Hypergraph Partitioning in the Cloud

Foad Lotfifar

Abstract

The thesis investigates the partitioning and load balancing problem which has many applications in High Performance Computing (HPC). The application to be partitioned is described with a graph or hypergraph. The latter is of greater interest as hypergraphs, compared to graphs, have a more general structure and can be used to model more complex relationships between groups of objects such as non-symmetric dependencies. Optimal graph and hypergraph partitioning is known to be NP-Hard but good polynomial time heuristic algorithms have been proposed.

In this thesis, we propose two multi-level hypergraph partitioning algorithms. The algorithms are based on rough set clustering techniques. The first algorithm, which is a serial algorithm, obtains high quality partitionings and improves the partitioning cut by up to 71% compared to the state-of-the-art serial hypergraph partitioning algorithms. Furthermore, the capacity of serial algorithms is limited due to the rapid growth of problem sizes of distributed applications. Consequently, we also propose a parallel hypergraph partitioning algorithm. Considering the generality of the hypergraph model, designing a parallel algorithm is difficult and the available parallel hypergraph algorithms offer less scalability compared to their graph counterparts. The issue is twofold: the parallel algorithm and the complexity of the hypergraph structure. Our parallel algorithm provides a trade-off between global and local vertex clustering decisions. By employing novel techniques and approaches, our algorithm achieves better scalability than the state-of-the-art parallel hypergraph partitioner in the *Zoltan* tool on a set of benchmarks, especially ones with irregular structure.

Furthermore, recent advances in cloud computing and the services they provide have led to a trend in moving HPC and large scale distributed applications into the cloud. Despite its advantages, some aspects of the cloud, such as limited network resources, present a challenge to running communication-intensive applications and

make them non-scalable in the cloud. While hypergraph partitioning is proposed as a solution for decreasing the communication overhead within parallel distributed applications, it can also offer advantages for running these applications in the cloud. The partitioning is usually done as a pre-processing step before running the parallel application. As parallel hypergraph partitioning itself is a communication-intensive operation, running it in the cloud is hard and suffers from poor scalability. The thesis also investigates the scalability of parallel hypergraph partitioning algorithms in the cloud, the challenges they present, and proposes solutions to improve the cost/performance ratio for running the partitioning problem in the cloud.

Our algorithms are implemented as a new hypergraph partitioning package within *Zoltan*. It is an open source Linux-based toolkit for parallel partitioning, load balancing and data-management designed at Sandia National Labs. The algorithms are known as *FEHG* and *PFEHG* algorithms.

Hypergraph Partitioning in the Cloud

Foad Lotfifar

A Thesis presented for the degree of
Doctor of Philosophy

Algorithms and Complexity Group
School of Engineering and Computing Sciences
University of Durham
United Kingdom

January 2016

Contents

List of Figures

List of Tables

Declaration

The work in this thesis is based on research carried out at the Algorithm and Complexity Group (ACiD), School of Engineering and Computing Sciences, Durham University, United Kingdom. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text. All parts of the thesis are my individual contribution.

How to Cite

Lotfifar, Foad. *Hypergraph Partitioning in the Cloud*, PhD Thesis, School of Engineering and Computing Sciences, Durham University, United Kingdom, January 2016.

Acknowledgements

First and foremost I want to express my special appreciation and thanks to my advisor Dr. Matthew Johnson for his help and support during my PhD at Durham University. I would like to thank the Efficient Computing and Storage Group at Johannes Gutenberg - Universität Mainz for providing me the resources for my simulations, Professor André Brinkmann and Dr. Lars Nagel for their comments and feedback during my doctoral studies, Markus Mäsker for his fruitful collaboration, and Krishnaprasad Narayanan for his help while doing my evaluations. I would also like to thank Dr. Tobias Weinzierl, and Dr. Hongjian Sun for interesting discussions. My thanks also goes to my examiners Professor William J. Knottenbelt and Dr. Maximilien Gadouleau for their useful comments.

I like to thank the European FP7 Marie Curie Initial Training Network "SCALUS (SCALing by means of Ubiquitous Storage)" under grant agreement No.238808 that funded my PhD at Durham University.

My special thanks to my family, specially my sisters who have been always with me in rough times, and my mother and father to whom I owe all of achievements in my life, the encouragement and support of my beloved wife who was my source of inspiration and energy, and my mother- and father-in-law for their awareness. Finally, my regards to all of my friends Hemn Gholami, Chia Qaderi, Misha Bordoloi Singh, Muhammad M. Saleh, Kaya Uzel, Dr. Massoud Sharifi, Rikan Kareem, Dr. Adel Feyzi, Hakim Herany, Amjed Rasheed, and and many others, specially in Durham. I greatly appreciate their friendship and their believe in me.

To my beloved wife, Fermesk

and

To my Parents

Chapter 1

Introduction

1.1 Motivations

1.1.1 Why Hypergraph Partitioning?

The *partitioning* problem is that of finding a way to decompose a set of interrelated objects (or jobs, tasks or components) into smaller fragments or *parts* such that intra-dependency between the objects in the same part is higher than inter-dependency of objects in separate parts. This provides advantages in data processing. As an example in distributed systems, the parts are dispatched into parallel machines for processing. The intra and inter object dependencies correspond to local and remote data accesses by processors, respectively. Minimising the object inter-dependencies will result in more data locality and, consequently, provide higher performance and speedup because the parts can be processed in parallel with less communication between the machines.

On the other hand, the performance of parallel systems is often limited by the response of the slowest system or responder. If a computer has more data to process than the others, then the parallel system must wait for this computer to finish its work while others stand idle. This causes not only performance degradation but also wastes system resources and provides poor resource utilisation. The aim of load balancing strategies is to assign the same amount of load to all machines in the parallel system in order to avoid this issue.

The partitioning problem with a load balancing constraint is to obtain an object decomposition such that the size of the parts are limited to a specified range in order to prevent load imbalance among the machines of the parallel system. Its applicability is not solely limited to the parallel and distributed systems and it has numerous applications in scientific and High Performance Computing (HPC), both in academia and industry. Examples are: design and partitioning of Very Large Scale Integrated (VLSI) systems [Len90], biology [MCS15], data mining [CJZM10], and domain decomposition (in areas such as fluid dynamics [PRN+11] and computational chemistry [Kim13]). Due to the importance of load balancing, whenever we mention the partitioning problem in the thesis, we mean the partitioning problem with load balancing constraint unless stated otherwise.

The partitioning problem requires defining the application workflow including specifying object dependency patterns and quantifying them. Among different approaches, graph and hypergraphs are two common data structures for this purpose.

1.1.2 Application Modelling

Following the discussion in the previous section, the application to be partitioned can be represented as a graph or hypergraph. The hypergraph is a generalisation of the graph model. Depending on the way the application is represented, the partitioning problem is categorised into either the graph partitioning problem or the hypergraph partitioning problem. In this section, we investigate each category and its advantages.

In the graph partitioning problem, the application is modelled with a graph that is composed of a set of objects and one-to-one relationships among them. This means that graphs only capture pair-object dependencies. In the graph model, objects are called vertices and pair-relationships are referred to as edges. These pair relationships provide some limitations. The limitations of graph partitioning in the context of sparse-matrix vector multiplication[1] is investigated by Hendrickson [Hen98] as follows:

[1]Sparse-matrix vector multiplication is represented as $y = \mathbf{A} \times x$. Vectors x and y are input and output vectors, respectively, and $\mathbf{A}$ is the sparse matrix.

1. The graph can only model square, symmetric matrices and it is unable to model non-square, non-symmetric matrices.

2. Graph partitioning can only provide symmetric partitions. Representing a graph as a sparse matrix[2], any partitioning on the rows of the matrix is identical to the same partitioning on the columns. This enforces the same partitioning on the input and output vectors. This restriction is not necessary for non-symmetric solvers.

3. Graph partitioning ignores the preconditioning, which is one of the methods for improving the performance of applications that include sparse-matrix vector multiplication. Graph partitioning only optimises the multiplication process, which itself is a small part of more complex operations in scientific computations. In order to obtain better performance improvements, graph partitioning should consider the whole process rather than only the multiplication part. For example, if the partitioning knows what calculations come next, it can further optimise the partitioning.

The other problem is that we can not always model object relationships as pair relationships. In the social networks such as Facebook, we are dealing with more complex relationships and interactions between a group of users can not be represented with edges or pair-wise relationships [WXSW14]. As another example, the graph model cannot fairly model the real inter-processor communication pattern in sparse matrix-vector multiplication operation when processors access data that resides on other processors [ÇA99].

A solution to some of the above mentioned problems related to graph modelling is to model applications with hypergraphs [DBH+05]. A hypergraph is a generalisation of the graph model in which object relationships are not limited to those that are between pairs of objects and complex relationships can be represented. In the hypergraph, objects are called vertices and group relations are referred to as

[2]A graph with N vertices is represented as a $N \times N$ matrix. An item at row i and column j of the matrix is one if there is an edge between vertex i and vertex j of the graph, otherwise zero.

hyperedges[3]. Unlike edges, hyperedges can contain any arbitrary number of vertices. Furthermore, hypergraphs have the ability to model non-symmetric matrices in a sparse-matrix vector applications [ÇA99]. In addition, one can obtain a non-identical partitioning on the rows or columns of the matrix by defining a partitioning on either the vertex set or the hyperedge set [KPÇA12].

The price that is paid for these advantages is the slower processing time compared to the graph partitioning problem. Although it is slower, employing hypergraph partitioning to partition HPC applications can result in much better performance [AK95]. The hypergraph partitioning is usually done as a pre-processing step before processing the HPC application. The performance improvement in the latter step compensates the longer processing time of the hypergraph partitioning. In the end, one can get higher performance than graph partitioning based solutions [ÇA99,TK08]. This has led to widespread usage and increased popularity of the hypergraph modelling and the hypergraph partitioning in scientific applications [Alp96,ÇA99,BJKT05,ZHS06,THK09,CJZM10,HC14,HLT+14,MCS15].

Optimal solutions to both graph and hypergraph variants of the partitioning problem are NP-hard [MJ79], but a number of good heuristic algorithms have been proposed to solve the problem sub-optimally [FM82,ÇA11,KK99,DBH+06,TK08].

1.1.3 Hypergraphs in the Cloud

The size of hypergraphs representing real applications is increasing. For example, hypergraphs that model social networks such as Facebook and Twitter have millions or billions of users (or vertices) with their interactions (hyperedges). The size of the hypergraphs is too large to be processed by only one computer. Therefore, the performance of serial algorithms limits the size of the problem that can be dealt with and we need parallel and scalable algorithms and tools [JNWH04,WXSW14,DBH+05].

There are two main specifications for parallel hypergraph partitioning algorithms in distributed multi-processor systems that need to be taken into account. The first is the quality: the quality of the partitioning should be retained as the size of the

[3]The term "hyperedge" is used to make a distinction between many-to-many relations in hypergraphs and one-to-one relations in graphs.

distributed system, in terms of the number of processing cores, increases. The larger the system is, the less the data locality among the processors is, and consequently, the worse the partitioning of the objects may be. The second is the scalability of the parallel algorithm. We are interested in a partitioning algorithm that is scalable and gives better speedup as we increase the number of processors in the distributed system. Achieving this is more difficult in hypergraph partitioning than in graph partitioning [4].

Taking a step further, the interest in moving distributed and scientific applications into the cloud has been increasing in recent years. The reason lies in the advantages that the cloud offers to distributed applications such as elasticity, small start-up and maintenance costs, dynamic resource allocation, and economies of scale and use. On the other hand, some characteristics of the cloud cause performance bottlenecks for running these applications such as hardware virtualisation, hardware heterogeneity, and multi-tenancy [GKG+13,YCD+11,Wal08,MDH+12]. Above all, the limited network resources of the cloud has made it a good candidate for running computation-intensive applications while the communication-intensive applications usually suffer from poor scalability [GKG+13]. In the latter, the scalability is dependant on some design specifications such as the communication pattern inside the application; for example, local and customised collective communications provide better scalability than global communications with short messages [JRM+10]. In addition, the structure of the application and how the application is designed both affect the scalability [GKG+13].

While an important application of the hypergraph partitioning in distributed systems is to decrease the communication volume and increase data locality, partitioning the application before running it in the cloud can provide considerable performance improvement. Considering the large size of HPC applications, we need

[4]Both requirements (quality and scalability) are important for designing a parallel partitioning algorithm. For more clarification, assume that the hypergraph partitioning is run as a pre-processing step every time the application is run. First, the poor scalability of the partitioning algorithm itself affects the application performance. On the other hand, if the algorithm does not give a quality comparable to the serial algorithm as the system scales up, the lower partitioning quality that is obtained by increasing the number of processors will result in higher communication overhead while processing the application; this also gives poorer performance.

a parallel hypergraph partitioner in the cloud.

This may be an issue when we run the partitioning process in the cloud. The problem is that the parallel hypergraph partitioning is one of those communication-intensive applications that not only provides challenges in the high performance computer clusters, but also can suffer from poor scalability in the cloud. The scalability problem, in either situation, is not only related to the partitioning algorithm itself but also the structure of the hypergraph that may impose high network communication overhead. The structure of the hypergraph and object dependency models vary from one application to another and make it difficult to define one framework that works the best for all types of hypergraphs. Even so, there is no known hypergraph partitioning algorithm that works well on all hypergraphs and there are always trade-offs. The more general structure of the hypergraph model compared to the graph model adds to the complexity [WXSW14]. Running the parallel hypergraph partitioning in the cloud and proposing a scalable algorithm is a challenging task and needs lots of provisioning and design techniques to make the problem feasible.

1.2 Thesis Objectives and Contributions

Our main objectives are summarized as follows:

1. To Propose a serial hypergraph partitioning algorithm that generates high quality partitioning results on small hypergraphs (hypergraphs that can be processed on one computing node). Our aim is to evaluate which design parameters affect the performance and the quality of the serial partitioning algorithm.

2. Considering the ever-increasing size of parallel and distributed applications, we try to design a parallel scalable hypergraph partitioning algorithm that generates partitioning quality comparable to the serial algorithm. The scalability is assessed based on achieved speedup over the serial hypergraph partitioning algorithm. An algorithm is considered as more scalable if it gives better speedup when the number of processors in the distributed system increases.

3. Considering the trend for running scientific applications in the cloud, we target the cloud as the testbed for parallel hypergraph partitioning. Considering the characteristics of the cloud and its runtime limitations, we try to propose algorithms and techniques to achieve scalability in the cloud. We also identify design issues that exist for running parallel hypergraph partitioning in the cloud.

4. To evaluate our algorithms against state-of-the-art hypergraph partitioning algorithms and using real application data and benchmarks for this purpose. The algorithms should be evaluated and compared based on two parameters: the quality of the partitioning and the scalability.

5. To implement our algorithms as a part of an open source software tool that is freely available.

The algorithms proposed in this thesis are of a type known as *multi-level* which are composed of three distinct phases [Kar02]. They first construct a sequence of approximations of the original hypergraph during the coarsening phase. The size reduction is done using data clustering techniques and vertex matching. In the second phase, which is called the initial partitioning phase, the partitioning problem is solved on the smallest or the coarsest hypergraph. In last phase, which is also called the uncoarsening phase, the coarsening stage is reversed and the solution obtained on the coarsest hypergraph is used to provide a solution on the input hypergraph. The coarsening phase is also known as the refinement phase.

Regarding the above objectives, the contributions of the thesis are as follows:

1. We propose a multi-level serial hypergraph partitioning algorithm that:

   - gives significant quality improvements over state-of-the-art algorithms on our evaluated benchmarks. It provides up to 71% improved partitioning cut on hypergraphs with irregular structure compared to the state-of-the-art serial algorithms.
   - is based on rough set clustering technique which is a global clustering technique for finding vertex matches in the coarsening phase. The algorithm

is designed based on removing *unimportant* and *redundant* information from the hypergraph for making better clustering decisions.

- provides a trade-off between global and local clustering decisions by categorising the vertices of the hypergraph.

2. We design a parallel hypergraph partitioning algorithm, developed from the serial algorithm in case one, such that:

   - it is based on the parallel rough set clustering techniques.
   - proposes a parallel scalable algorithm for attribute reduction in the hypergraph.
   - proposes a synchronised-based parallel FM refinement algorithm. Due to the serial nature of the FM algorithm, the refinement phase is the most challenging phase in the multi-level paradigm to parallelise.
   - as our partitioning algorithm is a recursive bipartitioning algorithm, which is considered to be a divide-and-conquer algorithm, we proposed a processor reconfiguration technique for each recursion of the algorithm. We show that our reconfiguration algorithm is an effective and easy-to-apply technique for providing a trade-off between the performance and the partitioning quality.
   - considering the ever-increasing scale of the current distributed systems, the parallel algorithm is evaluated in the HPC cluster with up to 1024 processing cores and the scalability is investigated.
   - the algorithm is evaluated in the cloud and the scalability is investigated. Considering the growing application of the hypergraph partitioning in the cloud, there is no prior work investigating the parallel hypergraph partitioning and its scalability in the cloud. We have identified the challenges on the way and propose solutions that will lead to a new approach to obtaining improvements in cost and performance when deploying distributed applications in the cloud.

3. We have implemented our parallel algorithm as a new library within the *Zoltan* toolkit [San14b] from the Sandia National Labs that retains the *Zoltan* interface. To date, there is no unified framework for hypergraph partitioning and available tools use different interfaces and frameworks. This contribution is of great importance because the increasing popularity of hypergraph partitioning demands a unified framework and programming interface.

1.3 Thesis Structure

The rest of the thesis is organised as follows.

Chapter 2 provides the background and necessary definitions used in the thesis. We start by providing the mathematical definition of the hypergraph partitioning problem. Then, we introduce the rough set clustering technique that is a powerful mathematical tool for data clustering and data analysis. Finally, we provide a brief overview of cloud computing, its specification and core features, and how the cloud can be employed for running HPC applications.

Chapter 3 is dedicated to the literature review. The chapter is divided into four sections. The first section describes different algorithms for the hypergraph partitioning problem. The proposed algorithms are categorised based on various aspects. It also covers related work for the parallel hypergraph partitioning algorithms. The second section provides an overview of the tools designed for hypergraph partitioning. Section three introduces applications of hypergraph partitioning in scientific computing. Finally, the last section concerns related work for transferring HPC applications into the cloud and investigates the challenges and limitations.

Chapter 4 proposes our serial multi-level algorithm known as *Feature Extraction Hypergraph Partitioning (FEHG)* algorithm. The algorithm makes novel use of the technique of rough set clustering in categorising the vertices of the hypergraph in the coarsening phase. *FEHG* considers hyperedges as the attributes and features of the hypergraph and tries to discard unimportant attributes to make better clustering decisions. The emphasis of the algorithm is on the coarsening phase of the multi-level paradigm as it is considered the most important phase. The chapter evaluates the

algorithm against the state-of-the-art hypergraph partitioning algorithms on a range of hypergraphs from real applications with different specifications.

Chapter 5 proposes our parallel multi-level hypergraph partitioning algorithm which is called the *Parallel Feature Extraction Hypergraph Partitioning (PFEHG)* algorithm. The algorithm is designed for scalability. The chapter first proposes the parallel coarsening phase that is based on the parallel rough set clustering techniques. A parallel algorithm is also proposed for attribute reduction and removing *unimportant hyperedges* which is based on constructing a bipartite graph from the hypergraph. Later on, a parallel synchronised-based refinement algorithm is proposed for the uncoarsening phase. This is the most difficult phase of the multi-level paradigm to parallelise. The reason is that the refinement algorithm is inherently sequential and vertex connectivities impose lots of network traffic during the refinement process. The parallel refinement algorithm is designed considering the specification of the refinement phase and using the lessons learned from the serial algorithm. The *PFEHG* algorithm uses a new one-dimensional initial hypergraph distribution among the processors and special processor reconfigurations in each recursion of the algorithm. Finally, the algorithm is evaluated against the state-of-the-art parallel hypergraph partitioner, *Zoltan* [San14b]. The evaluations are done in a HPC cluster as well as in the cloud and the performance, scalability, and the quality of the algorithms are compared. Algorithms are tested on a range of benchmarks from real applications with different specifications.

Chapter 6 concludes the thesis by providing a summary of the work and the evaluation results. It also proposes the opportunities for the future work.

Appendix A describes the set of benchmarks used in our thesis for evaluating the partitioning algorithms. The list of the hypergraphs, their applications, and their statistical specifications are proposed in more detail.

