REEVALUATING SPECTRAL PARTITIONING FOR UNSYMMETRIC MATRICES

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF APPLIED MATHEMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

EDA OKTAY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
SCIENTIFIC COMPUTING

SEPTEMBER 2020

Approval of the thesis:

REEVALUATING SPECTRAL PARTITIONING FOR UNSYMMETRIC MATRICES

submitted by **EDA OKTAY** in partial fulfillment of the requirements for the degree of **Master of Science in Scientific Computing Department, Middle East Technical University** by,

Prof. Dr. Ömür Uğur
Director, Graduate School of **Applied Mathematics** _____

Prof. Dr. Ömür Uğur
Head of Department, **Scientific Computing** _____

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

Assoc. Prof. Dr. Hamdulah Yücel
Co-supervisor, **Scientific Computing, METU** _____

Examining Committee Members:

Prof. Dr. Ömür Uğur
Institute of Applied Mathematics, METU _____

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

Prof. Dr. Cevdet Aykanat
Computer Engineering Department, Bilkent 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: EDA OKTAY

Signature :

ABSTRACT

REEVALUATING SPECTRAL PARTITIONING FOR UNSYMMETRIC MATRICES

Oktay, Eda

M.S., Department of Scientific Computing

Supervisor : Prof. Dr. Murat Manguoğlu

Co-Supervisor : Assoc. Prof. Dr. Hamdulah Yücel

September 2020, 89 pages

Parallel solutions to scientific problems having graph representation require efficient tasks and partitioning data. In this thesis, various parallel graph partitioning algorithms are studied. While these algorithms are applicable to both directed and undirected graphs, we focus on the directed case whose matrix representations are sparse and unsymmetric arising in linear system of equations representing various application domains such as computational fluid dynamics and thermal problems. Strategies inspected in this study are ParMETIS with the Multilevel Kernighan-Lin algorithm and the spectral partitioning algorithm with k-means clustering (SPEC) as well as the recursive spectral partitioning algorithm in CHACO. We have implemented SPEC in C programming language using PETSc and SLEPc libraries, whereas CHACO and ParMETIS are called from PETSc. Weighted partitioning is done under the consideration of the edge weights of the graph. SPEC is compared with the libraries only when the unweighted partitioning is made due to the limitations of the libraries for weighted partitioning. Hence, for weighted partitioning, only various eigensolver tolerances in SLEPc are studied in terms of the edge-cut and partitioning time. Another study is performed for the spectral partitioning algorithm based on eigensolver tolerance used with the k-means algorithm in MATLAB. The comparison is based on the quality of the partitioning (edge-cut and partition imbalance) and the number of iterations. The quality of partitioning is determined by the edge-cut and the load im-

balance, which could be based on the edge and vertex imbalance ratios of partitions depending on the application. Since the adjacency matrix of a graph is structurally symmetric, the eigenvalue problem can only be solved approximately when the matrix is unsymmetric. Thus, only approximate results are provided in this study.

It is deduced that using SPEC performs better than the existing software libraries when the number of cut edges is compared in unweighted partitioning of unsymmetric matrices.

Keywords: parallel graph partitioning, Laplacian, PETSc, SLEPc, ParMETIS, CHACO, spectral partitioning, domain decomposition, k-means clustering

ÖZ

SİMETRİK OLMAYAN MATRİSLER İÇİN SPEKTRAL BÖLÜMLEMEYİ YENİDEN DEĞERLENDİRME

Oktay, Eda
Yüksek Lisans, Bilimsel Hesaplama Bölümü
Tez Yöneticisi : Prof. Dr. Murat Manguoğlu
Ortak Tez Yöneticisi : Doç. Dr. Hamdulah Yücel

Eylül 2020, 89 sayfa

Grafik temsiline sahip bilimsel problemlerin paralel çözümleri, verimli görev ve veri bölümleme gerektirir. Bu tezde, çeşitli paralel grafik bölümleme algoritmaları incelenmiştir. Bu algoritmalar hem yönlendirilmiş hem de yönsüz grafiklere uygulanabilir olsa da, bu çalışmada, matris gösterimleri seyrek ve simetrik olmayan, hesaplamalı akışkanlar dinamiği ve termal problemler gibi çeşitli uygulama alanlarını temsil eden doğrusal denklem sisteminde ortaya çıkan yönlendirilmiş duruma odaklanıyoruz. Bu çalışmada incelenen stratejiler, Çok Seviyeli Kernighan-Lin algoritmasına sahip ParMETIS, k-ortalamalı kümeleme algoritması ile birlikte kullanılan spektral bölümleme (SPEC), ve CHACO içerisinde kullanılan spektral bölümleme algoritmasıdır. PETSc ve SLEPc kitaplıkları kullanılarak C programlama dilinde SPEC algoritması uygulanmış olup, CHACO ve ParMETIS ise PETSc'den çağrılmaktadır. Grafiğin kenar ağırlıkları dikkate alınarak ağırlıklı bölümlendirme yapılır. Ağırlıklı bölümleme yapıldığında kitaplıkların sınırlamaları nedeniyle SPEC, kitaplıklarla yalnızca ağırlıksız bölümleme yapıldığında karşılaştırılır. Bu nedenle, ağırlıklı bölümleme için, sadece SLEPc'deki çeşitli özdeğer çözücü toleransları, kenar kesme ve bölümleme süresi açısından incelenir. Başka bir araştırma ise MATLAB içerisinde k-ortalamalı kümeleme algoritması tarafından kullanıldığında, özdeğer çözücü toleransına dayalı spektral bölümleme algoritması için yapılmıştır. Karşılaştırma, bölümlemenin kalitesine (kenar kesimi ve bölüm dengesizliği) ve yineleme sayısı cinsinden maliyete dayanmaktadır.

Bir bölümlemenin kalitesi, uygulamaya bağlı olarak bölümlerin kenar ve tepe dengesizlik oranlarına bağlı olabilecek yük dengesizliğinin yanı sıra kesilen kenar sayısı ile belirlenir. Bir grafiğin bitişik matrisi yapısal olarak simetrik olduğundan, matris simetrik olmadığında özdeğer problemi ancak yaklaşık olarak çözülebilir. Bu nedenle, bu çalışmada yalnızca yaklaşık sonuçlar verilmiştir.

Simetrik olmayan matrislerin ağırlıksız bölümlemesinde kenar kesim sayısı karşılaştırıldığında, SPEC kullanımının mevcut yazılım kitaplıklarından daha iyi performans gösterdiği sonucuna varılmıştır.

Anahtar Kelimeler: paralel grafik bölümleme, Laplas, PETSc, SLEPc, ParMETIS, CHACO, spektral bölümleme, bölgesel ayrıştırma, k-means kümeleme yöntemi

To my family

ACKNOWLEDGMENTS

I would like to express my very great appreciation to my thesis supervisor Prof. Murat Manguoğlu and co-supervisor Assoc. Prof. Hamdullah Yücel for their patient guidance, encouragement, and valuable advice during this thesis's development and preparation. Their willingness to give their time and to share their experiences have brightened my path. I would also like to express my gratitude for Dr. Onur Tolga Şehitoğlu for his willingness to give his time and share his experiences.

I would also like to thank members of my thesis defense committee for their insightful comments and discussions.

I am thankful to my colleagues in the Institute of Applied Mathematics for providing such a friendly environment. Their experiences and sharing encouraged me not to give up. I am incredibly grateful to Mehmet Alp Üreten for his patient guidance during the preparation of this thesis.

Last but not least, I am deeply grateful to my family for their continuous support and encouragement through my graduate studies when I had doubted myself. I owe everything to them in this period and every moment of my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In many science and engineering problems, differential equations are used to model numerical problems and estimate systems' quantitative behaviors. Scientific computing tools are used for solving such equations to obtain some numerical simulations. In solving such problems, limited memory becomes one of the main bottlenecks since systems of equations are usually large and complicated. Hence, breaking down the system becomes crucial in scientific computing. In literature, there are various matrix partitioning methods such as Non-Rectangular Recursive Partitioning (NRRP) [5], and recursive partitioning method [8], depending on the sparsity (nonzero structure) of a matrix. Graph partitioning is one method to break down the coefficient side matrix if it is sparse [54]. The main goal of graph partitioning is to make the matrix suitable for parallel computing by splitting it into smaller sub-matrices. There are many partitioning strategies for graphs such as spectral, geometric, and multilevel partitioning schemes [54]. Therefore, for matrix partitioning, using graph partitioning techniques are preferred.

Graphs may or may not contain nodal coordinate information. In the former case, there are algorithms using the available coordinate information. The latter (also called non-spatial graph) is more general and applicable regardless of the availability of the nodal coordinate information. There are two main graph partitioning strategies for their use of nodal coordinates [6]: If a graph has coordinate information of its nodes, then the partitioning is done to assume nearest neighbor connectivity, meaning that the algorithm ignores edges. On the other hand, if the graph is non-spatial, then the partitioning will be based only on the graph's adjacency information since there is no

information about coordinates [6].

Moreover, non-spatial graph partitioning algorithms are categorized into two based on the approaches that work on the entire or local graph, namely global and local methods. Global methods work on the entire graph and compute solution directly, whereas local methods find a small cut near a specified starting vertex [1]. One of the most common examples of global methods is spectral partitioning [10] which is an approach based on *the Fiedler vector* [19]. A significant advantage of global methods is that they do not rely on an arbitrary initial partitioning. However, the essential drawback is that they are limited to bi-partitioning, which is partitioning a graph into two disjoint and independent sets. To overcome the limitation, these methods can be used with various clustering algorithms. For instance, the spectral method can do k partitioning when the k-means clustering algorithm is used on k eigenvectors [44]. Furthermore, local approaches are useful if the main concern is the cost in terms of time. Because instead of graph size, partitioning time of such algorithms are proportional to the edge-cut. However, these methods' main disadvantage is the vertex set's arbitrary initial partitioning since it affects the final partitioning quality. As an example of local approaches, the Kernighan-Lin algorithm [41] can be given.

As the problems become larger, solving and partitioning them become more difficult due to limited memory and require too much time to obtain a solution. Hence, the need for parallelism arises [36]. The main aim of parallel computing is to obtain solutions faster by using multiple resources. If some of the connections between vertices can be reduced in a graph, it can be partitioned in parallel faster, and hence if the application is solving a sparse linear system, it can be solved faster.

There are many libraries for graph partitioning using various sequential algorithms (e.g., Party [50], CHACO [32]) or parallel algorithms (e.g., ParMETIS [40], SCOTCH [49]). Using them in partitioning a graph can reduce cost in terms of communication time, and load balance can be achieved. However, the algorithms used in these libraries are heuristics, and their parallel implementations are known to be not scalable. To improve the quality and parallel scalability, we propose an algorithm using spectral partitioning [4] with the k-means clustering algorithm [43], that we call SPEC throughout the thesis.

We choose to study spectral graph partitioning since it is more amenable to parallelism and provides some flexibility via the precision of the eigenvalue computations. The spectral partitioning algorithm is first introduced by Donath and Hoffman [16]. In 1973, they proposed an algorithm for the construction of graph partitions based on the corresponding adjacency matrix's eigenvectors. In the same year, Fiedler [19] discovered the algebraic connectivity of graphs. Algebraic connectivity is based on the second smallest eigenvalue of Laplacian graph and its corresponding eigenvector. Hence, he suggested using this eigenvector to partition a graph. Then, to compute eigenvectors more efficiently, Parlett et al. [48] made improvements in algorithms to compute eigenvealues approximately. Later, the spectral partitioning method has become a simplified approach for graph partitioning, sparse matrix reorderings, and computing the Fiedler vector in parallel [45].

Another graph partitioning method is Kernighan-Lin (KL) algorithm. In 1970, KL algorithm is devised by Kernighan and Lin [41] for partitioning arbitrary graphs effectively in terms of time and the edge-cut. Later, in 1993, Bui and Jones [9] made improvements on the quality of the bisection returned by the KL and introduced three steps of multilevel partitioning: coarsening, partitioning, and uncoarsening. Later, Hendrickson and Leland [32] made improvements on this algorithm and presented another multilevel algorithm in based on the recursive usage of the spectral method. In 1998, Karypis and Kumar [39] presented a faster KL algorithm, multilevel KL, in which Fiduccia and Mattheyses linear-time version [18] of KL (KLFM) refinement is done during uncoarsening.

As partitioning large-scale matrices need to be faster, software packages have started to be developed while partitioning algorithms have been improved. In 1993, CHACO [31] was developed for the recursive graph partitioning. Later, to use multilevel methods, in 1997, Karypis and Kumar developed a software package called as METIS [38] for partitioning unstructured graphs. However, METIS is operating sequentially. Moreover, to partition graphs in parallel, they introduced ParMETIS [37] in the same year.

To partition a graph using the algorithms mentioned above, it is assumed that the graph is undirected. Therefore, there are many studies comparing the partitioning al-

gorithms when the corresponding matrix is symmetric. For instance, Gupta compared [27] graph partitioning and sparse matrix ordering package, namely WGPP, [26] and METIS, while Horst compared [55] recursive spectral bisection with recursive graph bisection, and recursive coordinate bisection in terms of the edge-cut.

There are several studies on structurally unsymmetric matrices. In 1998, Hendrickson and Kolda [28] partitioned unsymmetric and rectangular matrices by using bipartite graph concept and multilevel partitioning approaches. In 2000, they also tested several methods on unsymmetric matrices such as Fiduccia–Mattheyses [18], spectral, and alternating partitioning [42] methods with their multilevel implementations [30]. According to their research in [29], there are two types for modeling unsymmetric matrices for using symmetric matrix partitioning techniques. The first one is converting directed edges to undirected edges, i.e., using the matrix structure $|A| + |A|^T$. The second one is giving weights edges representing only one-way communication as ones, whereas giving the ones representing two-way communication as twos. For partitioning unsymmetric matrices, hypergraph partitioning strategies can also be used. If some edges connect more than two vertices, then these edges are said to be hyperedges, and the graph is said to be hypergraph. In 1999, Çatalyürek and Aykanat [13] proposed a generalized graph model along with hypergraph models for enabling the decomposition of unsymmetric matrices. In this study, unsymmetric matrices are partitioned using symmetric partitioning techniques by using the first modeling approach in [29]: For an unsymmetric matrix A, the problem is modeled as $|A| + |A|^T$ so that it becomes structurally symmetric. Even though this computation gives approximation results, it is an exact model for some applications, such as Hermitian and skew-Hermitian splitting [2].

Recently, with the advances of new computer architectures, more efficient algorithms and their implementations have been proposed in which they can solve eigenvalue problems faster and in parallel, such as Krylov methods [33]. Therefore, in this thesis, we focus on the reevaluation of spectral partitioning algorithms, which used to be considered as a slower alternative in the past. Using spectral partitioning algorithm has two main advantages. First of all, the tolerance of the eigenvalue problem used in the spectral partitioning provides flexibility that the other algorithms do not have. Moreover, when weighted graph partitioning is inspected, while the other partitioning

methods can only consider integer weights in the best case, the spectral algorithm can consider floating-point numbers as weights. This advantage makes the spectral graph partitioning method more accurate since, for other methods, a mapping is needed for integer weights, which causes information loss.

This thesis compares SPEC against CHACO and ParMETIS to partition a graph whose matrix representation is sparse unsymmetric arising in the linear systems representing the solution of various problems such as structural and mathematical problems. Furthermore, different eigensolver tolerances are inspected in the spectral partitioning algorithm in MATLAB to consider optimal tolerance when used with the k-means algorithm. The comparison is based on the quality of partitioning (edge-cut and partition imbalance), the cost in terms of time, and the number of iterations. The quality of a graph partition is determined from the edge-cut between vertices in different clusters and the load imbalance, which could be based on the edge and vertex imbalance ratios of partitions depending on the application. Since we use unsymmetric matrices even though the adjacency matrices have to be symmetric, the eigenvalue problem can only solve the partitioning problem approximately.

The rest of the thesis is organized as follows: in Chapter 2, the main ingredients of graph partitioning and graph partitioning algorithms for sparse matrices are explained, and parallelization of the methods are given; in Chapter 3, the application problems and the software infrastructure used in this study are introduced. In Chapter 4, the results of numerical experiments are presented and discussed, and finally, in Chapter 5, conclusions are given with possible future work.

CHAPTER 2

PRELIMINARIES

This chapter summarizes some of the most well-known graph partitioning methods for partitioning a sparse matrix. We will focus only on row block partitioning of sparse matrices.

2.1 Graph Partitioning

Graphs are one of the useful tools for modeling a problem. Especially in solving a linear system of equations when the coefficient matrix is sparse, the graph representation is frequently used to reduce fill-in and partition the problem. To partition the coefficient matrix, its graph representation is partitioned first, and the solution of the linear system is obtained, hopefully in a shorter amount of time.

A graph can be represented as

$$G = (V, E),$$

where V and E correspond to the set of vertices and edges, respectively. Graphs can be divided into two classes: Undirected and directed graphs. If for each edge $(v_1, v_2) \in E$, there exists $(v_2, v_1) \in E$, where v_1 and v_2 are vertices in V, and $(v_1, v_2) = (v_2, v_1)$, then the graph is said to be undirected. Otherwise, it is said to be directed. If the graph is undirected, then the concept of degree of a vertex can be introduced: The degree of vertex $v \in V$ is the number of edges incident to the vertex [6]:

$$deg(v) = |(v, v') \in E, v' \in V|. \tag{2.1}$$

Moreover, the vertex weight function $\omega_V : V(G) \rightarrow \mathbb{R}$ maps all vertices onto the set of real numbers. The weight of a vertex v is denoted as $\omega_V(v)$.

After defining graphs, now, graph partitioning problems (GPP) can be introduced [21]: For an undirected graph $G = (V, E)$ with non-negative edge weights, $\omega : E \rightarrow \mathbb{R}_{>0}$, for partitioning G into $k \in \mathbb{N}_{>1}$ parts, the graph partitioning problem (GPP) asks for a partition Π of V with blocks of nodes $\Pi = (V_1, ..., V_k)$:

1. $V_1 \cup ... \cup V_k = V$,
2. $V_i \cap V_j = \emptyset, \quad \forall i \neq j$.

The aim of GPP is partitioning V into k almost equal parts while minimizing the number of edges connecting vertices in different parts. In many numerical computations, such as sparse matrix-vector multiplication, GPP is used to reduce the communication between parts. GPP is introduced when the graph is undirected, i.e., the original matrix is symmetric. On the other hand, if it is unsymmetric, it is symmetrized using the computation $|A| + |A|^T$. Therefore, the model is approximate at multiple levels.

Next, vertex separator and the edge-cut concepts can be discussed since the purpose of GPP is to minimize the edge-cut. A vertex seperator is a subset of vertices such that removing those vertices divides the graph into k disconnected graphs [52]. Further, an edge separator E' is a subset of edges such that removing those edges divides the graph into k disconnected graphs. The norm of E' is the number of edge-cut.

To find optimal partitioning of a graph, load balance constraints and the edge-cut should be optimized simultaneously. With those two constraints, the problem is not easy, and it is known to be NP-Complete [22].

There are two matrices having crucial parts in graph partitioning: adjacency and Laplacian matrices.

2.1.1 Adjacency Matrix

To describe the connectivity of the nodes in a graph, adjacency matrix Adj is used. Since it contains only data for the nodes, the size of this matrix is $|V| \times |V|$, where

$|V|$ is the number of vertices. The absolute value of the nonzero elements $Adj_{u,v}$ of Adj shows the weight of the edge connecting node u to node v. If $Adj_{u,v}$ is zero, then there is no edge connecting node u to node v [7].

2.1.2 Laplacian Matrix

As the adjacency matrix, the Laplacian matrix $L(G)$ of an undirected graph G also plays an important role in graph partitioning. As will be mentioned in Section 2.2, spectral graph partitioning is based on the eigenvectors of the Laplacian matrix of a graph.

The Laplacian matrix shows the distribution of the edges in G, and it is of size $|V| \times |V|$ with one row and a column for each node. A Laplacian matrix can be unweighted ($L_u(G)$) or weighted ($L_w(G)$). In this study, weighted Laplacian is formed in two steps: First, $L = D - A$, where D is the diagonal of A, is found. Then, the absolute row-sums of each row is written as the diagonal element for this row. Each nonzero element in the row is said to be the weight $W_{(u,v)}$ of edge, (u, v). If the Laplacian is unweighted, then instead of absolute row-sums, the number of nonzero elements in the row is used. The unweighted and weighted Laplacian matrices are mathematically defined as

$$L_u(G)(u,v) = \begin{cases} deg(u), & \text{if } u = v, \\ -1, & \text{if } u \neq v \text{ and } (u,v) \in E, \\ 0, & \text{otherwise,} \end{cases}$$

and

$$L_w(G)(u,v) = \begin{cases} \sum\limits_{(u,v)\in E} W_{(u,v)}, & \text{if } u = v, \\ -W_{(u,v)}, & \text{if } u \neq v \text{ and } (u,v) \in E, \\ 0, & \text{otherwise,} \end{cases}$$

respectively.

From the definition of the Laplacian matrix, it can be said that it is symmetric, positive semi-definite, and for $\mathbf{e}$, $L\mathbf{e} = 0$. Hence, if the graph is connected, i.e., there is a path from any point to any other point in the graph, L has the smallest eigenvalue 0 with the corresponding eigenvector $\mathbf{e}$ [47]. Since L is positive semi-definite, if

G is connected then the second smallest eigenvalue λ_2 is positive [34]. λ_2 is also called as *algebraic connectivity* [19]. The eigenvector x_2 associated with λ_2 (called *the Fiedler vector*) contains important directional information about the graph: "the components of x_2 are weights on the corresponding vertices of G such that differences of the components provide information about the distances between the vertices and the graph is partitioned based on their signs. Furthermore, to minimize the distances between vertices, they can be sorted by using the Fiedler vector [4].

2.2 Spectral Graph Partitioning

One of the most common graph partitioning algorithms is based on the spectral bisection algorithm, called spectral graph partitioning [4]. The spectral bisection algorithm is implemented by using various eigensolver algorithms such as iterative methods (e.g., Lanczos) and Rayleigh Quotient Iteration (RQI) [23]. Spectral partitioning can be considered to be analogous to vibrating strings in physics. A string can be thought of as nodes connected by edges in one dimensional graph. In the case of the string being stable, the eigenvalue of the matrix L associated with the graph G becomes zero. Since the Laplacian is positive, this eigenvalue is smallest. During vibration, the eigenvalue is the second smallest one of L, and the eigenvector corresponding to this eigenvalue is called as *Fiedler vector*. To define algebraic connectivity of the graphs, Corollaries 1 and 2 are given below. If the vector's corresponding component is negative, then the node will be placed in the partition $N-$; otherwise, in $N+$.

Corollary 1. [20, Corollary 3.5] Let $G = (N, E)$ be connected and when the node becomes negative (positive), label nodes as $N-$ ($N+$). Then $N-$ is connected. If no $v_2(n) = 0$, then $N+$ is also connected.

Corollary 2. [20, Corollary 3.9] Let $G_1(N, E_1)$ be a subgraph of $G(N, E)$, so that G_1 is "less connected" than G. Then $\lambda_2(L(G_1)) \leq \lambda_2(L(G))$, i.e., the algebraic connectivity of G_1 is less than or equal to the algebraic connectivity of G.

2.3 Multilevel Graph Partitioning

To reduce the computation time of the solution of the graph partitioning problem, one can split the problem into a sequence of bisection steps. Partitioning can be done by dividing the graph into two pieces first and then bisecting the two subpieces independently and continue this process recursively. By this, an arbitrary number of almost equal-sized sets can be generated. However, since it does not reduce complexity, three stages of partitioning a graph in multiple levels are introduced in [9]: coarsening, partitioning, and uncoarsening, see Algorithm 1.

Algorithm 1 Multilevel Algorithm for Graph Partitioning [32]

while graph is not small enough **do**
  coarsen the graph.
end while
Partition graph.
while original graph is not achieved **do**
  uncoarsen the graph.
  uncoarsen the partitioning.
  locally refine partition if desired.
end while

2.3.1 Multilevel Kernighan-Lin Algorithm

One of the multilevel algorithms used in graph partitioning is the Multilevel Kernighan-Lin (KL) algorithm [35]. During the uncoarsening phase of the multilevel algorithm, Fiduccia and Mattheyses linear-time version [18] of KL (KLFM) refinement is used have better initial partition for the KL. This process is done because, since KL has a better initial partition, a smaller edge-cut can be achieved in fewer iterations.

The aim of the Kernighan-Lin algorithm [41] is to partition the vertices into two disjoint subsets A and B of (almost) equal size, such that the edge-cut between subsets are minimized. One of the most important advantages of this algorithm is that KL is more cost-effective than the bisection algorithm if a good initial partitioning is given. The pseudo-algorithm for KL is given in Algorithm 2 [51].

Algorithm 2 The pseudo-code for Kernighan-Lin algorithm for $G(V, E)$ [51]

Require: Initial partitioning of vertices into A and B.

Compute D values for all a in A and b in B.

Let gv, av, and bv be empty lists.

for $n = 1$ to $|V|/2$ **do**

Find a from A and b from B, such that $g = D[a] + D[b] - 2 \times c(a, b)$ is maximal.

Remove a and b from further consideration in this pass.

Add g to gv, a to av, and b to bv.

Update D values for the elements of $A = A\ a$ and $B = B\ b$.

end for

Find k which maximizes g_max, the sum of $gv[1], ..., gv[k]$.

while $g_max > 0$ **do**

Exchange $av[1], av[2], ..., av[k]$ with $bv[1], bv[2], ..., bv[k]$.

end while

return return $G(V, E)$.

CHAPTER 3

APPLICATIONS AND THE SOFTWARE INFRASTRUCTURE

In this chapter, the application problems with their properties and the software infrastructure used to partition the matrices arising from these problems are introduced.

The concept of parallel computing is solving a computational problem using multiple computational resources simultaneously. However, this also introduces the need for communication (or synchronization in shared address space platforms). By breaking down the problem into parts so that each one can be solved concurrently, solution time is reduced. Concurrent solution requires significant coordination. There are several ways to exchange data between processors, such as through a shared memory bus or over a network. In this thesis, the distributed memory model is used with the MPICH implementation [25] of the Message Passing Interface (MPI) library, which is used to coordinate the communication between processors.

In this study, three different graph partitioning algorithms are used: Spectral partitioning algorithm [4] with the k-means clustering [43] (SPEC), ParMETIS [40] (uses Multilevel Kerninghan-Lin Algorithm), and CHACO [32] (uses recursive spectral partitioning algorithm). Spectral partitioning algorithm is implemented in C language and PETSc [3], SLEPc [53] libraries, whereas CHACO and ParMETIS are implemented in C language and called from PETSc. Further, to compare load imbalance and the number of iterations based on the eigensolver's tolerance, MATLAB [46] is used to partition the graphs with SPEC.

3.1 Matrices

In the literature, there are studies based on symmetric sparse matrix partitioning comparisons based on CHACO, ParMETIS, Multilevel Kerninghan-Lin, and spectral partitioning methods. For the details of those comparisons for symmetric matrices, see [14, 24, 47, 57] and references therein.

Even though there are hypergraph partitioning models [12] for directed graphs, there are not enough studies evaluating the accuracy of the classical graph partitioning of unsymmetric matrices via graph partitioning models. From the adjacency matrix definition, the structure must be symmetric, and hence, only an approximate partitioning can be obtained. Matrices' sizes, symmetric structures, types of their elements, and problem kinds are given in Table 3.1. Except for Poisson(5) and Poisson(12), all matrices are obtained from the University of Florida Sparse Matrix Collection [15]. Poisson(5) and Poisson(12) are derived from the 5-point stencil solution of the Poisson equation in the domain of $[0, 1]$ with step size $1/(m + 1)$, where $m = 5$ and $m = 12$, respectively, and since they are not taken from the Collection, their types are denoted as (-).

In this thesis, matrices having a size less than 100×100 are used to visualize original and partitioned graphs by using GraphViz [17]. The ones having a size between 100×100 and 1000×1000 are used to compare the load imbalance, edge-cut, and the number of iterations based on the tolerance of eigensolver in MATLAB when SPEC is used. Finally, the larger ones are used to compare the algorithms used in this study in computational time and edge-cut.

Figures 3.1 - 3.4 show the small-sized matrices with their spy plots and graph representations in Table 3.1 before partitioning. From the spy representations, their vertex distribution can be observed, whereas, from their graphs, connections of the edges between vertices can be seen. Lastly, from the spy plots, their sparsity structures can be inspected.

