* iteration before entering the *while loop*. The *for loop* iterates four times like in [86], therefore the graph changes four times. In step 11, the change of edge weights has been updated in 5 different ways, which will be explained in the following subsection. In the *while loop*, a linear system is needed to be solved in step 19, which is the most critical step of the algorithm and determines the degree of parallelism as well as the number of iterations of the *while loop*. The coefficient matrix (A) of these linear systems is computed in step 14 and in order to avoid some unnecessary computations, the smaller values ($|A_{ij}| \leq 10^{-17}$) in A may be considered to be 0 in steps 15-17. Miyaji and Ohnishi [53] prove that A is a symmetric M-matrix. Although this is an important property to consider when solving the linear systems, there are not any earlier studies that uses M-matrix property of the coefficient matrix to solve the linear systems efficiently in *Physarum solver* as in the static case. In this study, Conjugate Gradient (CG) with symmetric Gauss-Seidel preconditioner is used to solve the linear systems using a moderate tolerance, which is the residual norm relative to the norm of the right hand side and is fixed at 10^{-3}. The preconditioner is specifically designed based on the properties of the coefficient matrix of those linear systems. The solution vector of the linear systems is the pressure $p = [p_i]$ for each vertex i and this step requires communication. We note that the performance of the parallel iterative solver depends on the iterative linear solver, preconditioner and the initial guess for the solution. In the rest of the thesis for simplicity we use the terms "outer" and "inner" iterations to refer to the *while loop* and iterative linear solver, respectively. In step 18, we propose to use an initial guess that is the previous solution vector multiplied by 2. The reason for this is that the conductivity values, computed by Equation 2.7

Algorithm 17 The Proposed Fully Dynamic Parallel Algorithm

1: *// L is an nxn graph, L_{ij} denotes the length between v_i and v_j*
2: *// n is the number of vertices of the graph*
3: *// k is the processor ID*
4: $[D_{ij}]_k \leftarrow 1$ where $L(i,j) \neq 0$
5: $[Q_{ij}]_k \leftarrow 0$
6: $[A_{ij}]_k \leftarrow 0$
7: $[T_{ij}]_k \leftarrow 0$
8: $iter \leftarrow 1$
9: $p \leftarrow 0$
10: **for** $the\ graph\ dynamically\ changes$ **do**
11: update of the edge weights in L_k
12: **while** $until\ a\ termination\ criterion\ is\ met$ **do**
13: $T_k \leftarrow \frac{D_k}{L_k}$
14: $[A_{ij}]_k = \begin{cases} \sum_{l \neq i}[T_{il}]_k & \text{if } i = j \\ -[T_{ij}]_k & otherwise \end{cases}$
15: **if** $|A_{ij}| \leq 10^{-17}$ **then**
16: $A_{ij} \leftarrow 0$
17: **end if**
18: $p_0 \leftarrow$ 2p
19: **solve** $Ap = b$ (Equation 2.5) **iteratively in parallel**
20: $p_n \leftarrow 0$
21: $[Q_{ij}]_k \leftarrow [T_{ij}]_k$ x $(p_i - p_j)$
22: $[D_{ij}]_k \leftarrow \frac{1}{2}(|[Q_{ij}]_k| + [D_{ij}]_k)$
23: $iter \leftarrow iter + 1$
24: **end while**
25: **end for**

approximately fall in half in step 22 (since frobenius norm of A approximately falls in half between two iterations) and the right hand side vector does not change. Next, in step 21, each process k computes corresponding flux values $[Q_{ij}]_k$ for each neighbour node j of node i based on the pressure values. It is important that computation of the flux matrix requires the edge weights $[L_{ij}]_k$, which changes in each *for loop* iteration in step 11. Moreover, computing the difference, $p_i - p_j$ requires some communication which is implemented using *MPI_Allgatherv()* operation since each processor k has its own $[p_i]_k$. After this, each process k computes corresponding conductivity matrix $[D_{ij}]_k$ using the previous one and the flux (using Equation 2.7) in step 22. The iterations continue until the frobenius norm of the change of the flux values between two consecutive iterations is smaller than a threshold which is set to 0.1. We point out that the result of each iteration is based on the previous iteration and fed into the next one. We also note that the proposed algorithm is parallel and does not require any communication except for steps 19 and 21.

5.1.1 Obtaining the dynamic graph

In this part, we provide some details on how the graph is dynamically changing in step 11 of Algorithm 17. The following parameters, which are variables of the *for loop*, indicates how many edges will be updated and in which ratio the edge weights are to be increased and/or decreased. We also note that edges whose weights will be changed are selected randomly.

- **Percentage of edges changed (pce)** presents the percentage of the edges changed in the graph. In our experiments, the *pce* is between 0 and 0.6. For instance, if the *pce* is 0.4 and the graph consists of 1000 edges, 400 edges are changed in the graph. That is, 40 percentage of the edges are changed.
- **Percentage of changed weight (pcw)** presents percentages of edges whose weights are decreased or increased compared to the original edge weight. While the *pcw* changes from 0 to 6 in the increasing case, it changes from 0 to 0.6 in the decreasing case. For instance, if the *pcw* is 4 and edge weight is 1000, new weight of this edge will be 5000. If the *pcw* is 0.4, new weight of this edge will be 600.

Our graphs are dynamically changing in each *for loop* iteration. In Algorithm 17, the *for loop* is iterated four times and in each iteration, dynamic graphs are obtained by changing the *pcw* or the *pce* parameters by increasing and/or decreasing edge weights. We use the same scheme for the changes as in the experiments in [86]. There are five cases and the graph changes for four times in each case:

1. **increase1:** The percentage of number of edges weight changes are set by *pce* parameter to 0, 0.2, 0.4, and 0.6, and the *pcw* is set to 1, which means that in the first iteration (the *pce*=0), the original graph is used. 20, 40, and 60 percentages of the total edges are selected and their edge weights are increased by adding its own edge weight.

2. **increase2:** The pce is a constant for each iteration and set to 0.2, but the edge weights are increased at different ratios by changing *pcw* from 0 to 6, which means that in the first iteration (the *pcw*=0), the original edge weights are used. The selected edge weights are increased by a factor of 2, 4, and 6 for second, third, and fourth iterations, respectively.

3. **decrease1:** The percentage of number of edges weight changes are set by *pce* parameter to 0, 0.2, 0.4, and 0.6, and the *pcw* is fixed at 0.2 for each iteration, meaning that in the first iteration (the *pce*=0), the original graph is used. 20, 40, and 60 percentages of the total edges are selected and their edge weights are decreased. For each case, the edge weight is decreased at 20 percent.

4. **decrease2:** The *pce* is constant at each iteration and set to 0.2, but the edge weights are decreased at different percentages by setting the *pcw* to 0, 0.2, 0.4, and 0.6, respectively. That is, in each iteration, 20 percentage of the total edges is selected and their edge weights are decreased by 20, 40, and 60 percent, respectively.

5. **mix:** The last case is the mix case. For each iteration, while some of edge weights are increased, some are decreased. In this case, the *pce* is set to 0, 0.2, 0.4, and 0.6, respectively and the *pcw* is 2 for increasing case and 0.2 for decreasing case. For instance, in the second iteration, the *pce* is 0.2, which means that while 10 percent of the total edges are increased, 10 percent of the total edges are decreased.

CHAPTER 6

HYBRID METHOD

In this chapter, we propose a novel hybrid method that first sparsifies a given graph by removing most edges which can not form the shortest path tree and then applies a classical shortest path algorithm on the sparser graph. Depending on the problem, the classical shortest path algorithm might be used for solving the single source shortest path or the source to target shortest path problems. In this section, we solve the single source shortest path problem. Removing all the edges that can not form the shortest path tree would be expensive since it is equivalent to solving the original problem. Therefore, we propose to use an iterative bio-inspired algorithm, namely the *Physarum* algorithm as the first stage to sparsify the graph. We prove that the resulting sparser graph always contains the shortest path tree of the original graph. Next, any shortest path algorithm can be used to find the single source shortest path on the resulting graph. The proposed method is, therefore, a two stage hybrid algorithm and it computes the single source shortest path exactly. Next, we give the details of the proposed hybrid method for solving single source shortest path problems.

6.1 Proposed Hybrid Method

When the number of the vertices and edges in a graph is larger, classical shortest path algorithms require excessive computational time to find the shortest path even though those edges may not be a part of the shortest path. For instance, a graph may include many cycles and at least one edge on a cycle can not be a part of the shortest path. Additionally, if the graph is complete, at least half of the edges can not be a part

of the single source shortest path tree. If those edges are removed from the graph, the computational time of any single source shortest path algorithm could be reduced significantly without compromising its accuracy.

In our proposed algorithm, after most edges which can not be a part of the single source shortest path are determined by using *Physarum* algorithm, those edges are removed from the graph and sparser graph is obtained. Next, Dijkstra's or breath first search (BFS) are used to find the single source shortest path on the sparser graph if the graph is unweighted or weighted, respectively. Thus, Dijkstra's or BFS leads to a significant reduction in computational time since search space of the algorithm is significantly reduced. Next, we prove a crucial theorem that shows in *Physarum* algorithm how eliminating edges that have fluxes less than zero does not disturb the shortest path tree. First we need to prove a few lemmas, leading to the proof of our theorem.

Lemma 6.1.1. *Any edge L_{ij} with negative Q_{ij} value can not be one of the edges forming the shortest path.*

Proof. Mathematical model of Physarum [78] based on Poiseuille flow and Krichooff's laws is represented by an electrical circuit system [89]. In this representation, the electrical current which is passing from node i to node j equals to the flux on the edge L_{ij}. The potential difference between node i and node j corresponds to the difference of the pressures $(p_i - p_j)$. Therefore, a mapping from Physarum network to an electrical circuit is constructed by Equation 6.1, 6.2, and 6.3

$$I_{ij} = Q_{ij} \tag{6.1}$$

$$V_{ij} = p_i - p_j \tag{6.2}$$

$$R_{ij} = \frac{L_{ij}}{D_{ij}} \tag{6.3}$$

where I_{ij} is the electrical current, V_{ij} is the potential, and R_{ij} is the resistance between vertex i and vertex j. In other words,

$$I_{ij} = \frac{V_{ij}}{R_{ij}} = \frac{D_{ij}}{L_{ij}}(p_i - p_j) = Q_{ij}. \tag{6.4}$$

Therefore, Equation 6.4 is a model to the flows thorough the tubes in the Physarum network (see Equation 2.1). We note that the flow direction is consistent with the edge orientation.

If the potential difference between node i and node j is positive, the electrical current flows from node i to node j by Ohm's law. Now we apply this information in the Physarum network. If Q_{ij} of an edge L_{ij} is negative, it means that the flow moves from node j to node i, which is the opposite of edge orientation. Thus, this edge can not be on the shortest path. In conclusion, the edge with negative Q value can not be one of the edges forming the shortest path from s to t since the orientation of the edge is not consistent with the flow direction. □

Lemma 6.1.2. *Let G be a graph with positive edges. There is at least one edge L_{ij} on a cycle in G whose Q_{ij} is negative.*

Proof. We assume that G includes a cycle whose length is k (i.e. there are k vertices on the cycle, $k \leq n$), v_x is the starting vertex of the cycle, and v_{x+k-1} is the ending vertex of the cycle. Moreover, we assume that all Q values of the edges on the cycle are positive. Then, Equation 6.5 holds

$$p_x > p_{x+1} \ldots > p_{x+k-1} > p_x. \tag{6.5}$$

This is a contradiction. Therefore, there is at least one edge L_{ij} on the cycle whose Q_{ij} is negative. □

Theorem 6.1.1. *Let G be a connected graph. If we remove the edges L_{ij} whose Q_{ij} values are less than zero and call the resulting graph as G_s, then G_s is a tree and the shortest path tree of the original graph G is a subtree of G_s.*

Proof. First, we prove that G_s is a tree. In order to prove this, we will show that G_s does not contain any cycle and is connected. First, we will show that G_s does not contain any cycle. If all edges whose Q values are negative are removed from G and in a cycle there is at least one edge with negative Q value by Lemma 6.1.2, at least one edge on a cycle is removed. Thus, the cycle is eliminated. Second, by Lemma 6.1.1, edges with negative Q values can not be on the shortest path. Therefore, removing such edges does not affect the connectivity of the graph since we only remove the

edges which can not be on the shortest path. As a result, G_s is a tree. Finally, since G_s includes all edges which may form the shortest path tree, the shortest path tree is a sub tree of G_s. □

Now, we give the pseudocode of the hybrid algorithm in Algorithm 18. The input of the algorithm is a graph $G = (V, E)$ where V and E are the set of vertices and edges, respectively. L is the corresponding adjecency matrix. In Line 3, the diagonal entries of L (i.e. self loops) are removed directly since the shortest path can not include the same node. Then, we apply Physarum algorithm in order to find the flux values (Q matrix Lines 8-13). This is called as *Stage1* of the hybrid algorithm. In Physarum model, Equation 6.6, which is derived from Equation 2.10 and Equation 2.13, is used since our goal is to compute the shortest path from a source vertex to the other vertices. In our algorithm, the *while* loop is iterated only once in order to determine direction of the flux. For determining more edges not forming the single source shortest path, the while loop could be iterated further. The algorithm requires to find the solution of linear systems in Line 10 and they are iteratively solved by using a moderate stopping tolerance, which is the residual norm relative to the norm of the right hand side vector and is fixed at 10^{-3}. After Q matrix is computed, the edges whose Q values are less than ϵ (machine precision) are removed (Lines 15-17) since they are practically zero. Therefore, the resulting graph G_s is simple and sparse and by Theorem 6.1.1, G_s includes all edges which may form the shortest path. Therefore, finding the single source shortest path on G corresponds the single source shortest path on G_s. Finally, BFS efficiently computes the shortest path using the simple and sparse graph, which is called as *Stage2* of the hybrid algorithm. To summarize, we present the flow of the algorithm in Figure 6.1.