Appendix B provides the programming interface for using our hypergraph partitioning algorithms. Our algorithms are implemented as a new hypergraph partitioning package in the *Zoltan* tool from Sandia National Labs [San14b] and use the same programming interface. It is implemented in C and MPI. Our algorithms have some parameters that are user defined and they are used to control the runtime behaviour.

The appendix describes the parameters and how they can be set. An example on how to use our partitioning algorithm in *Zoltan* is also given in the end.

1.4 Publications

1. Lotfifar, F., Johnson, M., "A Multi-level Hypergraph Partitioning Algorithm Using Rough Set Clustering", *In Euro-Par 2015: Parallel Processing*, volume 9233 of Lecture Notes in Computer Science, pp.159-170. Springer Berlin Heidelberg, 2015.

2. Lotfifar, F., Johnson, M., "A Scalable Multi-level Hypergraph Partitioning Algorithm", *ready for submission.*

3. Lotfifar, F., Johnson, M., "A Serial Multi-level Hypergraph Partitioning Algorithm", *submitted to the Cluster Computing journal.*

4. Masker, M., Nagel, L., Lotfifar, F., Brinkmann, A., Johnson, M., "Smart Grid-aware Scheduling in Data Centres", *in Sustainable Internet and ICT for Sustainability (SustainIT)*, pp.1-9, 14-15 April 2015. **(Best paper award)**

Chapter 2

Preliminaries

Hypergraph is a generalisation of graph in which edges can connect more than two vertices; an edge in hypergraph is called a hyperedge. Hypergraph has the ability to represent non-symmetric applications and provide a better connectivity model among a set of objects compared to its graph counterpart. Hypergraph partitioning, which is based on modelling the application with a hypergraph, is a recent improvement over graph partitioning. Its application in scientific computing for data partitioning has shown much better improvement than graph partitioning algorithms such as better data localisation [ÇA99] and data distribution [SK06].

In this chapter, we provide necessary definitions and preliminaries used in the rest of the thesis. These definitions are used for proposing our hypergraph partitioning algorithms in later chapters. First, hypergraph and the hypergraph partitioning problem are defined. Then we define the rough set data clustering technique which is a powerful mathematical tool for data analysis and classification. This is the basis of our vertex clustering algorithm that is used in both serial and parallel partitioning algorithms proposed in the thesis.

Furthermore, we provide a brief overview of cloud computing, its specification and core features, and how cloud computing can be employed for running scientific and distributed applications. The chapter also introduces OpenStack as an open source cloud operating system which enables the provision of a low-cost and scalable cloud environment for running distributed applications.

2.1 Hypergraphs

In mathematics and set theory, a multiset is defined as follows:

Definition 2.1 (Multiset) *A **multiset** is a generalisation of a set in which an element can occur several times. The **multiplicity** of an element is the number of times the element occurs in the multiset. The **cardinality** of a multiset is the sum of multiplicity of all elements in the multiset.*

As an example assume the multiset $\{a, b, b, c, c, c\}$. The cardinality of the multiset is 6 and multiplicities of a, b, and c in are 1,2, and 3, respectively. Accordingly, a hypergraph is defined as follows:

Definition 2.2 (Hypergraph) *A **hypergraph** $H = (V, E)$ (or simply $H(V, E)$) is a pair consists of a finite set of vertices V with size $|V| = n$ and a multiset $E \subseteq 2^V$ of hyperedges with size $|E| = m$.*

Let $e \in E$ and $v \in V$ be a hyperedge and a vertex of the hypergraph $H(V, E)$, respectively. The hyperedge e is said to be *incident* on v or *contains* v if $v \in e$ and it is shown as $e \triangleright v$. The pair $\langle e, v \rangle$ is further called a *pin* of H. The degree of v, which is represented as $d(v)$, is the number of hyperedges incident on v. The size or cardinality of e, which is shown as $|e|$, is the number of vertices it contains. According to this definition, a hypergraph is a generalisation of a graph in which there is no limitation on the size of hyperedges. A hypergraph is reduced to a graph if the cardinality of every hyperedge is two that is $|e| = 2, \forall e \in E$. Furthermore, the number of pins of the hypergraph is calculated as $pins(H) = \sum_{v \in V} d(v) = \sum_{e \in E} |e|$.

In some literature, a hyperedge is also called a *net*; therefore, we use hyperedge and net interchangeably in the rest of the thesis[1].

Definition 2.3 (Incidence Matrix) *The **Incidence Matrix** of a hypergraph $H(V, E)$ with $V = \{v_1, v_2, \cdots, v_n\}$ vertices and $E = \{e_1, e_2, \cdots, e_m\}$ hyperedges is the $n \times m$ matrix $\mathbf{\Theta}(H) = (\theta_{ij})$ with the entries calculated as follows:*

[1]The terminology originates from the application of hypergraph partitioning in VLSI circuit partitioning in which a hyperedge is a net (a set of wires) that connects a number of circuit components.

$$\theta_{ij} = \begin{cases} 1, & \text{if } v_i \in e_j \text{ (or } e_j \triangleright v_i) \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

We represent the incidence matrix of H as $\mathbf{\Theta}(H)$ or simply $\mathbf{\Theta}$. The number of non-zeros in the incidence matrix is equal to the number of pins in H.

Similarly, the adjacency matrix of a hypergraph is defined as follow.

Definition 2.4 (Adjacency Matrix) *The **Adjacency Matrix** of a hypergraph $H(V, E)$ with $V = \{v_1, v_2, \cdots, v_n\}$ vertices is the $n \times n$ matrix $\mathbf{A}(H) = (a_{ij})$ with entries calculated as follows:*

$$a_{ij} = \begin{cases} 1, & \text{if } \exists e \in E : e \triangleright v_i \text{ and } e \triangleright v_j, i \neq j \\ 0, & \text{otherwise} \end{cases} \tag{2.2}$$

The diagonal entries of the adjacency matrix are zero. By convention, the adjacency matrix of H is represented as $\mathbf{A}(H)$.

Let $\mathbf{D}$ be a diagonal matrix of size $|V| \times |V|$ whose entries are degrees of the vertices. The adjacency matrix can be calculated as follow:

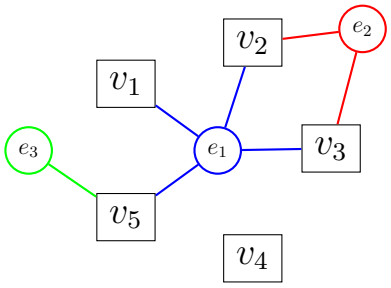
$$\mathbf{A}(H) = \mathbf{\Theta}\mathbf{\Theta}^T - \mathbf{D} \tag{2.3}$$

in which $\mathbf{\Theta}^T$ is the transpose of $\mathbf{\Theta}$.

The incidence matrix of a hypergraph is non-symmetric while the adjacency matrix is symmetric. In addition, Eq. (2.3) tells that the adjacency matrix of a given hypergraph can be quite dense even if its incidence matrix is sparse. This is a typical characteristic of hypergraphs that represent scientific applications. We will see in later chapters that this can provide challenges for partitioning some hypergraphs especially ones with very irregular structure.

Correspondingly, we define vertex and hyperedge adjacency as follows:

Definition 2.5 (Adjacent Vertices) *Given a hypergraph $H(V, E)$ and its adjacency matrix $\mathbf{A}(H)$, we say that two vertices $v, u \in V$ are **adjacent** if, and only if, its corresponding element in the $\mathbf{A}(H)$ is non-zero that is $a_{uv} \neq 0$.*



$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccc} e_1 & e_2 & e_3 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{pmatrix} \end{array} \qquad \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Figure 2.1: A sample hypergraph (left) with its incidence (middle) and adjacency (right) matrices.

Definition 2.6 *Given a hypergraph $H(V, E)$ and its incidence matrix $\boldsymbol{\Theta}(H)$, we say that two hyperedges $e, e' \in E$ are **adjacent** if, and only if, there is a vertex $v \in V$ such that $\theta_{ve} \neq 0$ and $\theta_{ve'} \neq 0$. Identically, e and e' are adjacent if, and only if, both contain v.*

An example of a hypergraph with vertex set $V = \{v_1, v_2, v_3, v_4, v_5\}$ and hyperedges set $E = \{e_1, e_2, e_3\}$ and its incidence and adjacency matrices are given in Fig. 2.1. The hyperedges e_1, e_2, and e_3 contain $\{v_1, v_2, v_3, v_5\}$, $\{v_2, v_3\}$, and $\{v_5\}$, respectively. The incidence matrix is of size 5×3 with 7 non-zeros which is equal to the number of pins in the hypergraph. The degree of a vertex v_i is the number of non-zeros in row i of incidence matrix, for example $d(v_1) = 1$ and $d(v_3) = 2$. All the vertices of this hypergraph except v_4 are adjacent. Furthermore, e_1 is adjacent to both e_2 and e_3, but e_2 is not adjacent to e_3.

2.2 Hypergraph Partitioning Problem

Given a hypergraph $H(V, E)$, let $\omega \colon V \mapsto \mathbb{N}$ be a function that assigns positive weights to the vertices of the hypergraph and let $\gamma \colon E \mapsto \mathbb{N}$ be function that assigns positive weights to the hyperedges.

Definition 2.7 (Hypergraph Partitioning) *Let k be a non-negative integer and let $H = (V, E)$ be a hypergraph. A **k-way partitioning** of H is a collection of sets $\Pi = \{\pi_1, \pi_2, \cdots, \pi_k\}$ such that $\bigcup_{i=1}^{k} \pi_i = V$ for which $\forall \pi_i, \pi_j \subseteq V, 1 \leqslant i \neq j \leqslant k$ and we have $\pi_i \neq \emptyset$ and $\pi_i \cap \pi_j = \emptyset$.*

The hypergraph partitioning problem is called a **bipartitioning** or a **bisectioning** problem if k is equal to two[2]. We say that vertex $v \in V$ is *assigned* to a part π if $v \in \pi$. Furthermore, the weight of a part is defined as follows:

Definition 2.8 (Part Weight) *Given a hypergraph $H(V, E)$ and a partitioning Π on the hypergraph, the weight of a part $\pi \in \Pi$ is sum of the weight of the vertices assigned to the part.*

$$\omega(\pi) = \sum_{v \in \pi} \omega(v). \tag{2.4}$$

A hyperedge $e \in E$ is said to be *connected* to (or *spans* on) the part π if $e \cap \pi \neq \emptyset$.

Definition 2.9 (Connectivity Degree) *For a given hypergraph $H(V, E)$ and a* k-way *partitioning Π, the **connectivity degree** of a hyperedge is the number of parts connected to the hyperedge. The connectivity degree of a hyperedge $e \in E$ is denoted as $\lambda_e(H, \Pi)$. A hyperedge is said to be **cut** if its connectivity degree is more than one.*

In the literature, hyperedges that are cut are said to be in the *cut set* of the partitioning. For the sample hypergraph given in Fig. 2.1, a possible bipartitioning Π_1 can be obtained as $\pi_{11} = \{v_1, v_2\}$ and $\pi_{12} = \{v_3, v_4, v_5\}$. In this bipartitioning, the connectivity degree of hyperedges e_1, e_2 and e_3 are 2, 2 and 1, respectively; therefore, hyperedges e_1 and e_2 are said to be cut.

In practical applications, we are interested in a partitioning of the hypergraph that optimises a cost function and imposes a constraint on the size of the parts. The first is called the *partitioning objective* and the the latter is called the *partitioning constraint* according to Karypis [Kar02]. In this thesis, we also refer to them as the *partitioning cost* and the *balance constraint*.

Alpert and Kahng [AK95] provide a survey of different partitioning costs. The most common partitioning cost objectives are minimising the hyperedge cut and

[2]We will later see in Eq. 2.6 that the weights of the parts can differ slightly and this is defined by introducing the *imbalance tolerance* to the hypergraph partitioning problem. In some literature, the bisectioning is defined as a bipartitioning that the weight of the parts are **exactly** equal. This means that the imbalance tolerance is zero. We avoid this distinction and use the bipartitioning and bisectioning, interchangeably, for a *2-way* partitioning with non-negative imbalance. We explicitly mention the word **exact** whenever we refer to exactly equal part sizes in a *2-way* partitioning.

minimising the Sum Of External Degrees (SOED). The first tries to minimise the sum of the weights of the hyperedges that are cut by a partitioning while the latter tries to minimise the sum of the hyperedge connectivity degree times their weights. The first objective mostly suits the graph partitioning problem (in which the size of all edges is two and an edge spans at most two parts) or the bipartitioning problem (the connectivity of the hyperedges are at most two). It is not a objective for the hypergraph partitioning problem because it does not consider the cardinality of hyperedges in the hypergraph (a hyperedge may spans several parts). Consequently, the second objective is considered to be a better for the hypergraph partitioning problem. There is a recently proposed objective which is derived from SOED and referred to as (*connectivity*−1) objective. This objective provides better modelling of the hyperedge cut in problems modelled with hypergraph. This objective is being used in most of recently proposed works on hypergraph partitioning [GL98,ÇA99,Kar13b,DBH+06,TK08]. In our thesis, we use the *connectivity*−1 objective as default unless stated otherwise.

Definition 2.10 (Hypergraph Partitioning Cost) *Given a hypergraph* $H(V, E)$ *and a partitioning* Π *on* H*, the partitioning cost is a cost function defined as follows:*

$$cost(H, \Pi) = \sum_{e \in E} (\gamma(e) \cdot (\lambda_e(H, \Pi) - 1)) \tag{2.5}$$

The objective of the hypergraph partitioning is to obtain a partitioning that minimises the cost[3]*.*

The cost of partitioning is also referred as the *quality* of partitioning [DBH+06]. In the rest of the thesis when we compare partitioning algorithms, we refer to the partitioning that gives smaller cost according to Eq. (2.5) as the partitioning with the higher quality.

[3]The majority of the applications that use hypergraph partitioning try to minimise the cost function. On contrary, there are some applications, such as data declustering, that are interested in the partitioning with the maximised cost function. An example is the work by Liu and Wu [LW01]. In the thesis, we consider a hypergraph partitioning problem that minimises the cost function unless stated otherwise.

As mentioned earlier, the weight of parts are usually bounded to a specified range in practical applications of the hypergraph partitioning. The constraint is referred to as the balance constraint and enforces the parts to have equal weights. The degree of freedom from the constraint is given by a real valued *imbalance tolerance* $\epsilon \in (0, 1)$. Given an imbalance tolerance, the weight of the parts should be limited as follows:

$$\overline{W} \cdot (1 - \epsilon) \leqslant \omega(\pi) \leqslant \overline{W} \cdot (1 + \epsilon), \ \forall \pi \in \Pi \tag{2.6}$$

where $\overline{W} = \sum_{v \in V} \omega(v)/k$.

In order to give and example of a partitioning with objectives we refer again to the example given in Fig. 2.1. Assume unit weights for all vertices and hyperedges in the hypergraph and a balance constraint $\epsilon = 0.2$. The partitioning Π_1 mentioned above is balanced, but it is not optimised. The cost of Π_1 is 2. A possible higher quality partitioning, and also optimal, is $\Pi_2 = \{\{v_2, v_3\}, \{v_1, v_4, v_5\}\}$ with unit cost.

In addition to the above single objective partitioning problem, there are some works proposed a *multi-objective* formulation [SKK99]. A multi-objective partitioning problem tries to optimise multiple objectives simultaneously. It can include both local and global objective functions, for example, it may try to minimise the cut while uniformly distributes cut set among the parts. In *multi-constraint* problem, a vector of weights is assigned to each vertex and the partitioning is done in a way such that the balance of the partitioning is preserved along each weight dimension while trying to optimise the cut. A use case of the multi-constraint problem is in VLSI circuit partitioning [Len90]. In addition to minimising the cut, the partitioning may also try to balance parameters such as: the noise, pins in each part, power consumption, and delay on the wires [Alp96].

Definition 2.11 (Hypergraph Partitioning Problem) *The hypergraph partitioning problem is finding a partitioning* Π *on the given hypergraph* $H(V, E)$ *according to Definition 2.7. This partitioning minimises the cost function that is given in Definition 2.10 and satisfies the balance requirement in Eq. (2.6).*

Finding an optimal solution to the hypergraph partitioning problem in Definition 2.11 above is shown to be NP-Hard [MJ79].

2.3 Rough Set Clustering

Rough set clustering is a mathematical approach to deal with uncertainty and vagueness in data analysis. The idea was first introduced by a Polish mathematician Zdzisław Pawlak in 1991 [Paw91]. The approach is different from statistical approaches, where the probability distribution of the data is needed, and fuzzy logic, where a degree of membership is required for an object to be a member of a set or cluster. The approach is based on the idea that every object in the universe is tied with some knowledge or attributes. Objects that are described with the same attributes are *indiscernible* and they can be put together in one category [PPS05]. The theory extracts a set of attributes for each object (also called *reduct set*) and performs classification and clustering based on these attributes [TP09]. It has found a lot of applications in engineering and data classifications and can be employed in applications such as feature selection and reduction, decision making rule generation, and data reduction. Thangavel and Pethalakshmi use rough sets for dimensionality reduction in high dimensional data sets [TP09]. In power engineering, Lambert-Torres employs rough set clustering to classify the current state of a power system [LT02]. Applications of rough sets in Artificial Intelligence (AI) and cognitive science is reviewed in [PRR12]. Lingras and West apply rough set clustering to classify web resources and web users for web mining [LW04]. Finally, Parmar et al. employ rough set theory to cluster categorical data in data mining [PWB07].

In rough set clustering, the data to be classified are called *objects* and they are described in an information system defined as follows:

Definition 2.12 (Information System) *An **information system** is a system represented as* $\mathfrak{I} = (\mathbb{U}, \mathbf{A}, \mathbf{V}, \mathcal{F})$ *where*

- $\mathbb{U}$ *is non-empty finite set of objects or the universe.*
- $\mathbf{A}$ *is a non-empty finite set of attributes.*
- $\mathbf{V}$ *is a multiset of attribute values such that* $\mathbf{V}_a \in \mathbf{V}$ *is a set of values for each* $a \in \mathbf{A}$.
- $\mathcal{F}$ *is a mapping function such that* $\mathcal{F}(u, a) \mapsto \mathbf{V}_a, \forall (a, u) \in \mathbf{A} \times \mathbb{U}$.

Table 2.1: An example of an information system with eight objects and five attributes. The value of each attribute is a non-negative integer number.

	a_1	a_2	a_3	a_4	a_5
$\mathbf{u_1}$	1	2	0	3	0
$\mathbf{u_2}$	2	0	0	1	3
$\mathbf{u_3}$	0	2	4	2	3
$\mathbf{u_4}$	1	2	0	3	0
$\mathbf{u_5}$	0	2	0	3	5
$\mathbf{u_6}$	1	2	0	3	0
$\mathbf{u_7}$	0	2	4	2	3
$\mathbf{u_8}$	2	0	0	1	3

Definition 2.13 (Decision Table) *An information system is called a* ***decision table*** *if* $\mathbf{A} = \mathbf{A}^C \cup \mathbf{A}^D$*, where* $\mathbf{A}^C$ *and* $\mathbf{A}^D$ *are sets of condition attributes and decision attributes, respectively, such that* $\mathbf{A}^C \cap \mathbf{A}^D = \emptyset$*.*

For any $\mathbf{B} = \{b_1, b_2, \cdots, b_j\} \subseteq \mathbf{A}$, an object $u \in \mathbb{U}$ can be denoted as a tuple $\vec{u}_{\mathbf{B}} = \langle \mathcal{F}(u, b_1), \mathcal{F}(u, b_2), \cdots, \mathcal{F}(u, b_j) \rangle$.

Definition 2.14 (Indiscernibility Relation) *For any* $\mathbf{B} \subseteq \mathbf{A}$ *there is an associated equivalence relation denoted as* $IND(\mathbf{B})$ *and called* ***B-Indiscernibility relation*** *such that:*

$$IND(\mathbf{B}) = \left\{(u, v) \in \mathbb{U}^2 \mid \forall b \in \mathbf{B},\, \mathcal{F}(u, b) = \mathcal{F}(v, b)\right\} \tag{2.7}$$

When $(u, v) \in \text{IND}(\mathbf{B})$, it is said that u and v are indiscernible under B and this is represented as an equivalence relation $u\mathcal{R}v$.