Table 3.1: Properties of the sparse matrices [15]

NAME	SIZE	SYMM	TYPE	KIND
Ragusa18	23×23	NO	INTEGER	Directed Weighted Graph
can_24	24×24	YES	BINARY	Structural Problem
GD01_b	18×18	NO	BINARY	Directed Graph
Poisson(5)	25×25	YES	INTEGER	-
Poisson(12)	144×144	YES	INTEGER	-
cz148	148×148	NO	REAL	2D/3D Problem
can_161	161×161	YES	BINARY	Structural Problem
lshp_265	265×265	YES	BINARY	Thermal Problem
FEM_3D_thermal1	17880×17880	NO	REAL	Thermal Problem
bcspwr10	5300×5300	YES	BINARY	Power Network Problem
epb2	25228×25228	NO	REAL	Thermal Problem
sme3Da	12504×12504	NO	REAL	Structural Problem
rw5151	5151×5151	NO	REAL	Statistical/Mathematical Problem
Zhao1	33861×33861	NO	REAL	Electromagnetics Problem
ns3Da	20414×20414	NO	REAL	Computational Fluid Dynamics Problem
poisson3Db	85623×85623	NO	REAL	Computational Fluid Dynamics Problem
chem_master1	40401×40401	NO	REAL	2D/3D Problem
av41092	41092×41092	NO	REAL	2D/3D Problem

$$36 \times \begin{bmatrix}
-4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -4 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -4
\end{bmatrix}$$

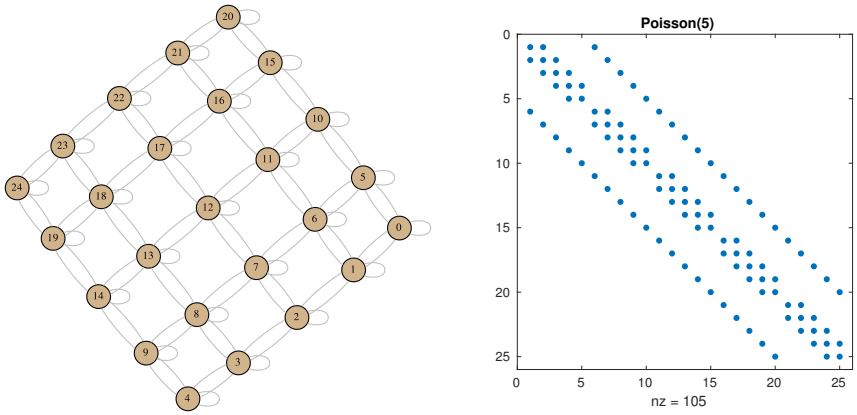


Figure 3.1: Matrix, graph, and spy representations of *Poisson(5)*

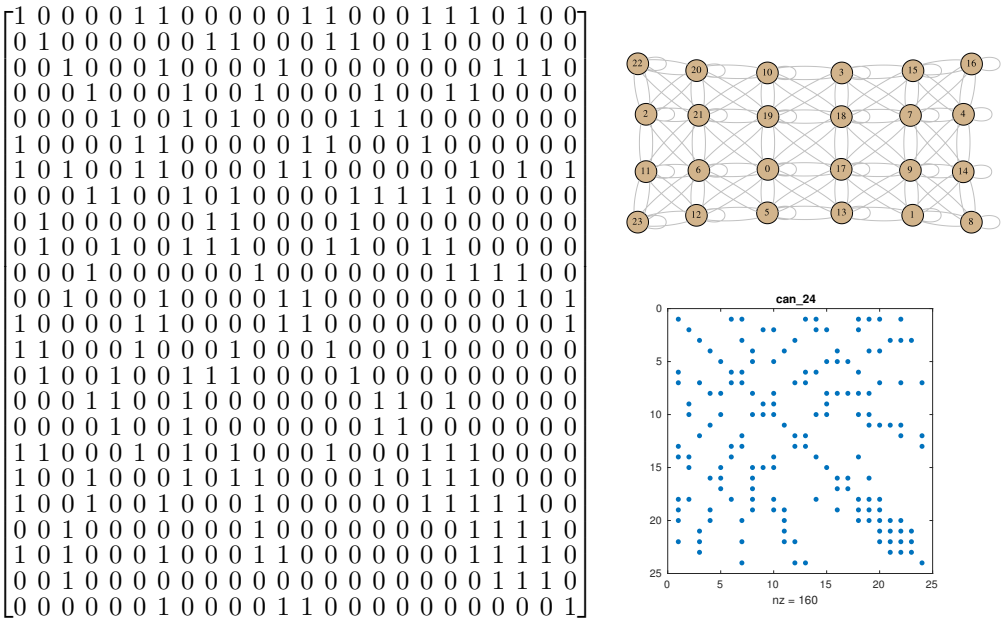


Figure 3.2: Matrix, graph, and spy representations of *can_24*

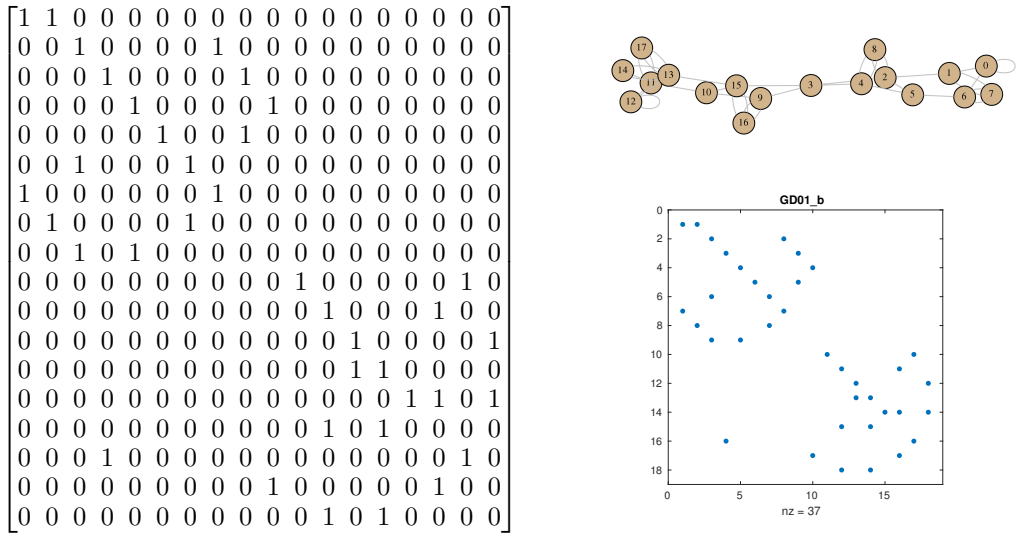


Figure 3.3: Matrix, graph, and spy representations of *GD01_b*

3.2 Comparison Metrics

This study's comparison metrics are determined as the edge-cut, partition imbalance, the number of iterations required for the eigensolver, and elapsed wall-clock time. The edge-cut and load imbalance give information about partitioning quality (the

$$\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 2 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 2 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{bmatrix}$$

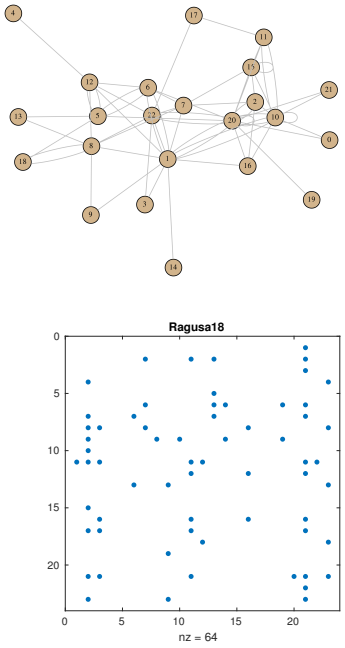


Figure 3.4: Matrix, graph, and spy representations of *Ragusa18*

smaller edge-cut and imbalance, the better the partitioning becomes for solving the linear system).

The computation of the edge-cut for all algorithms is given in Figure 3.5. If the matrix is symmetric, then the sum is halved since the graph is undirected, and the values are repeated in transposed blocks.

The load imbalances are obtained by the edge and vertex imbalance ratios of partitions. The computations of edge and vertex imbalances are given in Figure 3.6 and Figure 3.7, respectively. If the ratio is one, then the perfect imbalance is achieved.

For comparison, three different algorithms are used: SPEC, Multilevel KL algorithm by ParMETIS, and recursive spectral partitioning algorithm by CHACO. Unweighted (weighted) Laplacian is used for unweighted (weighted) partitioning in the spectral partitioning algorithm when tolerances of the eigensolver are compared.

The comparisons are also made based on the eigensolver tolerances to find the optimal tolerance for partitioning. In the spectral partitioning algorithm, each matrix is partitioned for 2, 4, 8, and 16 parts with eigensolver tolerances 10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, and 10^{-10}. The default subspace dimension in SLEPc is set to 17 since the par-

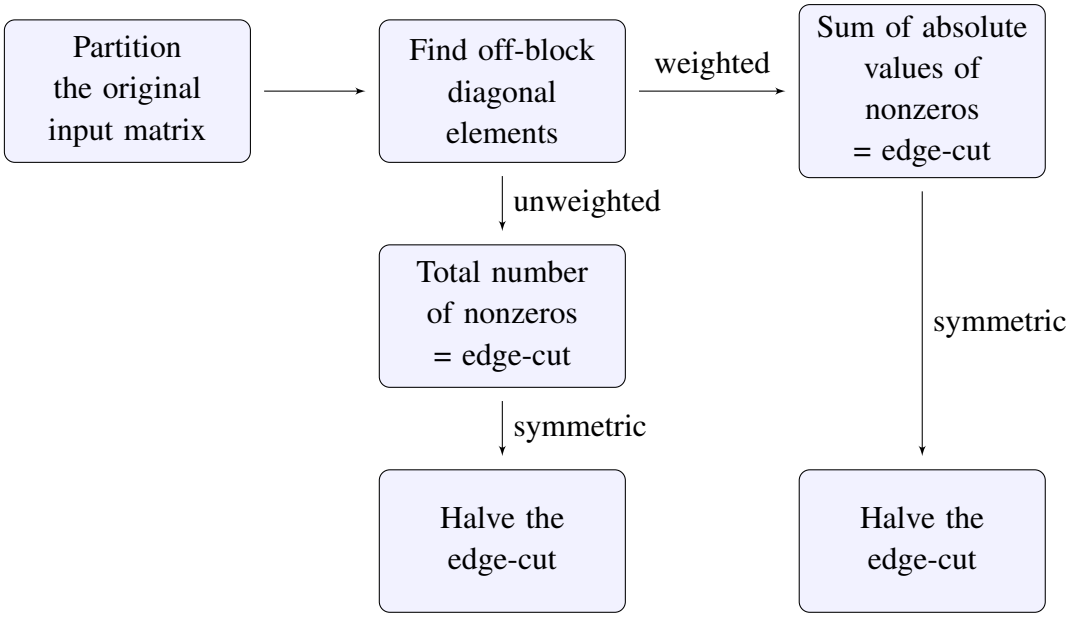


Figure 3.5: The flowchart of the edge-cut computation

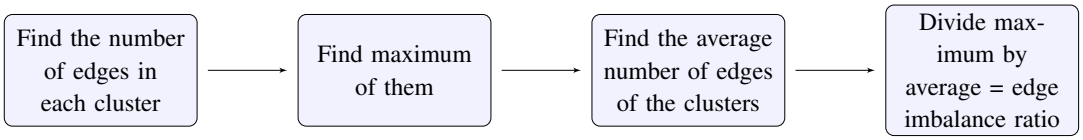


Figure 3.6: The flowchart of the edge imbalance ratio computation

titioning is done up to 16. However, for a subspace dimension, 17 is considered quite high for Krylov subspace methods. Therefore, changing the stopping criterion does not affect the quality of the partitions much, as observed later in Section 4. Lastly, the elapsed time is measured, starting after checking symmetry and ending after finding edge-cut.

3.3 Software Libraries

This section explains software libraries we use for this study: ParMETIS and CHACO are used as graph partitioning libraries, whereas PETSc is a scientific parallel application development environment, and SLEPc is a parallel eigensolver library. At the end of the section, we also explain the functions we used in MATLAB during the study.

ParMETIS is an MPI-based parallel library implementing various graph partitioning

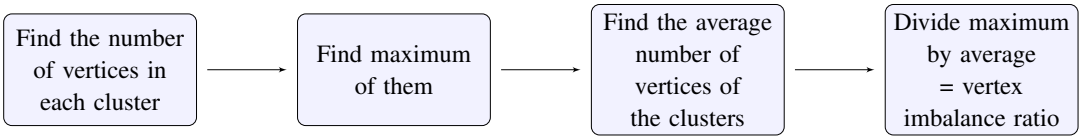


Figure 3.7: The flowchart of the vertex imbalance ratio computation

algorithms. ParMETIS_V3_PartKway is the routine in this library that is used to partition unstructured graphs. This routine takes a graph as an input. It computes a k-way partitioning while attempting to minimize the edge-cut and, at the same time, maintaining load balance (within a percentage of a user-defined parameter). The parallel graph partitioning algorithm used in this routine is based on the sequential multilevel k-way partitioning algorithm. The main advantage of ParMETIS is that it is being based on multilevel partitioning. Using a multilevel approach instead of recursive partitioning, a various number of partitions can be done instead of 2^n. Moreover, since it operates in parallel, using ParMETIS is advantageous when the large-scale numerical simulations are made. On the other hand, using Multilevel KL may not always be the best partitioning approach, since it is a local method. Depending on the graph structures, a global partitioning strategy can be more effective in partitioning a graph in terms of the edge-cut.

CHACO is also a software package designed to partition graphs [32]. Instead of multilevel algorithms, CHACO partitions the graphs recursively by using several methods such as inertial, spectral, KL. These approaches can partition the graph into two, four, or eight pieces at each recursion level. In this study, the spectral partitioning method is used with CHACO to compare the quality of partitioning and time with our parallel implementation of spectral partitioning using PETSc and SLEPc with the k-means clustering algorithm. Moreover, it is used to analyze the quality of the partitions in terms of the edge-cut when it is compared to ParMETIS. However, CHACO's main disadvantage is unlike ParMETIS, CHACO operating sequentially, and using recursive bisection instead of multilevel, which can cause CHACO to perform slower.

PETSc is a library for the implementation of large-scale applications in parallel or serial. For parallel implementation, PETSc uses MPI. Unfortunately, the number of processes being equal to the number of partitioning is a limitation of PETSc [3].

To solve large sparse eigenproblems in parallel, a general library, SLEPc [53], is also

used in this study. This software library can be considered an extension of PETSc, for solving eigenvalue problems.

For ParMETIS and CHACO, the input matrix A is considered as the parallel adjacency matrix of a graph [3]. Since the adjacency matrix should be symmetric as it is defined in Section 2.1, the symmetry of A should be checked first. If A is not symmetric, the algorithm uses $(|A| + |A|^T)$ to achieve symmetry. Then the partitioning is applied by the chosen algorithm.

MATLAB [46] is a numerical computing environment for matrix operations, plotting functions and implementation of algorithms. In this study, MATLAB is used to inspect the load imbalance of the partition by using SPEC, and the number of iterations is done to compute the eigenvalues. The eigenvalues are computed iteratively using the MATLAB function `eigs`, whereas the k-means clustering algorithm is applied by `kmeans`.

3.4 Spectral Graph Partitioning (SPEC)

Similarly, in the spectral partitioning algorithm, after checking the input matrix if it is symmetric or not, depending on user input, the weighted or unweighted Laplacian matrix L of the graph G having adjacency matrix A is obtained. If A is not symmetric, the algorithm uses $(|A| + |A|^T)$ to achieve symmetry. In the weighted case, the absolute value of the nonzeros is considered as the weights.

In the spectral graph partitioning algorithm defined in Section 2.2, based on the given number of processes $nproc$ and eigensolver tolerance given by the user, by calling SLEPc library, $\lambda_2, \lambda_3, \ldots, \lambda_{nproc+1}$ are found, where $0 = \lambda_1 < \lambda_2 < \lambda_3 < \cdots < \lambda_{nproc+1}$ are eigenvalues of the Laplacian matrix. There are many eigensolvers implemented in parallel in SLEPc. Power iteration and Krylov subspace methods are the most popular eigensolver methods for solving large sparse systems [44]. In [53], these techniques are not recommended due to the complexity of problems. Instead, the Krylov-Schur method is recommended because it is considered to be more efficient. Krylov-Schur can be considered a variation of Arnoldi/Lanczos algorithms [58] with effective restarting techniques [53]. We also investigate that using the Lanc-

zos algorithm as the eigensolver is not feasible for low tolerances since the number of converged eigenvalues is less than the number of partitions. Thus, Krylov-Schur method [59] is used as the eigensolver in this thesis.

After finding eigenvalues and the corresponding eigenvectors, an unnormalized spectral clustering algorithm in [44] is used with the k-means clustering algorithm presented in [11]. The pseudo-code for the k-means algorithm is given in Algorithm 3 [56]. A centroid of a cluster (cluster center) is a data point representing the center of the cluster. This data point can be imaginary or real. The original input matrix is permuted by using clustering information. This step is crucial for the approximation of partitioning the unsymmetric input matrix. After the permutation, the edge-cut is computed as described in Section 2.1.

Algorithm 3 The pseudo-code for k-means clustering algorithm [56]

Require: k (the number of clusters), D (data points)

Initialize k centroids randomly.

Associate each data point in D with the nearest centroid.

Recompute the position of centroids.

Repeat steps 2 and 3 until there are no more changes in the membership of the data points.

return Data points with cluster memberships.

A graph partitioning can be unweighted or weighted. If it is weighted, then as it is explained in Section 3.2, weights of edges will be considered. If not, then all edge weights will be assumed to be one. In this study, both cases are studied to have a reliable comparison between partitioning libraries and the proposed algorithm. A flowchart of the proposed algorithm is given in Figure 3.8.

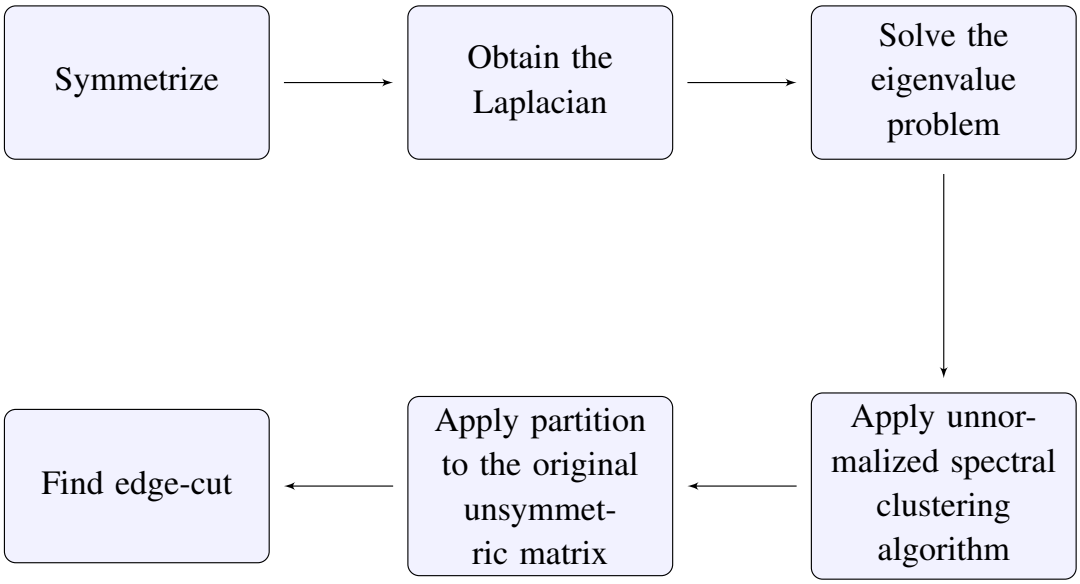


Figure 3.8: The flowchart of the proposed algorithm

CHAPTER 4

NUMERICAL RESULTS

In this chapter, the numerical results are obtained from three different graph partitioning algorithms with various libraries: SPEC, ParMETIS [40] (uses Multilevel Kerninghan-Lin Algorithm), and CHACO [32] (uses recursive spectral partitioning). Furthermore, MATLAB is used for comparison of load imbalance, edge-cut, and the number of iterations. The C codes for calling these algorithms within PETSc can be found in Appendix A, the numerical results in Appendix B, and figures of the partitioned small-sized matrices in Appendix C.

Even though we specifically target unsymmetric matrices for the comparison of the algorithms, in the numerical experiments, we include examples of symmetric ones as well. In the latter case, there is no need for symmetrizing the matrix. They are included as a baseline in order to observe the contribution of the symmetrizing step. Those symmetric matrices are shown in Table 3.1.

The computer environments used for numerical experiments in this study are introduced in Table 4.1. Greyfurt2 is located at the Department of Computer Engineering, Middle East Technical University. To compare large-sized matrices, 16 cores in Greyfurt2 are used for operating the partitioning with CHACO, ParMETIS, and spectral partitioning with the k-means clustering, whereas, for medium-sized matrices, 2 cores in TOSHIBA L50-C-172 are used in MATLAB.

Table 4.1: Computer environments

	OS	Processors	RAM	# Threads	# Cores	# Processors
Greyfurt2	Linux 5.3.0	AMD Opteron 6376	64GB	64	64	4
TOSHIBA L50-C-172	Linux 4.13.16	Intel i5-5200U	8GB	4	2	1

4.1 Load Imbalance and The Number of Iterations

In this section, edge-cut, edge, vertex imbalances, and the number of iterations required for the eigensolver are inspected using MATLAB. Several eigensolver tolerances are also used in this part to obtain the best choice. For comparison, the subspace dimension in MATLAB `eigs` function is set to be $nparts + 3$, where $nparts$ is the number of partitions. The maximum number of iteration is set to be 1000, and $nparts + 1$ eigenvalues are requested.

The numerical results of the number of iterations, edge-cut, edge and vertex imbalance ratios obtained from the partitioning of medium sized matrices can be found in Tables B.1 - B.4, whereas bar graphs representing the results are depicted in Figures 4.1 - 4.8.

Comparisons are interpreted as follows. The edge-cut gives an upper bound for the communication volume when a matrix-vector multiplication is performed. We also define a new metric called *the computational imbalance*. It is defined based on the context. For example, if only matrix-vector multiplications are performed, it is only the edge imbalance, and if only inner products are performed, it is only the vertex imbalance. Moreover, the number of inner products and matrix vector multiplications per iteration vary. Therefore, both inner products and matrix-vector multiplications are required in an iterative solver and the weights depend on the iterative solver. Hence, to solve sparse linear systems iteratively, the computational imbalance is a weighted combination of both edge and vertex imbalance.

4.1.1 Symmetric Medium-Sized Matrices

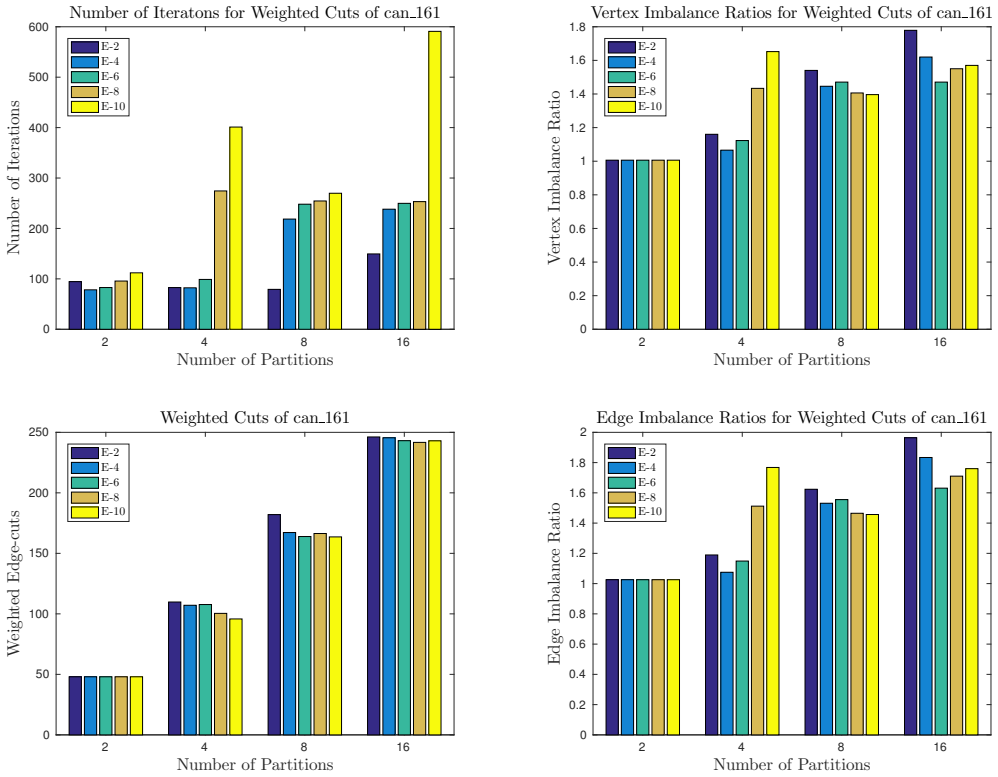


Figure 4.1: Results obtained from the weighted partitioning of *can_161*

When Figures 4.1 and 4.2 are observed, it is seen that partitioning of *can_161* into two gives the same result through the vertex and edge imbalance ratios, and edge-cut, whether the partitioning is weighted or not. Additionally, using 10^{-4} as tolerance requires the smallest number of iterations for partitioning into two. Furthermore, for any use of Laplacian, if the edge-cuts are considered, it is seen that for four and eight partitions, using 10^{-10} tolerance is the best choice for *can_161*. On the other hand, if the metric is edge and vertex imbalance, then using a larger tolerance becomes beneficial.

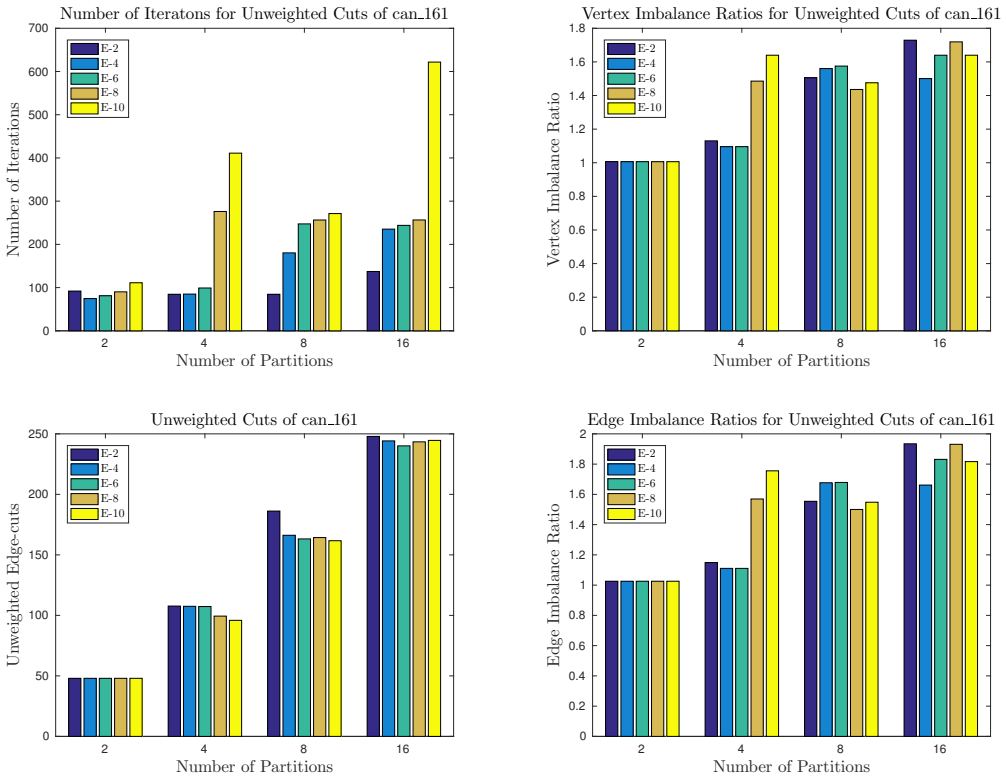


Figure 4.2: Results obtained from the unweighted partitioning of *can_161*

From Figures 4.3 and 4.4, it is inspected that when *lshp_265* is partitioned into two with any Laplacian, the same partitioning is obtained for each tolerance since all values are the same, except the number of iterations. It is seen that, for this case, 10^{-4} tolerance should be used to partition the matrix since its eigenvalues converge faster than those for any other tolerances. From the figures, it is also seen that when *lshp_265* is partitioned into eight, the smallest tolerance should be used for vertex and edge imbalance, whereas 10^{-8} should be used for less edge-cut for both partitions. Lastly, if the matrix is partitioned into sixteen, then for a reasonable imbalance ratio, 10^{-8} should be used as a tolerance if weighted Laplacian is used, while 10^{-2} will be enough for the unweighted case.