To illustrate how the proposed method works, we give a small example. The example graph G is given in Figure 6.2. There are 6 vertices (q_1, q_2, q_3, q_4, q_5, and q_6) and 19 edges in this graph. We assume that q_1 is the source vertex and all edge weights are 1. In order to remove unnecessary edges, Physarum approach is applied. Therefore, after the edges which can not form to the shortest path tree are removed by following the steps in Algorithm 18, the resulting sparse graph, G_s, is obtained. In Figure 6.3, G_s is shown by all edges with solid and dashed lines and the shortest path tree is

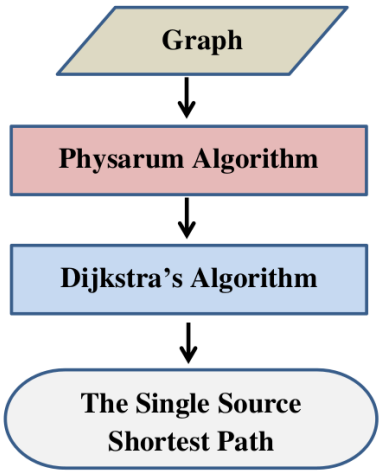


Figure 6.1: Flow of the hybrid algorithm, note that Dijkstra's algorithm is replaced with BFS if the graph is unweighted

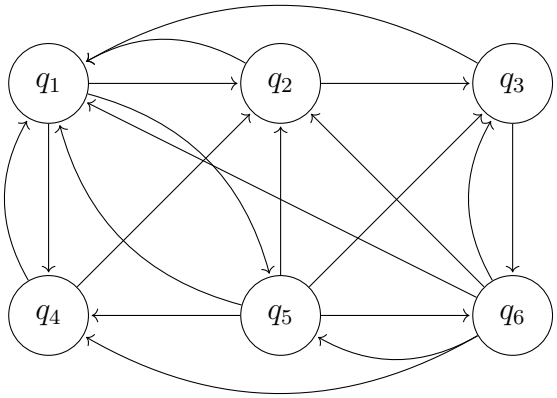


Figure 6.2: An example directed graph G

shown by only edges with solid lines. That is, the dashed edges belong to G_s, but not in the shortest path tree. To sum up, there are eight edges in G_s and only three of them are not included in the shortest path tree. Finally, the single source shortest path is computed by Dijkstra's or BFS algorithms using G_s graph. If we choose the vertex q_6 as a target vertex, the shortest path from q_1 to q_6 is computed as $s = q_1 - q_5 - q_6 = t$. In the next section, we show the performance of the proposed method on larger graphs.

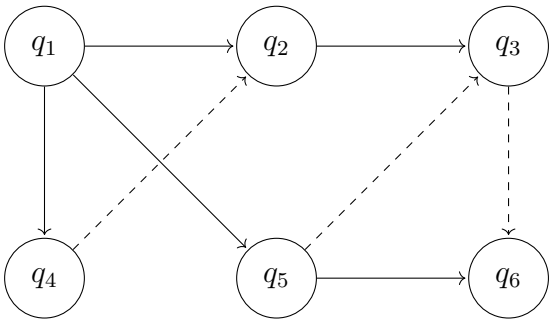


Figure 6.3: The resulting graph G_s (solid and dashed edges) where the solid edges indicate the shortest path tree

Algorithm 18 Hybrid Shortest Path Algorithm

Data: $G = (V, E)$ where V and E are the set of vertices and edges respectively of the graph (which is represented as an adjacency matrix L) and s is the source vertex

Result: the single source shortest path

$L^s = L - diag(L)$

//*Physarum* Algorithm

$D_{ij} \leftarrow 1$ where $L^s_{ij} \neq 0$

$Q \leftarrow 0$

$p \leftarrow 0$

while a termination condition is not met **do**

    $p_t \leftarrow 0$

    Computing the pressures by Equation 6.6

$$\sum_i (\frac{D_{ij}}{L^s_{ij}} + \frac{D_{ji}}{L^s_{ji}})(p_i - p_j) = \begin{cases} n-1 & \text{if } j = source \\ -1 & otherwise \end{cases} \tag{6.6}$$

    $Q_{ij} \leftarrow \frac{D_{ij}}{L_{ij}} \times (p_i - p_j)$

    $D \leftarrow (Q + D)/2$

end while

//Removing edges not forming the shortest path

if $Q_{ij} < \epsilon$ **then**

    $L^s_{ij} = 0$

end if

//*Dijkstra's* algorithm using G_s

[dist,path]=***graphshortestpath***(G_s, s) where $G_s = (V, E_s)$ represented by L^s

CHAPTER 7

EXPERIMENTAL RESULTS

The numerical results are obtained by using the High Performance and Grid Computing Center of the Scientific and Technological Research Council of Turkey. The cluster that we are using consists of 128 nodes connected via Mellanox Connect$\times$3 FDR Infiniband interconnection. Each node contains 2 $\times$ Xeon E5-2680v3 2.50GHz (28 cores in total) processors and 256GB memory in total. We have access to at most 96 cores.

In our experiments, we have used Δ-stepping implementation in Parallel Boost Graph Library (Parallel BGL) [41], version 1.63.0, which is a library that contains parallel and efficient implementations of graph algorithms for large and sparse graphs which are stored by using distributed adjacency list such that each processor has its local vertices and all edges which are directed from those vertices. Parallel BGL uses the Message Passing Interface (MPI) library for interprocess communication. Δ-stepping (in Parallel BGL) is a single source shortest path algorithm and computes the shortest path between the source node to all other nodes. There are no efficient MPI implementation of source to target shortest path in Boost library or in any other library that we know. Hence, we simply modify Δ-stepping algorithm in parallel BGL to terminate when no paths shorter than the shortest path from s-t existed by permitting to make early-termination for the s-t shortest paths (see steps 22-24 in Algorithm 4). We use the following function:

delta_stepping_shortest_paths
(const Graph& g,
typename graph_traits<Graph>::vertex_descriptor **s**,
typename graph_traits<Graph>::vertex_descriptor **t**,
PredecessorMap predecessor, DistanceMap distance, WeightMap weight)

Portable, Extensible Toolkit for Scientific Computation (PETSc) [3] by Argonne National Laboratory, version 3.7.5, is used for implementing PPA and dynPPA. PETSc is a high-performance linear algebra library and provides optimized parallel sparse matrix and vector routines including efficient implementations of iterative solvers such as Krylov subspace methods. It uses double precision arithmetic and also using MPI for interprocessor communication in C language.

In our implementation, the graphs are represented and stored as sparse matrices. In Algorithm 16 and 17, the MATMPIAIJ format of PETSc is used to store A matrix, and b vector is stored by using PETSc vector storage format which is quite suitable and efficient for PETSc linear solvers and preconditioners. To achieve good performance during the matrix assembly, it is crucial to preallocate the memory needed for the sparse matrices [3]. The other matrices (D, Q, and T) in Algorithm 16 and 17 are stored in coordinate format because of the fact that the assembly operation in coordinate format is cheaper. Details of these storage formats are explain below. All matrices and vectors in the algorithm have the same sparsity structures and they are conformably partitioned by block rows assigned to processors. For instance, the matrix A can be partitioned as in the following

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_t \end{pmatrix}$$

where t is the total number of processors and each process k owns the block row $A_k = [A_{ij}]_k$. Each block row has a dimension of $m_k \times n$ where

$$n = \sum_{k=1}^{t} m_k$$

n is the number of vertices of the graph.

7.1 Results of PPA

First, we give information about the dataset. As the basis for comparison, we use 12 undirected graphs in two groups: (1) Real-world graphs obtained from the 9^{th} DIMACS Implementation Challenge [1] and (2) Synthetic graphs. We used several different kinds of synthetic graphs: R-MAT, Erdos-Renyi, and small-world, generated by ParMAT [46], $erdos.renyi.game$ function [2] and $watts.strogatz.game$ function [4] of igraph package in R language, respectively. These different graph sets exhibit different properties. While Erdos-Renyi graphs violate power laws (degree distribution of Erdos-Renyi graphs converge to the Poisson distribution), R-MAT model fits a unimodal and power law graphs [20]. In addition, R-MAT graphs are realistic resembling social networks. Strogatz small-world networks have relatively smaller average shortest path lengths, and clustering property. On the other hand, real-world graphs have higher diameter and lower degree than the synthetic graphs.

We generated various size undirected synthetic graphs whose number of vertices range from 1 to 20 Millions and the number of edges are between 100 and 200 Millions. In R-MAT, the graph is recursively subdivided into four equal-sized partitions. The distributed edges within these partitions have the probabilities which are $a = 0.4$, $b = 0.1$, $c = 0.1$, and $d = 0.4$. The *watts.strogatz.game* function also includes some parameters which are rewiring probability ($p = 0.5$) which an edge is rewired. This means that the edge is disconnected from one of its vertices and then randomly connected to another vertices anywhere in the graph. Moreover, the neighborhood constant (nei) is set to 5. While we use distances for edge weights in the real-world graphs, we use unit weights in synthetic graphs. The properties of the graphs are given in Table 7.7.

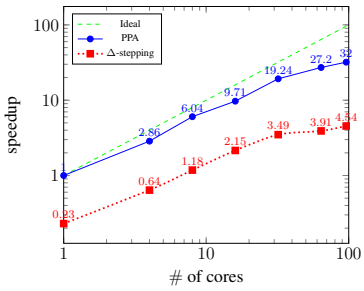
We evaluate our results based on the time to solution, parallel scalability as well as accuracy, and compare to the baseline algorithm namely, Δ-stepping algorithm which is considered to be the best performing variant of the Dijkstra's algorithm in Parallel BGL [31, 41]. Note that in Dijkstra's algorithm, we stop when the shortest path to the target is found. The comparisons are performed by using several kinds of real-world graphs for road networks of the USA and synthetic graphs.

Table 7.1: Graph Properties

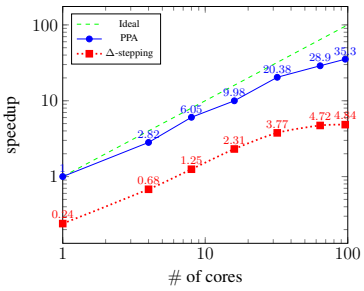
Graph Name	**Vertices**	**Edges**	**Kind**
Erdos1	1M	100M	Erdos-Renyi
Erdos2	1M	120M	Erdos-Renyi
Erdos3	1M	140M	Erdos-Renyi
Rmat1	1M	100M	R-MAT
Rmat2	1M	120M	R-MAT
Rmat3	1M	140M	R-MAT
Smallworld1	10M	100M	Small-world
Smallworld2	15M	150M	Small-world
Smallworld3	20M	200M	Small-world
USA CTR	$\sim$14M	$\sim$34M	Real-world
USA W	$\sim$6M	$\sim$15M	Real-world
USA E	$\sim$3.5M	$\sim$9M	Real-world

PPA requires some parameters. First one is the maximum number of outer iterations to break out the while loop which we set to be three in our experiments. The iteration number may be increased if more accuracy is desired, but for dataset we found this to be sufficient. CG method with GS preconditioner is used to solve the linear system. In order to terminate the inner iterations(CG iterations), combination of setting a limit on the maximum number of iterations and a stopping tolerance are used. In our experiments, the stopping tolerance, which is the residual norm relative to the norm of the right hand side, is called as the inner tolerance and is fixed at 0.01. Moreover, the maximum number of iterations is set to be 10^4.

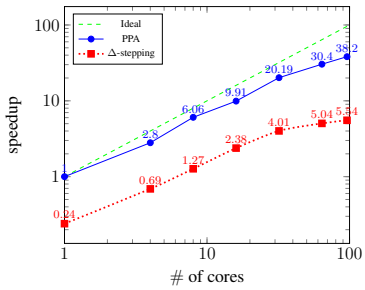
The baseline algorithm, Δ-stepping, uses the parameter Δ which is set to the maximum edge weight divided by the maximum degree for the synthetic graphs [52] and Δ is set to 400 for the real world graphs [31]. The sequential performances of both PPA and Δ-stepping algorithm are shown in Table 7.2. PPA is 5 times faster than Δ-stepping on average. We believe the sequential performance of PPA is due to the effective preconditioner for M-matrices which requires only 6 iterations to converge for the synthetic graphs and at most 67 iterations for real-world graphs to reach the desired inner tolerance.