Definition 2.15 (Equivalence Relation) *An* ***equivalence relation*** *is a binary relation* $\mathcal{R} \subseteq \mathbb{U} \times \mathbb{U}$ *which is*

- **Reflexive:** $u\mathcal{R}u$*.*
- **Symmetric:** *If* $u\mathcal{R}v$*, then* $v\mathcal{R}u$*.*
- **Transitive:** *If* $u\mathcal{R}v$ *and* $v\mathcal{R}z$*, then* $x\mathcal{R}z$*.*

Furthermore, the equivalence class of u with respect to B is $[u]_{\mathbf{B}} = \{v \in \mathbb{U} \mid u\mathcal{R}v\}$. The equivalence relation provides a partitioning of the universe and it is represented as $\mathbb{U}/\text{IND}(\mathbf{B})$ or simply $\mathbb{U}/\mathbf{B}$.

An example of an information system is represented in Table 2.1. The set of objects is $\mathbb{U} = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8\}$ and the set of attributes is $\mathbf{A} = \{a_1, a_2, a_3, a_4, a_5\}$. The mapping function assigns integer values to objects per each attribute. Objects u_1 and u_2 can be described as tuples $\langle 1, 2, 0, 3, 0\rangle$ and $\langle 2, 0, 0, 1, 3\rangle$, accordingly. For the attribute set $\mathbf{B} = \{a_2, a_3, a_5\} \subseteq \mathbf{A}$, the equivalence classes and the partitioning of the objects are:

1. **Part 1:** $[u_1]_{\mathbf{B}} = [u_4]_{\mathbf{B}} = [u_6]_{\mathbf{B}} = \{u_1, u_4, u_6\}$
2. **Part 2:** $[u_2]_{\mathbf{B}} = [u_8]_{\mathbf{B}} = \{u_2, u_8\}$
3. **Part 3:** $[u_3]_{\mathbf{B}} = [u_7]_{\mathbf{B}} = \{u_3, u_7\}$
4. **Part 4:** $[u_5]_{\mathbf{B}} = \{u_5\}$

therefore, $\mathbb{U}/\mathbf{B} = \{\{u_1, u_4, u_6\}, \{u_2, u_8\}, \{u_3, u_7\}, \{u_5\}\}$.

Definition 2.16 (Information Set) *For a subset of attributes* $\mathbf{B} \subseteq \mathbf{A}$*, the* ***information set*** *with respect to* $\mathbf{B}$ *for any* $C \in \mathbb{U}/\mathbf{B}$ *is defined as follows:*

$$\vec{C}_{\mathbf{B}} = \vec{c}_{\mathbf{B}}, \forall c \in C \tag{2.8}$$

Following the above example, the information set for part $C = [u_1]_{\mathbf{B}} = [u_4]_{\mathbf{B}} = [u_6]_{\mathbf{B}} = \{u_1, u_4, u_6\}$ is $\vec{C}_{\mathbf{B}} = \vec{u_1}_{\mathbf{B}} = \vec{u_4}_{\mathbf{B}} = \vec{u_6}_{\mathbf{B}} = \langle 2, 0, 0\rangle$.

Definition 2.17 (Set Approximation) *Let* $\mathbf{B} \subseteq \mathbf{A}$ *be a set of attributes. Every set* $X \subseteq \mathbb{U}$ *of objects can be approximated using the information in* $\mathbf{B}$ *by defining a* **B-*lower*** *and* **B-*upper*** *set approximations. The lower approximation is represented as* $\underline{\mathbf{B}X}$ *and contains objects that definitely belong to* X*. The upper approximation is denoted as* $\overline{\mathbf{B}X}$ *and contains objects that possibly belong to* X*.*

$$\begin{aligned} \underline{\mathbf{B}X} &= \{x \mid [x]_{\mathbf{B}} \subseteq X\} \\ \overline{\mathbf{B}X} &= \{x \mid [x]_{\mathbf{B}} \cap X \neq \emptyset\} \end{aligned} \tag{2.9}$$

Furthermore, the ***boundary region*** *is denoted as* $\overline{\mathbf{B}X} - \underline{\mathbf{B}X}$*. A set is said to be* ***rough*** *if its boundary region is non-empty.*

Table 2.2: An example of a decision table which decides about whether a person has flue according to her headache, cough, and body temperature as condition attributes.

	Headache	Cough	Temperature	Flu
$\mathbf{u_1}$	No	No	Normal	No
$\mathbf{u_2}$	No	Yes	High	Yes
$\mathbf{u_3}$	Yes	Yes	Normal	Yes
$\mathbf{u_4}$	Yes	No	Fever	Yes
$\mathbf{u_5}$	No	Yes	Normal	No
$\mathbf{u_6}$	Yes	Yes	High	Yes
$\mathbf{u_7}$	Yes	No	Normal	No
$\mathbf{u_8}$	Yes	No	High	Yes
$\mathbf{u_9}$	No	No	High	No
$\mathbf{u_{10}}$	Yes	Yes	Fever	Yes

As an example, consider the decision table depicted in Table 2.2 which decides about whether a person has flu according to her/his headache, cough, and body temperature as condition attributes. The attribute set $\mathbf{B} = \{\text{Headache}, \text{Cough}\}$ partitions the universe into $\mathbb{U}/\mathbf{B} = \{\{u_1, u_9\}, \{u_2, u_5\}, \{u_4, u_7, u_8\}, \{u_3, u_6, u_{10}\}\}$. Let $X = \{x \mid \text{Flu}(x) = \text{Yes}\}$. According to Definition 2.17, the lower and upper approximations of X are:

$$\underline{\mathbf{B}X} = \{u_3, u_6, u_{10}\}$$
$$\overline{\mathbf{B}X} = \{u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_{10}\}$$

The example tells us that people having both cough and headache are definitely recognised as having flu, otherwise we can not decide certainly. The boundary region $\overline{\mathbf{B}X} - \underline{\mathbf{B}X} = \{u_2, u_4, u_5, u_7, u_8\}$ contains objects that we can not definitely say that they are in X according to $\mathbf{B}$.

The set of attributes can contain some redundancy. Removing this redundancy could lead us to a better clustering decision and data categorisation while still preserves the indiscernibility relation among the objects.

Definition 2.18 (Reduct) *Let* $\mathbf{B} \subseteq \mathbf{A}$ *be a set of attributes.* $\mathbf{B}$ *is said to be a* ***reduct*** *of* $\mathbf{A}$ *if*

1. $IND(\mathbf{B}) = IND(\mathbf{A})$.
2. $\mathbf{B}$ *is* minimal *and no attribute can be removed from* $\mathbf{B}$ *without changing indiscernibility relations.*

We refer again to the information system depicted in Table 2.1 as an example. The attribute set $\mathbf{B} = \{a_4, a_5\}$ is a reduct and the attributes $\{a_1, a_2, a_3\}$ are redundant and can be removed. We achieve the same partitioning of objects with respect to $\mathbf{B}$ and it is minimal such that removing more attributes from $\mathbf{B}$ changes the indiscernibility relations.

The reduct of an information system is not unique. It is shown that finding a minimal reduct of an information system is an NP-hard problem [SR92b]. The number of reducts of an information system with k attributes may equal to $\binom{k}{\lceil \frac{k}{2} \rceil}$. Calculating the reduct is not a trivial task and it is one of computational bottlenecks of rough set clustering. A number of heuristic algorithms have been proposed for problems whose number of attributes is not very large. Examples are the work by Wroblewski [Wro95,Wró98], which is based on genetic algorithms, and the work by Ziarko and Shan [ZS95] that uses decision tables based on Boolean algebra. These methods are not applicable to hypergraphs which are usually representing applications with high dimensionality and very large number of attributes. In addition, the process would be much complicated when these operations have to be repeated several time during the partitioning process. We propose a method for calculating an approximation of the reduct set of hypergraphs when proposing our hypergraph partitioning algorithms.

2.4 Cloud Computing

Cloud computing has become a popular word nowadays. It refers, generally speaking, to a collection of integrated hardware and network resources, software, and internet infrastructures that provide a variety of services over the internet. In this model, users can access their desired services on-demand regardless of where and how these services are hosted or provided. This is usually described as pay-as-you-go computing model in which users can subscribe to a cloud services, use them and pay only for the time that the services have been used. In addition, users do not need to pay any upfront or maintenance costs. The term "services" is better described in the definition provided by Armbrust et al. [AFG+09]

Definition 2.19 (Cloud Computing) *Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and system software in the datacenter that provide those services.*

They refer to services as *Everything-as-a-Service*. In their terminology, it is referred as *XaaS* or *X-as-a-Service*. The most common examples are Software-as-a-Service (SaaS), Infrastructure-as-a-Service (IaaS), and Platform-as-a-Service (PaaS). A more precise definition of cloud computing is provided by U.S. National Institute of Standards and Technology (NIST)[4].

Definition 2.20 (Cloud Computing) *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (for example networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

NIST categorises cloud services into three different categories in different layers of service. The service layout which determines what is being provisioned is depicted in Fig. 2.2. User can access these services through the internet using a client device such as a web browser or a program interface. These services from bottom-up are:

Infrastructure-as-a-Service (IaaS) provides the physical infrastructure of cloud computing. This category itself is divided into two subcategories: *Hardware-as-a-Service (HaaS)* and *Storage-as-a-Service*. *HaaS* provides the processing, memory, networking and all other fundamental things that the user can deploy and run an arbitrary software on them. They are provided in the form of *Virtual Machine* (VM) instances. User can create VMs of desired configuration and install arbitrary software tools and interfaces on them. The latter provides virtualised storage in the form of raw disk space or object storage. The network is provided in the form of virtualised network that connects VMs to the internet or a private network. User does not control the cloud infrastructure, but only

[4] `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`

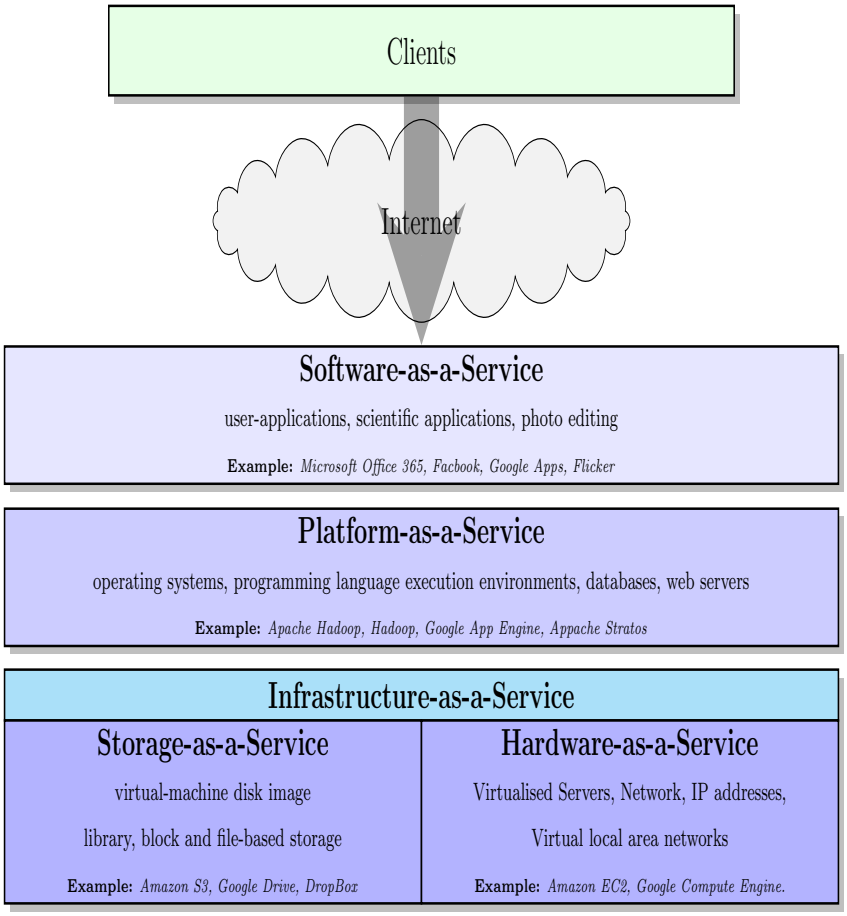


Figure 2.2: The general reference model of the cloud.

the operating system, VMs behaviour and software installed on them. She/He has a limited access over the networking capabilities such as firewall settings.

Platform-as-a-Service (PaaS) offers services such as computing platform, runtime environments, databases, and web servers. It provides a framework for the software and application developers to be able to develop and customise their applications and software. These services are backed by a core middleware platform that is responsible for creating the abstract environment where applications are deployed and executed [BVS13]. It means that service provider manages runtime environment, middleware, operating system, networking, fault tolerance, storage, etc., but users only concentrate on their application development and logic of their work.

Software-as-a-Service (SaaS) layer is build on the top of PaaS. It provides access to software applications which are referred to as *on-demand software*. Common

example are Adobe Photoshop, and Microsoft Office. Customers run these application on the cloud instead of a local desktop computer. These applications are usually shared among different users. Customers do not need to worry about installation, maintenance and running of the applications and it is the responsibility of the service provider. Users use a web interface program to access these services.

Users pay for the cloud services. The pricing model for the above services are described as *dollar per hour* and the cost is different based on the type of the service requested.

NIST provides five essential characteristics of the cloud as follows:

1. On-demand self service: A consumer can access computing resources automatically without requiring any human interaction.

2. Broad Network Access: Services and capabilities are available over the network and can be accessed through standard mechanisms anywhere regardless of the type of the device customers use.

3. Resource Pooling: The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model. Different physical and virtual resources are dynamically assigned and reassigned according to the consumer's demand. Consumers have no control over the exact location of the provided resources, but they might be able to specify location at a higher level of abstraction.

4. Rapid Elasticity: Resources can scale dynamically and rapidly both inward and outward based on the demand. From the customer's point of view, the services are unlimited and can be requested at any time.

5. Measured Service: Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (for example storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported

that provides transparency for both providers and consumers of the utilized service.

There are other characteristics such as *productivity*, in which multiple users can work on the same data simultaneously, *reliability*, such that user data is backed up on different places to guarantee that the data is not lost, *security*, in which the user has the confidence that his/her data is well protected and secured and it can not be accessed unauthorised users, and *scalability*, such that the resources can be scaled both horizontally (the number of resources) and vertically (using more powerful resources).

Furthermore, cloud systems can be categorised based their deployment model. They are divided into three categories [BVS13] as follows:

Public/Internet Clouds This is the most common form of cloud computing which is owned and operated by a third party organisation. Everybody can subscribe and use its services based on pay-as-you-go pricing model. This type of cloud is usually bigger in scale compared to others.

Private/Enterprise Clouds The cloud infrastructure is provided for exclusive use of a single organisation. There are two types of this cloud: *On-site private cloud* that is hosted within an organisation's own datacentre. It is suitable for organisations that need complete control of cloud configuration and security. The other is *externally hosted private cloud* in which the cloud infrastructure is hosted by a third party cloud provider.

Hybrid/Inter Clouds This type is a combination of both public and private clouds. For example organisations can lease public clouds when the capacity of their private cloud is insufficient. In another model, a company can store its sensitive data, which needs high security, on its private cloud and use a public cloud for other purposes.

As mentioned above, cloud services are offered in virtualised form. Virtualisation plays an important rule in cloud computing and it is the enabling technology of the cloud, because it allows a degree of customisation, configuration, security, isolation,

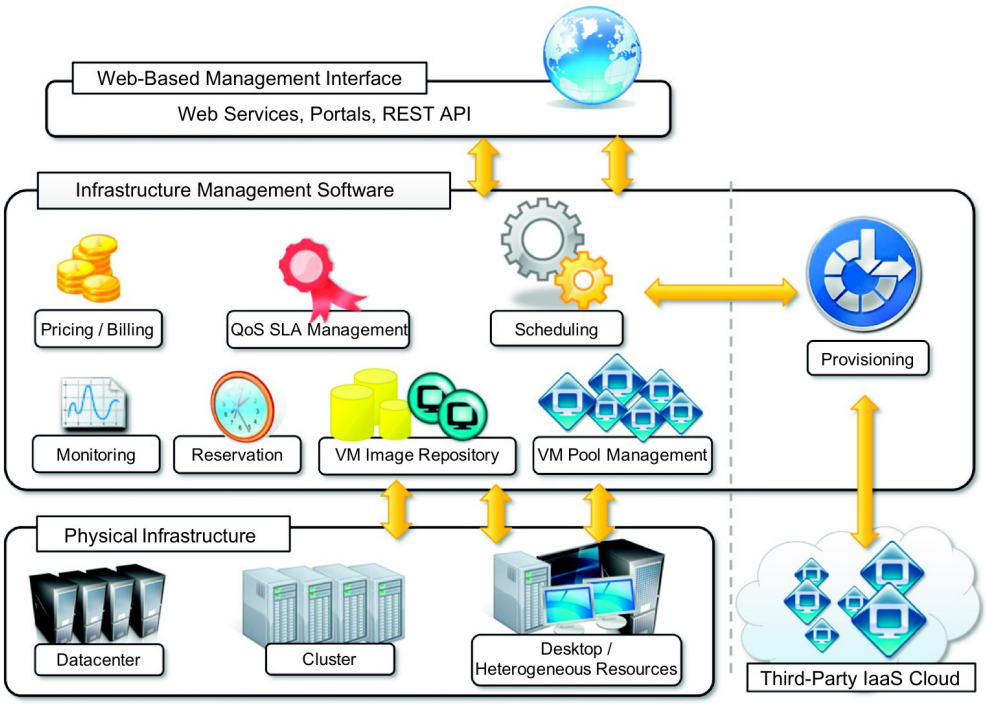


Figure 2.3: Infrastructure-as-a-Service (IaaS) architecture and its components [BVS13].

and manageability in cloud systems [BVS13]. Virtualisation is built on top of a physical computing node and separates it into one or more *virtual* instances. Instances share physical resources but they are consolidated and isolated and can be configured and controlled separately. Virtualisation in the cloud is mostly offered in the form of virtualised hardware and storage in IaaS service layer. Programming language virtualisation is offered as PaaS services. Virtualisation provides several advantages for the cloud computing. Besides from the isolated execution and manageable controllability of VM instances, portability is another important feature of virtualisation that allows moving one VM to another place with respect of physical systems. Furthermore, it provides efficient use of resources by sharing them. Examples of the virtualisation technologies are Xen[5], VMware[6], and Microsoft Hyper-V[7].

Virtualised resources are provided through IaaS solutions that is important for us as we work with this layer for our simulations and evaluations in the thesis. Figure

[5] http://www.xenproject.org/
[6] http://www.vmware.com/
[7] http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx

2.3 shows a reference architecture model for IaaS implementation. It is composed of three distinct layers: *physical infrastructure*, *infrastructure management software* and the *user interface* [BVS13]. The top layer provides access to the services which is based on Web 2.0 technologies. The middle layer is designed for management of the infrastructure and includes several components:

- **pricing/billing:** calculates the cost of using VMs.
- **monitoring:** tracks the execution of the VM instances for reporting and analysing purposes.
- **reservation:** manages the reservation of the VMs by the customers.
- **QoS/SLA management:** is responsible for the Quality of Service (QoS) and Service Level Agreement (SLA) with the customers to ensure that a certain level of agreed quality is met for the customers.
- **VM repository:** it has a database of VM images that customers can use for creating their VM instances. Users can upload their specific VM image.
- **VM pool manager:** manages running VM instances.

The bottom layer provides the physical infrastructure for running VMs in the cloud. *Datacentres*, which is a network of hundreds of thousands commodity hardware, is the most common infrastructure for hosting the cloud. It is where computing resources, storage and network are implemented and provided to the cloud in virtualised format. Cloud computing is built on top of one or more datacenters. In datacenters, the *price/performance* ratio is more important than performance alone, and the storage and energy efficiency are more important than shear speed performance [HDF11]. The use of commodity hardware as a low-end computing element benefit from the economies of scale and provides much better price/performance ratio compared to high-end computing nodes [BDH03,HDF11].