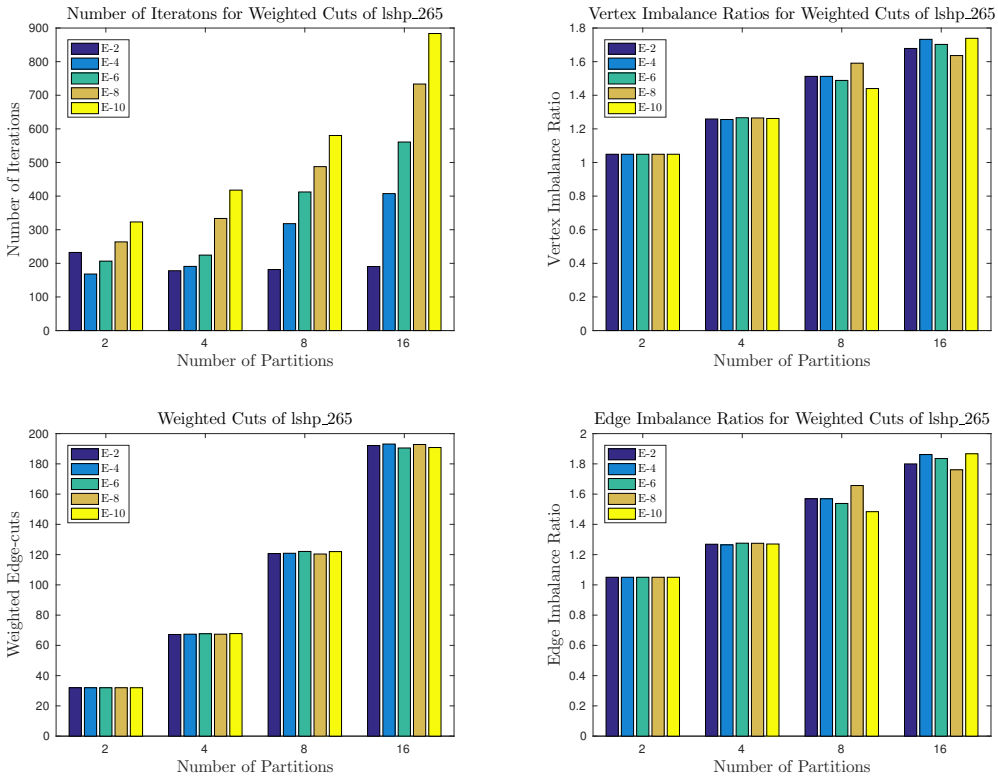


Figure 4.3: Results obtained from the weighted partitioning of *lshp_265*

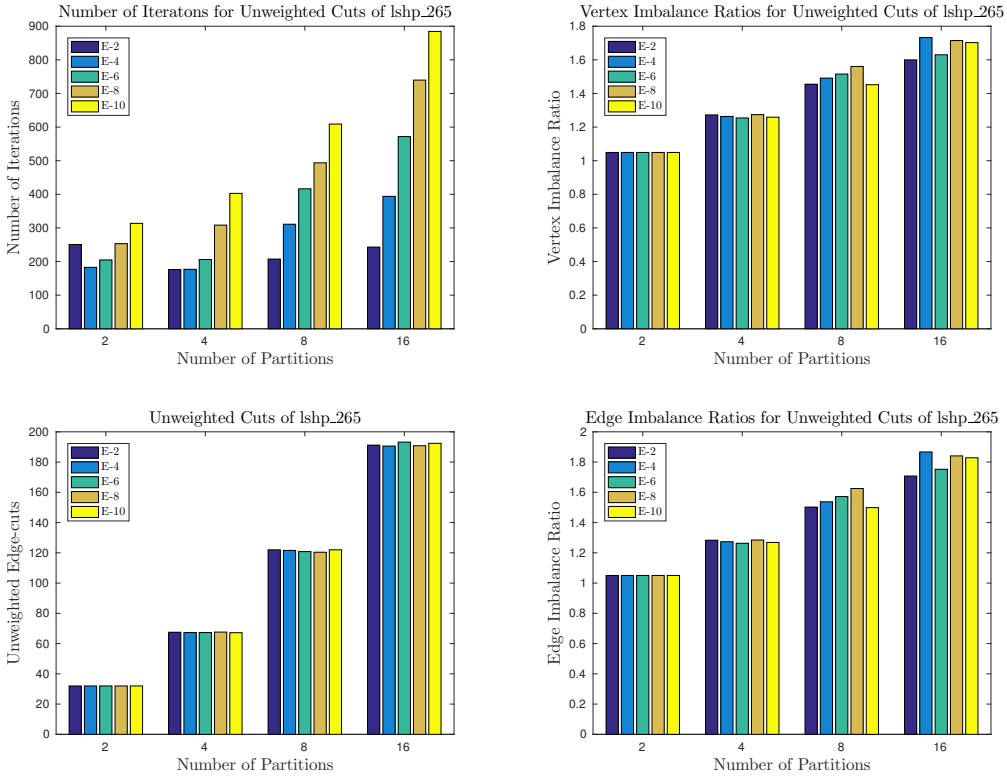


Figure 4.4: Results obtained from the unweighted partitioning of *lshp_265*

From Figures 4.5 and 4.6, it is seen that when weighted Laplacian is used to partition *Poisson(12)* up to sixteen, vertex and edge imbalance ratios are the closest to one when tolerance is the smallest. However, ratios become closer to one when tolerance is the highest if the partition number is raised. When the unweighted case is considered, using the largest tolerance gives smaller edge-cut and more balanced partitions. It is also seen that, when the matrix is partitioned into four, the load is completely balanced if 10^{-10} is chosen for tolerance since both vertex and edge imbalance ratios are 1, whether the Laplacian is weighted or not. In the same conditions, the edge-cut also becomes the smallest. Last but not least, when four or more partitions are required to cluster *Poisson(12)*, the largest tolerance requires the smallest number of iterations for all partitions, and this number increases dramatically as tolerance decreases, for both unweighted and weighted partitionings.

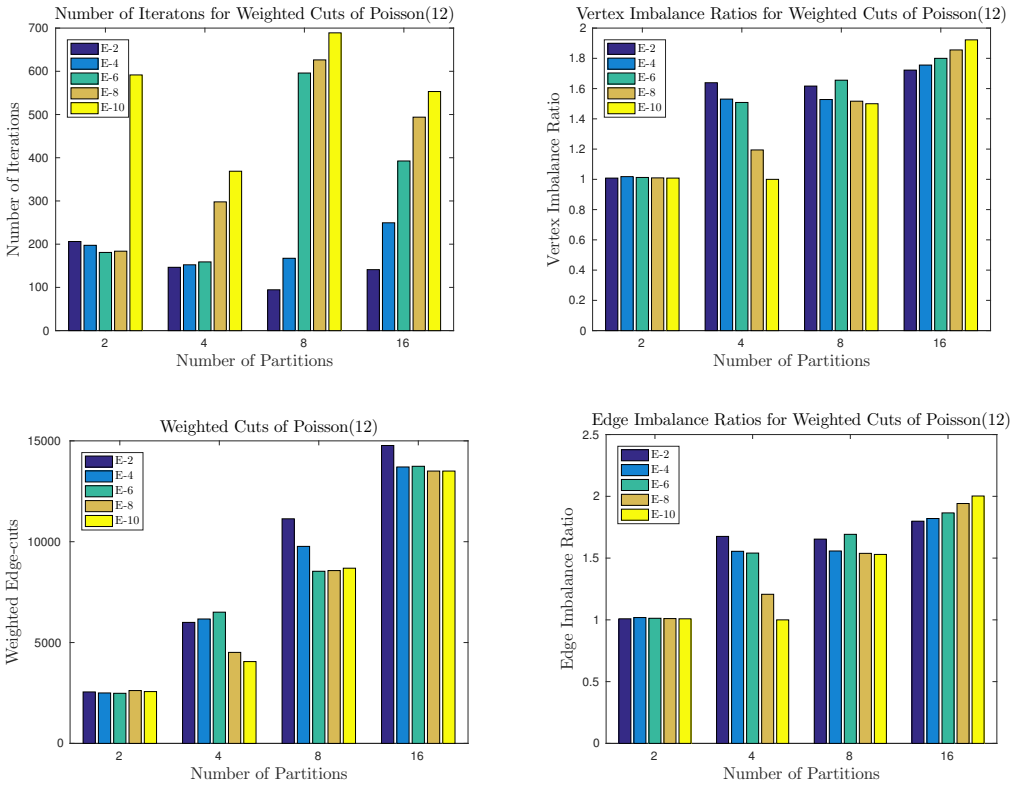


Figure 4.5: Results obtained from the weighted partitioning of *Poisson(12)*

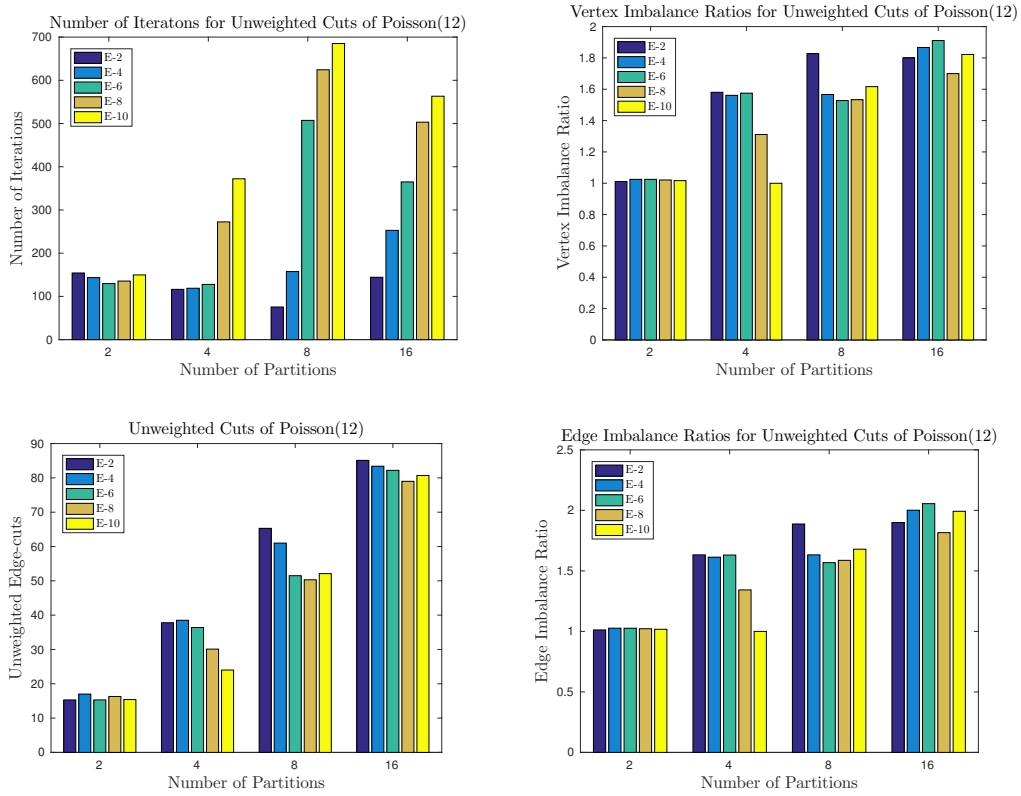


Figure 4.6: Results obtained from the unweighted partitioning of *Poisson(12)*

4.1.2 Unsymmetric Medium-Sized Matrices

From Figures 4.7 and 4.8, it is observed that when weighted Laplacian is used for partitioning *cz148* into two, the same partitioning is made for each tolerance since except the number of iterations, all values are the same although tolerance changes. However, the least number of iterations is achieved when it is 10^{-6}, whereas, in the unweighted case, 10^{-4} tolerance yields the least number. Moreover, if *cz148* is partitioned into eight and unweighted Laplacian is used, then for more balanced partitions, tolerance should be decreased to 10^{-10}. It is also observed that when unweighted Laplacian is used for partitioning the matrix into sixteen, both vertex and edge imbalance ratios become higher than two.

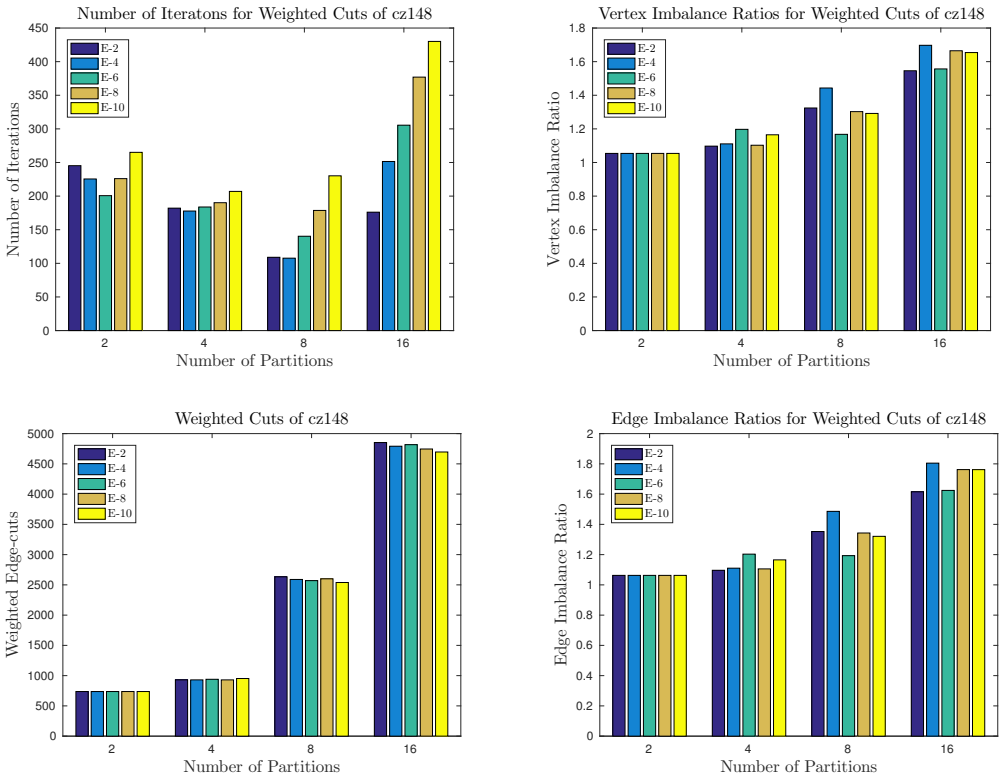


Figure 4.7: Results obtained from the weighted partitioning of *cz148*

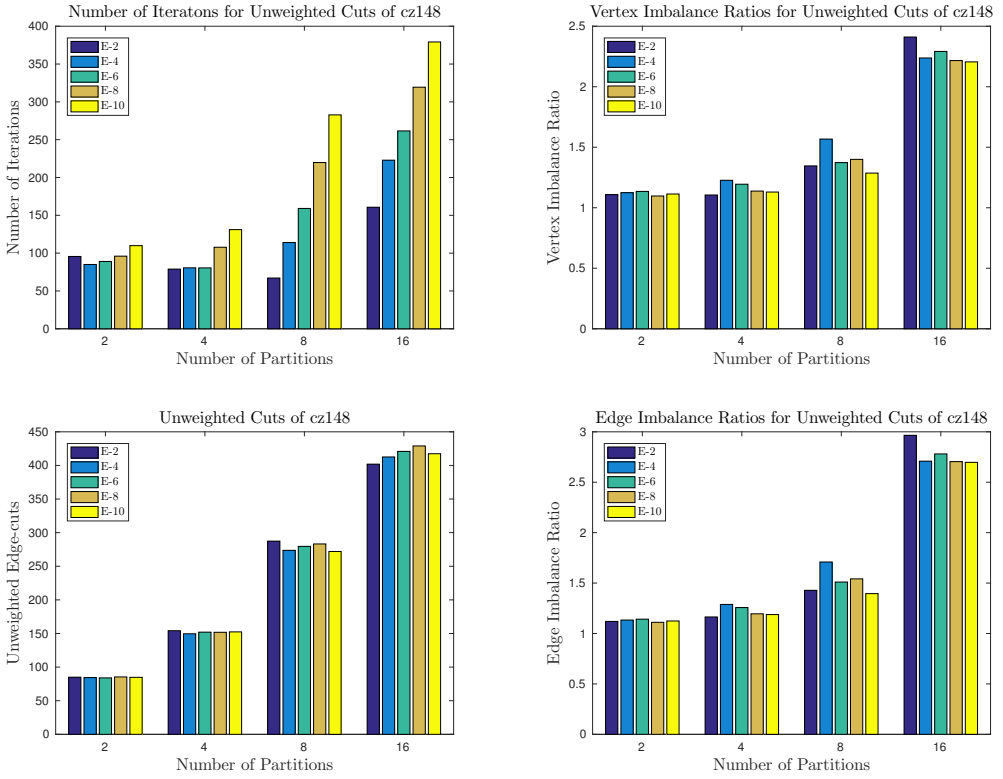


Figure 4.8: Results obtained from the unweighted partitioning of *cz148*

4.1.3 Results

Based on the numerical results obtained from the weighted (wgh) and unweighted (uwg) partitioning of 4 medium-sized matrices in terms of load imbalance, edge-cut, and the number of iterations, Tables 4.2-4.4 present the best tolerances for `eigs` to partition each matrix, respectively.

Table 4.2: The best tolerance for the eigensolver of eigs routine in MATLAB in terms of load imbalance when the medium-sized matrices are partitioned into 2, 4, 8, and 16 by using weighted (wgh) and unweighted (uwg) Laplacian.

nparts	Tolerance	cz148		Poisson(12)		lshp_265		can_161	
		wgh	uwg	wgh	uwg	wgh	uwg	wgh	uwg
2	10^{-2}	+			+	+	+	+	+
	10^{-4}	+				+	+	+	+
	10^{-6}	+				+	+	+	+
	10^{-8}	+	+			+	+	+	+
	10^{-10}	+		+		+	+	+	+
4	10^{-2}	+	+						
	10^{-4}					+		+	
	10^{-6}						+		+
	10^{-8}								
	10^{-10}			+	+				
8	10^{-2}								
	10^{-4}								
	10^{-6}	+			+				
	10^{-8}								+
	10^{-10}		+	+		+	+	+	
16	10^{-2}	+		+			+		
	10^{-4}								+
	10^{-6}							+	
	10^{-8}				+	+			
	10^{-10}		+						

Table 4.3: The best tolerance for the eigensolver of eigs routine in MATLAB in terms of the edge-cut when the medium-sized matrices are partitioned into 2, 4, 8, and 16 by using weighted (wgh) and unweighted (uwg) Laplacian.

nparts	Tolerance	cz148		Poisson(12)		lshp_65		can_61	
		wgh	uwg	wgh	uwg	wgh	uwg	wgh	uwg
2	10^{-2}	+			+	+	+	+	+
	10^{-4}	+				+	+	+	+
	10^{-6}	+				+	+	+	+
	10^{-8}	+	+			+	+	+	+
	10^{-10}	+		+		+	+	+	+
4	10^{-2}	+	+						
	10^{-4}					+		+	
	10^{-6}						+		+
	10^{-8}								
	10^{-10}			+	+				
8	10^{-2}								
	10^{-4}								
	10^{-6}	+			+				
	10^{-8}								+
	10^{-10}		+	+		+	+	+	
16	10^{-2}	+	+	+			+		
	10^{-4}								+
	10^{-6}							+	
	10^{-8}				+	+			
	10^{-10}								

Overall, from the numerical results, it is seen that the number of iterations usually increases as the number of partitions increases. However, as seen in Figures 4.1, 4.2, 4.5, and 4.6, *can_161* and *Poisson(12)* have some exceptions. When *can_161* is partitioned into four, finding eigenvalues when 10^{-8} and 10^{-10} are used as tolerances requires more iterations than those when eight partitions are made. Similarly, partitioning *Poisson(12)* with a tolerance of 10^{-10} needs more iteration than partitioning the matrix into four or sixteen. When eight partitions are made, only 10^{-2} or 10^{-4} should be used as tolerance.

It is seen that 2 of the 4 matrices yield more balanced results when the tolerance is 10^{-10}. For *cz148*, 10^{-2} tolerance is enough to have more balanced partitions, whereas for *can_161*, 10^{-4} tolerance should be used. Moreover, the best tolerance for better partitions (the least edge-cut) depends on the matrix. *cz148* achieves better partitions

Table 4.4: The best tolerance for the eigensolver of eigs routine in MATLAB in terms of the number of iterations when the medium-sized matrices are partitioned into 2, 4, 8, and 16 by using weighted (wgh) and unweighted (uwg) Laplacian.

nparts	Tolerance	cz148		Poisson(12)		lshp_65		can_61	
		wgh	uwg	wgh	uwg	wgh	uwg	wgh	uwg
2	10^{-2}		+			+	+	+	+
	10^{-4}								
	10^{-6}	+		+	+				
	10^{-8}								
	10^{-10}								
4	10^{-2}		+	+	+	+	+		+
	10^{-4}	+						+	
	10^{-6}								
	10^{-8}								
	10^{-10}								
8	10^{-2}		+	+	+	+	+	+	+
	10^{-4}	+							
	10^{-6}								
	10^{-8}								
	10^{-10}								
16	10^{-2}	+	+	+	+	+	+	+	+
	10^{-4}								
	10^{-6}								
	10^{-8}								
	10^{-10}								

when the tolerance is 10^{-2}, the two of them achieve the least edge-cut when the tolerance is decreased up to 10^{-10}, and *can_161* gets better partitions for various tolerances as the number of partitions varied.

When the comparison is made based on the number of partitions, it is observed that when unweighted partitioning is made, using the largest tolerance is enough to achieve a more balanced partition when the matrices are divided into two. However, in the weighted case, the tolerance should be decreased to 10^{-10} for *Poisson(12)*. The rest can be partitioned in a more balanced way when 10^{-2} is used as tolerance. Moreover, to partition the matrices into two in the least number of iterations, 10^{-4} tolerance is used for 3 matrices for the unweighted case, whereas it is decreased to 10^{-6} for *Poisson(12)*. If the partitioning is weighted, then the tolerance is decreased to 10^{-6} for *cz148*. Further, for 3 of the matrices, 10^{-2} is a sufficient tolerance for

the least edge-cut whether the partitioning into two is unweighted or not. For *Poisson(12)*, using 10^{-10} tolerance is beneficial if the partitioning is weighted. Lastly, when *cz148* is partitioned by using unweighted Laplacian, 10^{-8} tolerance should be used for less edge-cut.

When 4 matrices are partitioned into four, 2 of them give more balanced partitions when the tolerance is 10^{-4} for the weighted case and 10^{-6} for the weighted case. *cz148* needs to have 10^{-2}, and *Poisson(12)* needs to have 10^{-10} tolerance, whether the partitioning is weighted or not. If the number of iterations is considered, then all matrices are partitioned in the least iteration when the tolerance is the highest if the partitioning is unweighted. However, if it is weighted, 2 of them need 10^{-4} tolerance for fewer iterations. Last but not least, for any type of partitioning, *cz148* gives a less edge-cut when the tolerance is 10^{-2}, *Poisson(12)* gives when it is 10^{-10}. The remaining 2 matrices give the least edge-cut when the tolerance is 10^{-4} when weighted partitioning is made, whereas it is 10^{-6} when the partitioning is unweighted.

As the number of partitions increased to eight, optimum tolerance starts to decrease. To obtain more balanced partitions in weighted partitioning, *cz148* needs 10^{-6}, while the rest need to have 10^{-10} tolerance. When the partitioning is unweighted, *Poisson(12)* yields more balanced partitions when the tolerance is 10^{-6}, whereas *can_161* needs a tolerance of 10^{-8}. Moreover, 3 of 4 matrices give the least number of iterations when tolerance is the highest. When weighted partitioning is made, *cz148* is partitioned in less number of iterations when the tolerance is 10^{-4}. While *Poisson(12)* gives the less edge-cut in the tolerance of 10^{-6} when unweighted partitioning is made, *can_161* gives it when the tolerance is 10^{-8}. The remaining 2 matrices obtain less edge-cut when the tolerance is the smallest. Lastly, if the partitioning is made with weighted Laplacian, *cz148* needs 10^{-6} as tolerance, whereas 3 of them need the tolerance to be 10^{-10}.

When sixteen partitions are obtained from the matrices, *can_161* gives the most balanced partition when the tolerance is 10^{-6}, while 2 of them give when it is 10^{-2}. *lshp_265* needs to have 10^{-8} tolerance when weighted Laplacian is used for partitioning. On the other hand, when unweighted Laplacian is used, *lshp_265* gives more balanced partitions when the tolerance is the highest, whereas *can_161* and *Pois-*

son(12) need 10^{-4} and 10^{-8} tolerances, respectively. For unweighted partitioning of *cz148*, 10^{-10} should be used as tolerance. If the number of iterations is considered, all matrices give the least number of iterations when the tolerance is the highest, whether Laplacian is weighted or not. Lastly, *can_161* gives less edge-cut when the tolerance is 10^{-4}, 2 of them give when it is 10^{-2}, and *Poisson(12)* gives when the tolerance is 10^{-8} if unweighted partitioning is performed. Contrarily, when weighted partitioning is made, *can_161* and *Poisson(12)* need 10^{-6} and 10^{-8} tolerances for less edge-cuts, respectively. The remaining 2 can be partitioned with less edge-cuts when tolerance is the highest.

To summarize, choosing the optimal eigensolver tolerance for obtaining more balanced partitioning and less edge-cut depends on the matrix. It is also observed that as tolerance decreases, the edge-cut does not decrease. This is because the subspace dimension in `eigs` function becomes larger, especially when the number of partitions increases. Since the subspace dimension depends on the number of partitions, in the end, $16 + 3 = 19$ is quite a significant value as a subspace dimension for an eigensolver. Since MATLAB requires the subspace dimension being at least $nparts + 2$, an increase in dimension and a decrease in partitions' quality become inevitable.

4.2 Edge-cut and The Cost in terms of Time

In this section, edge-cut and the cost in terms of time are inspected when various algorithms are used to partition large-sized graphs. In comparing SPEC, various eigensolver tolerances are used to find the optimum value. All of these algorithms are used with C language and PETSc [3], SLEPc [53] libraries. The numerical results are presented in Tables B.5 - B.14, and graphs along with spy representations of the partitioned small-sized matrices with 2 and 4 processes (nproc) are given in Appendix C. Each color and shape in spy representations of graphs shows a different partition, where black dots represent off-block diagonal elements. Tables show the results of edge-cuts and elapsed time (sec) for SPEC, ParMETIS, and CHACO with a various number of processes (partitions) (NPROC). Moreover, only the unweighted partitions are made when ParMETIS is used since PETSc requires edge weights being integer and less than 10 if ParMETIS is used to partition a matrix with a weighted

option. Therefore, matrices should be mapped to achieve the desired edge weight properties. Also, CHACO does not support weighted partitioning, whereas SPEC can consider floating-point numbers as weights. Hence, only in the unweighted case, SPEC is compared with the libraries. That is why weighted results of CHACO and ParMETIS in tables are denoted as $(-)$. Furthermore, various eigensolver tolerances (10^{-2}, 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10}) and weighted (W), unweighted (U) Laplacian usages (LAP) are inspected for SPEC. Results presented in tables are rounded to two decimal digits from seven, and the best results are marked based on the seven decimal digits. Each numerical results in tables are computed as the average of 10 iterations to have reliable results. The numerical results obtained from the partitioning of large-sized graphs are presented in Figures 4.10 - 4.9.

For comparison, the default eigensolver in SLEPc is used (Krylov-Schur), with the subspace dimension 17. For SPEC, the number of eigenvalues we requested is equal to the number of processes. Moreover, due to a limitation of PETSc, the number of partitions is set to be equal to the number of processes [3].

4.2.1 Symmetric Large-Sized Matrices

Based on Figure 4.9, to partition *bcspwr10*, SPEC should be used when the number of processes is less than sixteen and unweighted edge-cut is compared. When sixteen processes are used, using CHACO performs better in terms of the edge-cut. If partitioning time is concerned for unweighted partitioning, using CHACO becomes preferable when two processes are used. Otherwise, ParMETIS should be chosen. When weighted partitioning of the matrix is considered, it is seen that if four or fewer processes are used, then using the highest tolerance results in the least edge-cut. If more processes are used, then the tolerance should be decreased. In terms of partitioning time, using the highest tolerance performs faster when weighted partitioning is made.