(a) Erdos1

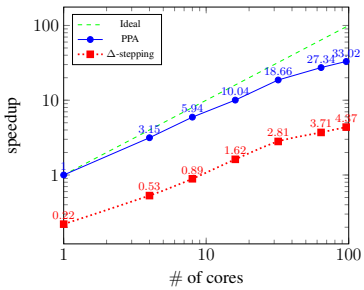


(b) Erdos2

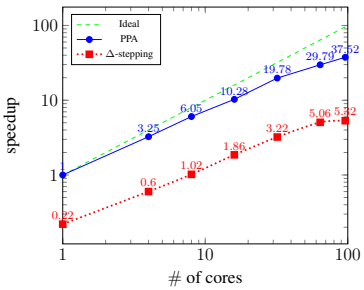


(c) Erdos3

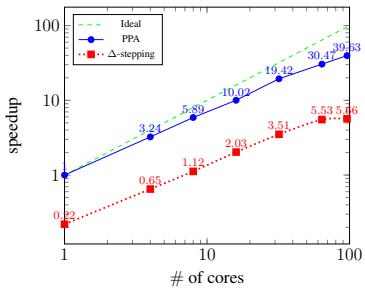
Figure 7.1: Speedup of Erdos-Renyi Graphs



(a) Rmat1

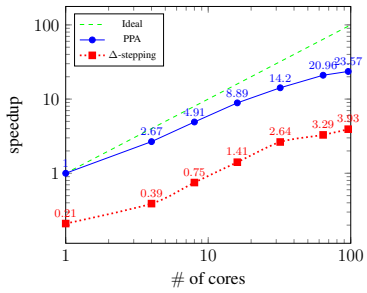


(b) Rmat2

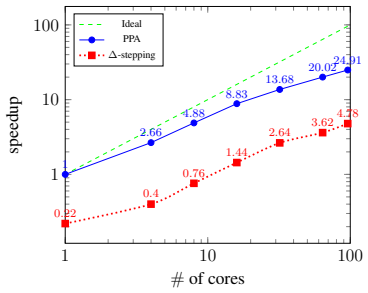


(c) Rmat3

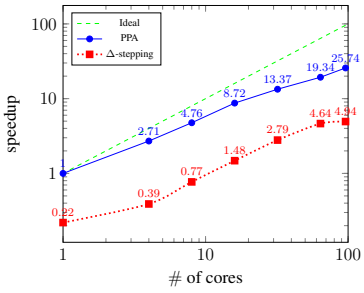
Figure 7.2: Speedup of RMAT Graphs



(a) Smallworld1

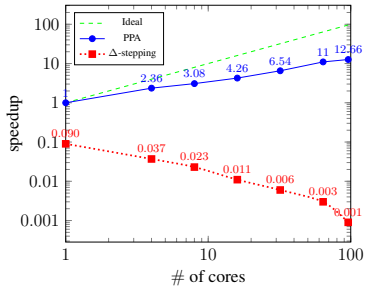


(b) Smallworld2

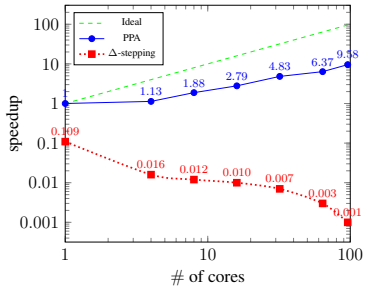


(c) Smallworld3

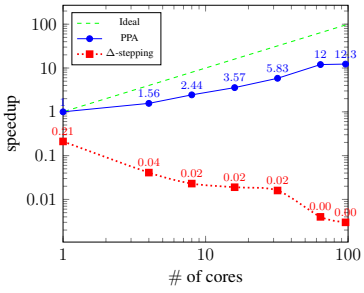
Figure 7.3: Speedup of small-world Graphs



(a) USA CTR



(b) USA E



(c) USA W

Figure 7.4: Speedup of real-world Graphs

Table 7.2: Sequential solution time for both PPA and Δ-stepping in seconds (using Conjugate Gradient linear solver with Gauss-Seidel preconditioner, outer iteration number is three) and the total number of inner iterations for PPA

	PPA		Δ-stepping
Graph	**Time**	**Inner Iter**	**Time**
Erdos1	59.3	6	260.4
Erdos2	74.5	6	308.1
Erdos3	88.1	6	363.4
Rmat1	55.0	6	251.7
Rmat2	68.9	6	309.9
Rmat3	81.2	6	363.3
Smallworld1	60.5	6	288.8
Smallworld2	92.8	6	459.2
Smallworld3	125.9	6	568.9
USA E	2.9	37	28.8
USA W	10.6	67	50.0
USA CTR	10.6	13	115.6

Furthermore, we study the parallel scalability of PPA and Δ-stepping algorithm by using the synthetic graphs and the real world graphs which have low and high diameters, respectively. Figure 7.1, 7.2, 7.3, and 7.4 illustrate the strong scalability of PPA and Δ-stepping algorithms with respect to the best sequential algorithm (in all cases this is the sequential PPA). The synthetic graphs have a relatively more uniform distribution of workloads shared by processors owing to the nature of random networks. Therefore, Erdos-Renyi, R-MAT, and small-world graphs in Figure 7.1, 7.2, and 7.3, respectively, are more scalable than the real world networks in Figure 7.4 for both algorithms. PPA achieves a speedup of 32 and 12 for the synthetic and real world graphs, respectively, using 96 cores (cluster) on average. On the other hand, Δ-stepping algorithm achieves a speedup of 5 for synthetic graphs using 96 cores on average and it is not able to achieve any speedup for real world graphs.

Next, we look into the results in more detail for each graph. Both Erdos-Renyi and RMAT graph sets contain three graphs with the same number of vertices and varying the number of edges. Figure 7.1 and Figure 7.2 depict the effect of the change

in the number of edges on the speedup in which the number of vertices are fixed (see Table 7.7) for all graphs in the set. When the number of edges increases, the speedup also increases due to the fact that there is more work per processsor. In Figure 7.1, the speedups of PPA and Δ-stepping are 32 and 4.5 for Erdos1 as shown in Figure 7.1a, 35.3 and 4.8 for Erdos2 as shown in Figure 7.1b, and 38.2 and 5.5 for Erdos3 as shown in Figure 7.1c, respectively using 96 cores. Similarly, in Figure 7.2 (Rmat1, Rmat2, and Rmat3) while PPA achieves the speedups of 33, 37.5, and 39.6, Δ-stepping achieves 4.4, 5.3, and 5.7, respectively as shown in Figure 7.2a, 7.2b, and 7.2c using 96 cores.

Figure 7.3 shows speedups for small-world graphs where the number of edges and number of vertices increase proportionally (see Table 7.7). As the number of processors increase, a better speedup for the larger problems is also observed. For Smallworld1, Smallworld2, and Smallworld3, the speedups of PPA are 23.6, 24.9, and 25.7; on the other hand, the speedups of Δ-stepping algorithm are 3.9 , 4.8, and 4.9 as shown in Figure 7.3a, 7.3b, and 7.3c, respectively using 96 cores.

In Figure 7.4, speedups for the real world graphs are given. PPA achieves a speedup of 9.6 for USA E, 12.3 for USA W, and 12.7 for USA CTR as shown in Figure 7.4b, 7.4c, and 7.4a, respectively using 96 cores. On the other hand, Δ-stepping does not actually give any speedup for all three problems. The main reason for the poor scalability of Δ-stepping for real world graphs is due to both the structure and the edge weights of the graphs. Edmonds et. al. [31] investigate the real road networks of the USA and they show that the reason for poor parallel scalability is that these graphs contain a large number of edges with large weights on the graph. Traversing such edges in the sparse regions may require a lot of iterations in Δ-stepping algorithm. Moreover, they investigate the effect of graph partitioning for such networks and they show that graph partitioning can not improve the performance. On the other hand, PPA shows a relatively better scalability than Δ-stepping for real-world networks. However, PPA requires more inner iterations for the real-world graphs compared to the synthetic graphs which is also due to the irregular distribution of edge weights.

Finally, we compare the accuracy of PPA and Δ-stepping algorithm. Note that PPA can compute the shortest path accurately provided that the number of outer iterations

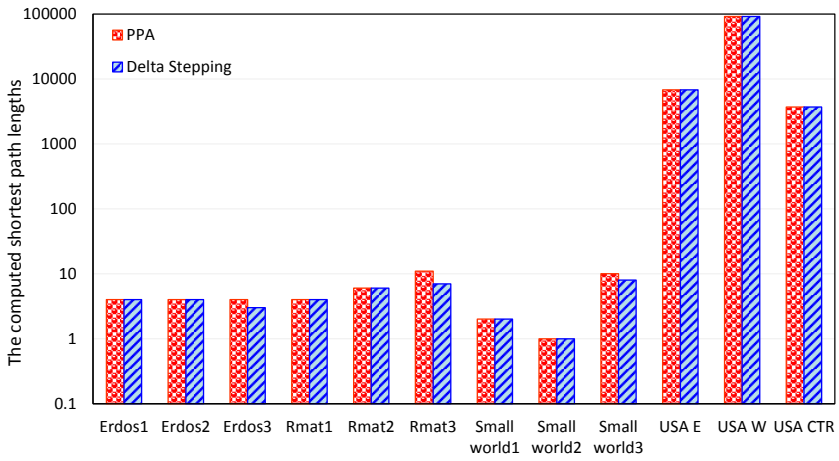


Figure 7.5: The computed shortest path lengths for PPA and Δ-stepping methods

and the accuracy of the linear system solution are set carefully. Figure 7.5 illustrates the computed shortest path length by PPA and Δ-stepping algorithm. PPA and Δ-stepping algorithm compute the same shortest path length for almost all test graphs, except for Erdos3, Rmat3, and Smallworld3 for which the differences are 1, 4, and 2, respectively and still quite accurate.

In conclusion, PPA achieves much better speedup compared to Δ-stepping algorithm for all graphs in the dataset. Furthermore, the speedup of PPA is near-linear for synthetic graphs that have lower diameter while the speedup is still acceptable for real world graphs that have higher diameter com- pared to Δ-stepping. On average, PPA can compute the shortest path roughly eight orders of magnitude faster than Δ-stepping algorithm with comparable accuracy in parallel using 96 cores.

7.2 Results of dynPPA

Our dataset includes various undirected graph models which are Erdos-Renyi, small-world and real-world, which are the-state-of-the-art models as well as they exhibit diverse properties. The dataset consists of four graphs. The properties of those graphs are shown in Table 7.7. The first graph is Erdos which is generated by using *Erdos-Renyi* model ($erdos.renyi.game$ function [2] of igraph library in R language). *Erdos-Renyi* model produces a random graph and each pair of vertices in the graph

Table 7.3: Graph Properties

Graph Name	**Vertices**	**Edges**	**Model**	**Description**
Erdos	1M	100M	Erdos-Renyi	Synthetic
Smallworld	1M	100M	Small-world	Synthetic
NW	∼1.2M	∼2.8M	Real-world	Northwest USA
CAL	∼2M	∼4.6M	Real-world	California and Nevada

is randomly connected with a probability. Degree distribution of *Erdos-Renyi* model is a binomial distribution. When the number of vertices goes to infinity, the degree distribution of Erdos-Renyi model converges to a Poisson distribution and this model violates to generate power-law degree distribution. Therefore, degrees of the vertices are high in such graphs. The second graph is Smallworld proposed by *Watts* and *Strogatz* and generated by using $watts.strogatz.game$ function [4] of igraph package in R language. Small-world graphs have short average path lengths like random graphs, but high clustering coefficient like real-world graphs. Therefore, small-world graphs are between regular and random graphs. Most large scale sparse graphs are found to be the type of the small-world graphs such as internet, neurons and social graphs. The last two graphs are selected from real-world applications, which are CAL and NW. The real-world graphs are obtained from 9^{th} DIMACS Implementation Challenge [1]. They tend to be clustered, so the neighboring vertices are most likely connected. When compared to the synthetic graphs, they have a higher diameter and lower vertex degrees. Moreover, the degree distribution of real world graphs satisfy the power law. While we use distances for edge weights in the real-world graphs, we use unit weights in synthetic graphs, namely Smallworld and Erdos.

7.2.1 Parallel Scalability Results of dynPPA

We evaluate our results based on the time to solution and accuracy, and compare to the parallel implementation of Dijkstra, namely Δ-stepping in parallel BGL. Moreover, we analyze the effect of the percentage of the edges changed, percentage of changed weight, and graph models to the parallel speedup of both dynPPA and Δ-stepping. Furthermore, in the proposed algorithm, previous iteration fully feeds into the next

Table 7.4: Total sequential time (in seconds) for synthetic graphs dynamically changed

	Smallworld		Erdos	
Edge weights	**dynPPA**	**Δ-stepping**	**dynPPA**	**Δ-stepping**
increase1	101.4	947.8	116.9	939.6
increase2	101.6	944.7	116.5	942.8
decrease1	102.5	937.5	117.0	933.4
decrease2	102.0	936.9	117.0	931.7
mix	103.0	968.6	116.1	956.1

Table 7.5: Total sequential time (in seconds) for real-world graphs dynamically changed

	NW		CAL	
Edge weights	**dynPPA**	**Δ-stepping**	**dynPPA**	**Δ-stepping**
increase1	3.9	38.0	60.6	74.1
increase2	3.2	1506	47.0	2846
decrease1	3.5	29.3	48.5	50.4
decrease2	3.4	29.4	54.3	50.4
mix	10.9	100.5	40.8	151.7

one. We discuss the effect of this adaptivity on the speedup.