The network connectivity in datacenters usually follows a hierarchical architecture. In general, low-end commodity servers are packed together in a unit or *rack* and they are interconnected using a local Ethernet switch. At a higher level, racks are

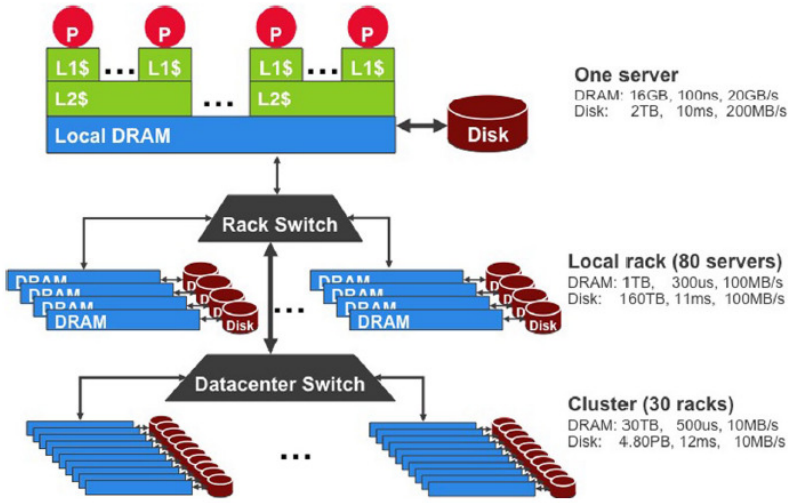


Figure 2.4: Storage hierarchy in a distributed storage datacenter from programmers point of view [BCH13].

connected using rack-level switches which themselves have upper links to cluster-level switches. As an example, a rack with 40 servers with 1-Gbps port have 4-8 uplinks to the cluster-level switch, which means 5 to 10 times degradation in inter-rack communication bandwidth compared to the intra-rack communication bandwidth [BCH13]. As a result, intra-rack communication is much faster than inter-rack communications, and they are both much faster than inter-cluster communications. This makes data locality an important factor of scalability and performance and it should be considered by the programmers and software developers while developing their applications in the cloud. This hierarchy of network connectivity can be a bottleneck specially for running applications with high volume of communication between servers.

Storage elements in datacenter can be connected to servers directly and managed by a distributed file system or cluster-level switches [BCH13] as a part of Network Attached Storage (NAS). The use of commodity hardware demands implementing a fault-tolerant file system and the most common ways are replication based strategies and error correction codes [CJZM10]. In a distributed storage architecture, storage hierarchy follows a similar architecture as network hierarchy. The storage hierarchy from the programmers point of view is depicted in Fig. 2.4. Barroso et al. [BCH13] have quantified latency, bandwidth, and capacity of memory accesses in the storage

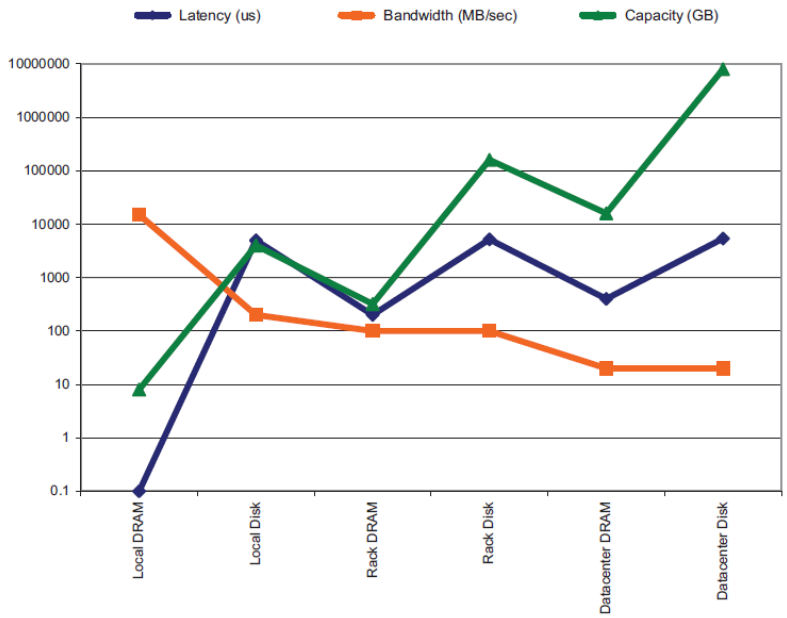


Figure 2.5: Latency, bandwidth and capacity of storage access in the storage hierarchy of a distributed storage datacenter [BCH13].

hierarchy as proposed in Fig. 2.4. Their model, despite being too simple, provides an idea of storage accessibility in the cloud among different layers of hierarchy. They assume a datacenter with 2000 servers, each with 8 GB of DRAM and two levels of cache and four 1 TB disk drives. Servers are arranged in racks of size 40 and they are connected to a 1 Gbps link to the rack-level switches. Each rack level switch has additional eight 1 Gbps ports to connect to cluster-level switches. Their evaluation is represented in Fig. 2.5. According to the results, cluster level storage, albeit being much bigger than local server storage, provides much higher latency and it can be a restricting factor of performance in the cloud (similar to the network latency mentioned above).

Recently, there is a great interest for running High Performance Computing (HPC) applications in the cloud [MAB+10]. Services are mostly offered in IaaS and PaaS layers. The cloud provides lots of advantages for scientific computing such as elasticity, virtualisation flexibility, low maintenance and setup costs, and dynamic reallocation. It provides a cost-effective running environment for HPC applications with faster turnaround time compared to private HPC clusters [YCD+11]. Despite these advantages, limited network interconnection capacity and the overhead imposed

by network and storage virtualisation are two major performance bottlenecks for HPC on the cloud [YCD+11,MDH+12]. Consequently, the cloud providers have decided to offer better cloud infrastructure for HPC applications such as Amazon Elastic Compute Cloud[8] (Amazon EC2), Magellan[9]: Cloud Computing for Science, and IBM Platform Computing[10]. Cloud computing provides a new opportunity for scientific applications and there is a great interest and effort for transferring these applications into the cloud [ZG11,You11,GKG+13].

2.4.1 OpenStack Cloud Software

In this section, we introduce the architecture of the OpenStack cloud software which provides cloud computing services mostly at IaaS level. It is an open source cloud operating system that is released under Apache 2.0 license and manages cloud resources in a datacenter. Users can can access their resources through a web interface (also called *dashboard*), command line tools, and RESTful APIs. It was founded in 2010 as a joint project between Rackspace[11] and National Aeronautics and Space Administration (NASA) to provide an open source software for organisations to create and offer cloud computing services on standardised hardware. OpenStack started with a missions:

> *"Produce the ubiquitous Open Source Cloud Computing platform that will meet the needs of public and private clouds regardless of size, by being simple to implement and massively scalable."*

and a motto:

> *"OpenStack is **open** source, **openly** designed, **openly** developed by an **open** community."*

Recently, more than 500 companies are working with the project and contribute with its development. As depicted in Fig. 2.6, its design architecture typically has the following components

[8] http://aws.amazon.com/ec2/
[9] http://www.alcf.anl.gov/magellan
[10] http://www-03.ibm.com/systems/platformcomputing/
[11] http://www.rackspace.com/

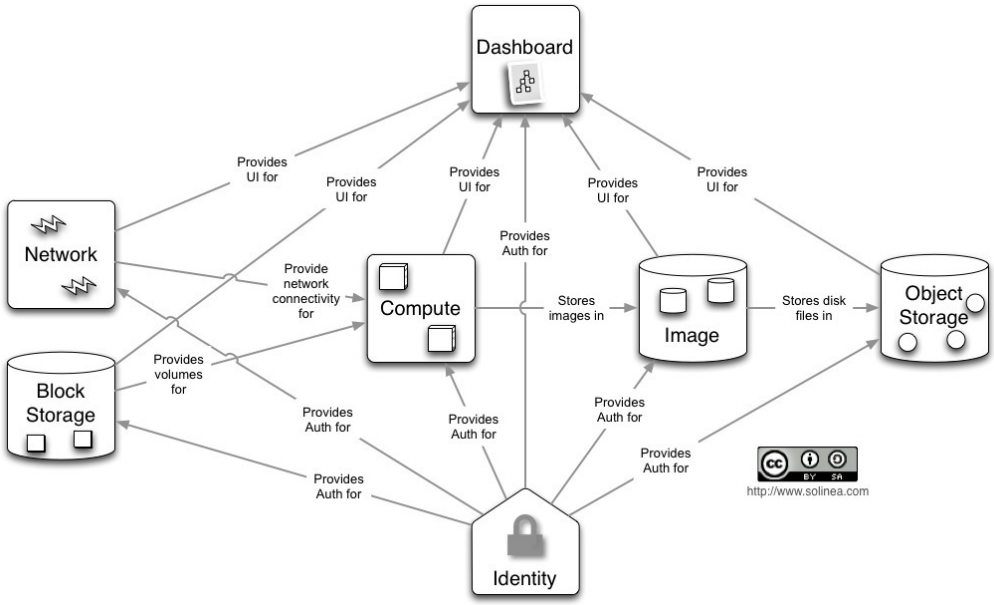


Figure 2.6: Architecture of the OpenStack cloud software.

- OpenStack Identity (keystone)
- OpenStack dashboard (horizon)
- OpenStack Compute (nova)
- OpenStack Object Storage (swift)
- OpenStack Block Storage (cinder)
- OpenStack Image service (glance)
- OpenStack Networking (neutron) or legacy networking (nova-network)
- OpenStack Orchestration (heat)
- OpenStack Database service (trove)
- OpenStack Data processing service (sahara)

Identity (keystone) is used to provide an authentication and authorization service for other OpenStack services. *Dashboard (horizon)* is a web interface that allow users to manage resources and services including OpenStack Compute cloud controller,

nova, and neutron. *Compute (nova)* provides control over VM instances and networks, and enables the user to manage access to the instances by defining users and project groups. *Object storage (swift)* is a component for creating redundant and scalable data storage using clusters of standardized servers to store big data. *Block Storage (cinder)* enables users to add extra block-storage to VMs and manage them. This service is similar to the Amazon EC2 Elastic Block Storage (EBS). *Image service (glance)* is used for storing and managing VM images. Users can also upload their own images. *Networking (neutron)* provides necessary API for defining network connectivity and addressing in the cloud and enables users to leverage different network technologies. It is used to manage VLAN, IP addresses to different VMs, and firewalls. *Orchestration (heat)* is used for managing multiple cloud applications by providing some APIs. *Database service (trove)* provides a scalable and reliable Database-as-a-Service (DaaS) in OpenStack for both relational and non-relational databases. Finally, *data processing service (sahara)* provides services to provision Hadoop[12] cluster in OpenStack by specifying parameters such as Hadoop version, cluster topology, and hardware details.

Creating instances are an easy process that can be done from the dashboard. Generally speaking, user first requests an VM instance from the console and she/he determines the image (operating system) to be loaded for this instance. *Horizon* passes the request to *nova*. *Nova* checks with *keystone* for identity checking and authentication. If it is authorised, *nova* asks *neutron* to provision networking and IP address for the instance. Finally, *nova* asks *glance* to load the requested OS image for the instance then it mounts the image on the VM and performs necessary configurations. On boot-up, the VM requests an IP address from the DHCP component on *neutron*. In this stage, VM is ready and can be customised by the user. The user can log into the node through services such as Remote Desktop Control (RDC), *ssh* connection, and from OpenStack dashboard using local VNC client.

[12] `https://hadoop.apache.org/`

Chapter 3

Related Work

Finding an optimal solution to the hypergraph partitioning problem considering the cost objective and balance constraint is known to be NP-Hard [MJ79], but a number of polynomial time heuristics have been proposed to produce a near-optimal solution for the problem. This chapter reviews the works related to the hypergraph partitioning problem. The content of the chapter is divided into four sections. Hypergraph partitioning algorithms are reviewed first. The proposed algorithms in the field can be categorised according to their specifications and the characteristics of the input hypergraph. In the second section, we review available software tools designed for hypergraph partitioning and the state-of-the-art tools are introduced. Section 3 reviews some of the applications of hypergraph partitioning. The application to be partitioned should be modelled first by a hypergraph. The modelling process is application specific and depends on the objectives of the problem under investigation. The aim of the chapter is to review algorithms, tools, and applications of hypergraph partitioning and we do not study the works related to graph partitioning. Works related to graph partitioning are studied whenever necessary, for example when a specific idea originates from the graph partitioning context.

In the last section, we study the related work for transferring scientific applications into the cloud and the challenges on the way. Despite the advantages that the cloud provides for HPC applications such as elasticity, small startup times, and maintenance costs, the transfer process is not straightforward and the specification of the cloud and the application should be considered. The section investigates the issues and

identifies the key characteristics of the cloud that can be problematic for this transfer.

3.1 Hypergraph Partitioning Algorithms

Heuristic algorithms for hypergraph partitioning can be categorised according to different aspects and specifications. This section reviews different categories of algorithms for the *k-way* hypergraph partitioning problem.

Graph partitioning is a well studied problem and efficient heuristic algorithms have been proposed; therefore, some algorithms try to overcome the hypergraph partitioning problem through graph partitioning. This needs the transformation of the hypergraph into a graph representation. The transformation should be done in a way to preserve the structure of the hypergraph. This is a hard task and there is no known algorithm with this specification [IWW93]. For example, it is difficult to define the weights of edges in the graph after transformation and the contribution of cut hyperedges to the partitioning cut. To clarify the issues, we study a bipartitioning on the hypergraph. One natural transformation of the hypergraph to a graph representation is to model a hyperedge and the vertices it contains with a clique with unit weight edges. In the bipartitioning problem, the contribution of a hyperedge with unit weight to the partitioning cut is one, while a clique that is evenly distributed on the cut contributes quadratically in the size of clique to the cost of bipartitioning. Another solution is to assign the weight $\frac{1}{|e|-1}$ to each edge of the clique for a hyperedge e of size $|e|$. In this case, a bipartition with one vertex on one side of partition and all other vertices on the other side gives unit cost partitioning while the cost of bipartitioning with half of the vertices on each side of partition is $\approx |e|/4$. Lengauer [Len90] shows that no matter how we choose the cost of each edge in the clique, the cost of bipartitioning always deviates $\Omega\left(\sqrt{|e|}\right)$ from the desired unit cost of a hyperedge. The problem is also impossible if we use different topologies other than clique for representing a hyperedge [Len90]. Due to the lack of a correct transformation, partitioning algorithms that work directly on the hypergraph are preferable for practical applications; therefore, we study this types of algorithms. An extended study about modelling hypergraphs with graphs

can be found in Alpert's work [Alp96].

In the second classification, algorithms can be **move-based** (also known as *flat* algorithms) [FM82] or **multi-level** (or *hierarchical*) [ÇA11]. Move-based algorithms are those that try to improve the cost objective by directly working on the original hypergraph. In constrast, multi-level algorithms provide a sequence of successive approximation of the original hypergraph. The size of the hypergraph is reduced with each approximation. The process continues until the smallest hypergraph has only few vertices. At this stage, a partitioning of the hypergraph is calculated on the smallest hypergraph and this partitioning is projected back to the original hypergraph by going through the same number of approximation levels. While projecting back, the cost function is further refined in each level.

In third classification, algorithms are **recursive** or **direct**. **Recursive bipartitioning** is a divide-and-conquer paradigm. First, the algorithm calculates a bipartitioning on the hypergraph. The hypergraph is then split into two sub-hypergraphs (one for each part) according to vertex-to-part assignments and the algorithm continues with a bipartitioning of each sub-hypergraph independently. The recursion stops when k parts are obtained. These algorithms are also known as **recursive bisectioning** [KAKS99]. Some parameters such as balance constraint and imbalance tolerance factor are changing and they should be re-adjusted after each bipartitioning recursion for each sub-hypergraph. **Direct *k-way* partitioning** algorithms directly calculate *k-way* partitioning and obtain k partitions without any recursion [KK99].

In the forth classification, algorithms are divided into **serial** [ÇA11] and **parallel** [TK08] algorithms. Serial algorithms are those that run on a standalone computer. However, the performance of serial algorithms are limited due to the limitation of the hardware resources on a single computing node. For example, we may have a very big hypergraph that can not fit into the computational capacity and the memory of a standalone computing node; therefore, the need for parallel hypergraph partitioning algorithms is inevitable.

Finally, there are some scientific applications in which the structure of the problem changes over time. An example of these applications is adaptive mesh refinement in which the structure of the mesh changes continuously. In parallel

implementation using k processors, the problem is modelled with a hypergraph and a *k-way* partitioning is calculated on the hypergraph and each parts is assigned to a processor. Although the hypergraph is balanced at the beginning of the process, structural changes generate unbalanced partitions over time. This results in unbalanced load among processors such that some processors would have more work to do compared to the others. This load imbalance provides performance degradation. Therefore, we need to tune the partitioning again and balance the load among processors by transferring some of the load from overloaded processors to underloaded processors. One solution is to re-run the whole partitioning process from the scratch and perform another run of the hypergraph partitioning to obtain a new load distribution. The other way is to repartition the hypergraph dynamically according to the current state of the partitioning. The algorithms of the second type are known as **dynamic** algorithms [ÇBD+07] and the first type are called **static** algorithms [TK08] that do not assume dynamic changes in the hypergraph under investigation.

3.1.1 Move-Based Heuristics

Move-based heuristics are those algorithms that build a new problem solution based on the neighbouring structure in the hypergraph. The neighbouring structure is built over a set of feasible solutions and the previous history of the algorithm. It is based on *moves* as an operator for transforming from one solution to the other. The algorithm stops when moves do not make any further improvement to the partitioning cost. By moves, we mean an operation that changes the state of a vertex; for example, moving the vertex from one partition to the other. Algorithms of this type are either *memoryless* such as Simulated Annealing (SA), which uses solely the current problem state to move to another neighbouring state, or *memory-based* that iteratively improves the problem solution using previous history of the moves. Algorithms in the second category are known as *iterative move-based* algorithms.

Iterative Move-Based Algorithms

Iterative move-based algorithms are a set of techniques that have been proved to be successful in practice and generate quite good partitioning qualities in a reasonable running time. They are applicable to all problem cases and various graph/hypergraph partitioning problems with different specifications and structures without any limitations. These algorithms start with an initial feasible solution of the problem and iteratively move vertices between part boundaries to generate better solutions. The algorithm stops when no further vertex move or further improvement of partitioning cost objective is possible. These algorithms converge into the local minima which depends on two factors: the initial distribution of the vertices and neighbouring structure among them.

Initial distribution is usually done using a randomised algorithm. In general, algorithms are composed of *passes* and all vertices are free to move at the beginning of each pass. Then, vertices are moved one at a time in each step of an iteration. The decision for selecting a vertex to be moved is based on the improvement that vertices provide to the partitioning cut if they move from one part to another. The degree of improvement for a vertex is usually expressed as the vertex *gain*. In each step, the vertex that gives the highest gain is moved to another part. A vertex is allowed to move at most once during a pass and it is locked after the move to prevent further moves of the vertex in the current pass[1]. The algorithm stops when no further improvement to the cost functions is possible.

Alpert [Alp96] gives several reasons why iterative move-based algorithms are suitable for practical applications. First, they are intuitive; it is an obvious way of improving a problem solution by iteratively making small improvements. Second, they are simple and easy to implement. Third, they are fast and can easily provide a trade-off between the running time and the quality of the partitioning by changing some parameters of the algorithm. For example, when a higher quality of partitioning is desired, the algorithm can be run several times with different initial distribution of vertices. Each run gives a partitioning on the hypergraph. Among them, the one

[1]Vertex locking is done in order to prevent vertex thrashing that is a vertex continuously moves in/out of a part while it does not make any improvement to the partitioning cut.

that gives the highest quality is selected as the final partitioning solution. Finally, they are independent of the partitioning objective while some other move-based algorithms may need the partitioning objective to be of a special type.

The two well-known algorithms in this category are Kernighan–Lin (KL) [KL70] and Fidduccia–Mattheyses (FM) [FM82] algorithms. These algorithms are basically proposed for the bipartitioning problem but they are also extended for the direct *k-way* partitioning such as direct *k-way* FM algorithm [San89]. We review these algorithms due to their importance as they are also employed in our partitioning algorithms proposed in later chapters.