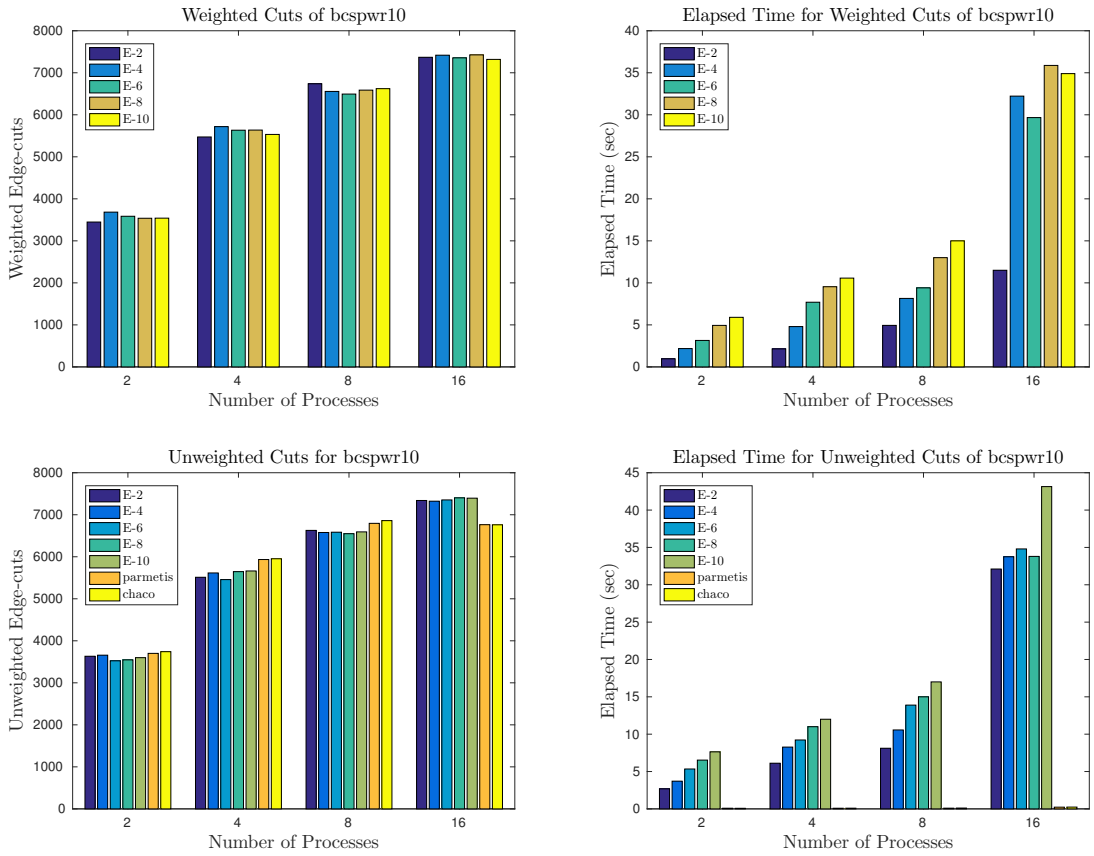


Figure 4.9: Results obtained from the partitioning of *bcspwr10*

4.2.2 Unsymmetric Large-Sized Matrices

Figure 4.10 shows that when the unweighted cut is compared, up to four partitions, CHACO gives better cut for *rw5151*. However, it is seen that SPEC is a better choice for more partitions. For this matrix, SPEC should be used with a tolerance of at least 10^{-10} for having a smaller edge-cut. Hence, to partition *rw5151*, if one needs to cluster the matrix into eight or more to achieve the smallest unweighted edge-cut, SPEC should be used. When the partitioning is made by using weighted Laplacian, it is seen that using the highest tolerance usually results in less edge-cut.

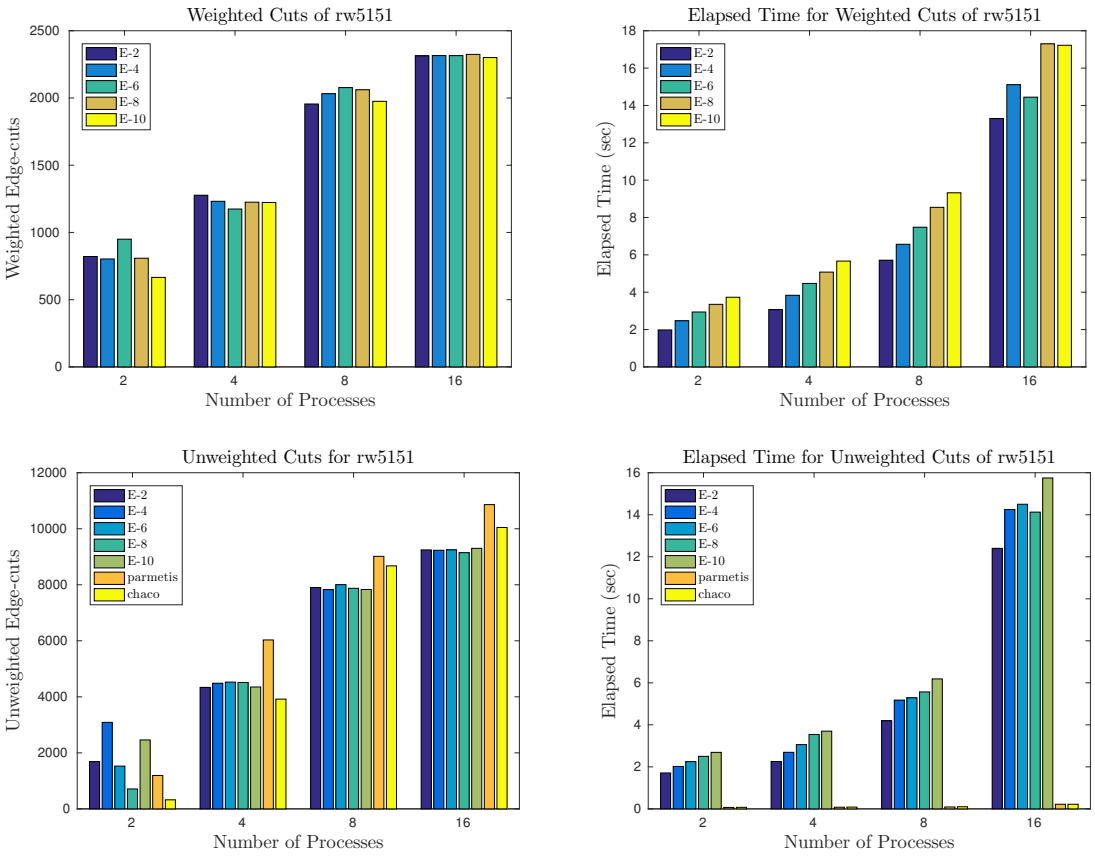


Figure 4.10: Results obtained from the partitioning of *rw5151*

From Figure 4.11, observations show that SPEC should be used with a tolerance of at least 10^{-8} when unweighted edge-cut is compared. On the other hand, when the cost is considered in terms of time, ParMETIS looks like the best option for obtaining unweighted edge-cut. It is also seen that when unweighted Laplacian is applied to *av41092*, partitioning into two is the slowest process. This is because of the number of strongly connected components in the resulting graph of this matrix being four, whereas it is one for other matrices. When the weighted Laplacian is used, partitioning is operated the fastest when the tolerance is the highest for all number of processes. Further, as the number of processes increases up to 16, the best tolerance for less edge-cut decreases. When 16 processes are used, using 10^{-4} as tolerance becomes optimum for less weighted edge-cut.

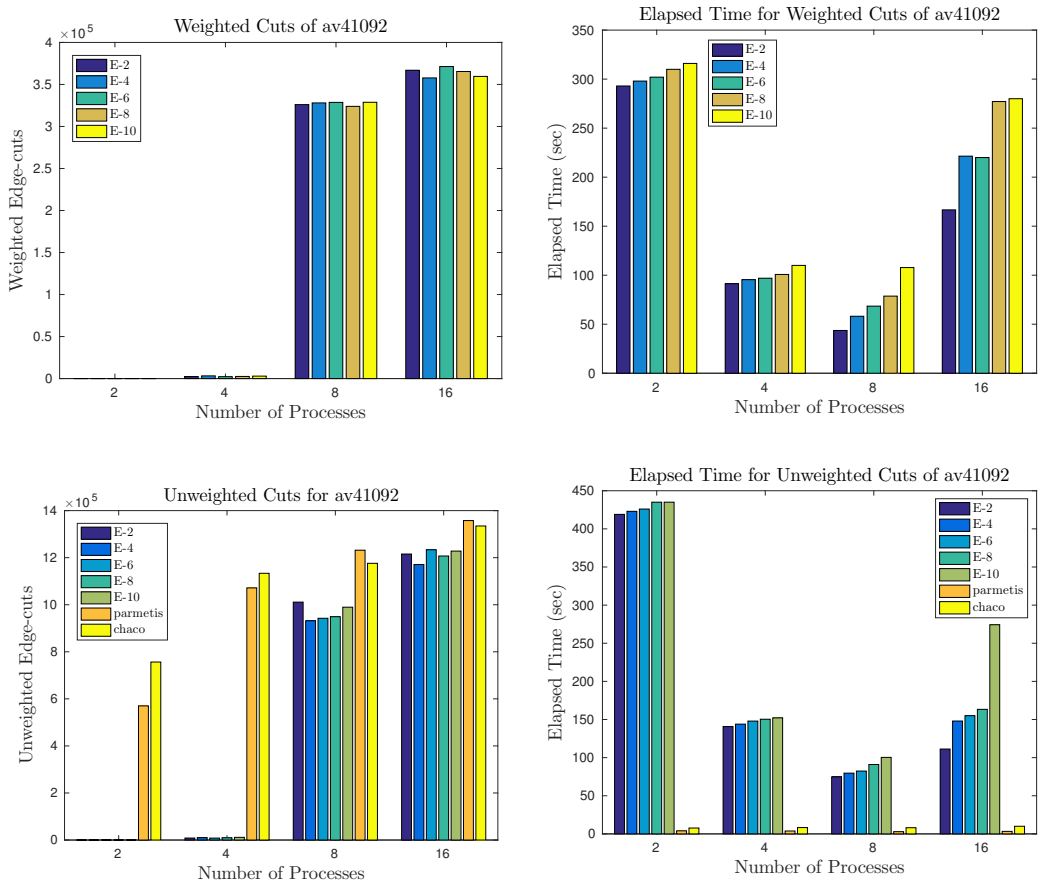


Figure 4.11: Results obtained from the partitioning of *av41092*

Figure 4.12 shows that when unweighted edge-cut is compared, up to four partitions, ParMETIS and CHACO give better cuts for *chem_master1*. When the matrix is partitioned into two, it is observed that ParMETIS is a better option, while for four partitions, CHACO performs better. However, SPEC becomes a better choice for more partitions. For this matrix, SPEC should be used with tolerance at least 10^{-8} if the weighted cut is considered; otherwise, 10^{-2} is enough for more than four partitions.

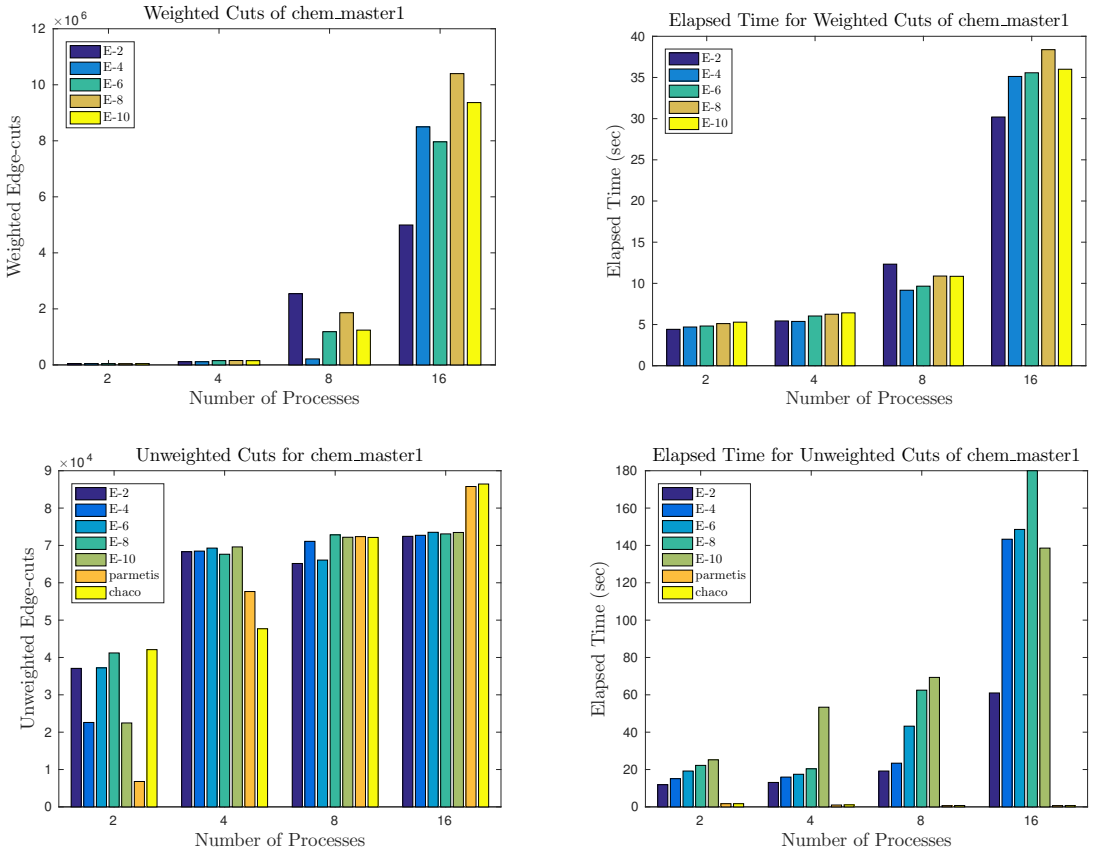


Figure 4.12: Results obtained from the partitioning of *chem_master1*

When Figure 4.13 is inspected, it is seen that for less unweighted edge-cut, SPEC should be used for partitioning of *epb2* by using any number of processes. However, when partitioning time is considered, it is inspected that ParMETIS is the fastest option for unweighted partitioning. Moreover, when the weighted Laplacian is used to partition the matrix, it is seen that using a tolerance of 10^{-6} usually gives the least edge-cut. Lastly, for partitioning the matrix faster into sixteen using weighted Laplacian, 10^{-8} tolerance should be used instead of 10^{-2}.

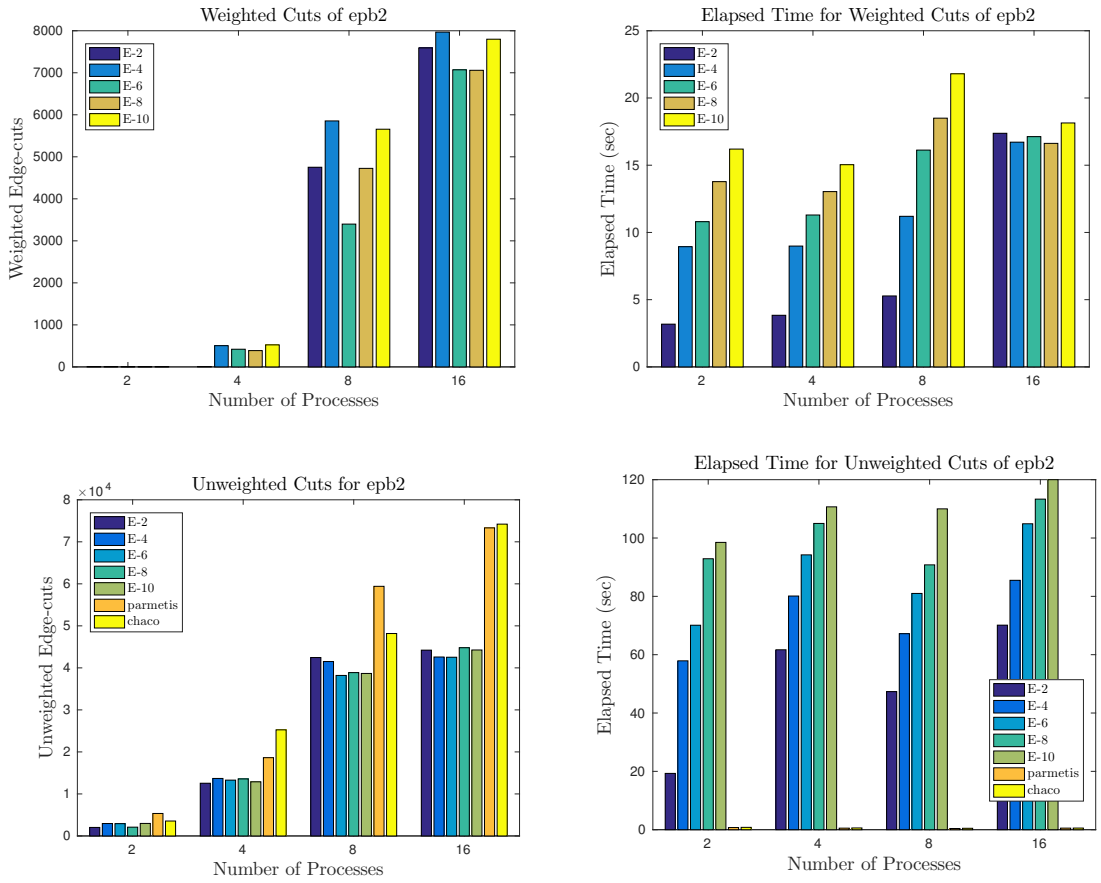


Figure 4.13: Results obtained from the partitioning of *epb2*

Figure 4.14 presents that if one needs to partition *FEM_3D_thermal1*, then using SPEC is beneficial for good partitions in terms of unweighted edge-cuts. For the algorithm, 10^{-10} tolerance is enough for the eigensolver. If one needs four partitions, choosing ParMETIS should be the best option in terms of the edge-cut. When weighted partitioning of the matrix is considered, it is seen that up to eight partitions, using the highest tolerance performs the fastest. When the number of processes is increased, using a tolerance of 10^{-4} is preferable. Lastly, it is seen that for sixteen processes, using the highest tolerance gives the least weighted edge-cut. When the number of processes is decreased, using smaller tolerances yields less edge-cut.

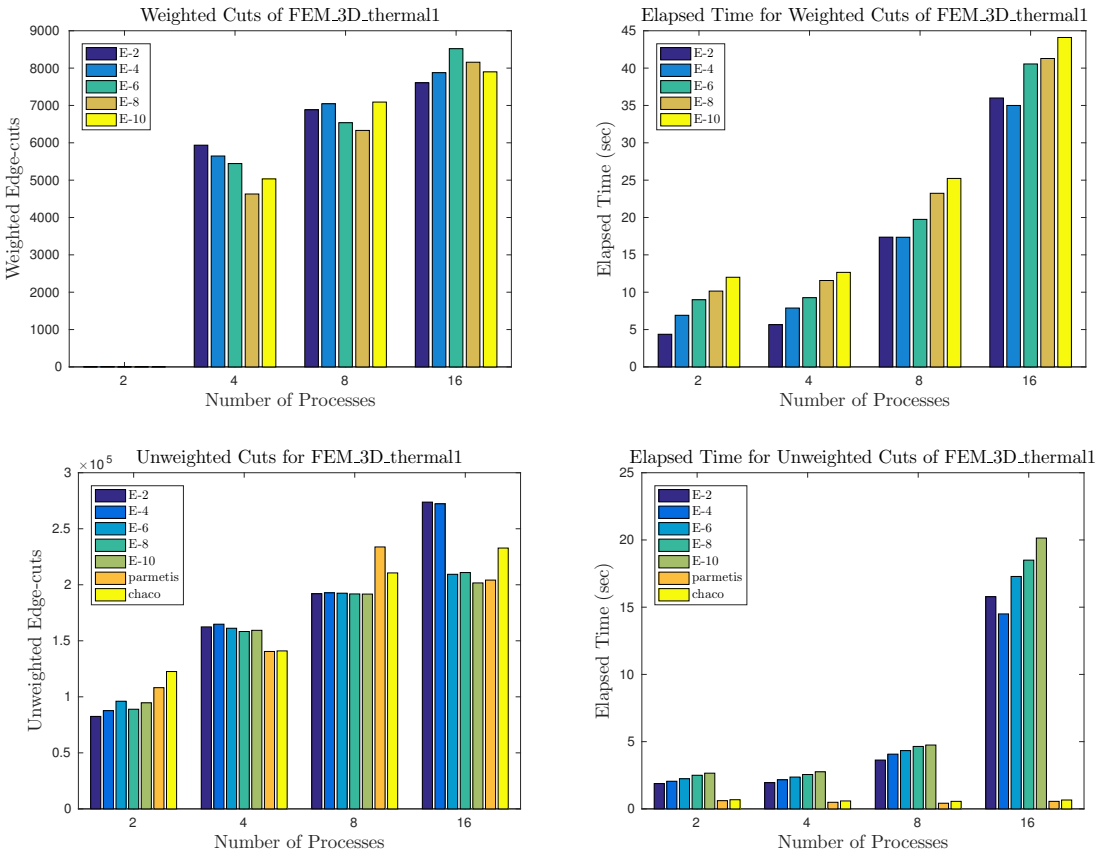


Figure 4.14: Results obtained from the partitioning of *FEM_3D_thermal1*

As shown in Figure 4.15, for the unweighted partitioning, up to sixteen partitions, SPEC should be used with a tolerance of at least 10^{-8}. Starting from sixteen partitions, using ParMETIS starts being a better choice for less edge-cut. If the elapsed time is the comparison metric for obtaining unweighted partitions, one should partition *ns3Da* by using ParMETIS. From the weighted partitioning of the matrix, it is seen that using the highest tolerance performs faster when the number of processes is less than sixteen. If it is increased to sixteen, then 10^{-6} should be used as tolerance. If the number of processes is more than four, then tolerance of 10^{-8} should be used for less weighted edge-cuts. If the number of processes is smaller, then higher tolerances should be used.

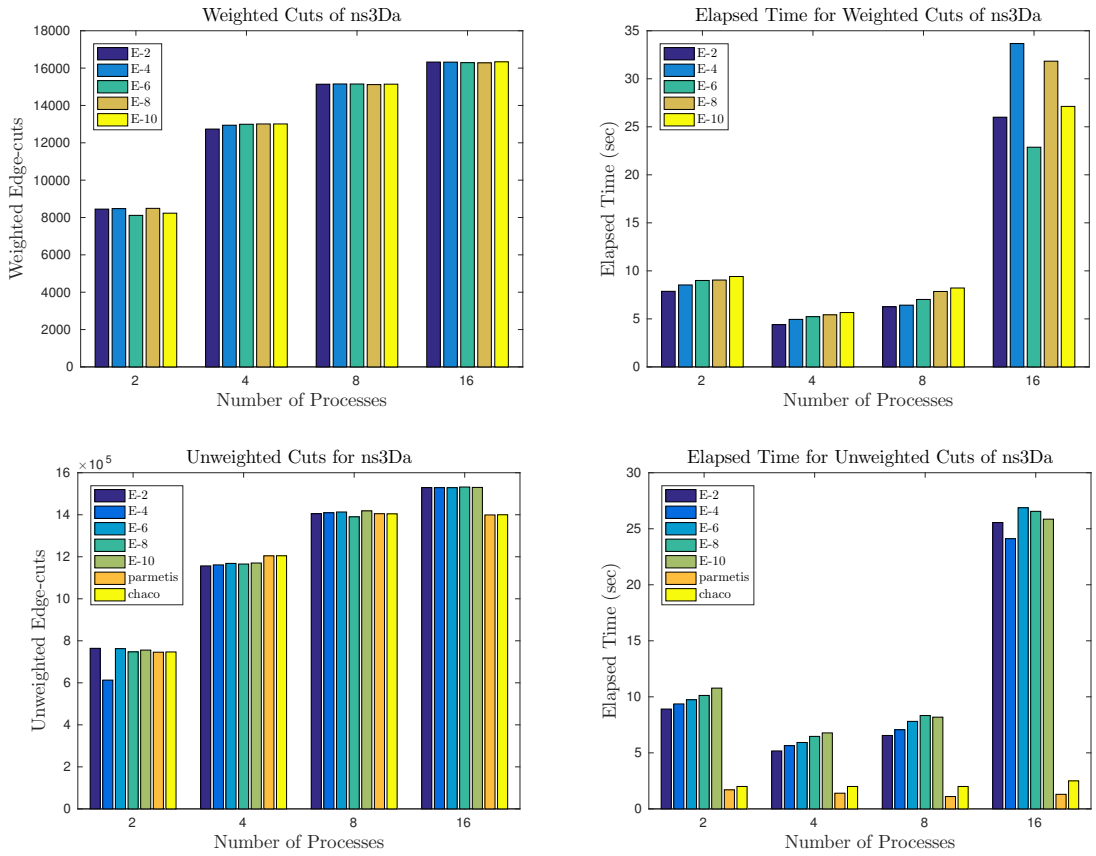


Figure 4.15: Results obtained from the partitioning of *ns3Da*

From Figure 4.16, results show that when unweighted edge-cut is compared, for partitioning, ParMETIS and CHACO give better cuts for *poisson3Db*. When software libraries are compared, CHACO is a suitable choice for two partitions. In contrast, for more partitions, ParMETIS becomes a better option. It is also seen that if time is compared for unweighted partitioning, ParMETIS should be the option regardless of the number of partitions. For the weighted case, SPEC should be used with a tolerance of at least 10^{-8} for less edge-cut. It is observed that as the number of partitions increases, the best tolerance for less edge-cut decreases to 10^{-10}. Lastly, if sixteen processes are used for weighted partitioning, then it is seen that using 10^{-6} tolerance performs faster.

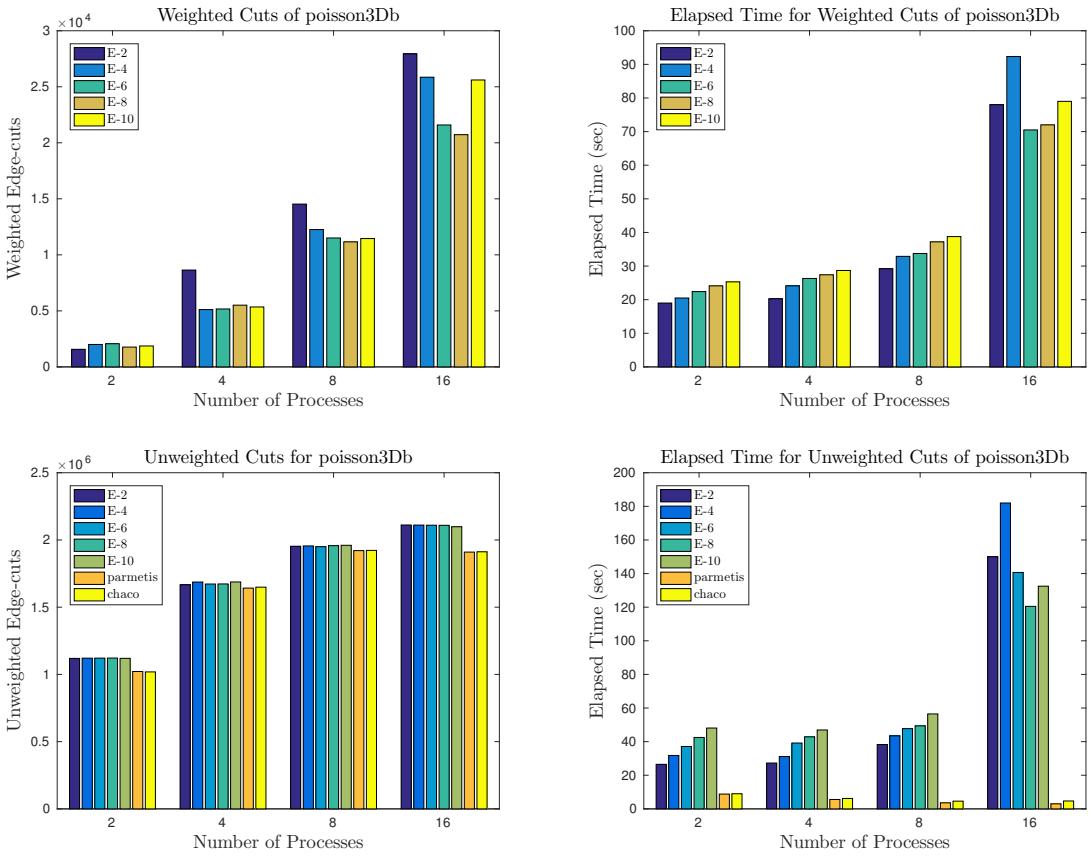


Figure 4.16: Results obtained from the partitioning of *poisson3Db*

From Figure 4.17, it is seen that for unweighted partitioning of *sme3Da*, up to sixteen processes, using SPEC results less edge-cut. However, if the number of processes becomes sixteen, then ParMETIS should be used instead. If the partitioning time is considered for the unweighted case, then ParMETIS is the fastest option. When weighted partitioning of the matrix is performed, it is seen that using the highest tolerance results in less edge-cut when at most four processes are used. If more processes are used, then the tolerance should be decreased.