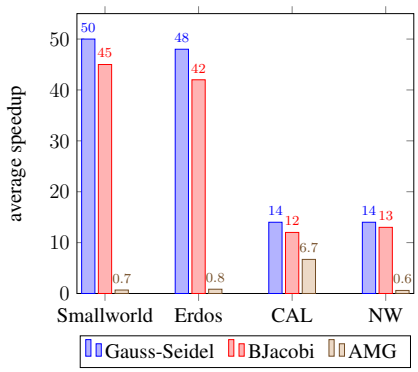
Now, we evaluate the sequential performance of both dynPPA and Δ-stepping shown in Table 7.4 and 7.5, for synthetic graphs and real-world graphs, respectively. First, we evaluate the sequential performance of the synthetic graphs, which are Small-world and Erdos in Table 7.4. We observe that for each method, five cases do not make much difference in terms of the sequential running time, and it is only slightly affected by the change of edge weights for five cases for both graphs in Table 7.4. dynPPA is $9\times$ faster than Δ-stepping in the sequential implementation. Second, we evaluate the sequential performance for the real-world graphs shown in Table 7.5 for the increasing, decreasing and mix cases. While dynPPA is still quite insensitive to the edge weight changes, the edge weight changes affect Δ-stepping dramatically. For *increase2*, Δ-stepping method requires a large amount of time to obtain the solution for both real-world graphs. The reason for this is that sparse regions contain many edges with large weights and if these edges are selected to be increased, so-

Table 7.6: Maximum edge weights while changing CAL graph in each iteration

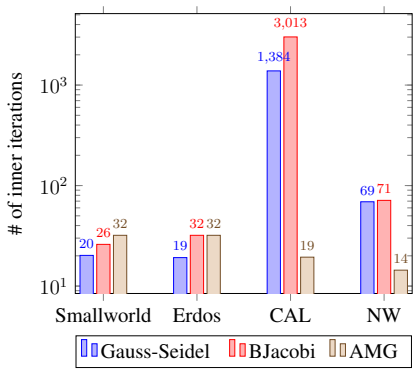
	the graph dynamically changes			
Edge weights	**iter1**	**iter2**	**iter3**	**iter4**
increase1	215,354	490,100	2,389,632	4,835,840
increase2	215,354	1,983,123	100,096,875	487,943,225
decrease1	215,354	215,354	182,492	172,283.2
decrease2	215,354	194,861	182,492	161,736
mix	215,354	1,260,378	4,776,408	15,367,903.2

lution time of Δ-stepping increases sharply since this results in a large number of reinsertions. To support this claim, we list max elements for dynamically changing CAL graph in Table 7.6. Note that Table 7.6 shows the largest elements for each iteration separately (the graph changes four times). In *increase2* case, the largest elements at the sparse regions are larger with increasing iteration count. This results in an increase in the solution time of Δ-stepping dramatically especially 3^{rd} and 4^{th} iterations. As a result, sum of the sequential running times for *increase2* case increases sharply. Secondly, when we study the results of decreasing cases (the *decrease1* and the *decrease2*), dynPPA is $8\times$ faster than Δ-stepping algorithm for NW graph. On the other hand, both methods require almost the same amount of time for CAL graph. The running time of Δ-stepping is reduced by decreasing large edge weights at sparse regions when compared to the increasing cases since the edge weights in the decreasing cases become more uniform. Lastly, for the *mix case*, dynPPA is affected by the edge weight changes and the sequential running time of dynPPA increases (especially fourth iteration in the *for loop*) since the change of the edge weights in the *mix* case between two consecutive iterations is higher than the other cases, so the number of inner iterations is higher and solving the linear systems consumes relatively more time. As a result, dynPPA is $10\times$ and $3.7\times$ faster than Δ-stepping for NW and CAL, respectively for the *mix* case. We also note that the sequential running time of dynPPA and Δ-stepping for the synthetic graphs are much higher than for the real-world graphs since the number of nonzeros in the synthetic graphs is much higher than the number of nonzeros in the real-world graphs.

Now, we give a comparison of the state-of-the-art preconditioners for dynPPA. The most critical step of the algorithm is to solve the linear systems. In this step, using



(a) The average speedups using 96 cores

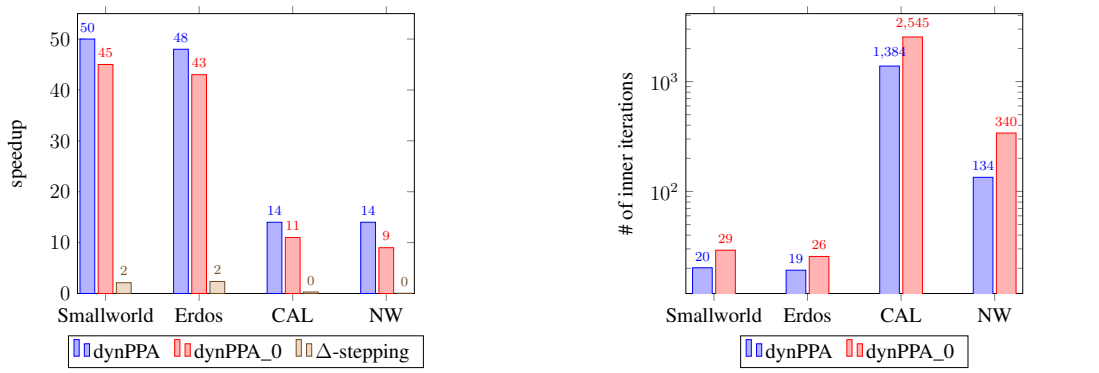


(b) Average of the total number of inner iterations

Figure 7.6: Comparison of the state-of-the-art implementation of the preconditioners

optimum linear solver and preconditioner is crucial based on the properties of the coefficient matrix. Figure 7.6 illustrates the comparison of Block Jacobi, AMG, and Gauss-Seidel preconditioners. Although we also used SAI preconditioner to solve the linear systems, it reaches the maximum number of iterations without convergence. In this study, the speedup is computed by dividing the best total sequential running time to total parallel running time using 96 cores. The average speedups in Figure 7.6a are computed by taking average of the speedups of the five cases (*increase1, increase2, decrease1, decrease2*, and *mix*) for each graph. Although the number of inner iterations is the lowest for the real-world graphs when using AMG preconditioner, the speedup achieved by AMG is poor. The reason for this is that the operations related to AMG preconditioner are costlier. Although both the parallel speedup and the number of inner iterations are relatively close to each other for BJacobi and Gauss-Seidel, dynPPA achieves the best speedups for all test graphs when Gauss-Seidel preconditioner is used, and it finds the solution in a fewer number of inner iterations in Figure 7.6b. Therefore, in the following experiments we use the Gauss-Seidel preconditioner in dynPPA. We have also experimented with recycling the Krylov supspace using RCG, however, it does not give any improvements since the change of the coefficient matrix of the linear systems among iterations is not small enough. In fact, the frobenius norm of the coefficient matrix approximately falls in half between two iterations. Therefore, RCG is not used in the following experiments.

Next, we discuss effect of the initial guess and adaptivity of the algorithm to the parallel speedup. In dynPPA, the values that are computed in the previous iteration feed



(a) Average speedups using 96 cores

(b) Average of the total number of inner iterations

Figure 7.7: The effect of initial guess and adaptivity of the algorithm to the parallel speedup in Algorithm 1

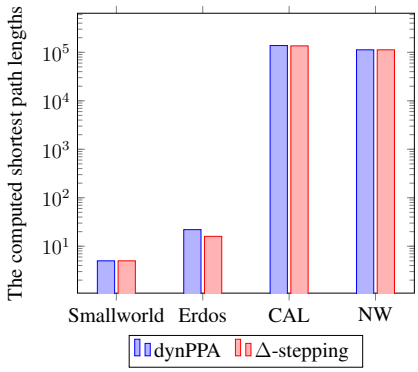
into the next one. That is, the conductivity values, flux values, and the solution of the linear systems are passed into the next *for loop* iteration. This emphasizes adaptivity of dynPPA in dynamically changing graph. dynPPA algorithm differentiates the affected edges and recomputes them spontaneously. Therefore, the operations containing unaffected edges remain the same and do not incur any additional cost. To reduce the number of iterations, hence reducing the parallel and sequential running time, the initial guess in the linear solver is improved in dynPPA. Figure 7.7 illustrates the effect of initial guess and adaptivity of the algorithm to the parallel speedup in Algorithm 17. Moreover, the speedup of Δ-stepping is also shown in Figure 7.7. If initial guesses are taken to be zero for all iterations, the resulting algorithm is called as dynPPA_0. That is, there is no transferring the information from previous iteration to the next one. As seen in Figure 7.7a, the average speedup of dynPPA is higher than both dynPPA_0 and Δ-stepping, and dynPPA finds to solution in a fewer number of inner iterations in Figure 7.7b. Furthermore, the speedup of Δ-stepping is too low especially for real-world graphs. The reasons for this will be explained later.

Now, we look into the results in terms of accuracy. Accuracy of dynPPA depends on both the number of iterations of *while loop* and the iterative solver to find the solution, and it has no dependence on whether the code is run in parallel or sequential. The number of the *while loop* is three and the number of inner iterations is shown in Figure 7.7b. Figure 7.8 illustrates the computed shortest path lengths by dynPPA and Δ-stepping for five cases, separately. Δ-stepping and dynPPA compute nearly the

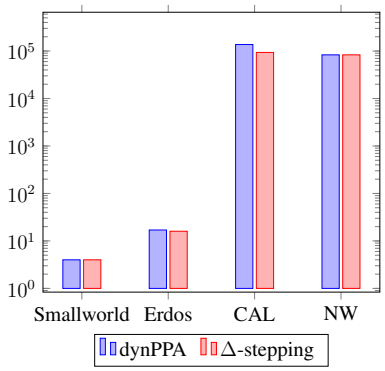
same shortest path lengths.

Next, we evaluate the effect of the percentage of the edges changed, percentage of changed weight, and graph models to the parallel speedup in detail. As in the sequential case, the results are again investigated into five cases, separately. The speedup is computed by dividing the best total sequential running time to total parallel running time.

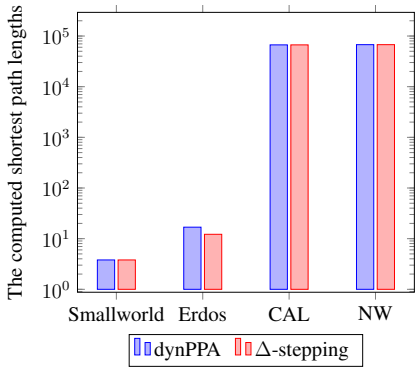
First, we evaluate results of the *increase1* case. In this case, while *pce* changes from 0 to 0.6 through *for loop* iterations, the *pcw* is constant and set to 1. Figure 7.9a, 7.9b, 7.9c, and 7.9d present the speedup of dynPPA and Δ-stepping for Smallworld, Erdos, CAL, and NW graphs, respectively. Synthetic graphs, which are Smallworld and Erdos show near linear speedup using dynPPA. While dynPPA achieves a speedup of 52.3 and 49.2, Δ-stepping achieves a speedup of 3.4 and 4.6 for Smallworld and Erdos, respectively. In other words, the speedup achieved by dynPPA is $17\times$ and $12\times$ higher than the speedup achieved by Δ-stepping for Smallworld and Erdos, respectively. On the other hand, while dynPPA achieves a speedup of 14.1 for both CAL and NW, the speedup of Δ-stepping is almost constant and smaller than 1. Next, we study the results by fixing the number of cores to 96 and varying *pce* from 0 to 0.6. In Figure 7.10a, 7.10b, 7.10c, and 7.10d, we present the results of the parallel running time for Smallworld, Erdos, CAL, and NW graphs, respectively. The parallel running time of dynPPA is almost constant. We observe that the change of the *pce* does not affect the parallel solution of the linear systems much (i.e. the number of inner iterations and the cost per iteration do not change). In fact, for CAL graph, the parallel running time decreases with improving the initial guess vector and adaptivity of dynPPA although the number of edges changed increases. On the other hand, for Smallworld graph, Δ-stepping method is sensitive to the change of the edge weights and the parallel running time increases sharply at *pce*=0.6 since at the last *for loop* iteration, edge weights of Smallworld graph are less uniform, so the graph has less uniform workloads assigned to processors. For other graphs, parallel running time of Δ-stepping among iterations is almost same. However, the parallel running time of Δ-stepping for real-world graphs is much higher than the synthetic graphs. To sum up, while dynPPA is not sensitive to the change of the parameter pce for all test graphs, Δ-stepping is slightly sensitive.



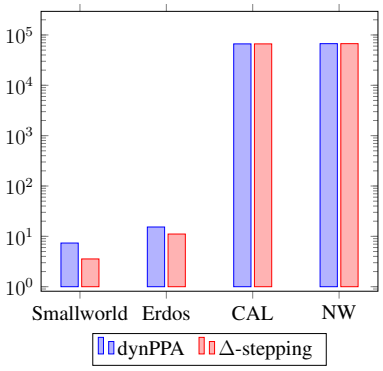
(a) *increase1*



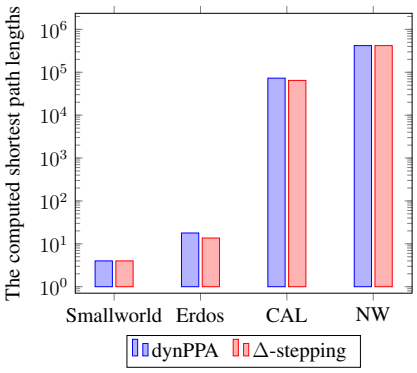
(b) *increase2*



(c) *decrease1*



(d) *decrease2*



(e) *mix*

Figure 7.8: The computed shortest path distances for both Δ-stepping and dynPPA

Second, we evaluate the results of *increase2* case. In this case, *pce* is a constant and set to 0.2. That is, 20 percent of total edges is randomly selected and their edge weights are increased with the *pcw* parameter. The *pcw* changes from 0 to 6. That is, the randomly selected edge weights will be increased by six times at the end of fourth iteration of the *for loop*. Figure 7.11a, 7.11b, 7.11c, and 7.11d present the speedup of dynPPA and Δ-stepping for Smallworld, Erdos, CAL, and NW graphs, respectively as the number of cores changes from 1 to 96. While dynPPA achieves a speedup of 50.6 and 49.4, Δ-stepping achieves a speedup of 2.7 and 2.2 for Smallworld and Erdos, respectively. In other words, the speedup of dynPPA is $17\times$ and $22\times$ higher than the speedup of Δ-stepping for Smallworld and Erdos, respectively. On the other hand, while dynPPA achieves a speedup of 14 and 11 for CAL and NW, respectively, the speedup of Δ-stepping for such graphs is smaller than 1. The speedup achieved by dynPPA increases when the number of cores increases for all graphs. However, the speedup achieved by Δ-stepping is almost constant for real-world graphs and increases for the synthetic graphs as the number of cores increases. Next, we study the results by fixing the number of cores to 96 and varying *pcw* from 0 to 6. Figure 7.12a, 7.12b, 7.12c, and 7.12d present results of the parallel running time for Smallworld, Erdos, CAL, and NW, respectively. The parallel running time of dynPPA is almost constant for all test graphs. We observe that the changes of the *pcw* do not affect the parallel running of the linear systems much in dynPPA. Therefore, dynPPA finds to solution quickly thanks to its adaptivity as well as the effect of improving the initial guess especially for CAL graph. On the other hand, parallel running time of Δ-stepping increases dramatically for all graphs, especially for real-world graphs as the *pcw* increases. This shows that the *pcw* parameter has a great effect on parallel running time of Δ-stepping method. We also point out that the parallel running time of Δ-stepping for the real-world graphs is much higher than for synthetic graphs.