Kernighan–Lin (KL)

KL algorithm is proposed for **exact** bisectioning[2] on graphs. The algorithm starts with an initial distribution of vertices. Then pairs of vertices are selected and exchanged between the parts if the exchange improves the partitioning cut. Assume a graph with n vertices and unit weight for each vertex. There are

$$\frac{1}{k}\binom{n}{p}\binom{n-p}{p}\cdots\binom{2p}{p}\binom{p}{p}$$

solutions to a *k-way* partitioning of the graph in which $p = n/k$ is the size of the parts (n is divisible by k). $\binom{n}{p}$ is the number of ways of choosing the first part, $\binom{n-p}{p}$ is the number of ways of choosing the second parts, and so on. The expression yields very large numbers even for small values of n and k. The random assignment is not a satisfactory solution as it is very unlikely to generate a near-optimal partitioning even on small graphs. It is shown that for graphs with incidence matrix of size 32×32, there are typically three to five optimal solutions among $\frac{1}{2}\binom{32}{16}$ available solutions.

Given a graph $G(V, E)$ on n vertices, KL starts with a arbitrary equally-sized parts A and B each having $n/2$ vertices. It tries to improve the cut size by a series of vertex interchanges between A and B. The algorithm stops when no further exchange

[2] As discussed in Chapter 2.2, the exact bisectioning is a bipartitioning on the graph such that the size of the parts are **exactly** equal. This is achieved when the imbalance tolerance is zero.

improves the cost function. Assume that A^* and B^* are an optimum bipartitioning solution. There are subsets $X \subset A$ and $Y \subset B$, with $|X| = |Y| \leqslant n/2$ such that interchanging X and Y between A and B produces A^* and B^* that is:

$$A^* = A - X + Y$$
$$B^* = B - Y + X$$

KL tries to identify X and Y approximately by sequentially identifying their elements. For every vertex in $v \in A$, the *external* ($C_{\text{ext}}(v)$) and *internal* ($C_{\text{int}}(v)$) costs are defined as follows:

$$C_{\text{ext}}(v) = \sum_{e_{uv}=\{v,u\} \in E, u \in B} \lambda(e_{uv})$$
$$C_{\text{int}}(v) = \sum_{e_{uv}=\{v,u\} \in E, u \in A} \lambda(e_{uv})$$

Similar applies to the vertices in B. Subsequently, the gain of a vertex is defined as:

$$g(v) = C_{\text{ext}}(v) - C_{\text{int}}(v).$$

The gain is technically the amount by which the cut is decreased if v is moved from A to B. The decrease in the partitioning cut size for an exchange (v, u) would be $g(v) + g(u) - 2\lambda(e_{uv})$, where $\lambda(e_{uv})$ is the cost of the edge connecting v and u, if one exists; otherwise, it is zero. The algorithm starts by initially calculating the gain of all vertices and proceeds in iterations. In each iteration, a pair of vertices that gives the maximum exchange gain is selected and swapped among the parts. After the move, the selected pair is locked to prevent further exchange and vertex thrashing between the parts. Then the gain of adjacent vertices on the moved vertices are updated accordingly. The number of iterations is $n/2$ for a graph with n vertices to consider all pair permutations. After $n/2$ iterations, the partitioning goes back to the initial state such that all vertices of A are now in B and vice versa. The algorithm keeps track of the pair swaps and the partitioning cost. At the end, the best cut size is calculated; all exchanges up to the point that gives the best cut size are kept and the others are reversed. All pairs of the vertices are considered during the partitioning cost minimisation process. The time complexity of the algorithm is

$O(n^2 \log n)$ if we keep a sorted list of best exchange pairs and a sorted list of vertex gains[3].

The algorithm, albeit simple and easy to implement, has some disadvantages. Langauer [Len90] describes the disadvantages of KL algorithm as the following:

1. The algorithm only works with unit vertex weights which makes it inapplicable to some problems such as VLSI circuit partitioning.

2. The algorithm is *exact bisectioning*, which is again not the case for most of practical applications.

3. The algorithm cannot be applied on hypergraphs.

4. The complexity of a pass is high. In practical applications, we are interested in linear time complexity in each pass.

5. The likelihood that the algorithm gets stuck in local minima is very high. Furthermore, it is tied with too much indeterminism such that the quality of the cut can vary dramatically according to the order we choose vertex pairs.

The algorithm encourages the introduction the FM algorithm on hypergraphs and resolves the above mentioned limitations.

Fidduccia–Mattheyses (FM)

FM algorithm resolves limitations of KL algorithms mentioned in Section 3.1.1 and it is applicable to hypergraphs. The logic of the algorithm stays the same except that FM moves only one vertex at a time. Like KL, the algorithm stops when it can not make any further improvements. Following the discussion about the KL algorithm, *external* and *internal* costs of a vertex $v \in A$ are defined as follows:

$$C_{\text{ext}}(v) = \sum_{e \in E_{\text{ext},v}} \lambda(e)$$

$$C_{\text{int}}(v) = \sum_{e \in E_{\text{int},v}} \lambda(e)$$

[3]All logarithms are base 2.

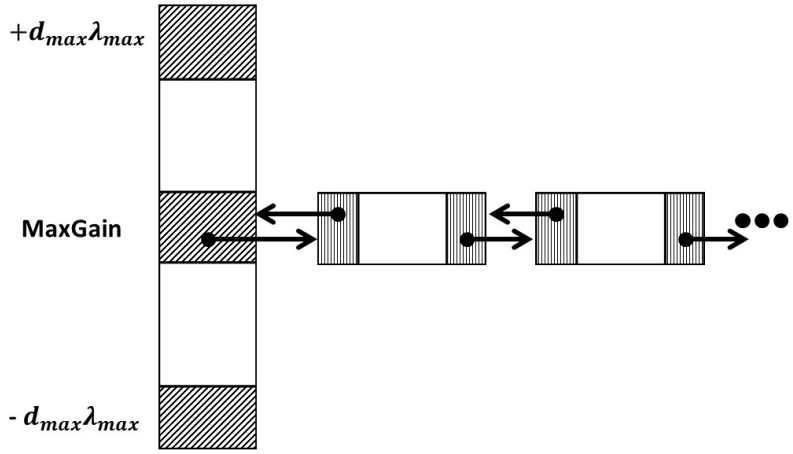


Figure 3.1: Gain bucket data structure in Fidduccia-Mattheyses (FM) algorithm.

where $E_{\text{ext},v} = \{e \in E \mid e \cap A = \{v\}\}$ is the set of hyperedges that are incident on v such that v is their only vertex in part A, and $E_{\text{int},v} = \{e \in E \mid v \in e, e \cap B = \emptyset\}$ is the set of hyperedges that have no vertex in B. By moving v to part B, all $E_{\text{ext},v}$ are removed from the cutset, but $E_{\text{int},v}$ are added to the cutset. The gain of the vertex move is calculated similar to KL that is $g(v) = C_{\text{ext}}(v) - C_{\text{int}}(v)$. Unlike KL, balance constraint is considered with each vertex move. In each iteration, a vertex with the maximum gain is selected as a move candidate if moving the vertex does not violate the balance constraint. Therefore, FM is not limited to unit vertex weights. The pseudo-code of the algorithm is depicted in Algorithm 1. The input to the algorithm is the hypergraph $H(V, E)$ and it calculates a bipartitioning of H.

Line 6 of the algorithm chooses a vertex to be moved (denoted as *cand*). For this purpose, the FM algorithm builds a gain bucket data structure depicted in Fig. 3.1 for each part. In the figure, $d_{\max} = \max\{d(v) \mid v \in V\}$ and $\lambda_{\max} = \max\{\lambda(e) \mid e \in E\}$. The size of each bucket is $[-d_{\max}\lambda_{\max}, +d_{\max}\lambda_{\max}]$, where each cell i points to a list of vertices in the hypergraph whose gain equals to i. *MaxGain* is a pointer that always points to the first non-empty bucket whose vertices have the highest gain. The vertex to be moved is selected from the maximum gain bucket. Ties are broken by selecting a vertex from the maximum gain bucket that gives a more balanced partition. The gain bucket is updated on each vertex move and *MaxGain* pointer is updated accordingly if there is a change in maximum gain value. When a vertex

Algorithm 1 Fidduccia-Mattheyses (FM)

```
1: procedure FM(H(V, E))
2:     Calculate gain values for every vertex
3:     Initialise gain buckets
4:     ∀v ∈ V, state(v) ← free
5:     while vertex moves improves the cut do
6:         cand ← a free vertex with the highest gain
7:         MoveVertex(H(V, E),cand,source)
8:         Update gain bucket data structure

Require: vertex v moves from source part π_s to destination part π_d.
9: procedure MoveVertex(H(V, E),v,π_s)
10:    Move vertex from π_d to π_d
11:    Update part weights
12:    state(v) ← locked
13:    Remove v from the gain bucket
14:    for all hyperedges e ∈ E that are incident on v do
       Phase 1 - Level 1 Update
15:        pins_s(e) = pins_s(e) − 1
16:        if pins_s(e) = 0 then
17:            for all {u | u ∈ e, u ∈ π_d, state(u) = free} do
18:                g(u) = g(u) − λ(e)
19:        else if pins_s(e) = 1 then
20:            for all {u | u ∈ e, u ∈ π_s, state(u) = free} do
21:                g(u) = g(u) + λ(e)

       Phase 2 - Level 2 Update
22:        pins_d(e) = pins_d(e) + 1
23:        if pins_d(e) = 1 then
24:            for all {u | u ∈ e, u ∈ π_s, state(u) = free} do
25:                g(u) = g(u) + λ(e)
26:        else if pins_s(e) = 2 then
27:            for all {u | u ∈ e, u ∈ π_d, state(u) = free} do
28:                g(u) = g(u) − λ(e)
29:    Update gain buckets
```

is moved, the gain of its adjacent vertices may change depending on the status of hyperedges incident on the vertex. The function *MoveVertex* updates the gains after a move. The function needs two arrays $pins_A$ and $pins_B$ for this purpose. They are arrays of length $|E|$. $pins_A$ and $pins_B$ keep the number of pins for each hyperedge that are in part A and part B, respectively. Moving a vertex is done in two phases.

First the vertex is removed from the source part in phase one, then the vertex is added to the destination part in phase two. The complexity of FM algorithm is shown to be $O\{d_{\max} \cdot \lambda_{\max} + pins(H) \cdot \lambda_{\max}\}$ [FM82,Len90]. In case of unit hyperedge weights, it is $O(pins(H))$.

The FM algorithm described above is known as a first level vertex gain algorithm. The most important hyperedges that have positive effect in calculating vertex gains in the initialisation step are those which have only one vertex in either of the parts and all of the other vertices fall into the other part. In addition, as described above, the only tie breaking strategy among the vertices in the *MaxGain* bucket is choosing a vertex that gives a more balanced partition. Based on these ideas, Krishnamurthy [Kri84] proposes a look ahead strategy for calculating different levels of vertex gains. Following the above definitions, he defines the number of *free* and *locked* vertices for each hyperedge $e \in E$ in a part. For part A, we have:

$$\phi_A(e) = |\{v \mid v \in A \text{ and } v \in e \text{ and } v \text{ is free}\}|$$
$$\chi_A(e) = |\{v \mid v \in A \text{ and } v \in e \text{ and } v \text{ is locked}\}|$$

and the binding number of the hyperedge e to part A is calculated as follows:

$$\beta_A(e) = \begin{cases} \phi_A(e) & \text{if } \chi_A(e) = 0 \\ \infty & \text{if } \chi_A(e) > 0 \end{cases}$$

The binding number shows how tightly a hyperedge is tied to a part. If the number of locked vertices for a hyperedge becomes greater than one, the hyperedge will be tied with that part for the rest of the pass. A hyperedge that gets locked vertices in both parts, is tied with both and it is impossible to take it out of the cut for the rest of the pass. Finally, level i gain of a vertex in part A is calculated as follow:

$$g_i(v) = \sum_{\substack{v \in e \\ \beta_A(e)=i \\ \beta_B(e)>0}} \lambda(e) - \sum_{\substack{v \in e \\ \beta_A(e)>0 \\ \beta_B(e)=i-1}} \lambda(e)$$

Having l gain levels, vertex gains can be represented as a tuple $\langle g_1, g_2, \cdots, g_l \rangle$. Ties are broken based on first, second, ..., l^{th} level gains during vertex moves. The

algorithm increases the complexity of the original FM algorithm by $O(l \cdot pins(H))$.

The FM algorithm, described above, calculates a bipartitioning of the hypergraph. Sanchis [San89] extends the algorithm to direct *k-way* FM algorithm. She uses $k(k-1)$ gain buckets in her algorithm, each for storing gains of vertices from part i to part j, $1 \leqslant i \neq j \leqslant k$. Accordingly, the complexity of the algorithm increases to $O(l \cdot pins(H) \cdot k(\log k + d_{\max} \cdot l))$. She found that using higher gain levels gives better results for larger number of parts. This algorithm is referred as the *k-way* FM algorithm or the K-FM algorithm in the rest of the thesis.

The algorithms proposed above (KL, FM, and K-FM algorithms) are known as the *basis* or *original* iterative move-based algorithms. Most of the other iterative move-based algorithms are developed from the original algorithms and referred as the *modifications* algorithms. As we have mentioned earlier, iterative move-based algorithms are local optimisations problems and the common drawback of these types of algorithms is that they are more likely to get stuck in local minima. Consequently, some of the modification algorithms propose strategies to prevent the local minima solutions in order to generate better partitioning solutions. Due to the diversity of the modification algorithms, we review only few of them and we refer to [Fjä98,AK95,Len90,Tri06] for further reading.

Modifications of FM/KL algorithm

In the FM algorithm introduced above, the algorithm stops when it can not make any further improvement; this is when all the remained vertices have negative gains. For hill climbing purposes and to get out of the local minima, some algorithms are proposed that allow a specified number of moves with negative gains [Kar02]. This strategy allows FM to makes a predefined number of negative moves, then it stops if FM can not make any improvement after those moves. When the algorithm stops, it looks back to the move history and the point that gives the best cut is calculated. In the end, it finalises all vertex moves upto the point and other vertex moves are reversed.

Cong et al. [CLL+97] propose an algorithm that is called Loose/Stable Net Removal. In their modified version of FM algorithm, each hyperedge follows *free* $\longrightarrow$

loose $\longrightarrow$ *locked* state transition. A net is free when all of its vertices are free. Then it becomes loose as soon as one of its vertices is locked to a part and has free vertices in the other part. Finally, the hyperedge is locked if it has atleast one locked vertex in each part. When the state is loose, the part in which the hyperedge has locked vertices is called the *anchor* and the other part, that contains some free vertices that belong to the hyperedge, is called the *tail*. To motivate a hyperedge in the loose state to free itself from the cut, we can intentionally drive the gain of the free vertices in the tail part; for example, their gain values can be increased. When the gain of a vertex is increased, the probability of selecting the vertex in subsequent moves is increased accordingly. This strategy gets a loose hyperedge out of the cut and helps the hyperedge to slide away from the cut.

Based on results reported by Shibuya et al. [SNK95], in which the authors report that 80% of hyperedges in the final cutset are stable and these locked hyperedges are the main reason for the FM algorithm to get stuck in local minima, Cong et al. [CLL+97] provide a hill climbing strategy called *Stable Net Transition* to give a chance to stable nets to get out of the cutset with the hope of getting better quality. They apply a multiple run FM algorithm and the stable nets are detected from the first run onward. Then some of them are chosen and all of their vertices are moved to the part with lower weight[4]. Vertices are allowed to move only once. The next run of FM is started using the output of the stable net removal. The evaluation of their partitioning algorithm combined with a net clustering algorithm to improve quality and speed of the partitioning process shows improvements to the original FM algorithm.

Cong et al. [CL98] provide an extended version of *k-way* FM algorithm. In their algorithm, parts are matched in pair, using different strategies, then a 2-way FM algorithm is run on the pairs. Among all part pairing strategies, the gain based strategy that pairs two parts, for which the cut size reduction is maximum during previous passes of the algorithm, gives better partitioning results. Evaluation shows up to 86.2% improvement on the *k-way* FM algorithm.

[4]They chose a predefined percentage of the hyperedges to be moved.

In the context of VLSI circuit partitioning, there are terminals such as I/O chips that are fixed and can not be moved; therefore, partitioning these circuits with a hypergraph partitioning algorithm contains some vertices that are fixed to the parts and can not be moved during optimisation process. Problems of this type are easier and takes less running time. These kind of partitioning are considered by some algorithms such as Alpert et al. [ACKM00], Çatalyürek and Aykanat [ÇA11], Aykanat et al. [ACU08].

Other Move-Based Algorithms

In addition to the iterative move-based algorithms, there are some other move-based algorithms that try to solve the hypergraph bipartitioning problem. An issue with these algorithms is that they should be chosen carefully with extensive study when applied to the partitioning problem because they do not exactly model the hypergraph partitioning problem.

Simulated Annealing (SA) is one of those algorithms that has been applied to graph bipartitioning such as the algorithm proposed by Johnson et al. [JAMS89]. Their algorithm does not exactly model the bipartitioning problem because it does not put any restriction on the part sizes. Consequently, they provided a customisation of the algorithm. An issue with SA-based is that the time taken by these algorithms is very high and it is not always affordable in practical applications. In addition, the partitioning quality is not always better than those generated by iterative move-based algorithms, which are popular in practical applications and they known for generating good partitioning quality in fastest time among other move-based algorithms. These two drawbacks limit their applicability and their usefulness is application-dependant [Len90].

Another set of algorithms are those model the bipartitioning problem with the maxflow-mincut problem. An example is the algorithm proposed by Bui et al. [BCLS87]. They put two distinguished vertices, one in each side, and bipartitioning is obtained by calculating max-flow between these two vertices. Then they calculate a bipartitioning by calculating max-flow between every pair of vertices. In their formulation, they do not put any restriction on the size of the parts and it cannot be

directly applied to the most partitioning problems in which the balance constraint is a necessity. For some special types of uniformly distributed random graphs, the algorithm can find optimum bisection with high probability but the probability distribution for generating the random graphs should be chosen carefully [Len90]. Their applicability in practice is limited because it is difficult to guarantee a certain probability distribution on the real graphs and, when it is found, it is more likely non-uniform. Another attribute of the algorithm is that it can only generate very good bipartitioning on graphs with small average vertex degrees [BCLS87]. When the average vertex degree of graph increases, simple heuristics generate very good results in less time.

Tabu search is another set of algorithms that keep the history of most recent moves of the iterative move-based algorithms to find a feasible solution in the problem space. The number of moves to be saved as history is given as an positive integer at the start of the partitioning. Examples of this type are [AV00,AV03]. The proposed algorithms mostly lack the comparison with the state-of-the-art algorithms in the field and one can not decide about their performance [Tri06].

Considering all of the above-mentioned facts, these algorithms including SA, maxflow-mincut, and Tabu search may give very good and close to optimal partitioning solutions. The main issue is their applicability and their dependency on the problem under investigation such that the effort needed to investigate those is better to be spent on the partitioning problem itself. As mentioned earlier, iterative move-based algorithms are preferred in practical applications and they have been already implemented in all software tools for graph and hypergraph partitioning. A big advantage of these algorithms is that they give very good partitioning quality in the fastest time compared to other move-based algorithms [Alp96,Tri06,Len90]. We refer to Trifunovic [Tri06] and Lengauer [Len90] for further reading and more related work on move-based algorithms.

3.1.2 Multi-level Hypergraph Partitioning

The main weakness of the move-based algorithms is their *unpredictability*. The solution found is a locally optimum solution. Whether the solution is also globally

optimum depends mostly on two factors: the density of the hypergraph and the initial distribution of the hypergraph among the parts. As mentioned earlier, the FM algorithm is usually run multiple times in practical applications; each time with a different assignment of vertices to the parts. Then the best partitioning among all runs is selected as the final partitioning result. The strategy tries to increase the chance of catching the global minimum solution by increasing the number of runs. However, this approach is too restrictive when the size of hypergraph is large. The reason is that the probability of a local minimum to be also a global minimum is decreasing as the size of hypergraph increases. In this situation, we need more runs of the algorithm. The number of required runs increases as the size of the hypergraph grows such that obtaining a globally optimum solution is almost impossible for very large hypergraphs.

Alpert [AK95] shows that the partitioning cost obtained for a locally minimum solution is only of the average quality of the globally minimum solution. Goldberg and Burstein [GB83] show that getting stuck in local minima only happens for sparse hypergraphs in which the average vertex degree of the vertices is low[5]. On the other hand, Lengauer [Len90] reports that it is more likely to achieve globally minimum solution on dense hypergraphs with large minimum vertex degree. As a result, the quality of algorithms is directly dependant on the hypergraph density. Saab and Rao [SR92a] show that the performance of the the KL algorithm improves as the graph density increases and they give a performance bound for 1-optimal heuristics[6] that becomes tighter as the number of edges in the graph increases.