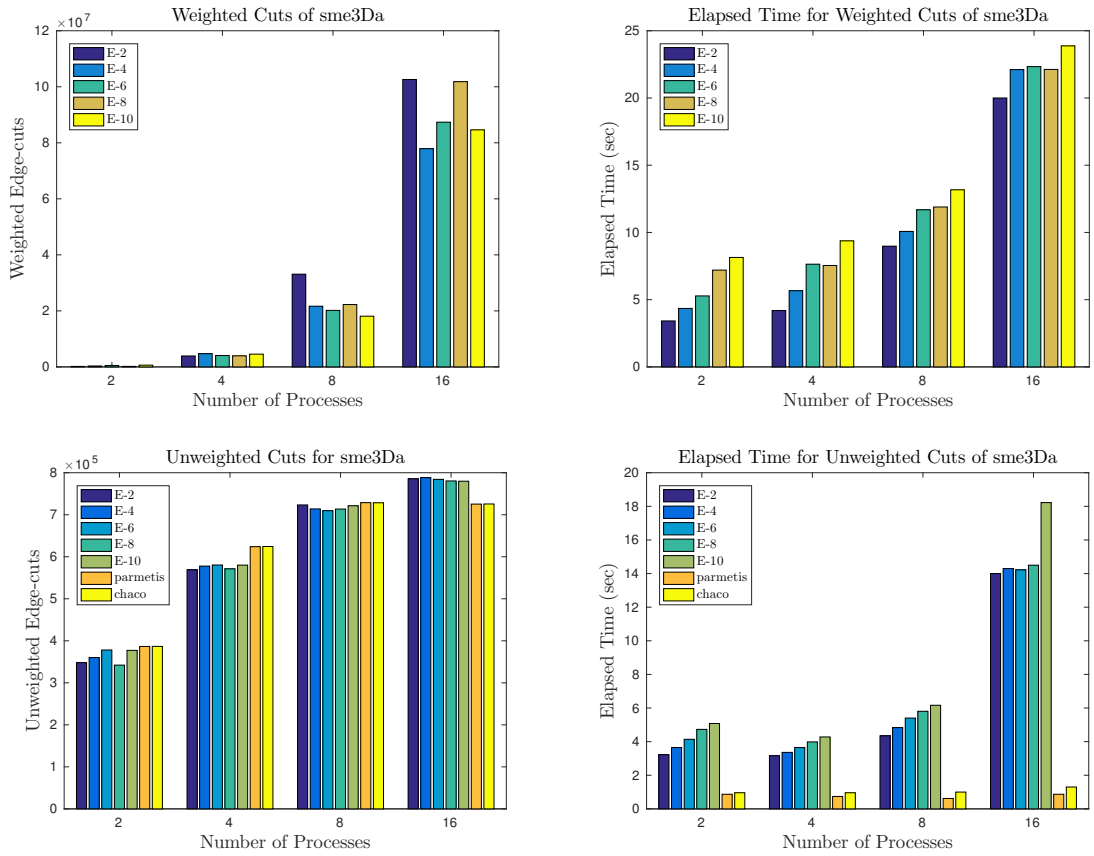


Figure 4.17: Results obtained from the partitioning of *sme3Da*

Based on the results in Figure 4.18, for weighted edge-cut, SPEC should be used to partition *Zhao1* into at most four. If one needs more partitions than four, CHACO should be used. It is also inspected that for any number of partitions when unweighted partitioning is made, CHACO performs faster than the other algorithms. When weighted partitioning of the matrix is considered, it is seen that using the highest tolerance performs faster. If less edge-cut is desired, then 10^{-2} should be used as tolerance for two and sixteen processes. Otherwise, 10^{-6} should be preferred.

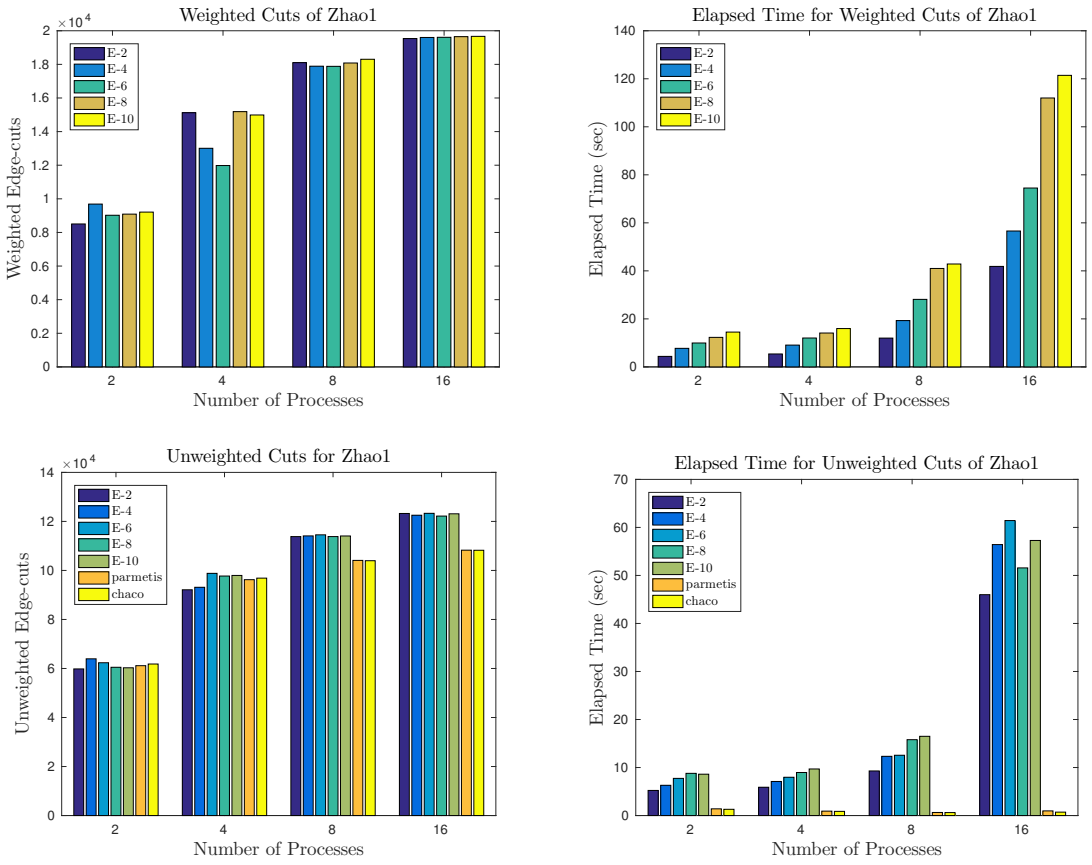


Figure 4.18: Results obtained from the partitioning of *Zhao1*

4.2.3 Results

Based on the numerical results obtained from the unweighted partitioning of 10 large-sized matrices in terms of the edge-cut and partitioning time, Tables 4.5-4.6 present the best algorithms for each matrix, respectively.

Table 4.5: The best algorithms (SPEC, ParMETIS (PAR), and CHACO) for large-sized matrices in terms of the unweighted edge-cut.

	2			4			8			16		
	SPEC	PAR	CHACO	SPEC	PAR	CHACO	SPEC	PAR	CHACO	SPEC	PAR	CHACO
bcspwr10	+			+			+					+
epb2	+			+			+			+		
sme3Da	+			+			+				+	
av41092	+			+			+			+		
poisson3Db			+		+			+			+	
rw5151			+			+	+			+		
FEM_3D_thermal1	+				+		+			+		
Zhao1	+			+					+			+
ns3Da	+			+			+				+	
chem_master1		+				+	+			+		

Table 4.6: The est algorithms (SPEC, ParMETIS (PAR), and CHACO) for large-sized matrices in terms of the unweighted partitioning time.

	2			4			8			16		
	SPEC	PAR	CHACO	SPEC	PAR	CHACO	SPEC	PAR	CHACO	SPEC	PAR	CHACO
bcspwr10			+		+,			+,			+,	
epb2		+			+			+			+	
sme3Da		+			+			+			+	
av41092		+			+			+			+	
poisson3Db		+			+			+			+	
rw5151		+			+			+			+	+
FEM_3D_thermal1		+			+			+			+	
Zhao1			+			+			+			+
ns3Da		+			+			+			+	
chem_master1		+			+			+			+	

As a summary, ParMETIS performs the fastest in partitioning 9 of the 10 matrices. For *Zhao1*, CHACO performs faster. The reason for ParMETIS operating faster than CHACO is that CHACO depending on the recursive usage of spectral bisection, where Multilevel Kernighan-Lin in ParMETIS is based on the partitioning in the coarsest level.

It is also observed that 7 of the 10 matrices perform better in terms of the edge-cut when only SPEC is used for unweighted partitioning. For 2 matrices, SPEC and CHACO perform better for the different number of partitions. For *poisson3Db*, ParMETIS gives the least edge-cut. Hence, SPEC becomes a preferable option for most cases.

When the comparison is made based on the number of processes, it is observed that 7 of 10 matrices give better performances in SPEC, 2 of them give in CHACO, and only one of them, *chem_master1* gives fewer edge-cuts in ParMETIS when two processes are used. On the other hand, when the number of processes is increased to four, 2 of the matrices yield less edge-cut when ParMETIS is used and the partitioning of *chem_master1* in less edge-cut is performed in CHACO. The remaining 7 matrices are partitioned in less edge-cut when SPEC is used. When eight processes are used to partition the matrices, SPEC operates better for 8 matrices, where for *Zhao1*, CHACO should be used for less edge-cut, and *poisson3Db* gives a better partition in ParMETIS. Lastly, using SPEC yields less edge-cut in 5 of matrices when sixteen processes are used. 3 of the matrices result in better partitions when ParMETIS is used. The remaining 2 are partitioned better by CHACO. Hence, SPEC performs better when the number of processes is small, whereas using libraries starts to perform

better in terms of the edge-cut when sixteen processes are used.

Based on the numerical results obtained from the weighted partitioning of 10 large-sized matrices, the best tolerances for SPEC in terms of the edge-cut and partitioning time are presented in Tables 4.7-4.8, respectively.

Table 4.7: The best tolerances for the eigensolver of SPEC in terms of weighted edge-cut when large-sized matrices are partitioned by using 2, 4, 8, and 16 processes.

nproc	Tolerance	bcspwr10	epb2	sme3Da	av41092	poisson3Db	rw5151	FEM_3D_thermal1	Zhao1	ns3Da	chem_master1
2	10^{-2}	+		+	+	+			+		
	10^{-4}										
	10^{-6}		+					+		+	
	10^{-8}										+
	10^{-10}						+				
4	10^{-2}	+	+	+						+	
	10^{-4}					+					+
	10^{-6}				+		+		+		
	10^{-8}							+			
	10^{-10}										
8	10^{-2}						+				
	10^{-4}										
	10^{-6}	+	+						+		+
	10^{-8}				+	+		+		+	
	10^{-10}			+							
16	10^{-2}							+	+		+
	10^{-4}			+	+						
	10^{-6}										
	10^{-8}		+			+				+	
	10^{-10}	+					+				

Table 4.8: The best tolerances for the eigensolver of SPEC in terms of partitioning time when large-sized matrices are partitioned with weighted Laplacian by using 2, 4, 8, and 16 processes.

nproc	Tolerance	bcspwr10	epb2	sme3Da	av41092	poisson3Db	rw5151	FEM_3D_thermal1	Zhao1	ns3Da	chem_master1
2	10^{-2}	+	+	+	+	+	+	+	+	+	+
	10^{-4}										
	10^{-6}										
	10^{-8}										
	10^{-10}										
4	10^{-2}	+	+	+	+	+	+	+	+	+	
	10^{-4}										+
	10^{-6}										
	10^{-8}										
	10^{-10}										
8	10^{-2}	+	+	+	+	+	+		+	+	
	10^{-4}							+			+
	10^{-6}										
	10^{-8}										
	10^{-10}										
16	10^{-2}	+		+	+		+		+		+
	10^{-4}							+			
	10^{-6}					+				+	
	10^{-8}		+								
	10^{-10}										

Lastly, if weighted partitioning is made for the matrices, it is observed that using the highest tolerance performs faster for every matrix. If the edge-cuts are considered, then the tolerance usually decreases as the number of processes increases up to six-

teen. If sixteen processes are used, then a higher tolerance usually yields less edge-cut for matrices.

In conclusion, when the cost is compared in terms of time, libraries give better performance for any partitions, while to achieve good partitions in terms of edge-cuts, SPEC should be preferred.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this thesis, we have inspected various applications of graph partitioning for matrix partitioning. Breaking down the system of equations and solving them in parallel becomes essential in scientific computing, and hence for making them suitable for this process, graph partitioning is used. For partitioning a graph, existing software libraries can be used as well as an additional algorithm to graph partitioning techniques such as the k-means clustering. To examine the effect of various graph partitioning techniques on the cost in terms of time and unweighted edge-cut, we compared CHACO and ParMETIS with SPEC. To obtain the optimum eigensolver tolerance for SLEPc, weighted edge-cuts are also inspected when the matrices are partitioned by SPEC.

To have a general opinion on the partitioning square structurally unsymmetric matrices, matrices having various properties have been used during the study: all of the large-sized matrices have different conditional numbers, least singular values, numbers of non-zeros, and symmetry rates. Hence, it is not suitable for generalization of the results based on a structure except being square structurally unsymmetric. For extending this study based on the structural properties, all application domains in the University of Florida Sparse Matrix Collection will be used in future work.

Using spectral graph partitioning leads to solving an eigenvalue problem. In this case, solving this problem and obtaining good partitioning depends on the eigensolver tolerance. Thus, this thesis's comparison is also based on the load imbalance and the number of iterations the eigensolver used when tolerance of the eigensolver changes.

The numerical results are obtained from the large-sized matrices show that when the cost is inspected in terms of time, the software libraries become more preferred to use. Moreover, it is seen that ParMETIS should be preferred over CHACO since CHACO is based on recursive bi-partitioning while ParMETIS uses multilevel algorithms.

Furthermore, a C code is developed for partitioning unsymmetric matrices by using SPEC to perform the solution of linear systems with an unsymmetric coefficient matrix more efficiently. When unweighted edge-cuts are inspected, using SPEC instead of graph partitioning libraries results in better partitions. Moreover, if a software package is needed to be used, then for less edge-cut, CHACO should be used instead of ParMETIS. Although ParMETIS gives the fastest performance for almost all matrices, it can be seen that there is no significant difference between the performance time of CHACO and ParMETIS. Thus, using CHACO instead of ParMETIS does not cause a worthwhile trade-off between the edge-cut and the cost in terms of time.

Finally, when the comparison is made based on the number of processes, it is observed that SPEC performs better when the number of processes is small, whereas using libraries starts to perform better when sixteen processes are used.

When the spectral partitioning is used, eigensolver tolerance should also be inspected. Hence, we have studied the eigensolver function in MATLAB to find an optimal tolerance for medium-sized matrices. From the results, it is seen that to get more balanced partitions, the smallest tolerance should be used with the eigensolver. It is also seen that the optimum tolerance for better partitions (the least edge-cut) depends on the matrix.

PETSc requires edge weights being integer and less than 10 if ParMETIS is used to partition a matrix with a weighted option. In other words, matrices should be mapped to achieve the desired edge-weight properties. However, mapping causes information loss, and SPEC uses more information on matrices since it does not require such mapping. Also, CHACO does not support weighted partitioning. Hence, using SPEC will give a better partitioning in terms of the edge-cut when weighted partitioning is made. Thus, a comparison based on the mapping of matrices for weighted partitioning will be studied for all application domains in the University of Florida Sparse Matrix Collection on ParMETIS in the future.

One of the significant drawbacks of the MATLAB eigensolver is the increase in the subspace dimension as the number of partitions increases. Therefore, as future work, each matrix's load imbalance will also be inspected in C language by using the SLEPc library. Another disadvantage encountered in this study is the subspace dimension in SLEPc being high. Hence, the optimum eigensolver tolerance with smaller subspace dimensions will be studied for all application domains in the University of Florida Sparse Matrix Collection.

REFERENCES

[1] R. Andersen, F. Chung, and K. Lang, Local Partitioning for Directed Graphs Using PageRank, in *Algorithms and Models for the Web-Graph*, pp. 166–178, Springer Berlin Heidelberg, 2007.

[2] Z.-Z. Bai, G. H. Golub, and M. K. Ng, Hermitian and Skew-Hermitian Splitting Methods for Non-Hermitian Positive Definite Linear Systems, SIAM Journal on Matrix Analysis and Applications, 24(3), pp. 603–626, 2003.

[3] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelma, L. Dalcin, A. Dene, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, D. May, L. C. McInnes, R. Mills, T. Munson, K. Rupp, P. Sanan, B.Smith, S. Zampini, H. Zhang, and H. Zhang, *PETSc Users Manual Revision 3.12*, Argonne National Laboratory, 3.12 edition, 2019.

[4] S. T. Barnard and H. D. Simon, Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems, Concurrency: Practice and Experience, 6(2), pp. 101–117, 1994.

[5] B. A. Becker and A. Lastovetsky, Matrix Multiplication on Two Interconnected Processors, in *2006 IEEE International Conference on Cluster Computing*, pp. 1–9, 2006.

[6] C.-E. Bichot, *Local Metaheuristics and Graph Partitioning*, chapter 6, pp. 137–161, John Wiley & Sons, Ltd, 2013.

[7] N. Biggs, *Algebraic Graph Theory*, Cambridge Mathematical Library, Cambridge University Press, 2 edition, 1974.

[8] E. G. Birgin, R. D. Lobato, and R. Morabito, An Effective Recursive Partitioning Approach for the Packing of Identical Rectangles in a Rectangle, Journal of the Operational Research Society, 61(2), pp. 306–320, 2010.

[9] T. N. Bui and C. Jones, A Heuristic for Reducing Fill-in in Sparse Matrix Factorization, Technical report, Society for Industrial and Applied Mathematics (SIAM), 1993.

[10] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, Recent Advances in Graph Partitioning, in *Algorithm Engineering*, pp. 117–158, Springer, 2016.

[11] D. Casella, k-means, `https://github.com/dcasella/k-means`, 2017.

[12] Ü. Çatalyürek and C. Aykanat, *PaToH (Partitioning Tool for Hypergraphs)*, pp. 1479–1487, Springer US, 2011.

[13] U. V. Catalyurek and C. Aykanat, Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication, IEEE Transactions on parallel and distributed systems, 10(7), pp. 673–693, 1999.

[14] T.-Y. Choe and C.-I. Park, A k-way Graph Partitioning Algorithm Based on Clustering by Eigenvector, in *Computational Science - ICCS 2004*, pp. 598–601, Springer Berlin Heidelberg, 2004.

[15] T. A. Davis and Y. Hu, The University of Florida Sparse Matrix Collection, ACM Transactions on Mathematical Software, 38(1), 2011.

[16] W. Donath and A. Hoffman, Lower Bounds for the Partitioning of Graphs, IBM Journal of Research and Development, 17(5), pp. 420–425, 1973.

[17] J. Ellson, E. Gansner, L. Koutsofios, S. North, G. Woodhull, S. Description, and L. Technologies, Graphviz - Open Source Graph Drawing Tools, in *Lecture Notes in Computer Science*, pp. 483–484, Springer-Verlag, 2001.

[18] C. M. Fiduccia and R. M. Mattheyses, A Linear-Time Heuristic for Improving Network Partitions, in *Proceedings of the 19th Design Automation Conference*, pp. 175–181, IEEE Press, 1982.

[19] M. Fiedler, Algebraic Connectivity of Graphs, Czechoslovak Mathematical Journal, 23, pp. 298–305, 1973.

[20] M. Fiedler, A Property of Eigenvectors of Nonnegative Symmetric Matrices and Its Application to Graph Theory, Czechoslovak Mathematical Journal, 25(4), pp. 619–633, 1975.

[21] P.-O. Fjällström, *Algorithms for Graph Partitioning: A Survey*, volume 3, Linköping University Electronic Press Linköping, 1998.

[22] M. Garey, D. Johnson, and L. Stockmeyer, Some Simplified NP-Complete Graph Problems, Theoretical Computer Science, 1(3), pp. 237 – 267, 1976.

[23] G. H. Golub and C. F. Van Loan, *Matrix Computations (3rd Ed.)*, Johns Hopkins University Press, 1996.

[24] L. Grady and E. Schwartz, Isoperimetric Partitioning: A New Algorithm for Graph Partitioning, SIAM Journal of Scientific Computing, 27, pp. 1844–1866, 2006.

[25] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, Parallel Computing, 22(6), pp. 789 – 828, 1996.

[26] A. Gupta, *WGPP: Watson Graph Partitioning (and sparse matrix ordering) Package*, IBM TJ Watson Research Center, 1996.

[27] A. Gupta, Fast and Effective Algorithms for Graph Partitioning and Sparse-Matrix Ordering, IBM Journal of Research and Development, 41(1.2), pp. 171–183, 1997.

[28] B. Hendrickson and T. G. Kolda, Partitioning sparse rectangular matrices for parallel computations of ax and atv, in *International Workshop on Applied Parallel Computing*, pp. 239–247, Springer, 1998.

[29] B. Hendrickson and T. G. Kolda, Graph Partitioning Models for Parallel Computing, Parallel Computing, 26(12), pp. 1519 – 1534, 2000.

[30] B. Hendrickson and T. G. Kolda, Partitioning Rectangular and Structurally Unsymmetric Sparse Matrices for Parallel Processing, SIAM Journal on Scientific Computing, 21(6), pp. 2048–2072, 2000.

[31] B. Hendrickson and R. Leland, *The Chaco User's Guide Version 1.0*, Sandia National Laboratories, 1993.

[32] B. Hendrickson and R. Leland, A Multi-Level Algorithm for Partitioning Graphs, in *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pp. 28–28, 1995.

[33] I. C. F. Ipsen and C. D. Meyer, The Idea Behind Krylov Methods, The American Mathematical Monthly, 105(10), pp. 889–899, 1998.

[34] W. N. A. Jr. and T. D. Morley, Eigenvalues of the Laplacian of a Graph, Linear and Multilinear Algebra, 18(2), pp. 141–145, 1985.

[35] G. Karypis and V. Kumar, Multilevel Graph Partitioning Schemes, in *Proceedings of 24th International Conference on Parallel Processing, III*, pp. 113–122, CRC Press, 1995.

[36] G. Karypis and V. Kumar, Parallel Multilevel Graph Partitioning, in *Proceedings of International Conference on Parallel Processing*, pp. 314–319, 1996.

[37] G. Karypis and V. Kumar, A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm, in *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[38] G. Karypis and V. Kumar, METIS — A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes and Computing Fill-Reducing Ordering of Sparse Matrices, 1997.

[39] G. Karypis and V. Kumar, A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, SIAM Journal on scientific Computing, 20(1), pp. 359–392, 1998.

[40] G. Karypis and K. Schloegel, *PARMETIS, Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 4.0*, University of Minnesota, Department of Computer Science and Engineering, 4 edition, 2013.

[41] B. W. Kernighan and S. Lin, An Efficient Heuristic Procedure for Partitioning Graphs, The Bell System Technical Journal, 49(2), pp. 291–307, 1970.

[42] T. G. Kolda, Partitioning Sparse Rectangular Matrices for Parallel Processing, in *Solving Irregularly Structured Problems in Parallel*, pp. 68–79, Springer Berlin Heidelberg, 1998.

[43] A. Likas, N. Vlassis, and J. J. Verbeek, The Global k-means Clustering Algorithm, Pattern Recognition, 36(2), pp. 451 – 461, 2003.

[44] U. Luxburg, A Tutorial on Spectral Clustering, Statistics and Computing, 17, pp. 395–416, 2004.

[45] M. Manguoglu, E. Cox, F. Saied, and A. Sameh, TRACEMIN-Fiedler: A Parallel Algorithm for Computing the Fiedler Vector, in *High Performance Computing for Computational Science – VECPAR 2010*, pp. 449–455, Springer Berlin Heidelberg, 2011.

[46] MATLAB, *version 7.10.0 (R2016a)*, The MathWorks Inc., 2016.

[47] M. Naumov and T. D. Moon, Parallel Spectral Graph Partitioning, Technical report, NVIDIA, 2016.

[48] B. Parlett, H. Simon, and L. Stringer, On Estimating the Largest Eigenvalue with the Lanczos Algorithm, Mathematics of Computation, 38, pp. 153–153, 1982.

[49] F. Pellegrini, Distillating Knowledge About Scotch, in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[50] R. Preis and R. Diekmann, PARTY - A Software Library for Graph Partitioning, 1998.

[51] C. P. Ravikumar and G. W. Zobrist, *Parallel Methods for VLSI Layout Design*, Greenwood Publishing Group Inc., 1995.

[52] F. Rendl and R. Sotirov, The Min-cut and Vertex Separator Problem, Computational Optimization and Applications, 69, pp. 159–187, 2017.

[53] J. E. Roman, C. Campos, E. Romero, and A. Tomas, SLEPc Users Manual: Scalable Library for Eigenvalue Problem Computations, Technical Report DSIC-II/24/02, Universitat Politecnica De Valencia, Departamento de Sistemas Informaticos y Computacion, 2019.

[54] K. Schloegel, G. Karypis, and V. Kumar, *Graph Partitioning for High-Performance Scientific Simulations*, pp. 491–541, Morgan Kaufmann Publishers Inc., 2003.

[55] H. D. Simon, Partitioning of Unstructured Problems for Parallel Processing, Computing systems in engineering, 2(2-3), pp. 135–148, 1991.

[56] D. Singh and C. K. Reddy, A Survey on Platforms for Big Data Analytics, Journal of big data, 2(8), 2015.

[57] A. Soper, C. Walshaw, and M. Cross, A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning, Journal of Global Optimization, 29, pp. 225–241, 2004.

[58] D. Sorensen, Implicitly Restarted Arnoldi/Lanczos Methods for Large Scale Eigenvalue Calculations, ICASE/LaRC Interdisciplinary Series in Science and Engineering, 4, 1996.

[59] G. W. Stewart, A Krylov–Schur Algorithm for Large Eigenproblems, SIAM Journal on Matrix Analysis and Applications, 23(3), pp. 601–614, 2002.