Third, we evaluate results of the *decrease1* case. In this case, we see the effect of the parameter pce to the parallel speedup and running time. While *pce* changes from 0 to 0.6, *pcw* is set to 0.2. Figure 7.13a, 7.13b, 7.13c, and 7.13d present the speedup of dynPPA and Δ-stepping for Smallworld, Erdos, CAL, and NW graphs, respectively. dynPPA achieves a speedup of 46.3 and 50.9 for Smallworld and Erdos, respectively

although Δ-stepping achieves a speedup of 1.5 for such graphs on average. In other words, the speedup achieved by dynPPA is $30\times$ and $35\times$ higher than the speedup achieved by Δ-stepping for Smallworld and Erdos, respectively. On the other hand, dynPPA achieves a speedup of 13 and 12 for CAL and NW graphs, respectively while Δ-stepping has poor scalability for such graphs. The speedup achieved by dynPPA increases when the number of cores increases for all graphs. However, the speedup achieved by Δ-stepping is almost constant for real-world graphs and generally increases for the synthetic graphs as the number of cores increases. We also point out that when the number of cores is 96, the speedup achieved by Δ-stepping decreases because of the communication overhead. Next, we study the results by fixing the number of cores to 96 and varying *pce* from 0 to 0.6. In Figure 7.14a, 7.14b, 7.14c, and 7.14d, we present parallel running time of dynPPA and Δ-stepping for Smallworld, Erdos, CAL, and NW graphs respectively. For all graphs, the parallel running time of dynPPA is almost constant since the change of the *pce* does not affect the parallel running time of the linear systems much. On the other hand, while the parallel running time of Δ-stepping increases as the pce increases for the synthetic graphs and is almost constant for the real-world graphs. To sum up, dynPPA is not sensitive to the change of the parameter pce, Δ-stepping is sensitive especially for the synthetic graphs. Compared to the *increase1* and *increase2* cases, the parallel running time of Δ-stepping for real-world graphs is significantly reduced in this case.

Fourth, we evaluate results of the *decrease2* case to see the effect of *pcw* parameter to the parallel speedup and running time. In this case, while the *pcw* changes from 0 to 0.6, the *pce* is fixed at 0.2. Figure 7.15a, 7.15b, 7.15c, and 7.15d present the speedup of dynPPA and Δ-stepping for Smallworld, Erdos, CAL, and NW graphs, respectively. dynPPA achieves a speedup of 52.6 and 48.6 although Δ-stepping achieves a speedup of 1.2 and 0.95 for Smallworld and Erdos, respectively. In other words, the speedup achieved by dynPPA is $43\times$ and $48\times$ higher than the speedups achieved by Δ-stepping for Smallworld and Erdos, respectively. On the other hand, dynPPA achieves 14 and 11 speedups for CAL and NW graphs while Δ-stepping has poor scalability for such graphs. The speedup achieved by dynPPA always increases when the number of cores increases for all graphs. However, the speedup achieved by Δ-stepping is almost constant for real-world graphs and generally increases for the

synthetic graphs as the number of cores increases. We also point out that when the number of cores is 96, the speedup achieved by Δ-stepping decreases because of the communication overhead. Next, we study the results by fixing the number of cores to 96 and varying *pcw* from 0 to 0.6. In Figure 7.16a, 7.16b, 7.16c, and 7.16d, we present parallel running time of dynPPA and Δ-stepping for Smallworld, Erdos, CAL, and NW graphs respectively. For all graphs, the parallel running time of dynPPA is almost constant and lower than the parallel running time of Δ-stepping. For the synthetic graphs, similar to the previous case, the parallel running time of Δ-stepping increases as pce increases and is almost constant for real-world graphs.

Finally, we evaluate the results of the *mix* case. In this case, edge weights are both decreased and increased at the same percentage. While the parameter *pce* changes from 0 to 0.6, the *pcw* is 2 and 0.2 for increasing and decreasing cases, respectively. For instance, if there are 1000 edges whose weights will be updated, 500 of them will be increased and 500 of them will be decreased. If the edge weight is 100 and it is decreased, it will be 80. Otherwise, it is increased, the new edge weight will be 300. Figure 7.17a, 7.17b, 7.17c, and 7.17d present the speedup results of dynPPA and Δ-stepping for Smallworld, Erdos, CAL, and NW graphs, respectively as the number of cores changes from 1 to 96. dynPPA achieves a speedup of 50.4 and 45 although Δ-stepping achieves a speedup of 1.8 and 2.6 for Smallworld and Erdos, respectively. In other words, the speedup achieved by dynPPA is $27\times$ and $17\times$ higher than the speedup achieved by Δ-stepping for Smallworld and Erdos, respectively. The speedup achieved by dynPPA is 14 and 17 for CAL and NW, respectively although the speedup of Δ-stepping is smaller than 1 for such graphs. This show that dynPPA efficiently overcomes to the real world graphs for the *mix* case. As the number of cores increases, the speedup of dynPPA increases for all graphs. Moreover, the speedup of Δ-stepping increases for Smallworld and Erdos while it is almost constant for real-world graphs as the number of cores increases. Next, we study the results by fixing the number of cores to 96 and varying *pce* from 0 to 0.6. In Figure 7.18a, 7.18b, 7.18c, and 7.18d, we present the parallel running time of dynPPA and Δ-stepping for Smallworld, Erdos, CAL, and NW graphs, respectively. For all graphs, while the parallel running time of dynPPA is almost constant, the parallel running time of Δ-stepping generally increases as the pce is increasing. This show that while dynPPA is not sen-

sitive to the changes of the pce for the *mix case*, Δ-stepping is quite sensitive. For all graphs, the parallel running time of Δ-stepping is higher than the parallel running time of dynPPA.

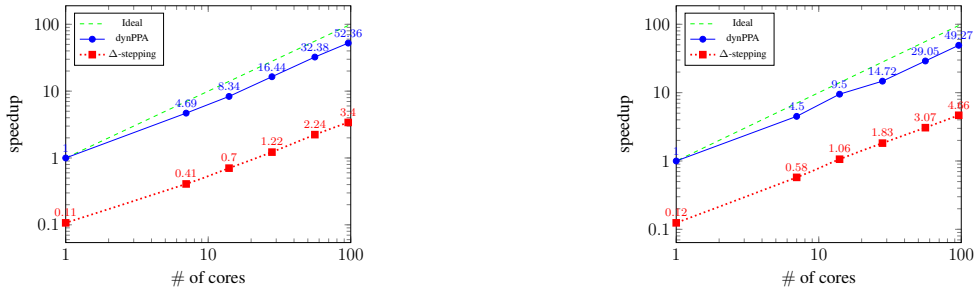
In the light of the information mentioned above, the speedup of dynPPA is close to linear for the synthetic graphs whose the mean-shortest path lengths are small, so they have a small diameter and high average degree. Therefore, dynPPA allows high parallelism on such graphs since they have relatively more uniform distribution of workloads shared by processors. On the other hand, the speedup achieved by dynPPA for real-world graphs decreases to a degree, but still acceptable and provides a speedup that ranges from $11\times$ to $17\times$ with respect to their sequential counterparts while Δ-stepping has poor scalability on such graphs. The reason for this is that the real-world graphs have very high diameter and this results in poor parallel speedup. Additionally, they are largely unbalanced graphs due to power law degree distribution. In such graphs, sparse regions contain heavy weights, which lead to removal of insufficient number of vertices and hence leading to a load imbalance using Δ-stepping. This underlines the effectiveness of dynPPA method to deal with such graphs. For all cases, the speedup achieved by dynPPA increases when the number of cores increases, however, the speedup achieved by Δ-stepping is almost constant for real-world graphs, and it decreases when the number of cores is 96 for the synthetic graphs in some cases due to the communication overhead. On the other hand, for five cases, the parallel running time of dynPPA is almost constant for all graphs as *pce*/*pcw* changes using 96 cores. In fact, for some graphs, the parallel running time of dynPPA decreases thanks to its adaptivity as well as the effect of improving the initial guess even though *pce*/*pcw* increases. However, the parallel running time of Δ-stepping generally increases while *pce*/*pcw* changes especially for the synthetic graphs. Moreover, the parallel running time of Δ-stepping for real-world graphs is much higher than for the synthetic graphs in the *increase1* and *increase2* cases. This is due to the variation of the edge weights since with increasing the pcw or the pce through the *for loop*, the edge weights at sparse regions will be larger (see Table 7.6) and more heterogeneous. This results in low degree of parallelism for Δ-stepping method. Contrary to the *increase1* and *increase2* cases, the parallel running time of Δ-stepping for real-world graphs in the *decrease1* and *decrease2* is not higher than for the synthetic graphs.

The reason for this is that if the edges at sparse regions whose weights are large are selected for decreasing their weights, the variation among the edge weights is smaller and this results in more parallelism using Δ-stepping. For five cases, the parallel running time of Δ-stepping is much higher than the parallel running time of dynPPA. While dynPPA is slightly sensitive to the changes of the parameters pce and pcw for five cases, Δ-stepping is quite sensitive to the changes of those parameters. Moreover, the parameter pcw has a more significant effect than pce for Δ-stepping.

To sum up, the speedup of dynPPA for Smallworld with the small average distance and clustering effect is 50 and the speedup of dynPPA for Erdos with high vertex degrees and very low diameter is 48 on average of five cases using 96 cores. That is, In conclusion, dynPPA provides a near-linear speedup for the synthetic graphs since small average distance and high average degree permit high parallelism. However, the speedup of Δ-stepping is 2 for both Erdos and Smallworld on average of five cases. On the other hand, the speedup of dynPPA for real-world graphs reaches 14 for both CAL and NW on average. Real-world graphs are characterized by low degree and a very high diameter. This results in a low parallel speedup when compared to the synthetic graphs since in such a graph topology, the frontier is propagated. Moreover, Δ-stepping presents poor scaling for such graphs. This underlines the effectiveness of the proposed method to deal with hard real-life problems requiring long time to solution using classical algorithms. We note that even though we have tried graph partitioning tools (such as METIS and PaToH) to improve the parallel scalability of the sparse matrix-vector multiplications in CG in PPA and DynPPA, we have not seen much improvement. However for much larger problems such partitioning can potentially improve the results.

7.3 Results of the Hybrid Method

In this part, we present various numerical experiments. In order to demonstrate efficiency of the proposed algorithm (Algorithm 18), as a baseline method, we use BFS using the *graphshortestpath* function of Matlab, which takes an *n* by *n* sparse graph and computes the single source shortest path. The proposed hybrid algorithm is implemented by using Matlab on a computer with an Intel Pentium Core i7 (2.60 GHz)



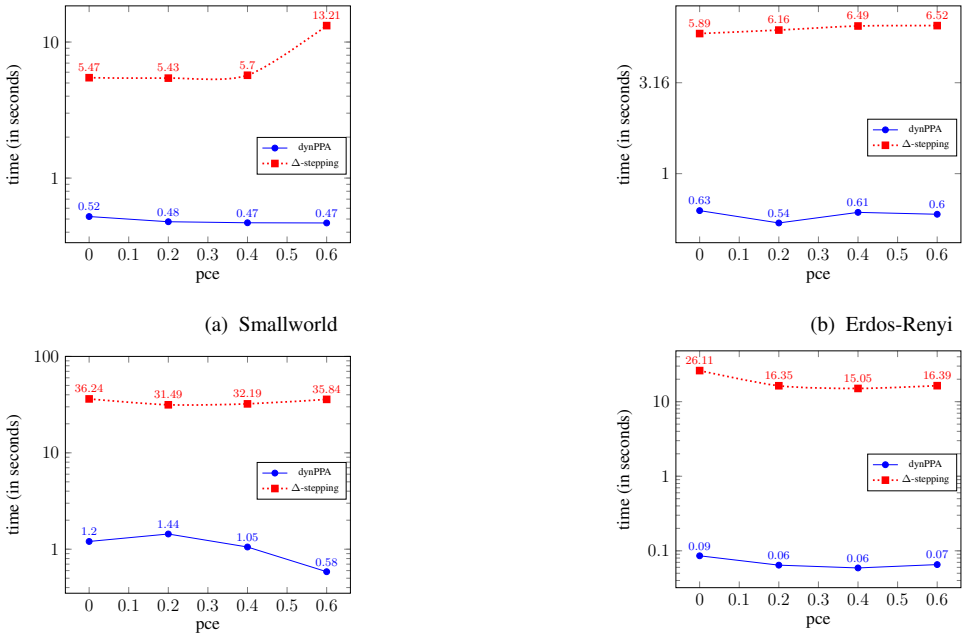
(a) Smallworld

(b) Erdos

(c) CAL

(d) NW

Figure 7.9: *Increase1*: Parallel speedup for sythetic and real-world graphs where pce changes from 0 to 0.6 and pcw is 1