The above discussion motivates using an algorithm in order to increase the hypergraph density for achieving a better partitioning quality. One solution is using net clustering techniques with move-based algorithms. The approach yields to advantages: increasing the hypergraph density and decreasing the problem size [HB97,HK92]. One of those techniques, which is also called two-phase approach, is to run move-based algorithm on the clustered hypergraph and use its output

[5]They claim that the average vertex degree of real VLSI circuits is between 1.8 and 2.5 that is considered as low vertex degree and the FM algorithm gives poor quality.

[6]A bipartitioning is called *r-optimal* if exchanging r modules between the two parts does not decrease the partitioning cost; therefore the KL algorithm is 1-optimal.

as the initial distribution of the second run that is executed on the original (non-clustered) hypergraph. Net clustering was a starting idea for introducing multi-level or hierarchical hypergraph partitioning algorithms.

Multi-level algorithms composed of three phases. First, it generates a sequence of hypergraphs each approximating the original hypergraph. The size of hypergraph is decreasing after each approximation. This phase is called the *coarsening* phase. It generates a sequence $H_i = (V_i, E_i), 0 \leqslant i \leqslant c$ such that $H = H_0$ and $|V_i| < |V_j|$ whenever $i > j$. The coarsening stops when the coarsest hypergraph H_c contains only few vertices (for example fewer than 100 vertices). The process continues through the second phase, which is called the *initial partitioning* phase. This phase calculates a partitioning of H_c using a randomised algorithm or one of move-base algorithms such as KL and FM. The final phase, which is called the *uncoarsening* phase, projects back the partitioning on H_c to the original hypergraph H by going back through the same coarsening levels that is $H_c \mapsto H_{c-1} \mapsto \cdots \mapsto H_1 \mapsto H_0$. In each level, a refinement of the partitioning is usually performed. The refinement process tries to move the vertices between the partitioning boundaries in order to further reduce the cost function. Consequently, the third phase is also referred as the *refinement* phase in literature. The multi-level bipartitioning process and its three phases are depicted in Fig. 3.2. The two-phase clustering algorithm described above is considered as one-level approximation. One-level approximation algorithms were first used on graphs and applied by Bui et al. [BHJL89]. Later on, the multi-level approximation graph algorithms are proposed such as the work by Hendrickson and Leland [HL95].

Karypis and Kumar [KK98a] show that a good partitioning of the coarsest hypergraph also yields a good partitioning of the original hypergraph, hence we need less effort during the refinement phase. This makes a multi-level algorithms a sustainable approach. We go through the details of each phase in the rest of this section.

Coarsening

Coarsening is described as the most important phase of the multi-level paradigm. Karypis [Kar02] provides two key requirements for the coarsening phase. For two

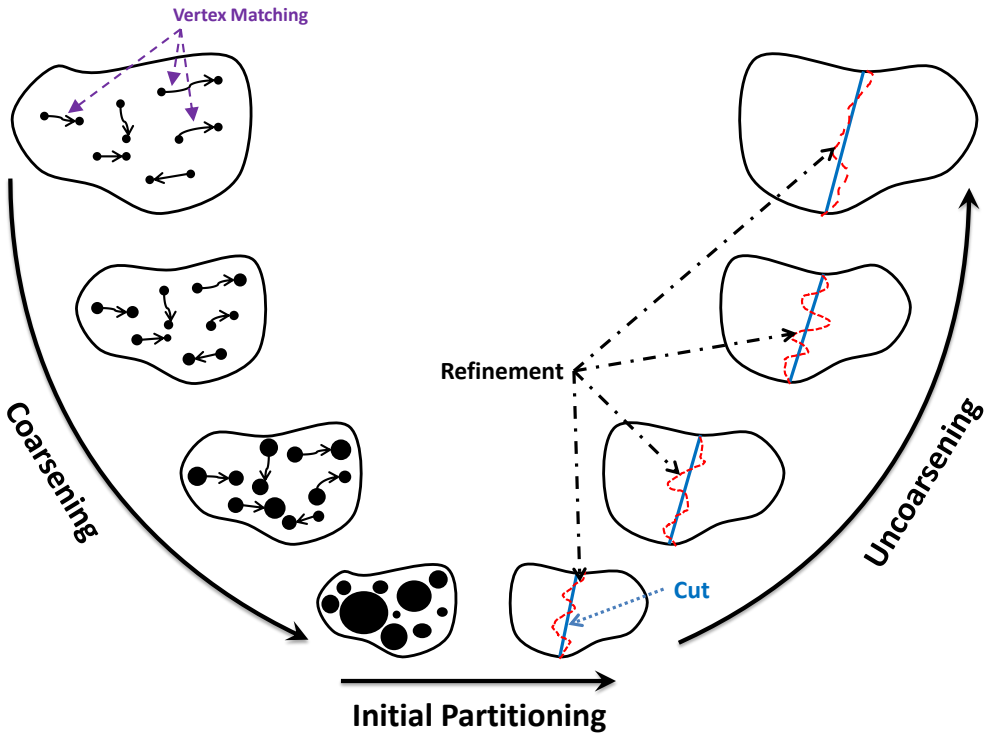


Figure 3.2: The multi-level hypergrapph bipartitioning paradigm and its three phases: coarsening, initial partitioning, and uncoarsening.

successive levels i and $i-1$, $1 \leqslant i \leqslant c$, the coarsening should have the following specifications:

1. Any partitioning on the coarser hypergraph H_i should be easily projected back to the finer hypergraph H_{i-1}.

2. The cost of the projected back partitioning on H_i is less than or equal to the cost of the partitioning on H_{i-1}.

He describes the second case as a necessity of the refinement process to be meaningful such that it decreases the cost progressively. The coarsening is done by vertex clustering; it finds clusters of vertices, matches vertices inside each cluster, and merges the matched vertices together to form coarser vertices in the coarser hypergraph. Clusters should not have any overlap and their union is equal to V. For example, when two vertices v, u are matched and form a cluster, the weight of the coarser vertex w is the sum of the weight of v and u that is $\omega(w) = \omega(v) + \omega(u)$.

Furthermore, the hyperedge set incident on w is the union of the hyperedges incident on v and u. Vertices that do not find any match, which may happen for some of vertices and depending on the coarsening approach, are simply copied to the next level without any change. When we move on to the next level of coarsening, the following are usually done to the hyperedge set:

1. Unit size hyperedges in the coarser hypergraph are removed because they do not participate for calculating the partitioning cost.

2. Identical hyperedges, which are hyperedges incident on the same vertex set, are identified; only one of them is kept in the coarser hypergraph and the others are removed. The weight of the kept hyperedge is set to the sum of the weight of all identical hyperedges.

As the coarsening proceeds, the average size of the hyperedges decreases but the average vertex degree increases as we go from one level of coarsening to the next level. Karypis [Kar02] further concludes that the coarser hypergraph should have fewer hyperedges than the original hypergraph and hyperedge weights should decrease quickly. These conditions are met when we remove unit size and identical hyperedges while coarsening. In the rest of this section we review some of the available coarsening methods.

One of the coarsening approaches is called *Edge Coarsening (EC)* in which pair-matches of vertices are found according to the maximal matching problem. In this approach, vertices are visited in random order. For each vertex, the algorithm visits all unmatched adjacent vertices of the vertex and the one is chosen as a pair-match that has the strongest connectivity with the vertex (maximal matching). In this approach, the hyperedges are implicitly treated as graph edges [Len90]. Karypis [Kar02] mentioned that this scheme can be further improved by giving priority to the smaller hyperedges with bigger weights among others. Giving priority to these hyperedges removes them from the hypergraph in the first few coarsening levels and leaves the coarsened hypergraph with fewer hyperedges. In addition, the average weight of the hyperedges decreases quickly.

In another strategy, the number of hyperedges and their average weight decrement is sped up by matching groups of vertices. The algorithm does not limit itself to pair-matching. In *Hyperedge Coarsening (HC)*, a set of independent hyperedges[7] is calculated. Then all vertices in each hyperedge are matched together to form a coarser vertex in the coarser hypergraph (multiple matching). Since hyperedges in this set do not have any vertex in common, there is no matching conflict. In order to choose this set, hyperedges are sorted in a list in decreasing order of their weights. The sorting algorithm breaks the ties by putting hyperedges with larger size in the list first. The list is traversed from the beginning and when the algorithm sees a hyperedge that none of its vertices are matched, all of its vertices are matched together. Although this approach speeds up the size decrement of the coarsened hypergraph, it is not guaranteed. For example, in hypergraphs with high connectivity (with large number of strongly connected components) most of the hyperedges are overlapped and one can find very few non-overlapped hyperedges. There are two drawbacks regarding this approach [Kar02]:

1. The size of some (or many) hyperedges do not change sufficiently (this causes starvation). This makes the refinement process difficult with very little space for optimisation.

2. As we proceed with the coarsening, the standard deviation of the vertex weights increases. This changes the structure of the hypergraph, affects the quality, and makes it hard to maintain the balance constraint.

Karypis [Kar02] improves the HC algorithm by traversing the hyperedge list twice. In the second traversal, if the algorithm sees a hyperedge with unmatched vertices, all of these vertices are matched.

The aim of the coarsening phase is to identify naturally existing clusters of vertices in the hypergraph. The independence (or the maximality) requirement exists in both EC and HC schemes, destroys these clusters and leads to less ideal coarse representation of the original hypergraph as the coarsening proceeds [Kar02]. In

[7]Independent hyperedges are those that do not have any vertex in common.

order to resolve this issue, the *First Choice (FC)* algorithm processes all adjacent matched and unmatched vertices when tries to find a match for a given vertex. The vertex is matched with the one that gives the highest connectivity. It means that the FC algorithm is not limited to pair-matches. When there are more than two adjacent vertices with the same connectivity, ties are broken by giving higher priority to unmatched vertices. Again, there are two risks. First, the reduction in the number of vertices from one coarsening step to another might be too high; this might not preserve the structure of the hypergraph. Second, it may form vertices with large differences among their weights (large vertex weight standard deviation) as we proceed to the coarsening[8].

Most of other coarsening algorithms in the literature propose a local measure of connectivity between the vertices of the hypergraph and match vertices according to this measure. Vertices are put in a list (in random order or sorted according to parameters such as vertex degrees). The list is traversed from the beginning, a vertex is selected, and a match is found according to the vertex connectivity metric. We review some of the connectivity metrics proposed in the literature. Alpert et al. [AHK98] propose a controlled vertex connectivity metric for two vertices v, u as:

$$connectivity(v,u) = \frac{1}{\omega(v)\omega(u)} \sum_{e \rhd v, e \rhd u, \forall e \in E} \frac{1}{|e|}$$

where dividing by the vertex weights discourages building large clusters. Caldwell et al. [AAI06] propose a vertex connectivity based on hyperedge bandwidth (denoted as a function $b(\cdot)$). The bandwidth is equal to two for hyperedges of size two, and one for others (> 2). The connectivity metric is defined as follows:

$$connectivity(u,v) = \frac{1}{\omega(v) + \omega(u)} \sum_{e \rhd v, e \rhd u, \forall e \in E} \frac{1}{b(e)}$$

[8]Having vertices with large differences among their weights makes it difficult to maintain the balance constraint. This is a major issue especially in the multi-constraint partitioning problems.

Catalyurek and Aykanat [ÇA99] defines the connectivity between a vertex and a cluster of vertices C as

$$connectivity(v, C) = \frac{|\{e \in E \mid v \in e, \forall u \in C \ s.t. \ u \in e\}|}{\omega(v) + \omega(C)}$$

where $\omega(C)$ is the weight of the cluster which is calculated as the sum of the weights of vertices in the cluster.

The above algorithms only define a metric of connectivity between the vertices and their performance highly depends on the structure of the hypergraph under investigation. Calculating vertex connectivity requires searching in the hypergraph adjacency matrix. As mentioned in Chapter 2 in Eq. 2.3, the adjacency matrix of a hypergraph might be very dense despite having a sparse incidence matrix. This is a case in scientific applications. This characteristic makes it difficult to define a good metric of connectivity between vertices. This subject is further investigated in Chapter 4 while proposing our serial hypergraph partitioner.

Finally, an advantage of the multi-level approach is that it provides a trade-off between the quality and the speedup. The more coarsening levels get us better partitioning quality but the algorithm runs slower with more memory consumption. On the other hand, decreasing the coarsening levels gets better runtime. Karypis [Kar02] defines the **compression ratio** between two levels of coarsening in a multi-level approach with c levels as follows:

$$r = \frac{|V_i|}{|V_{i+1}|}, \forall i, 0 \leqslant i < c \tag{3.1}$$

There are two things that should be taken into account. First, if we do not go through enough coarsening levels, the algorithm ends up with a big hypergraph in which it is not possible to find a good partitioning compared to the partitioning calculated on the original hypergraph. The performance may also decrease because we spend some time going through more coarsening levels without making enough improvement to the quality. Second, having lots of coarsening levels and obtaining very small coarsest hypergraph leaves us with few feasible solutions which may not result in a good partitioning quality. Karypis [Kar02] argues that a good trade-off

can be achieved by limiting the compression ratio between $1.5 \leqslant r \leqslant 1.8$. In the algorithm proposed by Devine et al. [DBH+06], the partitioning stops when the coarsest hypergraph has fewer than 100 vertices.

Initial Partitioning

The initial partitioning phase provides a partitioning on the coarsest hypergraph. The most common algorithms used for this purpose are move-based algorithms. The proposed heuristic algorithms usually execute multiple runs of different algorithms and the one that gives the best cut and meets the balance constraint is selected among them to be projected back to the original hypergraph. We should note that the size of H_c is very small compared to the original hypergraph and its partitioning can be calculated very fast in much less time. Some of the approaches are as follows:

1. **Random assignment:** Randomly assigns vertices to the parts.

2. **Linear assignment:** Linearly assigns vertices to the parts. It defines two counters: one for vertices c_V and for parts c_π. c_V is initialised randomly between zero and $|V-1|$ and $c_\pi = 0$. Vertex list is traversed starting from c_V'th vertex and assigned to p_π'th part. After each assignment, counters are updated as $c_V = ((c_V + 1) \mod |V|)$ and $c_\pi = ((c_\pi + 1) \mod k)$.

3. **Breadth-First and Depth-First assignment:** A vertex is selected randomly and the hypergraph is traversed by either breadth-first or depth-first algorithms and adjacent vertices are assigned to the same part.

4. **FM based assignment:** In bipartitioning, the approach selects a vertex randomly and assigns it to part 1 and all other vertices to part 0. Then the FM algorithm is run and the bipartitioning is calculated. In direct *k-way* partitioning, $k-1$ vertices are selected, one for each ith part $1 \leqslant i \leqslant k-1$ and all other vertices are assigned to part 0. Then a direct *k-way* FM algorithm calculates a *k-way* partitioning.

These approaches are used in algorithms and tools such as [DBH+06,AAI06,Kar07,ÇA11,RBT+13,San14b,Kar02,HB97]. Karypis et al.

[KAKS99] suggest that the algorithm can project back not only one but also some of partitions calculated in the initial partitioning phase to the original hypergraph. This approach is not memory efficient and is computationally expensive.

Uncoarsening and Refinement

The purpose of the uncoarsening phase is to project back the partitioning calculated on the coarsest hypergraph H_c to the original hypergraph by going through c levels. At each level, the partitioning cost is further refined as much as possible to improve the quality; therefore, this phase is also called the refinement phase. Another important specification of the refinement phase is to keep the balance constraint. Enforcing the balance constraint limits the movement of the vertices between partition boundaries especially in early uncoarsening levels. In these levels, the algorithm deals with clusters of vertices and the average vertex weights are higher such that moving one vertex could violate the balance constraint by a large percentage.

A common and popular refinement algorithm is FM algorithm [Kar07,ÇA11,RBT+13,San14b]. In each uncoarsening level, multiple passes of FM are usually applied. In each pass, FM is run on the hypergraph and the output of one pass is used as the initial partitioning of the next pass. It tries to find subsets of vertices that their movement improves the partitioning quality.

Karypis et al. [KAKS99] argue that most of the cut improvements can be achieved during the first and the second pass and the forthcoming passes improves the partitioning quality only by a small percentage. This fact can be used to limit the number of passes of FM algorithm in order to achieve a better runtime. They have also found that only a few percentage of vertices contribute to the cut reduction in the multi-level paradigm. In the original version of FM algorithm, the passes stop as soon as the algorithm can not make any further improvement to the cut. In their version of FM algorithm, moving vertices with negative gain is allowed in order to help the algorithm gets out of the local minima. It is observed that the hill-climbing after a large number of moves with negative gain is unlikely. Therefore, they stop the pass after a predefined number of moves have been made that do not improve the cost function. This strategy improves the runtime as well as the partitioning quality,

although the improvement in runtime is much greater. The prescribed number of moves is defined to be between 1% and 5% of the total number of vertices. This algorithm is known as *Early Exit* FM algorithm and denoted as *FM-EE*.

Karypis [Kar02] discusses that the hill climbing capability of FM algorithm is less important in multi-level paradigm. In the flat FM algorithm, the cost function is increasing when a cluster starts to move over the partition boundary. The cost decreases again when the whole cluster moves over. In the multi-level paradigm, clusters are integrated into vertices as we coarsen the hypergraph; therefore, this effect (increase and decrease in the cost function) does not happen. This means that the hill-climbing property becomes less important [9]. Considering this fact, the performance of FM can be improved by removing hill climbing property, which is removing the priority queues from FM algorithm. In this situation, vertices are traversed in random order and they are moved if they give a positive gain. The order is not important because vertices with large positive gains will be moved eventually at some point. This can provide much greater runtime improvement especially for direct *k-way* FM algorithm because the complexity of the algorithm is not dependant on the number of parts. This algorithm is known as *greedy refinement* and it is applied in *hMetis* hypergraph partitioning tool [Kar07].

Another variation of FM is known as the *Boundary FM (BFM)* algorithm [ÇA99,ÇA11,San14b]. This algorithm categorises vertices as *boundary* and *non-boundary* vertices. A vertex is called boundary if it is incident on at least one cut hyperedge, otherwise it is called non-boundary. BFM adds only boundary vertices to the gain bucket data structure; that is, only boundary vertices are moved. A non-boundary vertex may become boundary if the state of its adjacent vertices changes during the pass. Then, it is added to the gain bucket as soon as it becomes boundary and will be considered to be moved.

A single call of the multi-level algorithm is called a *V-cycle* (capital 'V'). In a multi-phase refinement algorithm, successive calls of the algorithm might be made

[9] In our opinion this is true if the coarser vertices represent natural clusters in the original hypergraph. Its validity decreases as the coarsening algorithm does not capture these natural clusters (as it happens in edge coarsening and hyperedge coarsening algorithms).

while projecting back the results. The call can be made from any intermediate level of the refinement. Each intermediate call is is a *v-cycle*[10] (small 'v'). Calls to v-cycle can stop if the last call did not provide any improvement to the partitioning cut or a specified number of calls is made. The use of v-cycle is usually avoided in parallel hypergraph partitioning algorithms because it limits the performance of the algorithm and imposes two much overhead [Tri06].

3.1.3 Recursive Bipartitioning vs Direct k-way Partitioning

A *k-way* partitioning on the hypergraph can be calculated recursively or directly. In recursive bipartitioning, the algorithm generates a bipartitioning of the original hypergraph. Then it is recursively applied to both parts independently. The process continues until we achieve k partitions. In direct partitioning, the algorithm calculates k partitions by directly working on the hypergraph. An example of *6-way* partitioning using both methods is given in Fig. 3.3. In recursive bipartitioning, the algorithm generates two equally sized parts at the first level. The part sizes are adjusted for each level of recursion. The hypergraph is *6-way* partitioned with three levels of recursion or $\lceil \log_2 6 \rceil$.