APPENDIX A

THE C CODES

In this appendix, C codes used for partitioning large-sized matrices are given. The first algorithm is used for partitioning by using ParMETIS and CHACO, whereas the latter one is by using SPEC algorithm. Each algorithm symmetrizes the original unsymmetric input matrix first.

```
static char help[] = "Matrix Partitioning by Using ParMETIS and
  ↪  CHACO.\n\n";
#include <petscmat.h>
#include <petscis.h>
#include <petscsys.h>
#include <stdio.h>
#include <stdlib.h>
#include <petsctime.h>
/*  Returns to the absolute value of matrix  */
PetscErrorCode absmat(Mat A,Mat *AbsA)
{
  PetscErrorCode    ierr;
  PetscInt          n,i,nc,j,rstart,rend;
  const PetscInt    *aj;
  const PetscScalar *aa;
  PetscScalar       *absaa;
  MatGetSize(A,&n,NULL);
  MatDuplicate(A,MAT_COPY_VALUES,AbsA);
  /* Copy over the matrix entries */
  MatGetOwnershipRange(A,&rstart,&rend);
  for (i=rstart; i<rend; i++) {
    MatGetRow(A,i,&nc,&aj,&aa);
  /* Replace the nonzero values with their absolute values */
    PetscMalloc1(nc,&absaa);
    for (j=0; j<nc; j++){
        absaa[j] = fabs(aa[j]);
    }
    MatSetValues(*AbsA,1,&i,nc,aj,absaa,INSERT_VALUES);
    MatRestoreRow(A,i,&nc,&aj,&aa);
  }
  return(0);
```

```
31  }
32
33  int main (int argc,char **argv)
34  {
35    Mat               A=NULL,AL;
36    IS                partitioning;
37    PetscViewer       fd;
38    char              file[PETSC_MAX_PATH_LEN];
39    PetscBool         flg;
40    PetscErrorCode    ierr;
41    MatPartitioning   part;
42
43    /* Read matrix from Petsc Binary File  */
44    PetscInitialize(&argc,&argv,(char*)0,help);if (ierr) return
        ↪  ierr;
45    /* Determine files from which we read matrix */
46    PetscOptionsGetString(NULL,NULL,"-f",file,PETSC_MAX_PATH_LEN,
        ↪  &flg);
47    if (!flg) SETERRQ(PETSC_COMM_WORLD,1,"Must indicate binary
        ↪  file with the -f option");
48    /* Open binary file */
49    PetscViewerBinaryOpen(PETSC_COMM_WORLD,file,FILE_MODE_READ,&f
        ↪  d);
50    /* Load the matrix; then destroy the viewer. */
51    MatCreate(PETSC_COMM_WORLD,&A);
52    MatSetType(A,MATMPIAIJ);
53    MatSetOptionsPrefix(A,"a_");
54    MatSetFromOptions(A);
55    MatLoad(A,fd);
56    PetscViewerDestroy(&fd);
57    /* Start the wall-clock time */
58    PetscLogDouble v1,v2,elapsed_time;
59    PetscTime(&v1);
60    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ↪  - - - - -
61                    Create Partitioning
62    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      ↪  - - - */
63    /* Symmetry check */
64    Mat       Atr,SymmA,Atrabs,Aabs;
65    PetscBool isEqual;
66    Vec       D;
67    PetscInt  i;
68    flg = PETSC_TRUE;
69    PetscOptionsGetBool(NULL,NULL, "-check_symmetry", &flg,NULL);
70    if (flg) {
71      MatIsSymmetric(A,0.0,&isEqual);
72      if (isEqual) {
73          PetscPrintf(PETSC_COMM_WORLD,"Input matrix is
            ↪  symmetric\n\n");
74          absmat(A,&SymmA);
75          } else {
76              PetscPrintf(PETSC_COMM_WORLD,"Input matrix is not
                ↪  symmetric\n\n");
```

```
77              MatTranspose(A,MAT_INITIAL_MATRIX,&Atr);
78              absmat(Atr,&Atrabs);
79              absmat(A,&Aabs);
80              MatAssemblyBegin(Atrabs,MAT_FINAL_ASSEMBLY);
81              MatAssemblyEnd(Atrabs,MAT_FINAL_ASSEMBLY);
82              MatDuplicate(Atrabs,MAT_COPY_VALUES,&SymmA);
83              MatAssemblyBegin(Aabs,MAT_FINAL_ASSEMBLY);
84              MatAssemblyEnd(Aabs,MAT_FINAL_ASSEMBLY);
85              MatAXPY(SymmA,1.,Aabs,DIFFERENT_NONZERO_PATTERN);
86          }
87    }
88    /*  Create adjacency matrix from the symmetric input matrix
      */
89    MatAssemblyBegin(SymmA,MAT_FINAL_ASSEMBLY);
90    MatAssemblyEnd(SymmA,MAT_FINAL_ASSEMBLY);
91    MatConvert(SymmA,MATMPIADJ,MAT_INITIAL_MATRIX,&AL);
92    MatPartitioningCreate(MPI_COMM_WORLD,&part);
93    MatPartitioningSetAdjacency(part,AL);
94    /* Create partitioning for symmetric matrix */
95    MatPartitioningSetFromOptions(part);
96    MatPartitioningApply(part,&partitioning);
97    /*  To find the approx cut, apply the partitioning to
        ↪  nonsymmetric input matrix */
98    PetscMPIInt rank,size;
99    MPI_Comm_rank(PETSC_COMM_WORLD, &rank);
100   MPI_Comm_size(PETSC_COMM_WORLD, &size);
101   PetscInt siz,sizm,j,k;
102   MatGetSize(A,&siz,&sizm);
103   /* Allgather the partitioning  */
104   const PetscInt *idUarr;
105   IS isall;
106   ISAllGather(partitioning,&isall);
107   ISGetIndices(isall,&idUarr);
108   PetscScalar *idUarrint;
109   PetscMalloc1(siz,&idUarrint);
110   for (i=0; i<siz; i++){
111     idUarrint[i] = idUarr[i];
112   }
113   /* Sort the partitioning set to permute A */
114   PetscInt *idxU,idm,idn;
115   PetscMalloc1(siz,&idxU);
116   for (i=0; i<siz;i++){
117       idxU[i] = i;
118   }
119   PetscSortRealWithPermutation(siz,idUarrint,idxU);
120   /* Scatter nodes into clusters based on partitioning
        ↪  information */
121   MatGetLocalSize(A,&idm,&idn);
122   PetscInt sizofidxU;
123   PetscInt *idxUsub;
124   PetscMalloc1(siz,&idxUsub);
125   for(i=0;i<siz;i++){
126       if (i != 0){
127           if (idUarrint[idxU[i]]== idUarrint[idxU[i-1]]){
```

```
                continue;
            }
        }
        if (idUarrint[idxU[i]]== rank){
            sizofidxU = 0;
            k = i;
            j = 0;
            while (idUarrint[idxU[k]] == rank){
                idxUsub[j] = idxU[k];
                sizofidxU++;
                k++;
                j++;
                if (k == siz){
                    break;
                }
            }
       }
    }
    IS idUi;
    ISCreateGeneral(PETSC_COMM_WORLD,sizofidxU,idxUsub,PETSC_COPY
      ↪  _VALUES,&idUi);
    /* Permute the input matrix */
    Mat Ais,PL;
    MatCreateSubMatrix(A,idUi,idUi,MAT_INITIAL_MATRIX,&Ais);
    ISSetPermutation(idUi);
    MatPermute(Ais,idUi,idUi,&PL);
    /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      ↪  - - - - -
         Find the edge-cut by finding off-block diagonal elements
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
    ↪  - - - */
    PetscInt Bm,Bn,Rm,Rn;
    MatGetOwnershipRangeColumn(PL,&Bm,&Bn);
    Mat dum;
    MatDuplicate(PL,MAT_COPY_VALUES,&dum);
    MatSetOption(dum,MAT_NEW_NONZERO_ALLOCATION_ERR,PETSC_FALSE);
    MatGetRowUpperTriangular(dum);
    const PetscInt *cold;
    const PetscScalar *vald;
    PetscInt nval,r;
    PetscScalar sum,sumr,tot;
    PetscScalar *valdn,*sumarr;
    PetscMalloc1(size,&sumarr);
    PetscInt cutt,partt;
    sumr = 0.0;
    for (r = Bm; r<Bn; r++){
        MatGetRow(dum,r,&nval,&cold,&vald);
        PetscMalloc1(nval,&valdn);
        for (j=0;j<nval;j++){
            if(cold[j]>=Bm && cold[j]<Bn){
                valdn[j]=0.0;
            }else{
                    valdn[j]= 1.0;
            }
```

```
179         }
180         sum=0.0;
181         for (j=0;j<nval;j++){
182             sum = sum + valdn[j];
183         }
184         sumr = sumr +sum;
185         MatRestoreRow(dum,r,&nval,&cold,&vald);
186     }
187     PetscScalar global_sum;
188     MPI_Reduce(&sumr, &global_sum, 1, MPIU_SCALAR, MPI_SUM, 0,
189                MPI_COMM_WORLD);
190     if (isEqual) {
191         global_sum = global_sum/2;
192         PetscPrintf(PETSC_COMM_WORLD,"edge cut %g\n",global_sum);
193     }else{
194             PetscPrintf(PETSC_COMM_WORLD,"edge cut
                ↪  %g\n",global_sum);
195     }
196     /*  Stop timer and calculate elapsed time */
197     PetscTime(&v2);
198     elapsed_time = v2 - v1;
199     PetscPrintf(PETSC_COMM_WORLD," \nElapsed time:
          ↪  %2.1e\n\n",elapsed_time);
200     ISDestroy(&partitioning);
201     MatPartitioningDestroy(&part);
202     MatDestroy(&AL);
203     MatDestroy(&A);
204     PetscFinalize();
205     return ierr;
206 }
```

Listing A.1: Matrix Partitioning by Using ParMETIS and CHACO

```c
static char help[] = "Matrix partitioning by using spectral
  ↪  partitioning algorithm with the k-means clustering\n\n";

#include <petscmat.h>
#include <petscis.h>
#include <petscsys.h>
#include <slepceps.h>
#include <stdio.h>
#include <stdlib.h>
#include <petsctime.h>
#include <petscdraw.h>
#include <petscviewer.h>
#include <petscdm.h>
#include <petscdmlabel.h>
#include <petscds.h>
#include <petscsf.h>
#include <mpi.h>
#include "../include/km.h"
#include <math.h>
#include <float.h>
#include <string.h>
#include <time.h>

#define ERR_NO_NUM -1
#define ERR_NO_MEM -2
#define FREED_RAND -3

int *clusters_sizes;

void print_vector(long double *vector, int vector_size) {
  printf("(");
  for (int i = 0; i < vector_size; ++i) {
    if (i > 0)
      printf(", ");
    printf("%Lf", vector[i]);
  }
  printf(")");
}

void print_observations(long double **observations, int
  ↪  observations_size, int vector_size) {
  printf("[");
  for (int i = 0; i < observations_size; ++i) {
    if (i > 0)
      printf(", ");
    print_vector(observations[i], vector_size);
  }
  printf("]");
}

void print_clusters(long double ***clusters, int k, int
  ↪  observations_size, int vector_size) {
  printf("{");
  for (int i = 0; i < k; ++i) {
```

```
52      if (i > 0)
53        printf(", ");
54      print_observations(clusters[i], clusters_sizes[i],
          ↪  vector_size);
55    }
56    free(clusters_sizes);
57    printf("}");
58  }
59
60  int compare_clusters(const int *clusters_map1, const int
      ↪  *clusters_map2, int clusters_size) {
61    int i = 0;
62    while (i < clusters_size) {
63      if (clusters_map1[i] != clusters_map2[i])
64        return 0;
65      ++i;
66    }
67    return 1;
68  }
69
70  long double ***km(long double **observations, int k, int
      ↪  observations_size, int vector_size) {
71    clusters_sizes = (int *) calloc(k, sizeof(int));
72    int *clusters_map = (int *) calloc(observations_size,
        ↪  sizeof(int));
73    long double **cs = initialize(observations, k,
        ↪  observations_size, vector_size);
74
75    if (observations_size < k) {
76      printf("Could not compute clusters.");
77      for (int i = 0; i < k; ++i)
78        free(cs[i]);
79      free(cs);
80      free(clusters_map);
81      free(clusters_sizes);
82      exit(1);
83    }
84    while (1) {
85      int *new_clusters_map = partition(observations, cs, k,
          ↪  observations_size, vector_size);
86      if (compare_clusters(clusters_map, new_clusters_map,
          ↪  observations_size)) {
87        long double ***clusters = map_clusters(clusters_map,
            ↪  observations, k, observations_size, vector_size);
88        for (int i = 0; i < k; ++i)
89          free(cs[i]);
90        free(cs);
91        free(clusters_map);
92        free(new_clusters_map);
93        return clusters;
94      }
95      for (int i = 0; i < k; ++i)
96        free(cs[i]);
97      free(cs);
```

```
98      free(clusters_map);
99      clusters_map = new_clusters_map;
100     cs = re_centroids(clusters_map, observations, k,
          ↪  observations_size, vector_size);
101   }
102 }
103
104 long double *centroid(long double **observations, int
      ↪  observations_size, int vector_size) {
105   long double *vector = (long double *) calloc(vector_size,
        ↪  sizeof(long double));
106
107   for (int i = 0; i < observations_size; ++i) {
108     long double *temp = vsum(vector, observations[i],
          ↪  vector_size);
109     free(vector);
110     vector = temp;
111   }
112   for (int i = 0; i < vector_size; ++i)
113     vector[i] /= observations_size;
114   return vector;
115 }
116
117 long double *vsum(const long double *vector1, const long double
      ↪  *vector2, int vector_size) {
118   long double *vector = (long double *) malloc(sizeof(long
        ↪  double) * vector_size);
119
120   for (int i = 0; i < vector_size; ++i)
121     vector[i] = vector1[i] + vector2[i];
122   return vector;
123 }
124
125 long double *vsub(const long double *vector1, const long double
      ↪  *vector2, int vector_size) {
126   long double *vector = (long double *) malloc(sizeof(long
        ↪  double) * vector_size);
127
128   for (int i = 0; i < vector_size; ++i)
129     vector[i] = vector1[i] - vector2[i];
130   return vector;
131 }
132
133 long double innerprod(const long double *vector1, const long
      ↪  double *vector2, int vector_size) {
134   long double prod = 0;
135
136   for (int i = 0; i < vector_size; ++i)
137     prod += vector1[i] * vector2[i];
138   return prod;
139 }
140
141 long double norm(const long double *vector, int vector_size) {
142   return sqrt(innerprod(vector, vector, vector_size));
```

```
143 }
144
145 /* Source for shuffling algorithm:
      ↪ http://stackoverflow.com/a/5064432 */
146 int rand_num(int size) {
147   static int *numArr = NULL;
148   static int numNums = 0;
149   int i, n;
150
151   if (size == -22) {
152     free(numArr);
153     return FREED_RAND;
154   }
155   if (size >= 0) {
156     if (numArr != NULL)
157       free(numArr);
158     if ((numArr = (int *) malloc(sizeof(int) * size)) == NULL)
159       return ERR_NO_MEM;
160     for (i = 0; i < size; ++i)
161       numArr[i] = i;
162     numNums = size;
163   }
164   if (numNums == 0)
165     return ERR_NO_NUM;
166   n = rand() % numNums;
167   i = numArr[n];
168   numArr[n] = numArr[numNums - 1];
169   numNums--;
170   if (numNums == 0) {
171     free(numArr);
172     numArr = 0;
173   }
174   return i;
175 }
176
177 long double **initialize(long double **observations, int k, int
      ↪ observations_size, int vector_size) {
178   long double **centroids = (long double **) malloc(sizeof(long
        ↪ double *) * k);
179
180   srand(time(NULL));
181   int r = rand_num(observations_size);
182   for (int i = 0; i < k; ++i) {
183     centroids[i] = (long double *) malloc(sizeof(long double) *
          ↪ vector_size);
184     for (int j = 0; j < vector_size; ++j) {
185       centroids[i][j] = observations[r][j];
186       r = rand_num(-1);
187     }
188   }
189   rand_num(-22);
190   return centroids;
191 }
192
```

```
193 int *partition(long double **observations, long double **cs,
      ↪  int k, int observations_size, int vector_size) {
194   int *clusters_map = (int *) malloc(sizeof(int) *
        ↪  observations_size);
195   float curr_distance;
196   int centroid;
197
198   for (int i = 0; i < observations_size; ++i) {
199     float min_distance = DBL_MAX;
200     for (int c = 0; c < k; ++c) {
201       long double *temp = vsub(observations[i], cs[c],
            ↪  vector_size);
202       if ((curr_distance = norm(temp, vector_size)) <
            ↪  min_distance) {
203         min_distance = curr_distance;
204         centroid = c;
205       }
206       free(temp);
207     }
208     clusters_map[i] = centroid;
209   }
210   return clusters_map;
211 }
212
213 long double **re_centroids(int *clusters_map, long double
      ↪  **observations, int k, int observations_size, int
      ↪  vector_size) {
214   long double **centroids = (long double **) malloc(sizeof(long
        ↪  double *) * k);
215   long double **temp_arr = (long double **) malloc(sizeof(long
        ↪  double *) * observations_size);
216
217   for (int c = 0, count = 0; c < k; ++c) {
218     for (int i = 0; i < observations_size; ++i) {
219       int curr = clusters_map[i];
220       if (curr == c) {
221         temp_arr[count] = observations[i];
222         ++count;
223       }
224     }
225     centroids[c] = centroid(temp_arr, count, vector_size);
226     count = 0;
227   }
228   free(temp_arr);
229   return centroids;
230 }
231
232 long double ***map_clusters(int *clusters_map, long double
      ↪  **observations, int k, int observations_size, int
      ↪  vector_size) {
233   long double ***clusters = (long double ***)
        ↪  malloc(sizeof(long double **) * k);
234
235   for (int i = 0; i < k; ++i)
```

```
236     clusters[i] = map_cluster(clusters_map, observations, i,
          ↪  observations_size, vector_size);
237   return clusters;
238 }
239
240 long double **map_cluster(const int *clusters_map, long double
      ↪  **observations, int c, int observations_size, int
      ↪  vector_size) {
241   int count = 0;
242   int *temp_arr = (int *) malloc(sizeof(int) *
        ↪  observations_size);
243
244   for (int i = 0; i < observations_size; ++i) {
245     if (clusters_map[i] == c) {
246       temp_arr[count] = i;
247       ++count;
248     }
249   }
250   long double **cluster = (long double **) malloc(sizeof(long
        ↪  double *) * count);
251   for (int i = 0; i < count; ++i)
252     cluster[i] = observations[temp_arr[i]];
253   free(temp_arr);
254   clusters_sizes[c] = count;
255   return cluster;
256 }
257
258
259 /* Absolute value of matrix */
260
261 PetscErrorCode absmat(Mat A,Mat *AbsA)
262 {
263   PetscErrorCode    ierr;
264   PetscInt          n,i,nc,j,rstart,rend;
265   const PetscInt    *aj;
266   const PetscScalar *aa;
267   PetscScalar   *absaa;
268
269   MatGetSize(A,&n,NULL);
270   MatDuplicate(A,MAT_COPY_VALUES,AbsA);
271
272   /* Copy over the matrix entries */
273   MatGetOwnershipRange(A,&rstart,&rend);
274   for (i=rstart; i<rend; i++) {
275     MatGetRow(A,i,&nc,&aj,&aa);
276
277     /* Replace the nonzero values with their absolute values */
278     PetscMalloc1(nc,&absaa);
279     for (j=0; j<nc; j++){
280         absaa[j] = fabs(aa[j]);
281     }
282     MatSetValues(*AbsA,1,&i,nc,aj,absaa,INSERT_VALUES);
283     MatRestoreRow(A,i,&nc,&aj,&aa);
284   }
```

```
285   return(0);
286 }
287
288 int main (int argc,char **argv)
289 {
290   EPS               eps;
291   Mat
        ↪  A=NULL,Atr,SymmA,Atrabs,Aabs,L,PL,NSymmA,NSymmAtrabs;
292   int               s;
293   Vec               x,D,DD,vr,DDD;
294   EPSType           type;
295   PetscInt          i,nev,*idx,mm,nn,rw,ncols,j,siz;
296   PetscScalar       kr,none=-1.0,*arr;
297   const PetscScalar *vals;
298   const PetscInt    *cols;
299   IS                is,partitioning;
300   PetscViewer       fd;
301   char              file[PETSC_MAX_PATH_LEN];
302   PetscBool         flg,isEqual,unw,wgh;
303   PetscErrorCode    ierr;
304   MatPartitioning   part;
305
306
307   SlepcInitialize(&argc,&argv,(char*)0,help);if (ierr) return
        ↪  ierr;
308   PetscInitialize(&argc,&argv,(char*)0,help);if (ierr) return
        ↪  ierr;
309
310   /* Determine files from which we read matrix */
311   PetscOptionsGetString(NULL,NULL,"-f",file,PETSC_MAX_PATH_LEN,
        ↪  &flg);
312   if (!flg) SETERRQ(PETSC_COMM_WORLD,1,"Must indicate binary
        ↪  file with the -f option");
313
314   /* Open binary file */
315   PetscViewerBinaryOpen(PETSC_COMM_WORLD,file,FILE_MODE_READ,&f
        ↪  d);
316
317   /* Load the matrix; then destroy the viewer. */
318   MatCreate(PETSC_COMM_WORLD,&A);
319   MatSetType(A,MATMPIAIJ);
320   MatSetOptionsPrefix(A,"a_");
321   MatSetFromOptions(A);
322   MatLoad(A,fd);
323   PetscViewerDestroy(&fd);
324
325    /* Start the wall-clock time */
326   PetscLogDouble v1,v2,elapsed_time;
327   total_elapsed_time = 0;
328   PetscTime(&v1);
329
330     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ↪  - - - - - -
331                 Create Laplacian
```

```c
    - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

    /* Symmetry check */
    flg = PETSC_TRUE;
    PetscOptionsGetBool(NULL,NULL, "-check_symmetry", &flg,NULL);

    if (flg) {
        MatIsSymmetric(A,0.0,&isEqual);
        if (isEqual) {
            PetscPrintf(PETSC_COMM_WORLD,"Input matrix is symmetric\n\n");
            absmat(A,&SymmA);
        } else {
            PetscPrintf(PETSC_COMM_WORLD,"Input matrix is not symmetric\n\n");
            MatTranspose(A,MAT_INITIAL_MATRIX,&Atr);
            absmat(Atr,&Atrabs);
            absmat(A,&Aabs);
            MatAssemblyBegin(Atrabs,MAT_FINAL_ASSEMBLY);
            MatAssemblyEnd(Atrabs,MAT_FINAL_ASSEMBLY);
            MatDuplicate(Atrabs,MAT_COPY_VALUES,&SymmA);
            MatAssemblyBegin(Aabs,MAT_FINAL_ASSEMBLY);
            MatAssemblyEnd(Aabs,MAT_FINAL_ASSEMBLY);
            MatAXPY(SymmA,1.,Aabs,DIFFERENT_NONZERO_PATTERN);
        }
    }
    PetscOptionsHasName(NULL,NULL,"-unweighted",&unw);
    PetscOptionsHasName(NULL,NULL,"-weighted",&wgh);

    /* Unweighted Laplacian */
    if (unw) {
        /* DD = diag(SymmA) */
        MatCreateVecs(SymmA,&DD,NULL);
        VecZeroEntries(DD);
        MatAssemblyBegin(SymmA,MAT_FINAL_ASSEMBLY);
        MatAssemblyEnd(SymmA,MAT_FINAL_ASSEMBLY);
        MatGetDiagonal(SymmA,DD);
        /* L = SymmN-diag(DD)  */
        VecScale(DD,none);
        MatDuplicate(SymmA,MAT_COPY_VALUES,&L);
        MatSetOption(L,MAT_NEW_NONZERO_ALLOCATION_ERR,PETSC_FALSE);
        MatDiagonalSet(L,DD,ADD_VALUES);
        /*
         [row,col,values] = find(L)
         values = -1
         RestNew = sparse(row,col,values,x,y)
        */
        MatGetOwnershipRange(L,&mm,&nn);
        MatDuplicate(L,MAT_DO_NOT_COPY_VALUES,&NSymmA);
        MatSetOption(NSymmA,MAT_NEW_NONZERO_ALLOCATION_ERR,PETSC_FALSE);
```

```
380         for(rw = mm; rw<nn; ++rw){
381             MatGetRow(L,rw,&ncols,&cols,&vals);
382             s = ncols;
383             PetscMalloc1(s,&arr);
384             for(j=0;j<s;++j){
385                 arr[j]=-1.0;
386             }
387             MatSetValues(NSymmA,1,&rw,ncols,cols,arr,INSERT_VALUES);
388             MatRestoreRow(L,rw,&ncols,&cols,&vals);
389         }
390         MatAssemblyBegin(NSymmA,MAT_FINAL_ASSEMBLY);
391         MatAssemblyEnd(NSymmA,MAT_FINAL_ASSEMBLY);
392         MatCreateVecs(NSymmA,&DDD,NULL);
393         VecZeroEntries(DDD);
394         MatDiagonalSet(NSymmA,DDD,INSERT_VALUES);
395         /* D = sum(abs(NSymmA')) */
396         absmat(NSymmA,&NSymmAtrabs);
397         MatCreateVecs(NSymmAtrabs,&D,NULL);
398         VecZeroEntries(D);
399         MatAssemblyBegin(NSymmAtrabs,MAT_FINAL_ASSEMBLY);
400         MatAssemblyEnd(NSymmAtrabs,MAT_FINAL_ASSEMBLY);
401         MatGetRowSum(NSymmAtrabs,D);
402         /* L = diag(D) + NSymmA */
403         MatDuplicate(NSymmA,MAT_COPY_VALUES,&L);
404         MatSetOption(L,MAT_NEW_NONZERO_ALLOCATION_ERR,PETSC_FALSE);
405         MatDiagonalSet(L,D,ADD_VALUES);
406     } else if (wgh) { /* Weighted Laplacian */
407         /* Initialize D as zero vector */
408         MatCreateVecs(SymmA,&D,NULL);
409         VecZeroEntries(D);
410         VecDuplicate(D,&DD);;
411         /* D = sum(SymmA) */
412         MatAssemblyBegin(SymmA,MAT_FINAL_ASSEMBLY);
413         MatAssemblyEnd(SymmA,MAT_FINAL_ASSEMBLY);
414         MatGetRowSum(SymmA,D);
415         /* L = -(SymmA-diag(diag(SymmA)))+diag(D) */
416         MatDuplicate(SymmA,MAT_DO_NOT_COPY_VALUES,&L);
417         MatSetOption(SymmA,MAT_NEW_NONZERO_ALLOCATION_ERR,PETSC_FALSE);
418         MatDiagonalSet(SymmA,DD,INSERT_VALUES);
419         MatAXPY(L,-1.,SymmA,DIFFERENT_NONZERO_PATTERN);
420         MatSetOption(L,MAT_NEW_NONZERO_ALLOCATION_ERR,PETSC_FALSE);
421         MatDiagonalSet(L,D,ADD_VALUES);
422     }
423     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
424                        Create the eigensolver and solve the
    eigensystem
425        - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
426     PetscMPIInt rank,size;
```

```
427   MPI_Comm_rank(PETSC_COMM_WORLD, &rank);
428   MPI_Comm_size(PETSC_COMM_WORLD, &size);
429   EPSCreate(PETSC_COMM_WORLD,&eps);
430   EPSSetOperators(eps,L,NULL);
431   EPSSetProblemType(eps,EPS_HEP);
432   EPSSetDimensions(eps,size,PETSC_DEFAULT,PETSC_DEFAULT);
433   EPSSetWhichEigenpairs(eps,EPS_SMALLEST_MAGNITUDE);
434   EPSSetFromOptions(eps);
435   MatCreateVecs(L,&x,NULL);
436   VecSet(x,1.0);
437   EPSSetDeflationSpace(eps,1,&x);
438   VecDestroy(&x);
439   EPSSolve(eps);
440   EPSGetType(eps,&type);
441   PetscPrintf(PETSC_COMM_WORLD," Solution method: %s\n\n",type);
442   EPSGetDimensions(eps,&nev,NULL,NULL);
443   PetscPrintf(PETSC_COMM_WORLD," Number of requested
        ↪  eigenvalues: %D\n",nev);
444   MatCreateVecs(L,&vr,NULL);
445   Vec *V;
446   VecDuplicateVecs(vr,nev,&V);
447   for (i=0; i<nev;i++){
448     EPSGetEigenpair(eps,i,&kr,NULL,V[i],NULL);
449   }
450   VecGetSize(vr,&siz);
451   /* Define a matrix U so that each eigenvector is a column of
        ↪  U */
452   Mat U;
453   PetscScalar *valsu;
454   const PetscScalar *va;
455   PetscInt urstart,urend;
456   PetscInt sizV;
457   PetscInt k;
458   MatCreateDense(PETSC_COMM_WORLD,PETSC_DECIDE,PETSC_DECIDE,siz ⌋
        ↪  ,nev,NULL,&U);
459   MatSetUp(U);
460   MatGetOwnershipRange(U,&urstart,&urend);
461   for (i=0; i<nev;i++){
462     MatDenseGetColumn(U,i,&valsu);
463     VecGetArrayRead(V[i],&va);
464     VecGetLocalSize(V[i],&sizV);
465     PetscInt mmm;
466     for (j=0; j<sizV; j++) {
467         mmm = rank*sizV+j;
468         MatSetValues(U,1,&mmm,1,&i,&va[j],INSERT_VALUES);
469     }
470     MatDenseRestoreColumn(U,&valsu);
471     VecRestoreArrayRead(V[i],&va);
472     }
473     MatAssemblyBegin(U,MAT_FINAL_ASSEMBLY);
474     MatAssemblyEnd(U,MAT_FINAL_ASSEMBLY);
475   /*  Get each row of U as row vector */
476   PetscInt *idxU,idm,idn;
477   PetscMalloc1(siz,&idxU);
```

```
478     for (i=0; i<siz;i++){
479         idxU[i] = i;
480     }
481     PetscInt *idxUc;
482     PetscMalloc1(nev,&idxUc);
483     for (i=0; i<nev;i++){
484         idxUc[i] = i;
485     }
486     IS idUi,idUic;
487     ISCreateGeneral(PETSC_COMM_WORLD,siz,idxU,PETSC_COPY_VALUES,&
          ↪  idUi);
488     ISCreateGeneral(PETSC_COMM_WORLD,nev,idxUc,PETSC_COPY_VALUES,
          ↪  &idUic);
489     Mat *submat;
490     MatCreateSubMatrices(U,1,&idUi,&idUic,MAT_INITIAL_MATRIX,&sub
          ↪  mat);
491     Vec duv,*UV;
492     VecCreate(PETSC_COMM_SELF,&duv);
493     VecSetSizes(duv,PETSC_DECIDE,nev);
494     VecSetUp(duv);
495     VecDuplicateVecs(duv,siz,&UV);
496     const PetscScalar *arrayone;
497     PetscScalar *arrUV;
498     MatDenseGetArrayRead(submat[0],&arrayone);
499     for (i=0;i<siz;i++){
500         VecGetArray(UV[i],&arrUV);
501         for(j=0;j<nev;j++){
502             arrUV[j]=arrayone[i+j*siz];
503         }
504         VecRestoreArray(UV[i],&arrUV);
505         VecRestoreArray(UV[i],&arrUV);
506     }
507     MatDenseRestoreArrayRead(submat[0],&arrayone);
508     MatDestroySubMatrices(1,&submat);
509     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          ↪  - - - - -
510                Apply k-means clustering algorithm to row vectors
511     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ↪  - - - */
512     PetscInt nA;
513     int nAi;
514     MatGetSize(A,&nA,NULL);
515     nAi = nA;
516     int observations_size = nAi;
517     int vector_size = nev;
518     int kc = nev;
519     long double **observations;
520     long double ***clusters;
521     observations = (long double **) malloc(sizeof(long double *)
          ↪  * observations_size);
522     for (int i = 0; i < observations_size; i++){
523           observations[i] = (long double *) malloc(sizeof(long
              ↪  double) * vector_size);
524     }
```

```
525     int stepsiz,kj;
526     stepsiz = sizV;
527     for (int i = 0; i < observations_size; i++) {
528           VecGetArray(UV[i],&arrUV);
529           if(i%stepsiz == 0){
530               kj = i/stepsiz;
531           }
532           for (int j = 0; j < vector_size; j++){
533               observations[i][j] = arrUV[j];
534           }
535           VecRestoreArray(UV[i],&arrUV);
536     }
537     clusters = km(observations, kc, observations_size,
        ↪  vector_size);
538     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ↪  - - - - -
539           Partition the nodes based on clustering information
540     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      ↪  - - - */
541     Vec vecidU;
542     PetscScalar *idUarrint;
543     PetscMalloc1(siz,&idUarrint);
544     PetscScalar *idUarridx;
545     PetscScalar *idxstop,idxs;
546     PetscMalloc1(nev,&idxstop);
547     for(j=0;j<nev;j++){
548         PetscMalloc1(clusters_sizes[j],&idUarridx);
549         if (j == rank){
550           for (k=0;k<clusters_sizes[j];k++){
551               i=0;
552               while(clusters[j][k] != observations[i]){
553                   i++;
554               }
555               idUarridx[k] = i;
556           }
557         }else {
558               continue;
559         }
560         idxstop[j]=k;
561         idxs = k;
562         VecCreateMPIWithArray(PETSC_COMM_WORLD,1,clusters_sizes[j
          ↪  ],siz,idUarridx,&vecidU);
563     }
564     VecScatter vecctx;
565     Vec idUall,idUalldum,vecidUdum;
566     PetscInt ls,vstart,vend,*vecar;
567     PetscScalar rk;
568     const PetscScalar *vecarr;
569     VecScatterCreateToAll(vecidU,&vecctx,&idUall);
570     VecScatterBegin(vecctx,vecidU,idUall,INSERT_VALUES,SCATTER_FO
        ↪  RWARD);
571     VecScatterEnd(vecctx,vecidU,idUall,INSERT_VALUES,SCATTER_FORW
        ↪  ARD);
572     VecDuplicate(vecidU,&vecidUdum);
```

```
573     VecGetLocalSize(vecidU,&ls);
574     VecGetOwnershipRange(vecidUdum,&vstart,&vend);
575     for(i=vstart;i<vend;i++){
576         rk = (PetscReal) (rank*1.0);
577         VecSetValues(vecidUdum,1,&i,&rk,INSERT_VALUES);
578     }
579     VecGetArrayRead(vecidU,&vecarr);
580     PetscMalloc1(vend-vstart,&vecar);
581     for(i=0;i<vend-vstart;i++){
582         vecar[i] = vecarr[i];
583     }
584     IS idvec;
585     ISCreateGeneral(PETSC_COMM_WORLD,vend-vstart,vecar,PETSC_COPY
          ↪  _VALUES,&idvec);
586     /* Permute the input matrix */
587     Mat Ais;
588     MatCreateSubMatrix(A,idvec,idvec,MAT_INITIAL_MATRIX,&Ais);
589     ISSetPermutation(idvec);
590     MatPermute(Ais,idvec,idvec,&PL);
591     /* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          ↪  - - - - -
592          Find the edge-cut by finding off-block diagonal elements
593     - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        ↪  - - - */
594     PetscBool cunw,cwgh;
595     PetscOptionsHasName(NULL,NULL,"-unweightedcut",&cunw);
596     PetscOptionsHasName(NULL,NULL,"-weightedcut",&cwgh);
597     PetscInt Bm,Bn,Rm,Rn;
598     MatGetOwnershipRange(PL,&Bm,&Bn);
599     const PetscInt *cold;
600     const PetscScalar *vald;
601     PetscInt nval,r;
602     PetscScalar sum,sumr,tot,*valdn,*sumarr;
603     PetscMalloc1(nev,&sumarr);
604     sumr = 0.0;
605     for (r = Bm; r<Bn; r++){
606         MatGetRow(PL,r,&nval,&cold,&vald);
607         PetscMalloc1(nval,&valdn);
608         for (j=0;j<nval;j++){
609            if(cold[j]>=Bm && cold[j]<=Bn){
610               valdn[j]=0.0;
611           }else{
612               if(cunw){
613                   valdn[j]= 1.0;
614               }else if(cwgh){
615
616                   if(vald[j]<0){
617                       valdn[j]= -1.0*vald[j];
618                   }else{
619                       valdn[j]= vald[j];
620                   }
621               }
622           }
623       }
```

```
624         sum=0.0;
625         for (j=0;j<nval;j++){
626             sum = sum + valdn[j];
627         }
628         sumr = sumr +sum;
629         MatRestoreRow(PL,r,&nval,&cold,&vald);
630     }
631     PetscScalar global_sum;
632     MPI_Reduce(&sumr, &global_sum, 1, MPIU_SCALAR, MPI_SUM,
        ↪  0,MPI_COMM_WORLD);
633     if (isEqual) {
634         global_sum = global_sum/2;
635         PetscPrintf(PETSC_COMM_WORLD,"edge cut %g\n",global_sum);
636     }else{
637         PetscPrintf(PETSC_COMM_WORLD,"edge cut %g\n",global_sum);
638     }
639     VecDestroy(&DD);
640     VecDestroy(&D);
641     MatDestroy(&SymmA);
642     MatDestroy(&L);
643     VecDestroy(&vr);
644     /*  Stop timer and calculate elapsed time */
645     PetscTime(&v2);
646     elapsed_time = v2 - v1;
647     PetscPrintf(PETSC_COMM_WORLD," \nElapsed time:
        ↪  %2.1e\n\n",elapsed_time);
648     MatDestroy(&A);
649     EPSDestroy(&eps);
650     SlepcFinalize();
651     return ierr;
652 }
```