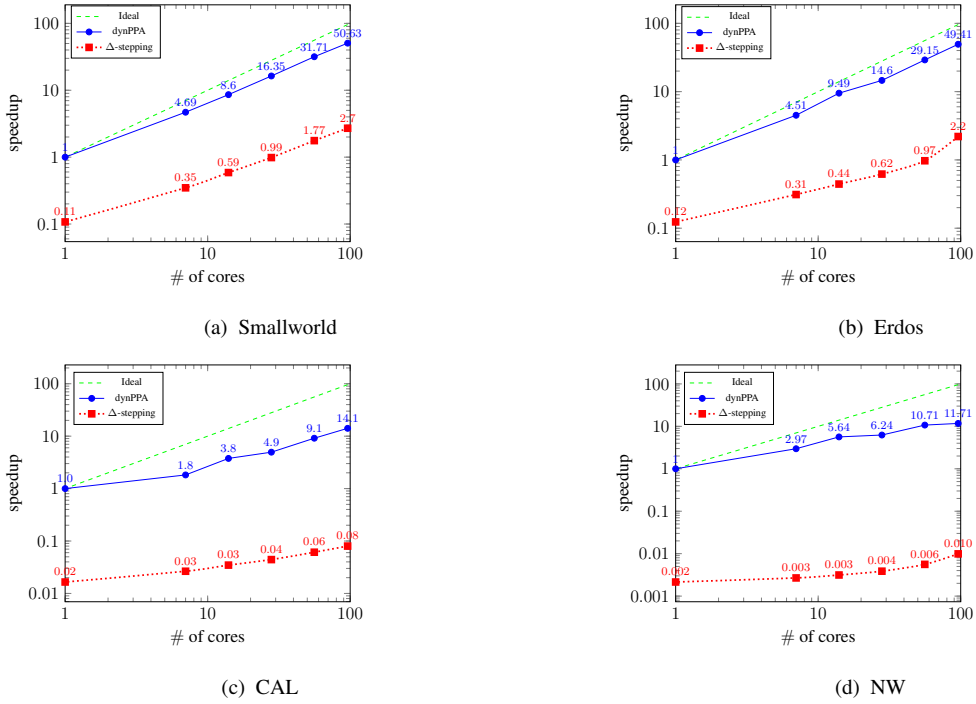
(a) Smallworld

(b) Erdos-Renyi

(c) CAL

(d) NW

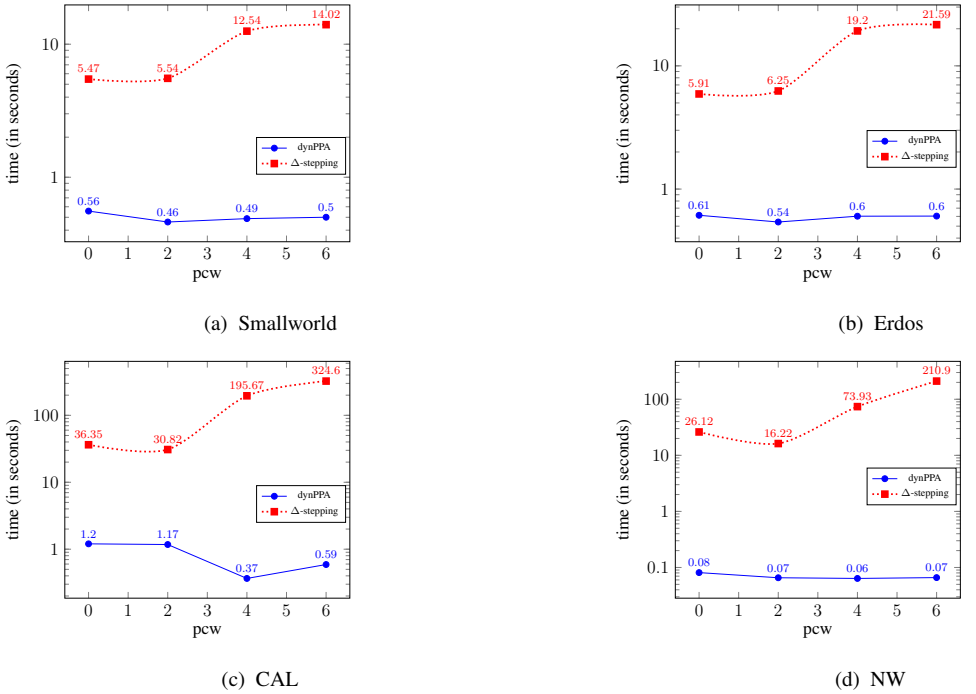
Figure 7.10: *Increase1*: Parallel running time (in seconds) for sythetic and real-world graphs using 96 cores



(a) Smallworld (b) Erdos

(c) CAL (d) NW

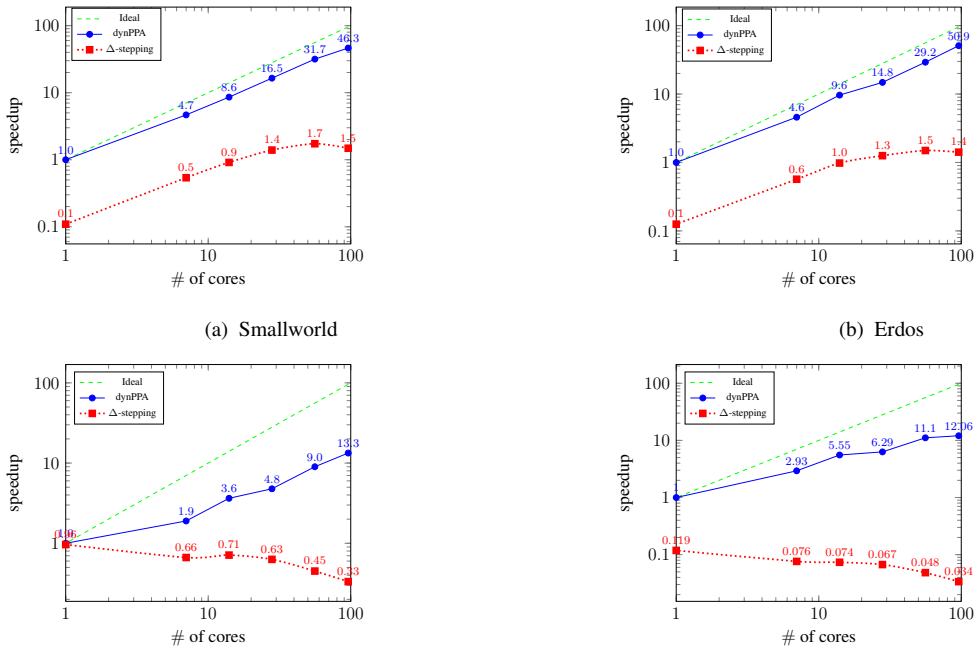
Figure 7.11: *Increase2*: Parallel speedup for sythetic and real-world graphs where pcw changes from 0 to 6 and pce = 0.2



(a) Smallworld (b) Erdos

(c) CAL (d) NW

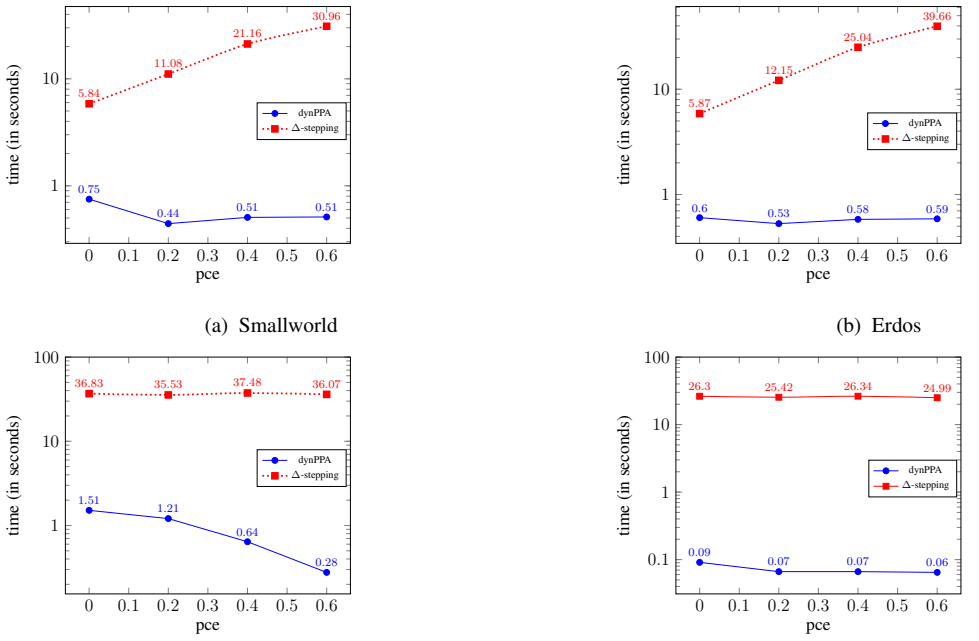
Figure 7.12: *increase2*: Parallel running time (in seconds) for sythetic and real-world graphs using 96 cores



(a) Smallworld (b) Erdos

(c) CAL (d) NW

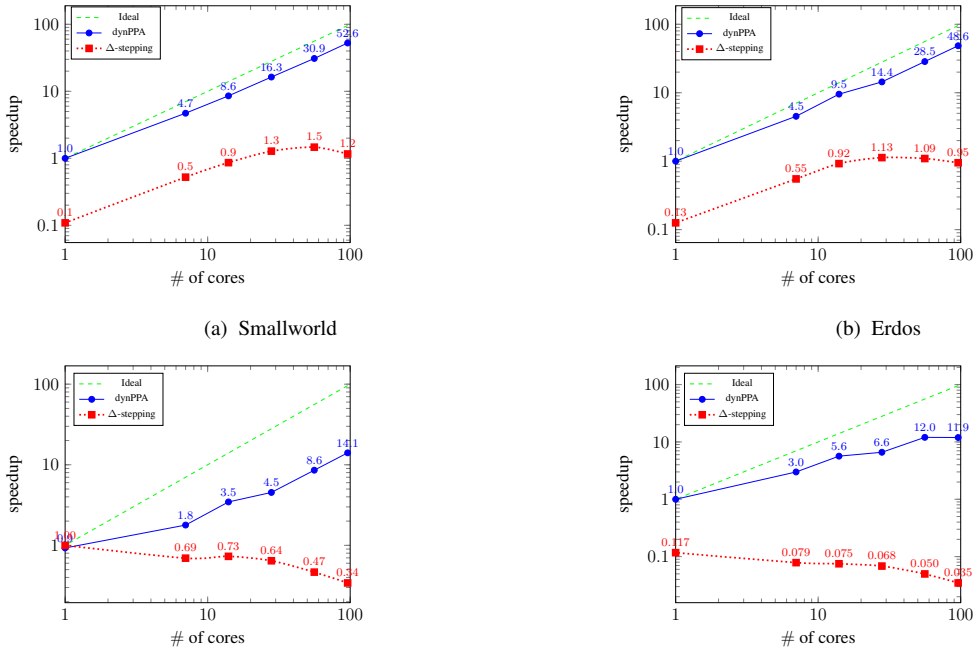
Figure 7.13: *decrease1*: Speedup of sythetic and real-world graphs where *pce* changes from 0 to 0.6 and *pcw*=0.2



(a) Smallworld (b) Erdos

(c) CAL (d) NW

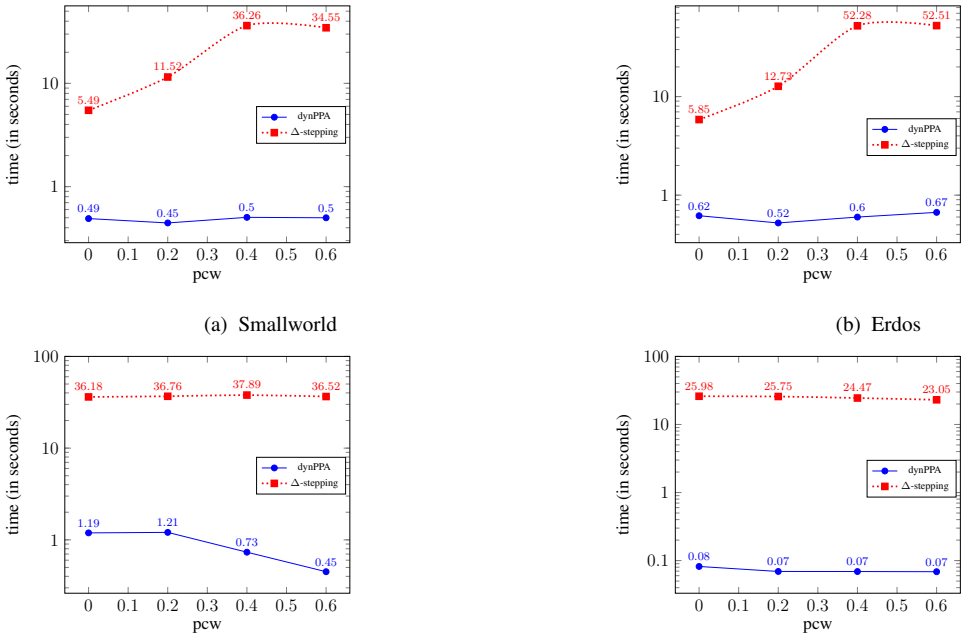
Figure 7.14: *decrease1*: Parallel running time (in seconds) for sythetic and real-world graphs using 96 cores