There is a debate in the literature on which paradigm gives better partitioning. Karypis [Kar02] reports some advantages of direct *k-way* partitioning over recursive algorithms. First, the recursive algorithms do not allow direct optimisation of the cost function such that it needs to know how hyperedges are cut among all k parts. Second, direct algorithms impose stricter balance constraints while trying to optimise the cost function, which is not possible in recursive bipartitioning algorithms. This is important when the partitioning algorithm is multi-constraint. Finally, the quality achieved by direct algorithms can be much better. Simon and Teng [ST97] report that the recursive bipartitioning almost always generates solutions within constant factor of the optimal solutions for well-shaped finite element/difference meshes. In

[10]The difference between v-cycle and V-Cycle is that the latter is the whole multi-level cycle that is $H_0 \mapsto \cdots \mapsto H_c \mapsto \cdots \mapsto H_0$, while the first can be called multiple times from any uncoarsening level; for example, when it is called from uncoarsening level $0 \leq i < c$, the algorithm goes through $H_i \mapsto \cdots \mapsto H_c \mapsto \cdots \mapsto H_i$ levels.

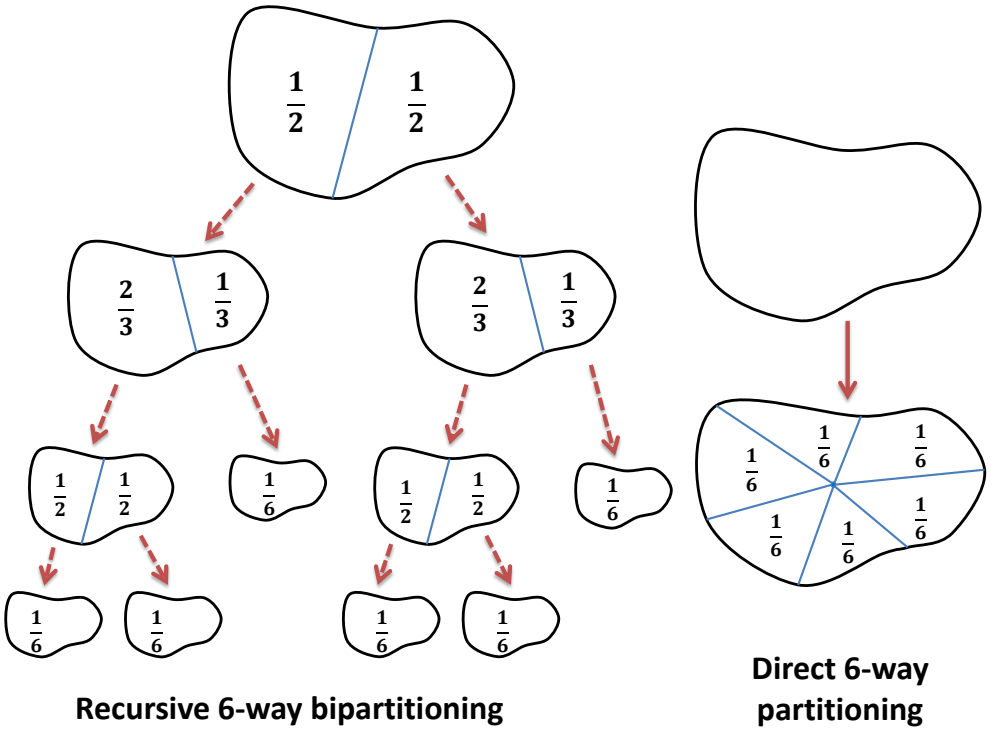


Figure 3.3: Recursive 6-way bipartitioning of the hypergraph vs direct 6-way partitioning

situations that the balance constraint is bounded by $2|V|/k$, the solution is a factor of $O(\log p)$ worse than the optimal solution.

Others report the superiority of recursive bipartitioning algorithms over direct algorithms. Cong et al. [CL98] report that the quality of direct algorithms are worse because they are most likely to get stuck in local minima. They propose a modification of the K-FM algorithm. They improve the K-FM by pairwise matching of parts and run 2-way FM on them. Their method improves K-FM by 86.2% and FM by 17.3% on ISPD-98 [Alp98] and MCNC [Yan88] benchmarks. Karypis and Kumar [KK00] show that recursive algorithms tend to be better than direct algorithms because they have more relaxed balance constraint and do not get trapped in local minima as easily as direct *k-way* FM algorithm. They propose a variation of direct multi-level FM algorithm which has qualities competitive with recursive FM and has lower execution time.

Wang et al. [WLCS00] compare two categories of algorithms on graphs. First, algorithms that obtain *k-way* partitioning through all-way bipartitioning. Algorithms

in this category start with an initial *k-way* partitioning. Then they select pair-wise partitions and improve the cut between them. Second category is the recursive bipartitioning algorithms. Assume that the optimal cost of a partitioning on a given hypergraph is $Cost_{opt}$. They use $\delta-$approximation[11] bipartition heuristics. They argue that the recursive bipartitioning algorithms have an upper bound of $\delta Cost_{opt} \log_2 k$ while the first category has an upper bound of $\delta k Cost_{opt}$. Furthermore, the second approach obtains better partitioning quality and it is much faster. They show that the partitioning cost at each level of recursive bipartitioning is no more than δ times worse than the cost of optimal solution at that level. Generally speaking, the cost of overall algorithm is no more than $\delta Cost_{opt} \log_2 k$ for $\log(k)$ recursions.

Aykanat et al. [ACU08] investigate the comparison between direct *k-way* and recursive algorithms when the hypergraph partitioning is multi-constraint with fixed vertices. The results are reported for the *PaToH* hypergraph partitioning tool [ÇA11]. The algorithms are denoted as *kPaToH* (direct *k-way*) and *PaToH* (recursive bipartitioning). They found that *kPaToH* gets better quality and runs faster compared to *PaToH* when k is large. The algorithms are evaluated on some hypergraphs in the University of Florida Sparse Matrix Collection [DH11] benchmark. For $k = 32$ and $k = 256$, *kPaToH* gives 4.82% and 6.81% better partitioning cuts, respectively. The runtime is 1.7 times better on average. They show that the two most time consuming operation in *PaToH* are matching and hypergraph construction at each recursion, while the most time consuming operation of *kPaToH* is the uncoarsening phase. The uncoarsening runtime is getting worse when k increases. An exception happens when the average net size is low; in this case the increase in uncoarsening time is smooth. When considering fixed vertices, they report better average cut size 5.82% in single constraint case, 20.98% in two–constraint case, and 40.02% in four–constraint case for *kPaToH*. In addition, *kPaToH* gives better partitioning quality for partitioning with fixed vertices when k decreases while the number of fixed vertices increases.

[11] Their analysis is not based on a specific algorithm; therefore, they refer to the bipartitioning algorithm as $\delta-$approximation and it means that the quality of partitioning generated by the algorithm is no more that δ worse than the optimal solution.

The question of which partitioning method is better is not completely known. Their efficiency depends on the structure of the hypergraph and the problem under investigation (for example, partitioning with fixed vertices or not). This is one of the fields of the hypergraph partitioning which needs further investigation and research. In practice, the recursive bipartitioning methodology tied with multi-level paradigm is known to generate good partitioning qualities and gives reasonable performance [Kar07,ÇA11,San14b,RBT+13,AAI06].

3.1.4 Serial and Parallel Partitioning Algorithms

The ever-increasing size of graphs and hypergraphs makes it impossible to fit them into the memory of a standalone computer or process them with the existing computational power of one computer. For example, graphs and hypergraphs representing social networks such as Facebook and Twitter have billions of vertices and edges [HC14] and needs terabytes of data to be saved. Bradley et al. [JNWH04] have investigated the use of the hypergraph partitioning in iterative Laplace transform inversion algorithm for analysing the response time of queueing systems. A hypergraph is used for modelling sparse matrix decomposition. The authors report that the performance of the application is limited by serial hypergraph partitioning algorithms and they need a parallel algorithm to be able to process some practical models.

Consequently, we need parallel and scalable hypergraph partitioning algorithms. There are two objectives for designing a parallel algorithm as follows:

1. The parallel algorithm should be designed in a way to generate partitioning quality comparable to serial algorithms. No parallel algorithm can generate partitioning quality better than serial partitioning algorithms. The reason is related to the data locality issue in distributed systems. The input hypergraph is distributed among a set of processors. Each processor knows about a portion of the problem and makes some decisions locally. This deteriorates the partitioning quality.

2. The parallel algorithm should be scalable in term of computations and memory usage. This is also not an easy task because the scalability of the proposed

parallel partitioning algorithm not only depends on the parallel algorithm itself, but also the structure of the input hypergraph. An algorithm might be scalable and work very well for some hypergraphs, but it might get bad scalability on the others.

In this section we focus on parallelisation of multi-level algorithms and the challenges on the way. In multi-level paradigm, the two phases that are difficult to parallelise are the coarsening and uncoarsening phases. The initial partitioning phase is not a problem because the coarsest hypergraph is small enough and can be processed on one computer very quickly. The performance of these two phases (coarsening and uncoarsening) depends on the way we distribute the input hypergraph on the processors. Bad distribution can generate high network traffic, limit the scalability, and interfere with vertex matching decisions. In the following, the palalellisation of these three phases are discussed.

Hypergraph Distribution

An important decision in parallel hypergraph partitioning algorithms is how to initially distribute the input hypergraph among processors in order to increase data locality. There are mainly two strategies for this purpose which are encouraged by parallel graph partitioning algorithms. We refer to them as one-Dimensional (1D) and two-Dimensional (2D) initial hypergraph distributions. The input hypergraph is represented as $H(V, E)$, the number of processors is p, and the number of partitions is denoted as k (or *k-way* partitioning of H on p processors).

The 1D distribution is the natural distribution of vertices on processors and it is applied in *Parkway* parallel hypergraph partitioning tool [TK08]. The distribution is previously used in parallel graph partitioning algorithms such as the work by Karypis and Kumar [KK97]. In this configuration, each processor stores $\frac{|V|}{p}$ vertices and $\frac{|E|}{p}$ hyperedges. Vertices are assigned to processors in lexical order such that the first portion is assigned to the first processor, the second portion to the second processor, and so on. In this method, there might be some hyperedges whose vertices are stored on different processors. *Parkway* calls these hyperedges *frontier* hyperedges. the 1D distribution does not guarantee that hyperedges incident on a given vertex will be on

the same processor who owns the vertex. Even if there is no frontier hyperedge, there is no guarantee that all hyperedges incident on locally owned vertices by processors exist on the same processor. While all hyperedges incident on a given vertex are needed in order to process the vertex in the coarsening and refinement phases, incident hyperedges are collected using an all-to-all communication in the beginning of each coarsening level. The communication replicates the frontier hyperedges on the processors. Replicated hyperedges are deleted at the end of current coarsening level in order to save memory.

For the purpose of fast hyperedge comparison, *Parkway* uses hash functions. Each hyperedge is hashed to an integer value based on the vertices contained in the hyperedge. When the algorithm wants to compare two hyperedges (in order to check if they are identical), the hash values are compared. Collisions may occur; the probability of collisions using a $64-$bit hash-keys for $|E| = 10^8$ is reported to be less than 0.0003. In case of collision, the full content of hyperedges are compared.

The second distribution, the 2D distribution, is originally inspired by two-dimensional graph to processor assignment similar to distribution of graph adjacency matrix on the processor set by Grama [Gra03]. This distribution is employed by *Zoltan* parallel hypergraph partitioner [DBH+06]. In this method, the processor set is logically arranged in a grid $p = p_x \times p_y$ in which p_x and p_y are the number of processors in rows and columns, respectively. The vertex set is distributed on p_x processors each holding $\frac{|V|}{p_x}$ vertices. The hyperedge set is similarly distributed on p_y processors. This provides a Cartesian distribution of the hypergraph on the processor set such that each processor stores a subblock of the hypergraph. In this distribution, only p_x or p_y processors need to contribute in collective communications. The authors suggest that having $p_x \approx O\left(\sqrt{p}\right)$ gives the highest performance.

The authors show that the 2D distribution fits the hypergraph partitioning context rather than than graph partitioning. In the parallel graph partitioning algorithm proposed by Karypis and Kumar algorithm [KK98c], the processor set is arranged in a $\sqrt{p} \times \sqrt{p}$ matrix and the vertex set is distributed into $\sqrt{p}$ subsets and assigned to processors in a cyclic mapping. Furthermore, the adjacency matrix is distributed among all p processors such that processor p_{ij} (on row i and column j)

stores the edge set between vertices in i^{th} and j^{th} vertex set. The matching decision is made by the $\sqrt{p}$ diagonal processors; this creates a bottleneck and restricts the scalability of the algorithm to $\sqrt{p}$. For this reason, authors do not recommend 2D distribution for parallel graph partitioning.

The evaluations report better scalability of *Zoltan* compared to 1D distribution. In addition, analysing the algorithm shows that the *natural way* of distributing vertices among processors and keep the full connectivity of vertices on all processors ($p_x = p, p_y = 1$) gives the worst performance, while distributing the hyperedges among processors and maintaining full vertex information for each hyperedge ($p_x = 1, p_y = p$) is a better approach. The distribution with $p_x > 1$ gets better performance on average.

Parallel Coarsening

The initial distribution of the hypergraph affects how to find a pair-match for the vertices. Conflicts may occur as the data is distributed and processors find pair-matches for the vertices independently. Conflict means that a target vertex may receive several matching requests from other vertices. As each vertex should end up in exactly one cluster, conflicts should be resolved before proceeding with the next step. We propose some of the parallel coarsening algorithms in this section.

The parallel coarsening algorithm proposed by Trifunovic and Knottenbelt in *Parkway*, which uses the 1D distribution, is as follows [TK04]. In the beginning of each coarsening step, the hyperedges incident to each vertex are collected with a special all-to-all communication. Then processors visit their local vertices in random order and find the best match for them using the *FirstChoice(FC)* vertex similarity metric. If processor i finds vertex u that resides on processor j as the best match for its local vertex v, u is put in request set $S_{i,j}$ of v on processor i. A local vertex can request a match with only one remote vertex. The matching stops when a user specified coarsening ratio is met. Later on, request sets are communicated among processors and they decide about matching and breaking conflicts together. We denote the set of vertices from remote processor i that requested to match with w on local processor j by ${M^w}_{i,j}$. Processor j considers these sets for each local vertex in turn and decides about them as follows:

1. if w is unmatched, matched locally, or already matched remotely, then a match with $M^w{}_{i,j}$ is granted to the processor i if we do not exceed cluster weight criteria.

2. If w has been sent to processor $k \neq i$ as a part of a request, then processor j informs processor i that the match with $M^w{}_{i,j}$ has been rejected (only because we may exceed the cluster weight criteria). When processor i is informed about the rejection, it locally matches all vertices in $M^w{}_{i,j}$ into a single vertex.

A special case may occur if processor i sends a matching request for its local vertex v to vertex w on processor j and vice versa; that is a mutual match. This case might be rejected by both. In order to resolve this, the following operations are done before we perform the above-mentioned two cases to resolve mutual-matches with one customised all-to-all communication. The communication is split into two steps. First all $S_{i,j}$ are communicated if $i < j$, and in the next step all $S_{i,j}$ with $j < i$ is communicated. When all matching decisions are finalised, hyperedges are contracted locally and identical hyperedges are removed. The removal of identical hyperedges is done based on 64-bit hashing function that is explained in Section 3.1.4.

In *Zoltan* [DBH+06], which applies the 2D hypergraph distribution, the preferred connectivity metric between two vertices is the *inner product* of their incident hyperedges. This is the same connectivity metric employed in tools such as *Mondriaan* [RBT+13] and *hMetis* [Kar07]. The inner product between two vertices is the Euclidean inner product between their hyperedge incidence vectors with one modification; instead of summing binary hyperedge incidence vectors, the weight of hyperedges are added up. When a vertex is selected for finding a pair-match, the inner product with all adjacent vertices are calculated and the vertex with the highest non-zero value is selected as a match candidate. The matching process is split among rounds. In each round, every processor selects a subset of its vertices as candidates. Then candidates are broadcast to all other row processors. The receiving processors compute the *inner product* of the received vertices with their local vertices. These are only the partial products and the communication among column processors are required to get the global inner products.

At this point, the potential matches on processor columns are send to the processors on the master row (processors at row 0). The master row, first greedily decides about the local match for each candidate. Then they are locked which means that they can not match with any other vertex in the current round[12]. Later on, a global row communication decides about the global match for each candidate. The conflicts are prevented by the locking mechanism which guarantees no conflict.

The authors argue that the matching is the most time consuming operation during the partitioning process. In order to improve the run time, they rely on partial matching solutions. One of those methods is that they limit the matching to pair of vertices in the same processor column and no horizontal communication is required. Using this method, the communication costs significantly drops but the quality suffers as well.

The next step after the matching is the hypergraph contraction. Matched vertices are merged together to produce coarser vertices. The weight of a coarser vertex is the sum of the weight of the merged vertices. The union of the hyperedges incident on any of the merged vertices is the set of incident hyperedges for any coarsened vertex. Furthermore, hyperedges of unit size are removed from the coarsened hypergraph as they do not contribute to the cut. Identical hyperedges are also detected and they are collapsed into single hyperedges. Detecting identical hyperedges is done in the same way as *Parkway* that is using hash functions. Hyperedges with different hash values are not identical. In order to identify identical hyperedges, the hash values are calculated locally. Then one horizontal communication followed by a vertical communication is needed to identify identical hyperedges. When hyperedges are collapsed, the weight of a new hyperedge is the sum of the weight of all hyperedges it represents.

Initial Partitioning

In this stage, the size of the coarsest hypergraph is small and a partitioning on it can be calculated much quicker than the original hypergraph. Because of the small

[12]This is done to prevent matching conflicts, because a vertex might be the best candidate for a number of vertices.

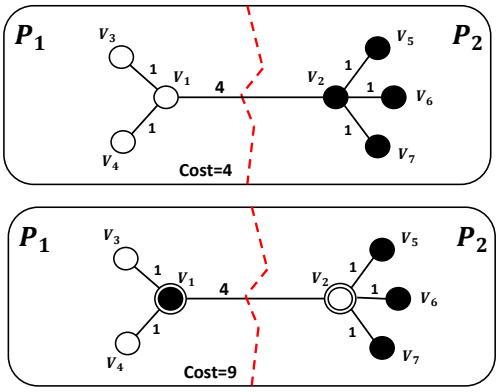
size of the hypergraph, it can be replicated on every processor. Each processor then partitions the hypergraph using a number of randomised algorithms. Processors use any of the randomised algorithms described in Section 3.1.2 or any flat move-based partitioning algorithm for this purpose. Finally, the best partitioning amongst them is selected to be projected back to the original hypergraph.

Parallel Uncoarsening

The parallel refinement algorithms is a challenging phase of the parallel algorithm. The algorithm is inherently serial and make it work in parallel is difficult. As mentioned earlier, the FM algorithm is shown to be successful in multi-level refinement. We base our analysis on the *2-way* FM algorithm but it can be also applied to other iterative move-based algorithm. In the refinement phase, vertices are moved between partition boundaries to further improve the cost function. The algorithm runs in iterations and, in each iteration, a vertex with the maximum gain is selected to be moved to the other part. When a vertex moves, the gain of all adjacent vertices should be updated. Two problems may occur for each vertex move in the parallel refinement algorithm and they are explained in the following.

First, the hypergraph is distributed among processors and each processor stores a sub-hypergraph in which adjacent vertices may reside on different processors. When a vertex is moved, updating the gain of all adjacent vertices that reside on other processors creates lots of network communications and degrades the performance. In the end, the time of the refinement phase can simply spoil the speedup of the parallel algorithm. The data to be communicated for each vertex move includes the ID of the vertex that is moved and the number of pins for each hyperedge in either parts (that changes for every hyperedge incident on the moved vertex and it is explained in Section 3.1.1).

The other issue is the conflicts that may occur when processors move their vertices in parallel if processors decide about vertex moves independently. Consider a situation in which a vertex v on processor p_i is adjacent to a number of vertices on another remote processor such as p_j. Let say p_i decides to move v to the other part because it has the highest gain among all other vertices. Processor p_i decides about



ed

Figure 3.4: An example of a conflict that happens in the parallel FM refinement algorithm when processors decide about vertex moves independently. Processor p_1 and p_2 move v_1 and v_2 to the other part, respectively. This increases the cost of partitioning from 4 to 9. The red line shows the boundary between two processors.

this move assuming that none of the adjacent vertices of v on p_j change their parts. The problem is, if p_j moves one of those vertices we may end up in a configuration in which the partitioning cut is got worse [KK96,Sch09]. An example of this situation is depicted in Fig. 3.4 for a graph. White circles show vertices in the first part and black circles represent vertices on the other part. In this example, the weight of the edge connecting v_1 and v_2 is +4 and the weight of all other edges are unity. Processors p_1 and p_2 independently decide to move v_1 and v_2 because they give the highest gain. After the move, the cost of the partitioning changes from 4 to 9. Both v_1 and v_2 are locked after the move to prevent further moves and thrashing between the parts.