Listing A.2: Matrix partitioning by using spectral partitioning algorithm with the k-means clustering

APPENDIX B

TABLES

In this appendix, numerical results obtained from the partitioning of medium and large-sized matrices are given. For medium-sized matrices, load imbalance, the number of iterations, and edge-cuts are obtained by using `eigs` and `kmeans` routines of MATLAB. For large-sized matrices, edge-cut and partitioning time are given as results when SPEC, ParMETIS, and CHACO are used for partitioning.

Table B.1: Results obtained from the partitioning of *cz148*

	NPARTS	2		4		8		16	
	LAP	W	U	W	U	W	U	W	U
TOL = E-2	**ITER**	245.3	95.6	182.1	**78.9**	109	**67.1**	**176.1**	**160.7**
	V.RATIO	**1.05**	1.11	**1.10**	**1.11**	1.32	1.34	**1.55**	2.41
	CUT	**737.26**	85	930.22	154.2	2633.31	287.4	4852.13	**401.9**
	E.RATIO	**1.06**	1.12	**1.10**	**1.16**	1.35	1.43	**1.62**	2.97
TOL = E-4	**ITER**	225.5	**85**	**177.8**	80.6	**107.7**	114	251.5	222.9
	V.RATIO	**1.05**	1.12	1.11	1.22	1.44	1.57	1.70	2.24
	CUT	**737.26**	84.4	930.07	**149.6**	2589.66	273.7	4791.64	412.6
	E.RATIO	**1.06**	1.13	1.11	1.29	1.49	1.71	1.80	2.71
TOL = E-6	**ITER**	**200.7**	89	183.8	80.5	140.3	159.1	305.5	261.6
	V.RATIO	**1.05**	1.14	1.20	1.19	**1.17**	1.37	1.56	2.29
	CUT	**737.26**	**84**	939.69	152	2570.21	279.6	4817.13	420.9
	E.RATIO	**1.06**	1.14	1.20	1.26	**1.19**	1.51	1.62	2.78
TOL = E-8	**ITER**	226	96	190.2	107.8	178.7	219.8	377	319.4
	V.RATIO	**1.05**	**1.10**	1.10	1.14	1.30	1.4	1.66	2.22
	CUT	**737.26**	85.4	**929.91**	151.8	2600.31	283.2	4745.64	429
	E.RATIO	**1.06**	**1.11**	1.11	1.20	1.34	1.54	1.76	2.70
TOL = E-10	**ITER**	265.1	109.9	207	131.1	230.2	282.8	430.1	379.2
	V.RATIO	**1.05**	1.113	1.16	1.13	1.29	**1.29**	1.65	**2.21**
	CUT	**737.26**	84.8	952.36	152.4	**2539.11**	**272**	**4696.18**	417.4
	E.RATIO	**1.06**	1.12	1.17	1.19	1.32	**1.40**	1.76	**2.70**

Table B.2: Results obtained from the partitioning of *Poisson(12)*

	NPARTS	2		4		8		16	
	LAP	W	U	W	U	W	U	W	U
TOL = E-2	**ITER**	206.2	154.2	**146.5**	**116.3**	**94.5**	**75.5**	**141.1**	**144.3**
	V.RATIO	**1.01**	**1.01**	1.64	1.58	1.62	1.83	**1.72**	1.8
	CUT	2551.9	**15.3**	5999.5	37.8	11137.1	65.3	14770.6	85.1
	E.RATIO	1.01	**1.01**	1.68	1.63	1.65	1.89	**1.80**	1.90
TOL = E-4	**ITER**	197.4	143.6	152.3	118.9	167.4	157.5	249.5	252.8
	V.RATIO	1.02	1.03	1.53	1.56	1.53	1.57	1.76	1.87
	CUT	2501.2	17	6168.5	38.5	9768.2	61	13705.9	83.4
	E.RATIO	1.02	1.03	1.56	1.61	1.56	1.63	1.82	2.00
TOL = E-6	**ITER**	**180.9**	**129.6**	159	127.8	596	507.3	392.5	364.9
	V.RATIO	1.01	1.03	1.51	1.58	1.66	**1.58**	1.8	1.91
	CUT	**2484.3**	**15.3**	6506.5	36.4	**8534.5**	51.5	13739.7	82.2
	E.RATIO	1.01	1.03	1.54	1.63	1.69	**1.57**	1.87	2.06
TOL = E-8	**ITER**	183.8	135.5	297.9	272.4	626.4	624.4	493.9	503.1
	V.RATIO	1.01	1.02	1.19	1.31	1.52	1.53	1.86	**1.7**
	CUT	2619.5	16.3	4512.3	30.1	8568.3	**50.3**	**13503.1**	**79**
	E.RATIO	1.01	1.02	1.21	1.34	1.54	1.59	1.94	**1.82**
TOL = E-10	**ITER**	591.6	149.7	368.8	372.1	689	685.1	553.2	563.2
	V.RATIO	**1.01**	1.02	**1**	**1**	**1.5**	1.62	1.92	1.82
	CUT	2568.8	15.4	**4056**	**24**	8686.6	52.1	**13503.1**	80.7
	E.RATIO	**1.01**	1.02	**1**	**1**	**1.53**	1.68	2.02	1.99

Table B.3: Results obtained from the partitioning of *lshp_265*

	NPARTS	2		4		8		16	
	LAP	W	U	W	U	W	U	W	U
TOL = E-2	**ITER**	232.4	250.7	**178.1**	**175.9**	**181.5**	**207.4**	**190.6**	**243**
	V.RATIO	**1.05**	**1.05**	1.26	1.27	1.51	1.46	1.68	**1.6**
	CUT	**32**	**32**	**67.2**	67.5	120.7	122	192.1	191.2
	E.RATIO	**1.05**	**1.05**	1.27	1.28	1.57	1.50	1.80	**1.71**
TOL = E-4	**ITER**	**168.2**	**182.7**	190.9	176.9	318.1	311	407.5	393.9
	V.RATIO	**1.05**	**1.05**	**1.26**	1.26	1.51	1.49	1.73	1.73
	CUT	**32**	**32**	67.4	67.3	120.9	121.6	193.1	**190.6**
	E.RATIO	**1.05**	**1.05**	**1.26**	1.27	1.57	1.54	1.86	1.87
TOL = E-6	**ITER**	206.6	204.8	224.5	206	412.3	416.4	560.9	571.7
	V.RATIO	**1.05**	**1.05**	1.27	**1.25**	1.49	1.52	1.70	1.63
	CUT	**32**	**32**	67.7	67.3	122.1	120.8	**190.5**	193.2
	E.RATIO	**1.05**	**1.05**	1.28	**1.26**	1.54	1.58	1.84	1.75
TOL = E-8	**ITER**	263.8	253.1	333.5	308.3	487.7	493.7	733.4	739.8
	V.RATIO	**1.05**	**1.05**	1.26	1.27	1.59	1.56	**1.64**	1.71
	CUT	**32**	**32**	67.4	67.6	**120.4**	**120.4**	192.8	190.8
	E.RATIO	**1.05**	**1.05**	1.28	1.28	1.66	1.63	**1.76**	1.84
TOL = E-10	**ITER**	323.1	313.6	417.8	402.8	580.3	609	883.7	884.6
	V.RATIO	**1.05**	**1.05**	1.26	1.26	**1.44**	**1.45**	1.74	1.70
	CUT	**32**	**32**	67.8	**67.2**	122	122	190.8	192.4
	E.RATIO	**1.05**	**1.05**	1.27	1.27	**1.48**	**1.50**	1.87	1.83

Table B.4: Results obtained from the partitioning of *can_161*

	NPARTS	2		4		8		16	
	LAP	W	U	W	U	W	U	W	U
TOL = E-2	**ITER**	94.5	92	82.7	**84.6**	**79.2**	**84.6**	**149.5**	**137.3**
	V.RATIO	**1.01**	**1.01**	1.16	1.13	1.54	1.51	1.78	1.73
	CUT	**48**	**48**	109.8	107.7	182	186.2	246.2	247.9
	E.RATIO	**1.025761**	**1.03**	1.19	1.15	1.62	1.55	1.96	1.93
TOL = E-4	**ITER**	**78.3**	**74.6**	**82.3**	85	218.6	180.4	238.2	235.3
	V.RATIO	**1.01**	**1.01**	**1.07**	**1.10**	1.45	1.56	1.62	**1.50**
	CUT	**48**	**48**	107.1	107.5	167.2	166.2	245.5	244.2
	E.RATIO	**1.03**	**1.03**	**1.07**	1.11	1.53	1.68	1.83	**1.66**
TOL = E-6	**ITER**	82.8	81.1	98.9	99.1	248.1	247.4	249.8	244
	V.RATIO	**1.01**	**1.01**	1.12	**1.10**	1.47	1.58	**1.47**	1.64
	CUT	**48**	**48**	107.7	107.3	163.9	163.2	243.1	**240.1**
	E.RATIO	**1.03**	**1.03**	1.15	**1.11**	1.56	1.68	**1.63**	1.83
TOL = E-8	**ITER**	95.7	90.2	274.4	276.1	254.5	256.4	253.2	256.5
	V.RATIO	**1.01**	**1.01**	1.43	1.49	1.41	**1.44**	1.55	1.72
	CUT	**48**	**48**	100.4	99.4	166.4	164.3	**241.7**	243.4
	E.RATIO	**1.03**	**1.03**	1.51	1.57	1.46	**1.50**	1.71	1.93
TOL = E-10	**ITER**	112	111.2	401.1	411.1	269.8	271.3	590.9	621.8
	V.RATIO	**1.01**	**1.01**	1.65	1.64	**1.40**	1.48	1.57	1.64
	CUT	**48**	**48**	**95.8**	**95.9**	**163.6**	**161.7**	243	244.6
	E.RATIO	**1.03**	**1.03**	1.77	1.76	**1.46**	1.55	1.76	1.82

Table B.5: Results obtained from the partitioning of *bcspwr10*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	**3447.8**	3632.4	**5472.57**	5512	6738.17	6627.77	7368.8	7339.55
		TIME	**0.97**	2.68	**2.143**	6.11	**4.94**	8.12	**11.5**	32.11
	E-4	CUT	3683.6	3657.9	5718.17	5615.14	6556.2	6577.87	7418.44	7322.75
		TIME	2.18	3.71	4.80	8.27	8.15	10.57	32.22	33.75
	E-6	CUT	3585.1	**3524**	5633.33	**5456.77**	**6493.45**	6584.53	7355.55	7353.1
		TIME	3.15	5.34	7.70	9.22	9.41	13.89	29.67	34.8
	E-8	CUT	3537.8	3546.5	5636.43	5645.83	6587.1	**6548.96**	7426	7402
		TIME	4.94	6.53	9.54	11	13	15	35.88	33.8
	E-10	CUT	3540.2	3598.44	5532.67	5662.23	6621.83	6592.37	**7318.4**	7394.71
		TIME	5.9	7.63	10.57	12	15	17	34.9	43.14
PARMETIS		CUT	-	3700.8	-	5934.19	-	6795.48	-	6763.5
		TIME	-	0.087	-	**0.09**	-	**0.10**	-	**0.23**
CHACO		CUT	-	3741.3	-	5952.58	-	6860.32	-	**6760.8**
		TIME	-	**0.08**	-	0.09	-	0.11	-	0.24

Table B.6: Results obtained from the partitioning of *epb2*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	7.15	**2027.4**	**9.26**	**12546.58**	4751.72	42456.67	7592.20	44220.38
		TIME	**3.18**	19.3	**3.84**	61.69	**5.28**	47.33	17.38	70.125
	E-4	CUT	6.28	2956.7	505.25	13684.70	5854.22	41508.1	7968.41	42587.83
		TIME	8.94	57.9	8.99	80.11	11.2	67.2	16.71	85.5
	E-6	CUT	**5.53**	2925.3	420.71	13285.27	**3398.72**	**38200.44**	7070.59	**42534.11**
		TIME	10.8	70.1	11.30	94.22	16.125	81	17.12	104.89
	E-8	CUT	5.66	2091.8	387.89	13572.43	4725.07	38872.1	**7059.14**	44797.33
		TIME	13.78	92.9	13.04	105	18.5	90.8	**16.62**	113.33
	E-10	CUT	6.32	2979.1	524.43	12883.21	5656.77	38673.17	7799.03	44259.88
		TIME	16.2	98.5	15.04	110.69	21.8	110	18.14	120
PARMETIS		CUT	-	5349.6	-	18628.1	-	59409.9	-	73315.8
		TIME	-	**0.74**	-	**0.53**	-	**0.41**	-	**0.54**
CHACO		CUT	-	3552.3	-	25253.2	-	48176.1	-	74211.3
		TIME	-	0.8	-	0.6	-	0.48	-	0.56

Table B.7: Results obtained from the partitioning of *sme3Da*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	**177564.25**	347856.67	**3906704.24**	**568983.07**	33099257.9	723371.54	102601278	785580.6
		TIME	**3.42**	3.22	**4.19**	3.16	**8.98**	4.35	**20**	14
	E-4	CUT	325709.16	360182.2	4732248.33	577783.69	21666568.9	713939.43	**77908288.9**	788251.3
		TIME	4.35	3.65	5.66	3.36	10.08	4.83	22.11	14.3
	E-6	CUT	490991.77	378146.6	4061762.75	580390.52	20173584.9	**709642.93**	87369850	784247
		TIME	5.28	4.14	7.64	3.65	11.69	5.40	22.33	14.22
	E-8	CUT	186245.28	**342154.2**	3955129	571332.96	22265248.2	713608.64	101834075	780475.67
		TIME	7.2	4.73	7.54	3.99	11.89	5.81	22.12	14.5
	E-10	CUT	626977.78	377228.3	4550442.76	580133.83	**18103760.7**	721403.36	84627162.5	779831.22
		TIME	8.14	5.08	9.38	4.28	13.17	6.16	23.88	18.22
PARMETIS		CUT	-	386491	-	624002	-	728797	-	**725362**
		TIME	-	**0.87**	-	**0.74**	-	**0.62**	-	**0.87**
CHACO		CUT	-	386726	-	624429	-	728487	-	725628
		TIME	-	0.96	-	0.96	-	1	-	1.3

Table B.8: Results obtained from the partitioning of *av41092*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	**3.61**	**54**	2421.38	8485.89	326045.62	1011116.22	366911.44	1215482.86
		TIME	**293**	419	**91.29**	140.74	**43.62**	74.78	**166.67**	111.29
	E-4	CUT	5.52	**54**	3245.53	10232.86	328031.38	**931911**	**357798.86**	**1170944**
		TIME	298	423	95.44	143.93	58.12	79.67	221.43	148
	E-6	CUT	7.65	**54**	**2311.72**	**8294.48**	328670.67	942487.4	371301.6	1233671.67
		TIME	302	426	96.93	147.93	68.44	82.4	220	155
	E-8	CUT	7.65	**54**	2529.52	9602.14	**323929.67**	949196.3	365404.86	1207084.44
		TIME	310	435	100.77	150.36	78.67	91	277.14	163.33
	E-10	CUT	6.70	**54**	2916.34	11409.61	328781.33	989577.33	359577.75	1228014.29
		TIME	316	435	110	152.14	107.78	100.33	280	274.29
PARMETIS		CUT	-	570071	-	1071490	-	1231760	-	1357700
		TIME	-	**4**	-	**3.8**	-	**2.9**	-	**3.3**
CHACO		CUT	-	756486	-	1133640	-	1176020	-	1334670
		TIME	-	7.7	-	8.3	-	8.1	-	10

Table B.9: Results obtained from the partitioning of *poisson3Db*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	**1569.63**	1119165	8642.87	1666558.57	14530.88	1953335.88	27935.2	2111390
		TIME	**19**	26.5	**20.26**	27.29	**29.2**	38.29	78	150
	E-4	CUT	2002.61	1120924.44	**5115.87**	1687069.58	12259.07	1955761.11	25855.53	2110710
		TIME	20.5	31.78	24.12	31.17	32.89	43.44	92.33	182
	E-6	CUT	2068.89	1121311.11	5169.94	1672333.13	11506.42	1950050.53	21594.2	2109636.67
		TIME	22.4	37.11	26.31	39.19	33.74	47.74	**70.5**	140.67
	E-8	CUT	1768.73	1121640	5509.39	1672930.37	**11160.70**	1957599.44	**20725.4**	2109037.5
		TIME	24.11	42.5	27.4	42.85	37.2	49.39	72	120.5
	E-10	CUT	1868.28	1119676	5349.22	1687595.79	11453.59	1960292.5	25606.7	2098330
		TIME	25.3	48.1	28.69	46.95	38.76	56.5	79	132.5
PARMETIS		CUT	-	1021800	-	**1642050**	-	**1920710**	-	**1909780**
		TIME	-	**8.8**	-	**5.6**	-	**3.6**	-	**3**
CHACO		CUT	-	**1018730**	-	1648950	-	1922440	-	1912030
		TIME	-	9	-	6.2	-	4.6	-	4.7

Table B.10: Results obtained from the partitioning of *rw5151*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	820.98	1686.22	1276.79	4336.67	**1955.34**	7902.7	2313.21	9245.5
		TIME	**1.98**	1.71	**3.07**	2.25	**5.71**	4.19	**13.3**	12.4
	E-4	CUT	802.58	3088.5	1231.70	4485.59	2031.85	**7827.44**	2315.51	9232.25
		TIME	2.47	2.02	3.83	2.69	6.57	5.18	15.11	14.25
	E-6	CUT	949.86	1527.12	**1174.25**	4525.59	2077.29	8004.38	2314.92	9247.8
		TIME	2.94	2.25	4.47	3.06	7.48	5.29	14.44	14.5
	E-8	CUT	808.19	711.6	1225.14	4512.38	2061.27	7874.32	2324.18	**9145.12**
		TIME	3.35	2.5	5.08	3.54	8.54	5.57	17.3	14.12
	E-10	CUT	**665.61**	2461	1222.98	4351.10	1975.16	7831.34	**2300.82**	9300.75
		TIME	3.73	2.69	5.67	3.7	9.32	6.19	17.22	15.75
PARMETIS		CUT	-	1192.5	-	6030.97	-	9015.48	-	10861.2
		TIME	-	**0.07**	-	**0.079**	-	**0.09**	-	**0.22**
CHACO		CUT	-	**323.1**	-	**3918.39**	-	8674.84	-	10044.9
		TIME	-	0.07	-	0.09	-	0.1	-	**0.22**

Table B.11: Results obtained from the partitioning of *FEM_3D_thermal1*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	3.58	**82531.8**	5936.78	162424	6883.76	191996.21	**7606.36**	273786.44
		TIME	**4.37**	1.88	**5.66**	1.95	17.37	3.63	36	15.78
	E-4	CUT	6.50	87664.44	5646.61	164832.6	7045.44	192814.29	7877.52	272248.2
		TIME	6.91	2.05	7.88	2.16	**17.36**	4.07	**35**	14.5
	E-6	CUT	**3.38**	96077.6	5444.14	161212.73	6537.87	192500.22	8519.99	209367.71
		TIME	8.99	2.24	9.27	2.37	19.75	4.33	40.55	17.29
	E-8	CUT	4.88	88925.8	**4629.26**	158303.10	**6331.74**	191827.52	8157.62	210908
		TIME	10.15	2.5	11.57	2.55	23.24	4.64	41.29	18.5
	E-10	CUT	4.33	94653.11	5033.79	159361.14	7090.62	**191703.41**	7899.69	**201653.43**
		TIME	12	2.65	12.65	2.76	25.23	4.75	44.1	20.14
PARMETIS		CUT	-	108160	-	**140489**	-	233764	-	204192
		TIME	-	**0.61**	-	**0.49**	-	**0.42**	-	**0.56**
CHACO		CUT	-	122573	-	140988	-	210583	-	232781
		TIME	-	0.68	-	0.59	-	0.56	-	0.66

Table B.12: Results obtained from the partitioning of *Zhao1*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	**8505.36**	**59824.7**	15122.27	**92118.83**	18101.9	113814.56	**19534.71**	123243
		TIME	**4.38**	5.23	**5.39**	5.89	**12**	9.29	**41.86**	46
	E-4	CUT	9688.05	63947.88	13007.34	93127	17891.54	114094.44	19601.18	122528.57
		TIME	7.73	6.3	9.08	7.10	19.29	12.33	56.6	56.43
	E-6	CUT	9022.98	62347.2	**11981.19**	98811.13	**17883.05**	114532.11	19611.05	123302.71
		TIME	9.95	7.74	12.03	7.97	28.12	12.55	74.5	61.43
	E-8	CUT	9090.23	60492.5	15187.34	97716.87	18081.9	113819.7	19648.6	122180.86
		TIME	12.3	8.79	14.11	8.97	41	15.8	112	51.57
	E-10	CUT	9210.666	60288	14989.20	97960.96	18303.63	114061.5	19666.88	123110.14
		TIME	14.5	8.61	15.97	9.7	42.86	16.5	121.43	57.29
PARMETIS		CUT	-	61144.2	-	96234.2	-	104099	-	108267
		TIME	-	1.4	-	0.92	-	0.64	-	0.97
CHACO		CUT	-	61821.9	-	96860.3	-	**103972**	-	**108230**
		TIME	-	**1.3**	-	**0.87**	-	**0.62**	-	**0.71**

Table B.13: Results obtained from the partitioning of *ns3Da*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	E-2	CUT	8449.75	764245.4	**12736.39**	1156616.21	15138.3	1405029.26	16323.83	1529212.22
		TIME	**7.87**	8.91	**4.4**	5.17	**6.26**	6.55	26	25.55
	E-4	CUT	8475.48	**612952.33**	12941.87	1161404.48	15151.03	1409480.4	16320.29	1528741.11
		TIME	8.53	9.37	4.95	5.64	6.43	7.06	33.67	24.11
	E-6	CUT	**8115.13**	762546.2	12993.67	1168459.11	15148.18	1413132.38	16292.01	1528748.75
		TIME	9	9.74	5.23	5.92	7.02	7.81	**22.88**	26.88
	E-8	CUT	8490.44	747369.38	13009.4	**1165376.33**	**15116.47**	**1390162.31**	**16282.37**	1531977.78
		TIME	9.04	10.12	5.42	6.47	7.85	8.33	31.83	26.55
	E-10	CUT	8231.30	755795.67	13010.19	1170253.08	15139.90	1418514.64	16336.81	1530112.86
		TIME	9.41	10.78	5.66	6.78	8.21	8.19	27.12	25.86
PARMETIS		CUT	-	745817	-	1204450	-	1404950	-	**1399100**
		TIME	-	**1.7**	-	**1.4**	-	**1.1**	-	**1.3**
CHACO		CUT	-	746896	-	1204720	-	1404300	-	1399910
		TIME	-	2	-	2	-	2	-	2.5

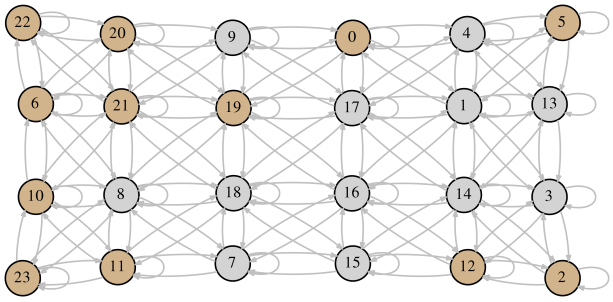
Table B.14: Results obtained from the partitioning of *chem_master1*

		NPROC	2		4		8		16	
		LAP	W	U	W	U	W	U	W	U
SPEC	**E-2**	**CUT**	47395.73	37084.25	113098.54	68315.58	2541911.67	**65168.2**	**4995050**	**72436.86**
		TIME	**4.42**	11.88	5.44	13.08	12.33	19.2	**30.2**	61
	E-4	**CUT**	46925.84	22600.6	**112034.41**	68483.04	213577.44	71096.6	8497460	72714.67
		TIME	4.7	15.1	**5.38**	15.96	**9.17**	23.4	35.12	143.33
	E-6	**CUT**	46526.87	37219	152669.74	69289.36	**1185086.5**	66079.73	7963524.29	73519.71
		TIME	4.82	19.2	6.04	17.46	9.66	43.22	35.57	148.57
	E-8	**CUT**	**43565.46**	41214.22	156653.79	67643.57	1862043.22	72831	10398187.5	73089.25
		TIME	5.11	22.22	6.25	20.39	10.89	62.5	38.38	180
	E-10	**CUT**	46659.86	22467.8	151536.81	69602.83	1242225.71	72190.89	9364298.57	73470.29
		TIME	5.29	25.2	6.42	53.38	10.86	69.33	36	138.57
PARMETIS		**CUT**	-	**6796.8**	-	57652.3	-	72358.2	-	85775.4
		TIME	-	**1.7**	-	**1**	-	**0.66**	-	**0.68**
CHACO		**CUT**	-	42096.6	-	**47692.3**	-	72144	-	86443.2
		TIME	-	1.7	-	1.1	-	0.72	-	0.7

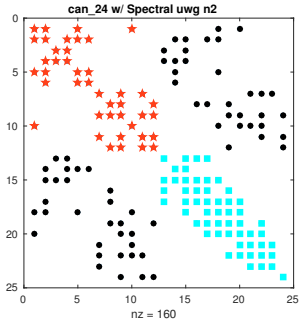
APPENDIX C

FIGURES

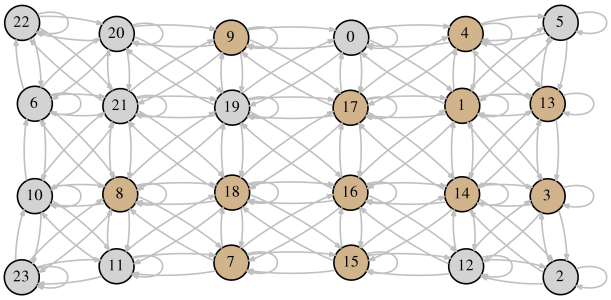
In this appendix, graph and spy representations of the partitioned small-sized matrices are given. For spy representation, MATLAB is used, whereas for graph representations, GraphViz is used. Each color in representations show different cluster.