(a) Smallworld (b) Erdos

(c) CAL (d) NW

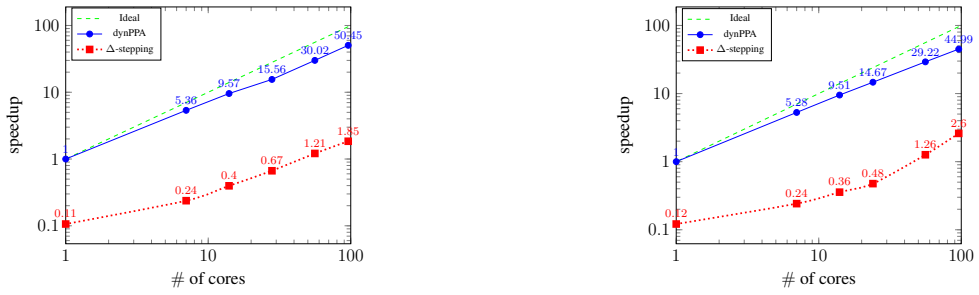
Figure 7.15: *decrease2*: Parallel speedup for sythetic and real-world graphs where pcw changes from 0 to 0.6 and pce = 0.2



(a) Smallworld (b) Erdos

(c) CAL (d) NW

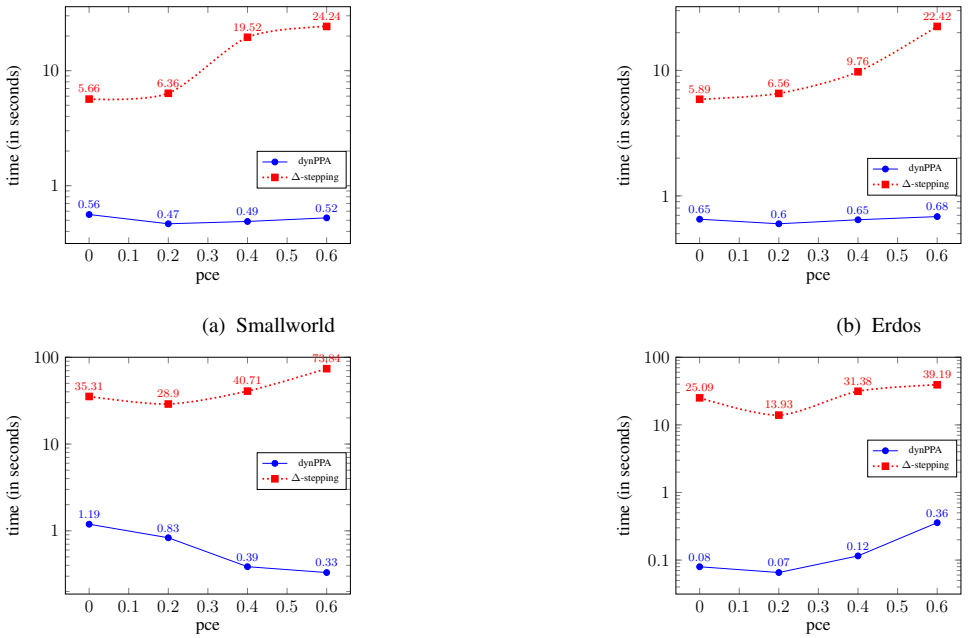
Figure 7.16: *decrease2*: Parallel running time (in seconds) for sythetic and real-world graphs using 96 cores



(a) Smallworld (b) Erdos

(c) CAL (d) NW

Figure 7.17: *Mix case*: Speedup for sythetic and real-world graphs where rue changes from 0 to 0.6 and rcw is 2 for the increasing and 0.2 for the decreasing case



(a) Smallworld (b) Erdos

(c) CAL (d) NW

Figure 7.18: *Mix case*: Parallel running time (in seconds) for sythetic and real-world graphs using 96 cores

Table 7.7: Graph Properties

Graph Name	**Vertices**	**Edges**	**Kind**
Rmat1	3500	6M	R-MAT
Rmat2	4000	7M	R-MAT
Rmat3	5000	12M	R-MAT
Rmat4	6000	18M	R-MAT
Erdos1	7000	24M	Erdos-Renyi
Erdos2	8000	32M	Erdos-Renyi
Erdos3	9000	40M	Erdos-Renyi
Erdos4	10000	50M	Erdos-Renyi

processor and 8GB RAM on Linux operating system. The dataset consists of eight sparse unweighted directed graphs. The number of vertices, edges, and type of the graphs are shown in Table 7.7. The first four graphs are generated by using Erdos-Renyi model (*erdos.renyi.game* function [2] of igraph library in R language). The last four graphs, which are R-MAT graph models, are generated by using PaRMAT library [46]. Diverse real graphs can be well approximated by these models [20]. In the dataset, the graph size varies from 3500 to 10000 and the number of vertices varies from 6M to 50M. We use the unit edge weights.

We evaluate the algorithms based on required time to solution as well as accuracy. In the hybrid algorithm, first, the edges which can not form the shortest path tree are removed. Thus, a sparser graph is obtained. Figure 7.19 presents the number of nonzero elements of G_s (in which the edges which can not form the shortest path are removed) for each graph, separately. Moreover, the number of nonzero elements for the original test graphs G are shown in this figure. As shown in Figure 7.19, there are large number of edges which can not form the shortest path tree and the proposed algorithm identifies and removes such edges efficiently. Second, after removing a large number of unused edges in the shortest path tree, BFS is used to compute the single source shortest path by *graphshortestpath()* function in Matlab. Therefore, BFS algorithm efficiently computes the single source shortest path. Next, we look into the timing results. Hybrid algorithm consists of two stages and the solution time of these stages and baseline algorithm are shown in Table 7.8 in seconds. The algorithm detects the edges which are not a part of the single source shortest path quickly in *Stage1* and the

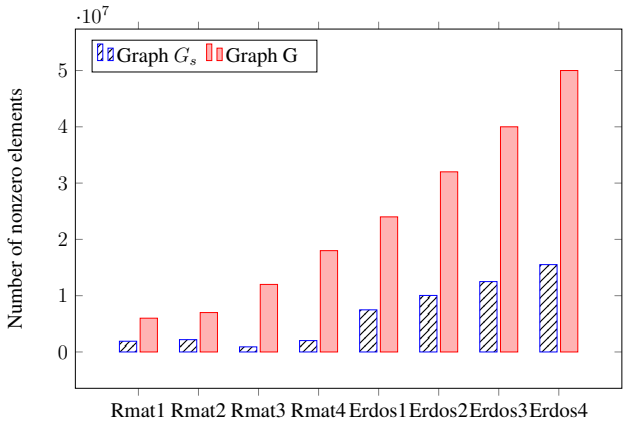


Figure 7.19: Total number of nonzeros elements for the adjacency matrices corresponding to the original (G) and sparsified (G_s) graphs

solution time of this stage is shown in Table 7.8. *Stage2* employs to BFS algorithm on a simpler sparse graph and when we compare timing results of *Stage2* and the baseline algorithm, *Stage2* is $4\times$ faster than the baseline algorithm on average. The reason for this is that the search space in *Stage2* is significantly reduced when compared to the search space in the baseline algorithm. We also note that for graph Rmat3, the number of the edges which is removed from the graph in Figure 7.19 is larger and therefore the required time for *Stage2* is significantly reduced for this graph. Now, we look into the results in terms of the total solution time. Total solution time of the hybrid algorithm is the sum of the time of *Stage1* and *Stage2*. In order to compare the results, we apply BFS algorithm for each test graph and Figure 7.20 presents total solution time of the baseline and the hybrid algorithm for each test graph separately. As shown in Figure 7.20, the time of the hybrid algorithm is less than the baseline for all test graphs. In fact, the proposed hybrid algorithm is about $2.5\times$ faster than the baseline on average. Removing the edges which cannot form the shortest path tree saves significant amount of time. When the size of the graph increases, the time also increases as we expect.

Now, we look into the results in terms of accuracy. We compute the shortest path from the source vertex to the other vertices. Hybrid Algorithm and the baseline algorithm compute exactly the same shortest path with the same distance. On the other hand, if one had used the *Physarum* algorithm alone to find the shortest path it would be only an approximation [12].

Table 7.8: Time (in seconds, rounded to two decimal places) for Hybrid and the baseline algorithms. Hybrid algorithm consists of two stages and time for these stages is shown separately

Graph Name	Hybrid Algorithm			Baseline Algorithm
	Stage1	***Stage2***	***Total***	**BFS**
Rmat1	0.31	1.31	1.62	2.69
Rmat2	0.40	1.51	1.91	3.29
Rmat3	0.62	0.44	1.07	4.88
Rmat4	0.81	0.91	1.72	7.00
Erdos1	1.18	3.11	4.29	11.64
Erdos2	1.51	4.21	5.72	16.20
Erdos3	1.90	5.54	7.44	20.95
Erdos4	2.34	9.65	11.99	26.88

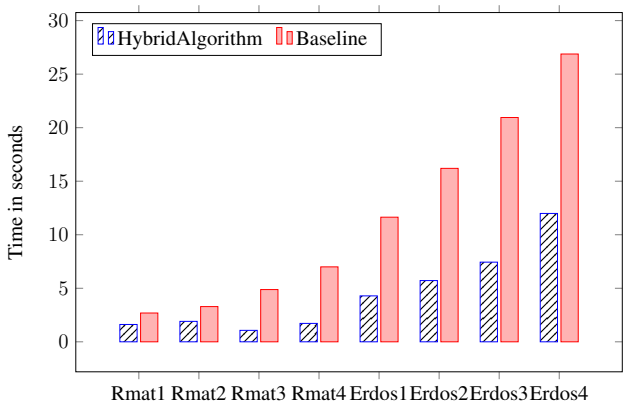


Figure 7.20: Time in seconds of the baseline and Hybrid algorithms for each test graph

CHAPTER 8

CONCLUSION AND FUTURE WORK

As parallelism became more common with the advent of multi-core architectures as well as large and complex networks have begun to emerge in many settings, it is inevitable to come up with algorithms that take advantage of the current architectures. Moreover, bio-inspired models which are more amenable to parallelism have became a promising alternative to handle large scale problems efficiently. Motivated by this we propose iterative shortest path algorithms for static and dynamically changing graphs based on *Physarum solver* on a multicore-cluster architecture. The proposed algorithms include various improvements and optimizations for *Physarum solver*. The linear systems are efficiently solved by both using a parallel iterative solver with a preconditioner. The effect of the preconditioner to the time to solution is discussed by comparing the state-of-the-art preconditioners. Moreover, for dynPPA, previous iteration feeds completely into the next one including initial guess of the iterative solution. The effect of this improvement is also analysed in our study. Finally, in order to compute the shortest path exactly, we propose a novel hybrid algorithm since Physarum Solver is not guaranteed to find the exact shortest path. This underlines the accuracy of the hybrid method to compute the shortest path exactly. Physarum Solver is used in order to detect the edges which cannot form the shortest path tree in the hybrid algorithm. Experimental results have been conducted on graph models with different characteristics to compare the proposed method with the most representative parallel implementation of Dijkstra's algorithm, Δ-stepping. The results are evaluated based on the required time to solution, the parallel speedup as well as accuracy. The proposed algorithms achieve much better speedup compared to Δ-stepping algorithm for all graphs in the datasets. While the speedups of the

proposed methods are near-linear for synthetic graphs that have lower diameter, the speedup is still acceptable for real world graphs that have higher diameter compared to Δ-stepping. As a future study we plan to develop multiple source multiple target parallel shortest path algorithm based on the *Physarum Solver*.

REFERENCES

[1] 9th DIMACS Implementation Challenge - Shortest Paths. http://www.dis.uniroma1.it/challenge9/download.shtml. Accessed: January-2017.

[2] Erdos-Renyi Graphs. http://cneurocvs.rmki.kfki.hu/igraph/doc/R/erdos.renyi.game.html. Accessed: January-2017.

[3] Portable, Extensible Toolkit for Scientific Computing, version 3.6.3. http://www.mcs.anl.gov/petsc. Accessed: 2016-04-1.

[4] The Watts-Strogatz small-world model. http://cneurocvs.rmki.kfki.hu/igraph/doc/R/watts.strogatz.game.html. Accessed: January-2017.

[5] C. W. Ahn and R. S. Ramakrishna. A genetic algorithm for shortest path routing problem and the sizing of populations. *IEEE Transactions on Evolutionary Computation*, 6(6):566–579, 12 2002.

[6] L. Aleksandrov, A. Maheshwari, and J. R. Sack. Determining Approximate Shortest Paths on Weighted Polyhedral Surfaces. *Journal of the ACM*, 52(1):25–53, 1 2005.

[7] W. E. Arnoldi. The principle of minimized iteration in the solution of the matrix eigenvalue problem. *Quart. Appl. Math.*, 9:17–29, 1951.

[8] B. Awerbuch, B. Berger, L. Cowen, and D. Peleg. Near-Linear Time Construction of Sparse Neighborhood Covers. *SIAM J. Comput.*, 28(1):263–277, 7 2006.

[9] R. E. Bank and T. F. Chan. A composite step bi-conjugate gradient algorithm for nonsymmetric linear systems. *Numerical Algorithms*, 7(1):1–16, Mar 1994.

[10] M. Barbehenn and S. Hutchinson. Efficient search and hierarchical motion planning by dynamically maintaining single-source shortest paths trees. *IEEE Transactions on Robotics and Automation*, 11(2):198–214, 1995.