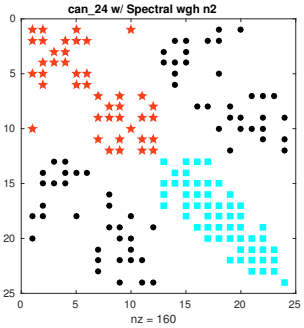
(a) Unweighted, nproc = 2, graph rep.



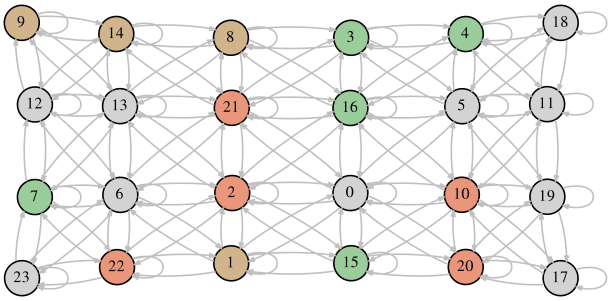
(b) Unweighted, nproc = 2, spy rep.



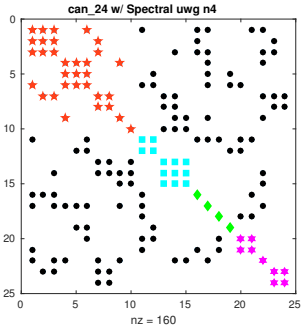
(c) Weighted, nproc = 2, graph rep.



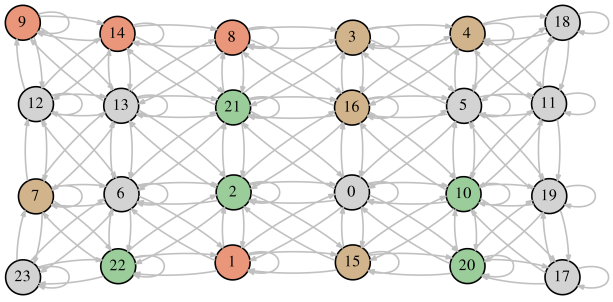
(d) Weighted, nproc = 2, spy rep.



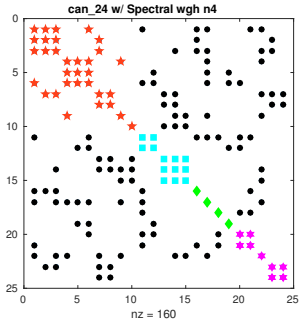
(e) Unweighted, nproc = 4, graph rep.



(f) Unweighted, nproc = 4, spy rep.

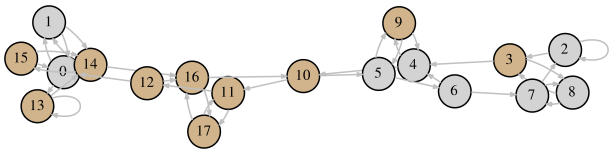


(g) Weighted, nproc = 4, graph rep.

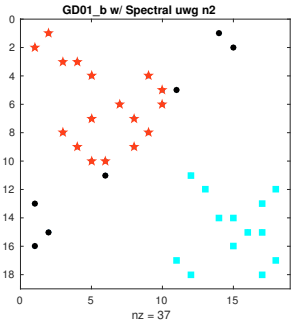


(h) Weighted, nproc = 4, spy rep.

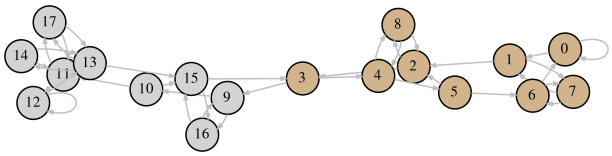
Figure C.1: Graph and spy representations of the partitioned *can_24* by spectral partitioning with k-means clustering



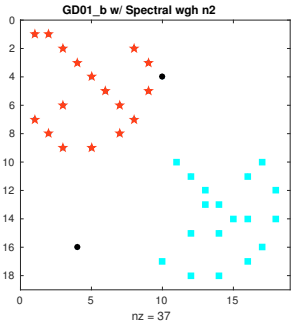
(a) Unweighted, nproc = 2, graph rep.



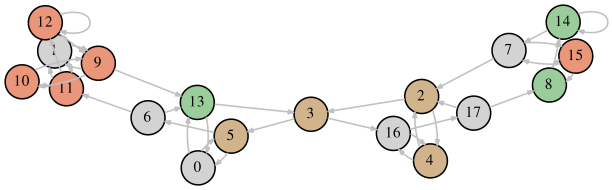
(b) Unweighted, nproc = 2, spy rep.



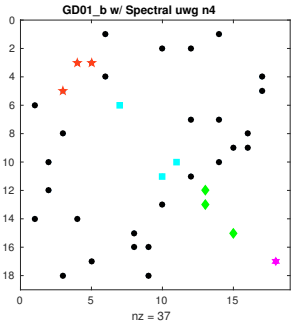
(c) Weighted, nproc = 2, graph rep.



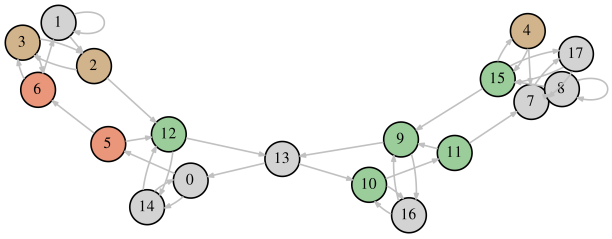
(d) Weighted, nproc = 2, spy rep.



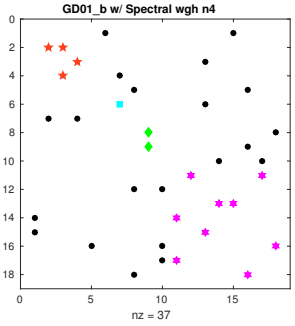
(e) Unweighted, nproc = 4, graph rep.



(f) Unweighted, nproc = 4, spy rep.

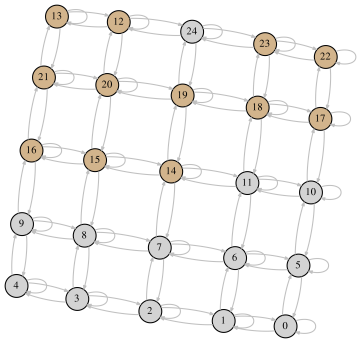


(g) Weighted, nproc = 4, graph rep.

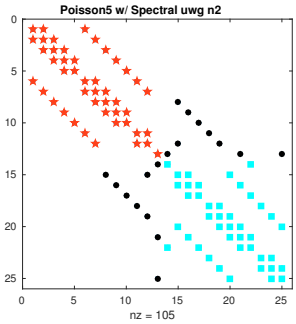


(h) Weighted, nproc = 4, spy rep.

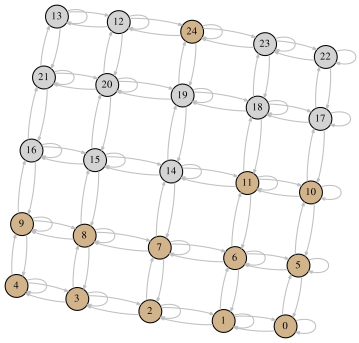
Figure C.2: Graph and spy representations of the partitioned *GD01_b* by spectral partitioning with k-means clustering



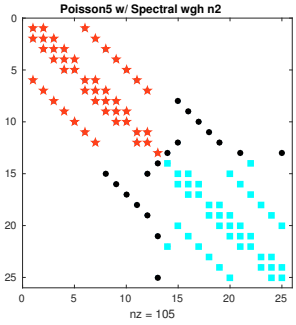
(a) Unweighted, nproc = 2, graph rep.



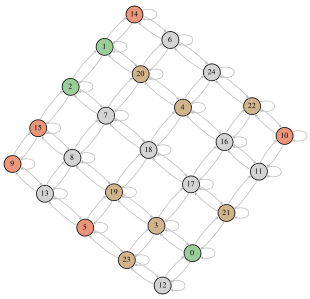
(b) Unweighted, nproc = 2, spy rep.



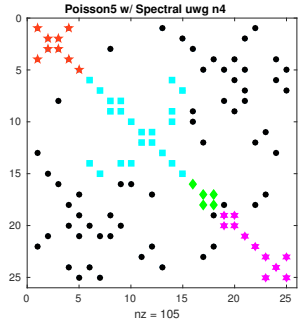
(c) Weighted, nproc = 2, graph rep.



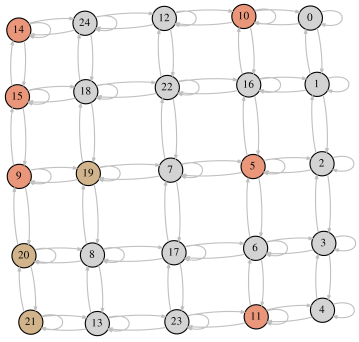
(d) Weighted, nproc = 2, spy rep.



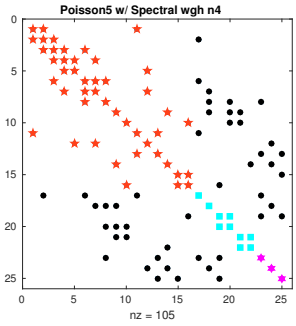
(e) Unweighted, nproc = 4, graph rep.



(f) Unweighted, nproc = 4, spy rep.

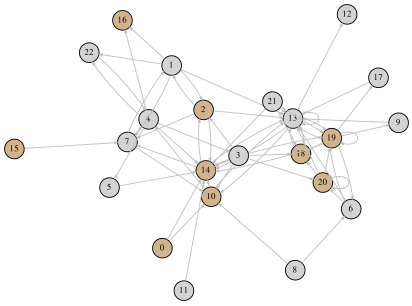


(g) Weighted, nproc = 4, graph rep.

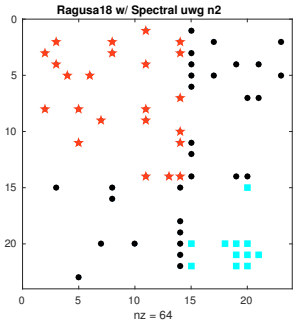


(h) Weighted, nproc = 4, spy rep.

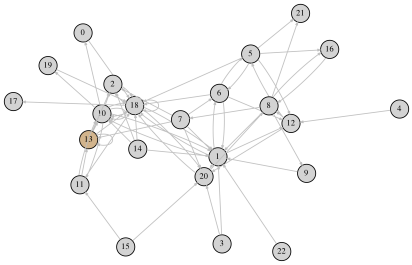
Figure C.3: Graph and spy representations of the partitioned *Poisson(5)* by spectral partitioning with k-means clustering



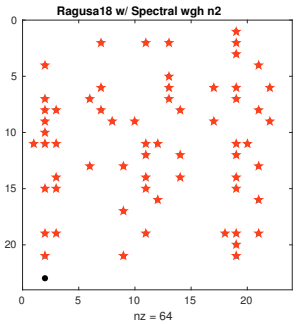
(a) Unweighted, nproc = 2, graph rep.



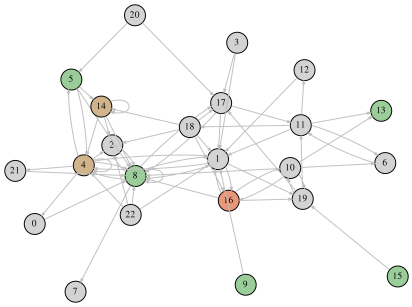
(b) Unweighted, nproc = 2, spy rep.



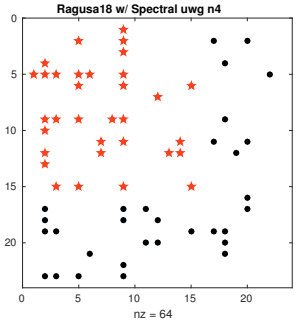
(c) Weighted, nproc = 2, graph rep.



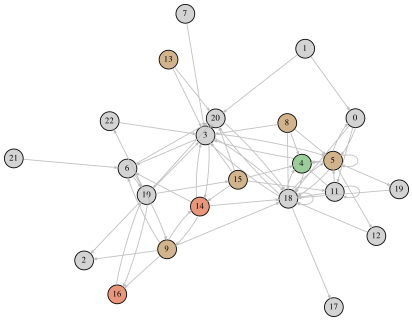
(d) Weighted, nproc = 2, spy rep.



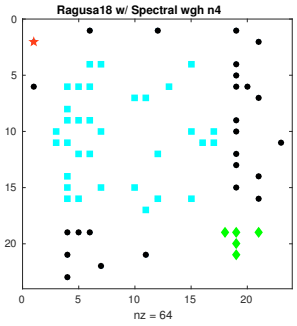
(e) Unweighted, nproc = 4, graph rep.



(f) Unweighted, nproc = 4, spy rep.

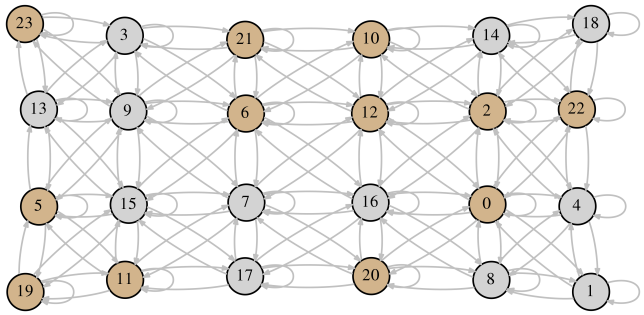


(g) Weighted, nproc = 4, graph rep.

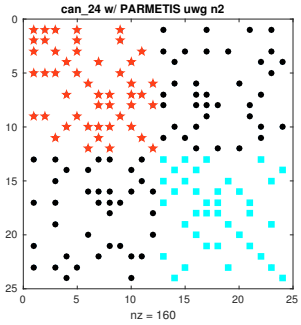


(h) Weighted, nproc = 4, spy rep.

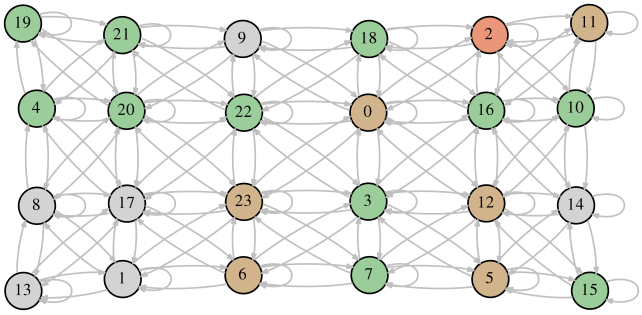
Figure C.4: Graph and spy representations of the partitioned *Ragusa18* by spectral partitioning with k-means clustering



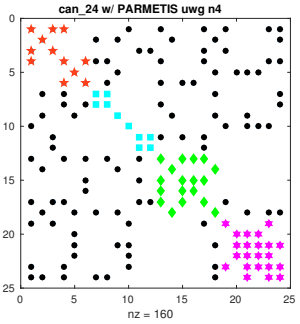
(a) nproc = 2, graph rep.



(b) nproc = 2, spy rep.

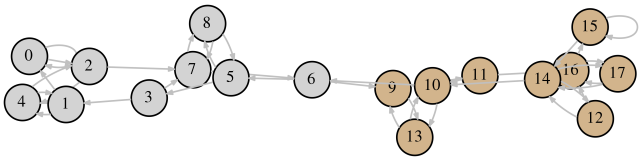


(c) nproc = 4, graph rep.

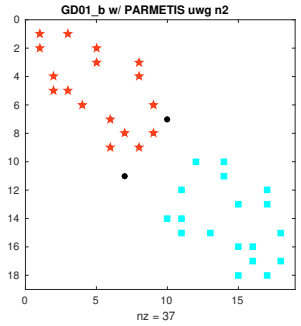


(d) nproc = 4, spy rep.

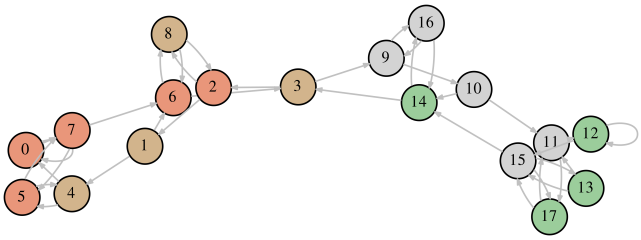
Figure C.5: Graph and spy representations of the partitioned *can_24* by ParMETIS



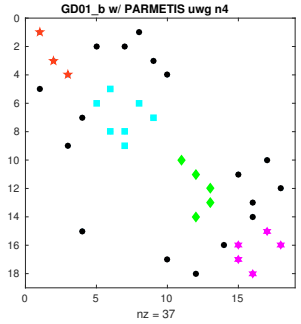
(a) nproc = 2, graph rep.



(b) nproc = 2, spy rep.

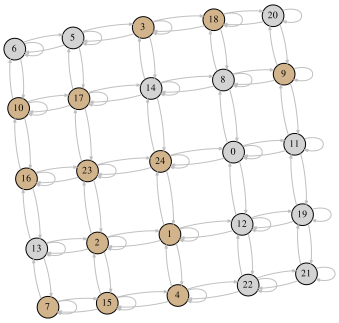


(c) nproc = 4, graph rep.

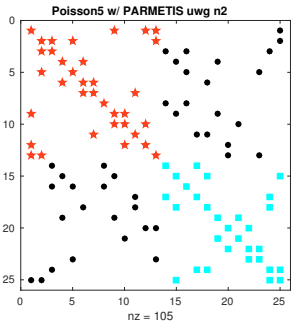


(d) nproc = 4, spy rep.

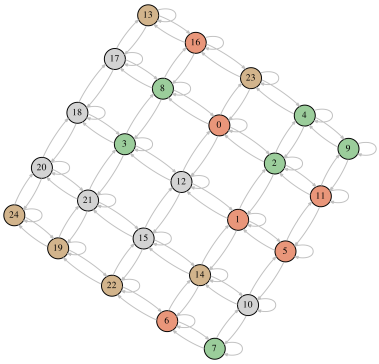
Figure C.6: Graph and spy representations of the partitioned *GD01_b* by ParMETIS



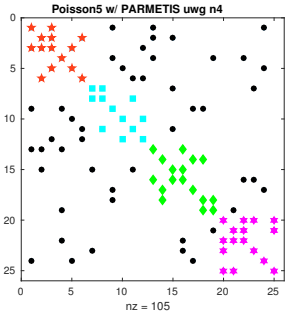
(a) nproc = 2, graph rep.



(b) nproc = 2, spy rep.

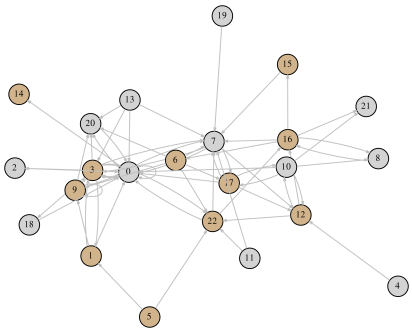


(c) nproc = 4, graph rep.

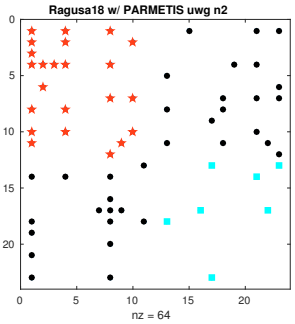


(d) nproc = 4, spy rep.

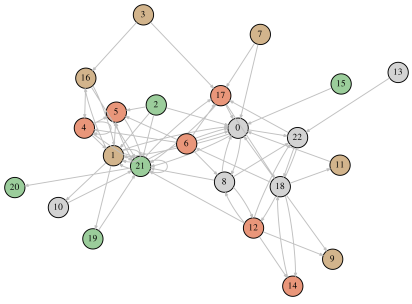
Figure C.7: Graph and spy representations of the partitioned *Poisson(5)* by ParMETIS



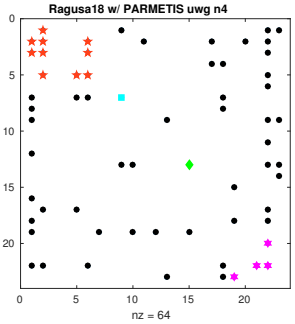
(a) nproc = 2, graph rep.



(b) nproc = 2, spy rep.

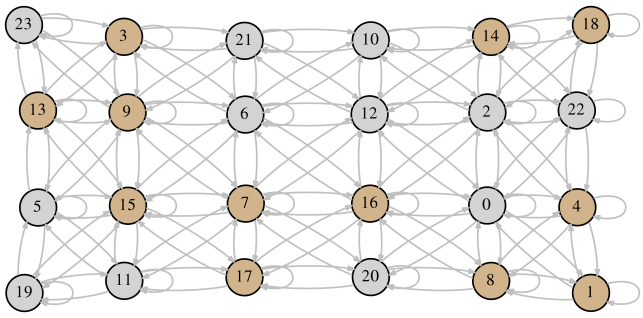


(c) nproc = 4, graph rep.

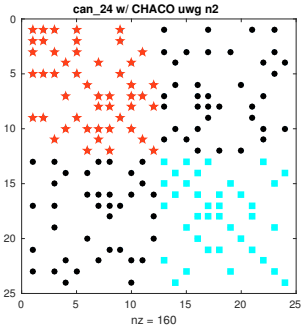


(d) nproc = 4, spy rep.

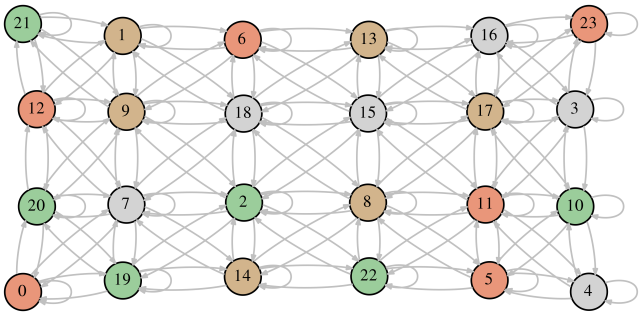
Figure C.8: Graph and spy representations of the partitioned *Ragusa18* by ParMETIS



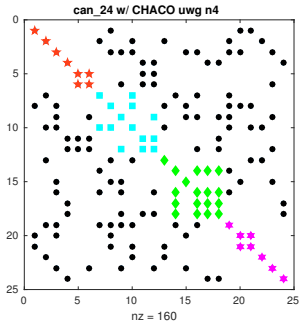
(a) nproc = 2, graph rep.



(b) nproc = 2, spy rep.

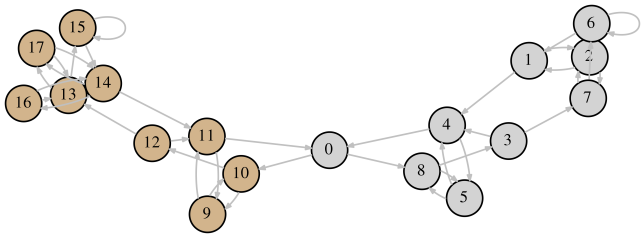


(c) nproc = 4, graph rep.

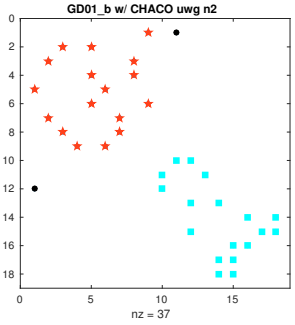


(d) nproc = 4, spy rep.

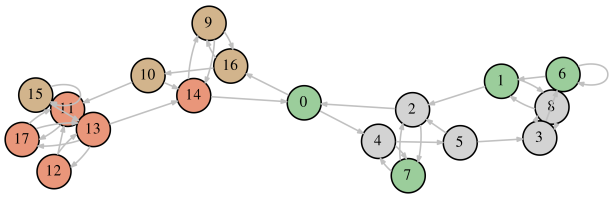
Figure C.9: Graph and spy representations of the partitioned *can_24* by CHACO



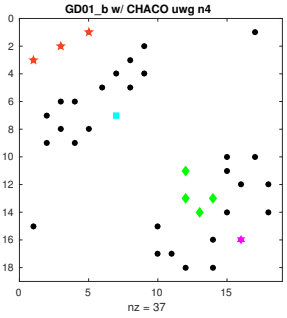
(a) nproc = 2, graph rep.



(b) nproc = 2, spy rep.

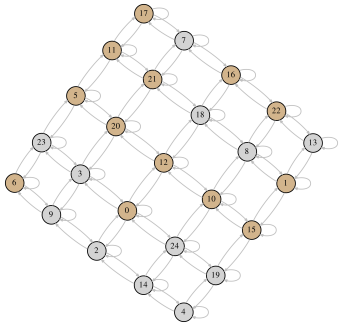


(c) nproc = 4, graph rep.

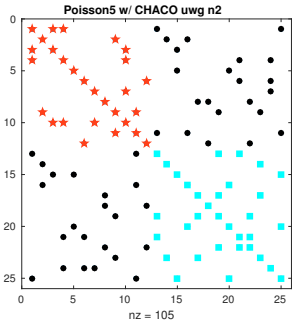


(d) nproc = 4, spy rep.

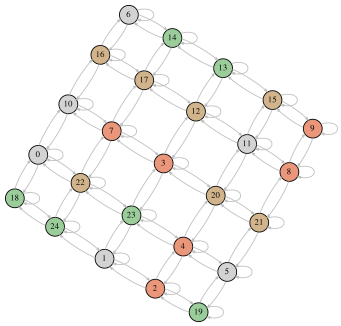
Figure C.10: Graph and spy representations of the partitioned *GD01_b* by CHACO



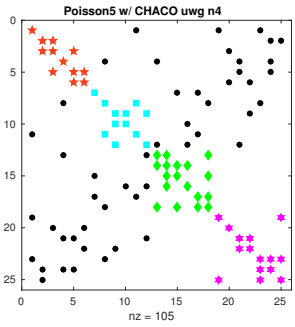
(a) nproc = 2, graph rep.



(b) nproc = 2, spy rep.

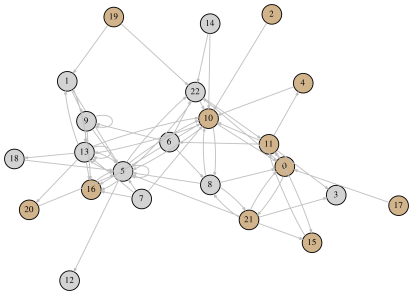


(c) nproc = 4, graph rep.

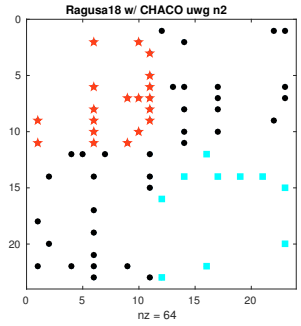


(d) nproc = 4, spy rep.

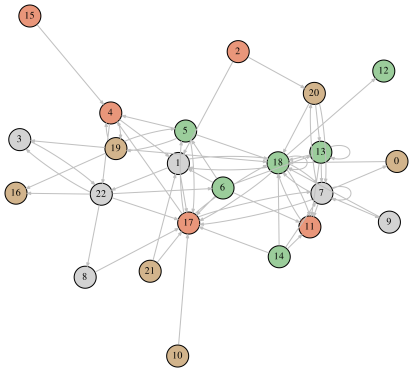
Figure C.11: Graph and spy representations of the partitioned *Poisson(5)* by CHACO



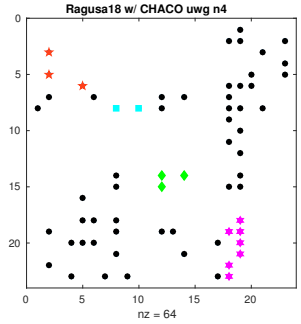
(a) nproc = 2, graph rep.



(b) nproc = 2, spy rep.



(c) nproc = 4, graph rep.



(d) nproc = 4, spy rep.

Figure C.12: Graph and spy representations of the partitioned *Ragusa18* by CHACO