[11] F. Bauer and A. Varma. Distributed algorithms for multicast path setup in data networks. In *Global Telecommunications Conference, 1995. GLOBECOM '95., IEEE*, volume 2, pages 1374–1378, Nov 1995.

[12] L. Becchetti, V. Bonifaci, M. Dirnberger, A. Karrenbauer, and K. Mehlhorn. Physarum Can Compute Shortest Paths: Convergence Proofs and Complexity Bounds. In F.V. Fomin, R. Freivalds, M. Kwiatkowska, and D. Peleg, editors, *Automata Languages and Programming: 40th International Colloquium and ICALP 2013 Riga and Latvia and July 8-12 and 2013 and Proceedings and Part II*, pages 472–483, Berlin and Heidelberg, 2013. Springer Berlin Heidelberg.

[13] R. Bellman. On A Routing Problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 12 1958.

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

[15] M. Benzi and M. Tuma. A comparative study of sparse approximate inverse preconditioners. *Applied Numerical Mathematics*, 30(2):305 – 340, 1999.

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

[17] I. Chabini and S. Ganugapati. Design and implementation of parallel dynamic shortest path algorithms for intelligent transportation systems application. *Transportation Research Records*, 1771:219–228, 2001.

[18] A. Chaibou and O. Sie. Improving Global Performance on GPU for Algorithms with Main Loop Containing a Reduction Operation: Case of Dijkstra's Algorithm. *Journal of Computer and Communications*, 3:41–54, 2015.

[19] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 889–901, 2014.

[20] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004.

[21] E. P. F. Chan and Y. Yang. Shortest Path Tree Computation in Dynamic Graphs. *IEEE Transactions on Computers*, 58(4):541–557, April 2009.

[22] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. part ii: Optimal parallel algorithms. *Theoretical Computer Science*, 203(2):205–223, 1998.

[23] B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest path algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

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

[25] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. *A parallelization of Dijkstra's shortest path algorithm*, pages 722–731. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[26] C. Demetrescu and G. F. Italiano. Fully dynamic all pairs shortest paths with real edge weights. In *Proceedings 2001 IEEE International Conference on Cluster Computing*, pages 260–267, Oct 2001.

[27] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematics*, 1:269–271, 1959.

[28] H. N. Djidjev, G. E. Pantziou, and C. D. Zaroliagis. On-line and dynamic algorithms for shortest path problems. In *STACS 95: 12th Annual Symposium on Theoretical Aspects of Computer Science Munich, Germany, March 2–4, 1995 Proceedings*, pages 193–204, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[29] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 4 1997.

[30] N. Edmonds, A. Breuer, D. Gregor, and A. Lumsdaine. Single-Source Shortest Paths with the Parallel Boost Graph Library. *Open Systems Laboratory, Indiana University*.

[31] N. Edmonds, A. Breuer, D. Gregor, and Andrew Lumsdaine. Single-Source Shortest Paths with the Parallel Boost Graph Library. In *The Ninth DIMACS Implementation Challenge: The Shortest Path Problem*, 2006.

[32] S. C. Eisenstat. Efficient Implementation of a Class of Preconditioned Conjugate Gradient Methods. *SIAM Journal on Scientific and Statistical Computing*, 2:1–4, 1981.

[33] M. Elkin. Computing Almost Shortest Paths. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, pages 53–62. ACM, 2001.

[34] G. Ertl. Shortest path calculation in large road networks. *Operations-Research-Spektrum*, 20(1):15–20, 1998.

[35] D. Frigioni, M. Ioffreda, U. Nanni, and G. Pasqualone. Experimental analysis of dynamic algorithms for the single source shortest paths problem. *J. Exp. Algorithmics*, 3, 1998.

[36] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic output bounded single source shortest path problem. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 212–221, 1996.

[37] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2):251–281, 2000.

[38] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. *Reach for A*: Efficient Point-to-Point Shortest Path Algorithms*, pages 129–143.

[39] M. Gong, G. Li, Z. Wang, L. Ma, and D. Tian. An efficient shortest path approach for social networks based on community structure. *{CAAI} Transactions on Intelligence Technology*, 1(1):114 – 123, 2016.

[40] R. Grech, T. Cassar, J. Muscat, K. P. Camilleri, S. G. Fabri, M. Zervakis, P. Xanthopoulos, V. Sakkalis, and B. Vanrumste. Review on solving the inverse problem in eeg source analysis. *Journal of NeuroEngineering and Rehabilitation*, 5(1):4–46, Nov 2008.

[41] D. Gregor and A. Lumsdaine. The Parallel BGL: A Generic Library for Distributed Graph Computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, 2005.

[42] A. Gubichev, S. Bedathur, S. Seufert, and G. Weikum. Fast and Accurate Estimation of Shortest Paths in Large Graphs. In *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, CIKM '10, pages 499–508. ACM, 2010.

[43] A. Hadjidimos, D. Noutsos, and M. Tzoumas. More on modifications and improvements of classical iterative schemes for M-matrices. *Linear Algebra and its Applications*, 364:253–279, 2003.

[44] V. E. Henson and U. M. Yang. Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155 – 177, 2002.

[45] T. Ikeda, Min-Yao Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. A fast algorithm for finding better routes by ai search techniques. In *Vehicle Navigation and Information Systems Conference, 1994. Proceedings., 1994*, pages 291–296, 1994.

[46] Khorasani, Farzad, Vora, Keval, Gupta, and Rajiv. PaRMAT: A Parallel Generator for Large R-MAT Graphs. `https://github.com/farkhor/PaRMAT`, 2015.

[47] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*, pages 81–89, 1999.

[48] E. J. Lee and J. Zhang. A two-phase preconditioning strategy of sparse approximate inverse for indefinite matrices. *Computers and Mathematics with Applications*, 58(6):1152 – 1159, 2009.

[49] M. Liang, C. Gao, and Z. Zhang. A new genetic algorithm based on modified physarum network model for bandwidth-delay constrained least-cost multicast routing. *Natural Computing*, 16(1):85–98, Mar 2017.

[50] L. Liu, Y. Song, H. Zhang, and H. Ma. Physarum Optimization: A Biology-Inspired Algorithm for the Steiner Tree Problem in Networks. *IEEE Transaction on Computers*, 64:818–831, 2015.

[51] M. Manguoglu, M. Koyutürk, A. H. Sameh, and A. Grama. Weighted matrix ordering and parallel banded preconditioners for iterative linear system solvers. *SIAM J. Sci. Comput.*, 32(3):1201–1216, April 2010.

[52] U. Meyer and P. Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49:114–152, 2003.

[53] T. Miyaji and I. Ohnishi. Physarum Can Solve the Shortest Path Problem on Riemann Surface Mathematically Rigorously. *International Journal of Pure and Applied Mathematics*, 47:353–369, 2008.

[54] P. Narvaez, K. Y. Siu, and H. Y. Tzeng. New dynamic spt algorithm based on a ball-and-string model. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 973–981, 1999.

[55] P. Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic algorithms for shortest path tree computation. *IEEE/ACM Transactions on Networking*, 8(6):734–746, Dec 2000.

[56] M. Neumann and R. J. Plemmons. Convergence of Parallel Multisplitting Iterative Methods for M-Matrices. *Linear Algebra and its Applications*, 88:559–573, 1987.

[57] U. T. Nguyen and J. Xu. Multicast Routing in Wireless Mesh Networks: Minimum Cost Trees or Shortest Path Trees? *IEEE Communications Magazine*, 45(11):72–77, 2007.

[58] J.M Ortega and C.H Romine. The ijk forms of factorization methods ii. parallel systems. *Parallel Computing*, 7(2):149 – 162, 1988.

[59] C. C. Paige and M. A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM Journal on Numerical Analysis*, 12(4):617–629, 1975.

[60] M. L. Parks, E. Sturler, G. Mackey, D. D. Johnson, and S. Maiti. Recycling krylov subspaces for sequences of linear systems. *SIAM J. Sci. Comput.*, 28:1651–1674, 2006.

[61] D. Peleg and J. D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18:740–747, 1989.

[62] R.J. Plemmons. M-matrix characterizations.I-nonsingular M-matrices. *Linear Algebra and its Applications*, 18(2):175–188, 1977.

[63] I. Pohl. Bi-directional search. *Machine Intelligence*, 6:124–140, 1971.

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

[65] B.T. Polyak. The conjugate gradient method in extremal problems. *USSR Computational Mathematics and Mathematical Physics*, 9(4):94 – 112, 1969.

[66] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.

[67] G. Ramalingam and T. Reps. An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms*, 21(2):267–305, 1996.

[68] S.Chandra Sekhara Rao. Existence and uniqueness of wz factorization. *Parallel Computing*, 23(8):1129 – 1139, 1997.

[69] A. A. Rescigno. Optimally balanced spanning tree of the star network. *IEEE Transactions on Computers*, 50(1):88–91, Jan 2001.

[70] W. C. Rheinboldt. On M-functions and their application to nonlinear Gauss-Seidel iterations and to network flows. *Journal of Mathematical Analysis and Applications*, 32:274–307, 1970.

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

[72] Y. Saad and M. H. Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[73] B. Schutter and B. Moor. The qr decomposition and the singular value decomposition in the symmetrized max-plus algebra revisited. *SIAM Review*, 44(3):417–454, 2002.

[74] R. Sedgewick and J. Vitter. Shortest paths in euclidean graphs. *Algorithmica*, 1:31–48, 1986.

[75] S. Sen. Approximating Shortest Paths in Graphs. In *3rd International Workshop on Algorithms and Computation (WALCOM)*, pages 32–43, 2009.

[76] K. Stuben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128(1):281 – 309, 2001.

[77] K. Stüben, U. Trottenberg, and K. Witsch. Software development based on multigrid techniques. In *Proc. Of the Sixth Int'L. Symposium on Computing Methods in Applied Sciences and Engineering, VI*, pages 187–, 1985.

[78] A. Tero, R. Kobayashi, and T. Nakagaki. Physarum solver: A biologically inspired method of road-network navigation. *Physica A: Statistical Mechanics and its Applications*, 363(1):115–119, 4 2006.

[79] 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, 2 2007.

[80] S. Xu and W. Jiang. A physarum-inspired model for the probit-based stochastic user equilibrium problem. *CoRR*, abs/1703.01880, 2017.

[81] R. Yuster. Approximate shortest paths in weighted graphs. *Journal of Computer and System Sciences*, 78:632–637, 2012.

[82] F. B. Zhan and C. E. Noon. A comparison between label-setting and label-correcting algorithms for computing one-to-one shortest paths. *Journal of Geographic Information and Decision Analysis*, 4, 2000.

[83] J. Zhang. A sparse approximate inverse preconditioner for parallel preconditioning of general sparse matrices. *Applied Mathematics and Computation*, 130(1):63 – 85, 2002.

[84] X. Zhang, A. Adamatzky, F. T. S. Chan, Y. Deng, H. Yang, X. S. Yang, M.A.I. Tsompanas, G. Sirakoulis, and S. Mahadevan. A biologically inspired network design model. *Scientific Reports*, 5:10794, 2015.

[85] X. Zhang, A. Adamatzky, H. Yang, S. Mahadaven, X. S. Yang, Q. Wang, and Y. Deng. A bio-inspired algorithm for identification of critical components in the transportation networks. *Applied Mathematics and Computation*, 248:18–27, 2014.

[86] X. Zhang, F. T.S. Chan, H. Yang, and Y. Deng. An adaptive amoeba algorithm for shortest path tree computation in dynamic graphs. *Information Sciences*, 405:123–140, 2017.

[87] X. Zhang, S. Huang, Y. Hu, Y. Zhang, S. Mahadevan, and Y. Deng. Solving 0-1 Knapsack problems based on amoeboid organism algorithm. *Applied Mathematics and Computation*, 219:9959–9970, 2013.

[88] X. Zhang, X. Zhang, Y. Zhang, D. Wei, and Y. Deng. Route selection for emergency logistics management: A bio-inspired algorithm. *Safety Science*, 54:87–91, 2013.

[89] X. Zhang, Y. Zhang, and Y. Deng. An improved bio-inspired algorithm for the directed shortest path problem. *Bioinspiration & Biomimetics*, 9(4):046016, 2014.

[90] Z. Zhang, C. Gao, Y. Liu, and T. Qian. A universal optimization strategy for ant colony optimization algorithms based on the physarum -inspired mathematical model. *Bioinspiration and Biomimetics*, 9(3):036006, 2014.

CURRICULUM VITAE

PERSONAL INFORMATION

Surname, Name: Arslan, Hilal
Nationality: Turkish (TC)
Date and Place of Birth: 12.03.1984, Ankara
Marital Status: Married
Phone: 0 312 2102080
Fax: 0 312 2105544

EDUCATION

Degree	Institution	Year of Graduation
Ph.D.	Department of Computer Engineering, METU	2017
M.S.	Department of Computer Engineering, METU	2011
M.S.	Department of Mathematics, Ankara University	2007
B.S.	Department of Mathematics, Ankara University	2005
High School	Fatih Sultan Mehmet High School	2001

SCHOLARSHIP

TUBİTAK (The Scientific and Technological Research Council of Turkey) higher education scholarship for Ph.D. Degree (2012-2017).

PROFESSIONAL EXPERIENCE

Year	Place	Enrollment
2007-2018	Department of Computer Engineering, METU	Teaching Assistant

PUBLICATIONS

Paper submitted to international journal

H. Arslan and M. Manguoğlu, *A Parallel Bio-Inspried Shortest Path Algorithm*, Computing, 2018.

International Conference Publication

H. Kılıç, Y. İlgüner, and T. Can, *ProSVM and ProK-means: Novel methods for promoter prediction*, Health Informatics and Bioinformatics (HIBIT), 2011.