In order to prevent these two problems, parallel refinement algorithms use a modification of the above mentioned algorithm [DBH+06,TK04]. To resolve the first issue, processors make vertex move decisions only based on local information. This means that only the gain of the vertices on the local processor is updated after a move and no communication between processors is done.

To overcome the second issue, another restriction is added to the refinement algorithm the algorithm. In each pass of the FM algorithm, vertices are only allowed to move in one direction; for example, from part 0 to part 1 in the bipartitioning of the hypergraph or from a higher part number to a lower part numbers in direct

k-way partitioning. The number of passes is selected to be even and the direction of the move alternates between the passes. This restriction guarantees that no move conflict happens during the parallel refinement. Another benefit of this method is that none of the hyperedges are locked into the cut because no vertex is moving in the opposite direction.

These two modification come with an extra cost such that it is hard to meet the balance constraint. While decisions are made locally and processors move vertices independently, the balance constraint may be violated. For this purpose, the modified algorithm converts the global balance constraint into a local balance constraint for each processor based on the vertex-part distribution and the distribution of the hypergraph on the processor set. Processors are obliged to satisfy the local balance constraint but they may violate the global balance constraint while moving vertices. In the end of each refinement pass, one processor is selected as the root processor (usually processor with rank 0). If the partitioning does not meet the global balance constraint, the root processor determines which processors should undo some of their vertex moves in order to meet the balance constraint.

3.1.5 Other Hypergraph Partitioning Algorithms

Çatalyürek et al. [ÇBD+07] propose a dynamic hypergraph repartitioning algorithm for problems in which the structure of the input hypergraph is changing dynamically such as adaptive mesh refinement. When hypergraph changes, instead of performing the whole partitioning from the scratch, they decide to repartition the hypergraph by moving vertices from some parts to others in order to keep the balance constraint and minimise the cost objective. They provide a unified model that combines both communication and migration cost and tries to solve a multi-objective problem. The cost function is defined as $t_{tot} = \alpha(t_{comp} + t_{comm}) + t_{mig} + t_{repart}$ that is kept at minimum when making decisions about the repartitioning. The algorithm is divided into *epochs* such that the computation in all epochs is the same but the structure of the hypergraph is different. The hypergraph in epoch j is denoted as H^j. Parameter α in the cost function represents the number of iterations in each *epoch*. The migration cost is modelled by adding H^j, k new vertices $u_1, u_2, ..., u_k$,

and some new hyperedges. For each vertex v in H^j and every u_i, a net is added to H^j that shows the migration cost of moving v to part i at the end of epoch j. Before starting the repartitioning process, u_i vertices are fixed to one of the parts and the repartitioning is reduced to hypergraph partitioning with fixed vertices.

Aykanat et al. [ACU08] propose a multi-constraint hypergraph partitioning algorithm with fixed vertices. In multi-constraint cases, the solution space is limited and it is hard to find a globally optimum solution because the vertex movement is further restricted. The multi-constraint is defined by assigning a weight vector to each vertex or hyperedge. The support for fixed vertices is also provided with some simple rules:

1. No two fixed vertices are allowed to match during coarsening if they are fixed to different parts.

2. At the initial partitioning phase, first a temporary hypergraph H' is built that is free from the fixed vertices. The nets in H' are only those nets in the coarsest hypergraph H^c that have at least two non-fixed vertices. The partitioning of H' gives a lower bound on the partitioning of H^c. Each net in H^c has the potential to increase the cut size by its weight times the number of parts it is connected to by fixed vertices. The latter is an upper bound on the cut size of H^c. At this point, a relabelling of the fixed vertices is found. Then a bipartite graph is formed with the fixed vertices of H^c on one side and the non-fixed vertices on the other side. Each non-fixed vertex is connected with an edge of weight zero to every fixed vertices on the other side (a complete bipartite graph with all edges weights initialised to zero).

   For every hyperedge e in H^c and for every possible pair of vertices (u, v), such that $u, v \in e$, u is a fixed vertex and v is a non-fixed vertex, the algorithm recursively increases the edge weight u, v in the bipartite graph by the weight of e ($\gamma(e)$). Now, finding the maximum-weight matching in the bipartite graph corresponds to finding a match between fixed vertices and non-fixed vertices that has the minimum effect on the cut size when fixed vertices are added to H'.

3. During the uncoarsening phase, fixed vertices are locked to their parts and they are not allowed to move during the refinement process. Then refinement is done like the normal refinement algorithm with no change.

Selvakkumaran and Karypis [SK06] propose an algorithm for multi-objective cut minimisation that minimises the cut and the subdomain degree. The subdomain degree of a part is defined as the sum of the weight of hyperedges that has at least one vertex in the part[13]. The authors argue that despite the success of hypergraph partitioning for minimising the cut, the cut hyperedges are not uniformly distributed among the parts. For many VLSI applications, a partitioning is preferred that minimises the cut and the subdomain degree. The latter is important in order to avoid high density interconnect regions on the circuit board by evenly distributing the interconnections across the physical device. They propose a family of multi-level partitioning algorithms that are capable of minimising both the cut and the maximum subdomain degree.

Hypergraph bipartitioning algorithms always concentrate on reducing the cut hyperedges between the two parts without considering the previous history of the partitioning. Additional information can propagate from upper layers of recursion to inferior layers. However, algorithms following this strategy usually *over-constrain* the problem. Authors suggest that direct *k-way* algorithms with the ability to consider hyperedge cut over all parts are better candidates for these types of problems. A number of solutions are proposed for reducing subdomain degree that are based on, for example, sum-of-external-degree, absorption, scaled cost, and they use a combination of recursive bipartitioning and direct *k-way* algorithms.

The proposed algorithms run as follows. First, the best partition is calculated using a state-of-the-art recursive bisection algorithm. Then, a direct *k-way* partitioning with a different objective is applied to reduce subdomain degrees. The direct *k-way* algorithm applies three rules. First, it explicitly minimises subdomain degrees of the parts. Second, increasing the cut is inevitable, when reducing the subdomain degree. How much the algorithm cares about minimising the maximum subdomain degree is

[13]Each subdomain is a part in our partitioning problem.

a user defined parameter. Tighter restrictions provide more limited space for vertex moves between the parts. Finally, the direct *k-way* algorithm always maintains the balance constraint.

In multi-objective graph/hypergraph partitioning algorithms, objectives are usually assigned priorities. The objective with the highest priority is treated as the main objective and the others as tie-breaking conditions. But this paper uses a weighted cost function, that is $Cost = \alpha \cdot (maximum\ subdomain\ degree) + \beta \cdot (Cut)$, where α and β are user defined parameters. The cost function assigns each objective a weight and considers objectives according to the weight assigned to them. In direct multi-phase refinement, a vertex is chosen randomly. If the vertex is internal to the part then no move occurs; otherwise it is moved to one of the parts in which it has some adjacent vertices such that the balance constraint is not violated and the subdomain degree is decreased. The main drawback is that it cannot make large perturbations to the initial partitioning because the hill-climbing capability of refinement algorithms is often limited. In addition, it cannot make large improvements to the subdomain degree if the cut objective has higher priority ($\beta > \alpha$). In another method, which is called aggressive method, every part is collapsed into 2^l macro parts (l is user defined). Then a $2^l \cdot k\text{-}way$ partitioning is performed on the macro parts using the previous method. Finally, the $2^l k$ macro parts are partitioned again into k parts such that each parts has exactly 2^l macro parts. The aggressive method has better chance of escaping local minima and simulations show that it improves the multi-objective cost compared to the direct multi-phase refinement. It is reported that increasing l causes higher hyperedge cuts.

3.2 Hypergraph Partitioning Tools

In this section we provide a summary of the tools available for hypergraph partitioning. To date, there is no unified framework for hypergraph partitioning and available tools use different user interface and input formats. The need for a framework is necessary like the one exists for graph partitioning tools. As mentioned in the beginning of the chapter, we focus on the work related to the hypergraph partitioning (not

graph partitioning tools and algorithms). We only name few graph partitioning tools and the reader is referred to the tool manuals for further information about them. Examples are: the serial graph partitioner *METIS* [Kar13a] and parallel graph partitioner *ParMETIS* [Kar13b] from the Karypis lab, *JOSTLE* [WC07] that is a parallel graph partitioning software for partitioning unstructed meshes (for example, finite element or finite volume meshes), the scalable multi-level partitioner *KaPPa* [HSS10], the Karlsruhe Fast Flow Partitioner *KaFFPa* [PC10], Karlsruhe High Quality Partitioning *KaHIP*[14] and the distributed graph partitioner *JA-BE-JA* [RPG+13].

Zoltan data management services for parallel dynamic applications [San14b] is a toolkit developed at Sandia National Laboratories. The library includes a wide range of tools such as dynamic load balancing, graph/hypergraph colouring, matrix operations, data migration, unstructured communications, distributed directories, and graph/hypergraph partitioning. It follows a distributed memory model and uses MPI for inter-processor communications. It is available in *Trilinos* which is an open source software project for scientific applications [San14a]. It has a lot of applications in large-scale experiments and simulations such as:

1. *Interoperable Technologies for Advanced Petascale Simulations (ITAPS)* for the partitioning of their mesh databases [Sci15] in projects *iZoltan*, *iMeshP*, and Scaling Unstructured Mesh Computations

2. Geometrical particle partitioning in 3D parallel finite-element simulations at *Scientific Discovery through Advanced Computing (SciDAC)* [Sci08].

The source code of *Zoltan* is written in C with interfaces for *C++* and *Fortran*. It has no restriction on the input data format. Applications can use their own format but they should tell *Zoltan* how the input should be interpreted; the application does this by providing query functions. *Zoltan* queries the application for the necessary information such as vertex IDs, hyperedge IDs, hypergraph incidence matrix, vertex weights, hyperedge weighs, and object coordinates (in case of geometric partitioning).

[14] `http://algo2.iti.kit.edu/documents/kahip/`.

The runtime behaviour of the algorithm is controlled by providing a number of input parameters. Input parameters are set by the user. *Zoltan* supports geometric, graph, and hypergraph partitioning as well as static and dynamic hypergraph partitioning. It also supports multi-criteria load balancing by assigning a weight vector to vertices and/or edges/hyperedge. The heterogeneity of processors is supported by giving an input vector that defines different sizes for the parts. Furthermore, unstructured communication is supported by providing communication utility package.

The latest version of *Zoltan* is released in May 2015. Currently, there is a newer version of *Zoltan* that is called *Zoltan2*. This is a new project and the developers are rewriting the source code in modern templated C++. It is available in *Trilinos* release 12.2.1. Currently, *Zoltan2* only supports a few partitioning and ordering methods but the package is actively developed. It supports arbitrary index types; therefore, it solves problems with more than 2 billion elements (32-bit limit). Much of *Zoltan2* should be considered experimental code and the feature set is currently small compared to Zoltan.

hMetis [Kar07] is the earliest tool for serial hypergraph partitioning developed by Karypis and Kumar at the University of Minnesota. It is specially designed for VLSI circuit partitioning. The algorithms are based on multi-level partitioning schemes and support recursive bisectioning *shmetis, hmetis*, direct *k-way* partitioning *kmetis*, and partitioning with fixed vertices. The license is free for educational and research purposes by non-profit institutions and US government agencies. The supported platforms are Linux, Mac OS, Windows, Sun, IBM AIX, and IRIX. It is callable as a standalone program from the terminal with command line parameters. There are a variety of supported coarsening schemes and the algorithms for the initial partitioning and uncoarsening phases of the multi-level paradigm are modified versions of *FM* algorithm. *V-Cycle* refinement is also provided. User can define some input parameters in order to control the runtime behaviour of the algorithms. Algorithms can also be accessed form a standalone library libmetis.a. The latest stable version is released in November 2008.

PaToH [ÇA11] is a serial hypergraph partitioner developed by Ümit V. Çatalyürek at the Bilkent University. It is a fast multi-level recursive bipartitioning based tool

that supports partitioning with fixed vertices and multi-constraint objectives. The supported platforms are Linux (32-bit and 64-bit), Mac OS, Sun Solaris, and IBM AIX. An interface is also provided for Matlab. The binary distribution is available free of charge for non-commercial and research purposes; commercial use of the software needs a license. It supports a set of agglomerative (vertex clusters are formed one at a time) and hierarchical clustering (several cluster of vertices can be formed simultaneously) algorithms for the coarsening phase. The *Greedy Hypergraph Growing (GHG)* algorithm is used for the initial partitioning which is an extension of the *GGGP* algorithm in *hMetis*. Furthermore, a variety of *KL-FM* based refinement algorithms are provided for the uncoarsening phase like *Boundary FM (BFM)* algorithm. In addition to the library interface, the partitioner can be invoked from the terminal. The tool includes direct *k-way* hypergraph partitioner *kPaToH*. The latest Linux version is dated back to November 2008.

Parkway is a parallel multi-level hypergraph partitioner developed by Aleksandar Trifunovic during his PhD research. It is the first parallel hypergraph partitioning tool. It is written in C++ and is Linux-based. Interfaces are provided to *hMetis*, *PaToH*, hypergraph partitioners and *SPRNG* libraries for parallel pseudorandom number generator [Flo14]. The latest optimised version is 2.11. The algorithms for each phase of partitioning are 1) Coarsening: parallel *FirstChoice* algorithm, 2) Initial Partitioning: generic recursive bisection or using interfaces to *hMetis* or *PaToH*, and 3) Refinement: greedy *k-way* refinement algorithm; *V-Cycle* refinement is also supported. The runtime behaviour of the partitioner is controlled by some input parameters. The output is also returned in binary format. The latest version is Parkway 2.11 from May 2008.

Mondriaan [RBT+13,VB05] is a sequential hypergraph partitioner especially designed for rectangular sparse matrix-vector multiplications. It is recursive multi-level hypergraph bipartitioning algorithm written in C. The first release of the software was in 2002 and the last update is dated August 2010. It supports a variety of platforms with interfaces to *PaToH* and Matlab since version 3.0. The input has to be given in Matrix Market (MM) format and after the partitioning, several output files are generated including distributed matrix, processor indices for each part,

row/column permutations, input/output vector distribution, etc. The output of the partitioning can be seen as an image and the whole partitioning process can be seen as an animated GIF image. The user has the capability of setting runtime parameters and partitioning methods before calling *Mondriaan* by setting some non-numerical values. Furthermore, numerical options are used to fine-tune partitioning options. The latest version is released in August 2013.

MLPart [AAI06] is a hypergraph circuit partitioner developed in UCLA. The source code is written in C++ and the supported platforms are Intel Linux, Sun Solaris 2.7, and Microsoft Windows (95/98/NT). The move-based algorithm is a modification of the FM algorithm in which the algorithm uses a form of randomisation for computing gains of the legal moves at the beginning of each pass in order to escape local minima [PM07]. The algorithm also implements a slightly different data structure for the gain bucket to manage fixed nodes. The algorithm is evaluated against *hMetis* and it generates improved partition quality for some of the hypergraphs in IBM ISPD-99 circuit benchmarks [CKM99]. For the coarsening phase, it implements a linear time clustering algorithm for the edge coarsening proposed in *hMetis* library [Kar07]. This is followed by *CLIP-FM* [DD97] for the initial partitioning and *LIFO-FM* [Kar07] for the uncoarsening phase. The implication of the proposed algorithms is based on simplicity of design. It has also support for fixed vertices. The last update is dated back to October 2004.

SCOTCH and *PT-SCOTCH* are projects developed by the Satanas team of the Laboratoire Bordelais de Recherche en Informatique (LaBRI) in INRIA Bordeaux - Sud-Ouest and they are designed for serial and parallel graph partitioning, respectively. The tools are designed for graph algorithms with the divide and conquer approach to scientific computing problems such as: graph and mesh partitioning, static mapping, and sparse matrix ordering. These algorithms have applications in various domains ranging from structural mechanics to operating systems and bio-chemistry. It supports C and Fortran. The tool is not specifically designed for hypergraph partitioning, but it provides algorithms to partition graph and mesh structures. Because the mesh structure can be defined as a node-element bipartite graph, it can represent a hypergraph. We consider this tool as a graph partitioner rather than a

hypergraph partitioner. The latest versions are released in December 2012.

3.3 Applications of Hypergraph Partitioning

There are lots of applications for hypergraph partitioning and we discuss some of them in this section. The first and a long standing application of hypergraph partitioning is netlist partitioning for the computer aided design of VLSI circuits [Len90,She12]. As the size and complexity of today's VLSI circuits are increasing, partitioning the VLSI circuit into clusters with minimised interconnection among them is important and critical. The circuit is composed of a number of pre-designed *components* with input and output *terminals*. Each component is represented as a vertex in the hypergraph. A net is a collection of interconnected input and output terminals and it is represented as a hyperedge. The hypergraph partitioning can be used in different design stages and for various purposes as reported by Alpert [Alp96]:

1. The partitioning divides the system into smaller sub-circuits such that the signals between these sub-circuits correspond to the interconnections between them. The partitioning can be used to reduce circuit design complexity in hierarchical design such that the design process will be more manageable and feasible by automatic design tools and software.

2. Partitioning increases system performance. As the size of the VLSI circuits is increasing, wire delays surpasses gate delays. In addition, long off-chip signal delays are much bigger than on-chip signal delays and their power consumption is higher. The partitioning is important in identifying the interconnect structure and decreasing off-chip signals as well as long global wires.

3. It reduces layout area. In top-down hierarchical design, wires between sub-circuits at top levels of hierarchy are longer than wires between sub-circuits at lower levels of hierarchy. In this situation, the problem can be directly expressed by the hypergraph partitioning problem with minimum cut objective in order to reduce layout area and total wire length.

Çatalyürek and Aykanat [ÇA99] apply hypergraph partitioning to sparse matrix-vector multiplication. Iterative methods such as the conjugate gradient normal equation error and residual methods (CGNE and CGNR) and the standard quasi-minimal residual method (QMR) (that is used for solving unsymmetric linear systems) need computations of the form $y = \mathbf{A}x$ (or $y = \mathbf{A}^T x$). There is an unsymmetric square coefficient matrix in each iteration. In parallel computation of this multiplication, the sparse matrix $\mathbf{A}$, the input (x), and the output (y) vectors are distributed among a number of processors and multiplication is done in parallel. During the multiplication, processors regularly access parts of the matrix that are stored on other processors. This imposes lots of network communications and does not allow the problem to scale up by increasing the number of processors. The communication volume also scales up with increasing the problem size. The authors propose a decomposition of the sparse matrix among the processor set based on hypergraph partitioning such that it also balances the computational load. They propose two models of decomposition: column-net and row-net that are used for row-wise decomposition (that needs processor pre-communication to collect matrix data before calculation the multiplication) and column-wise decomposition (that needs processor post-communication to finalise multiplication results), respectively. In row-wise decomposition, the sparse matrix is represented as a hypergraph in which rows and columns of the matrix correspond to vertices and hyperedges of the hypergraph, respectively. Each hyperedge at column j contains a vertex at row i if its corresponding item $A[i, j]$ in the matrix is non-zero. The column-wise decomposition is obtained similarly with rows of matrix as hyperedges and columns as vertices. They achieved between 30% to 38% less communication volume on average between processors compared to graph partitioning models. There are similar works for modelling inter-processor communication in parallel sparse matrix-vector multiplication such as Vastenhouw and Bisseling [VB05], Uçar and Aykanat [UA04].

Curino et al. [CJZM10] proposes a workload-aware database partitioning and replication strategy to improve the scalability of shared nothing relational databases

in OLTP[15] systems. The database is distributed among a number of computer nodes. The workload is a set of transactions that access the database. The target of the partitioning is to increase the scalability and improve the availability. The latter is is achieved by insuring that the system still can answer transaction queries in case of failure; when one partition fails the remaining partitions should be able to answer some of the transactions. The traditional partitioning schemes, such as round-robin and random hashes, fails to improve performance especially when transactions access various parts of the database. Running distributed transactions are expensive and the aim is to run them locally. In their model, they first characterise transactions in the input workload and their access pattern. Then a hypergraph representation of the database is built. Each tuple of the database is represented as a vertex. Each hyperedge represents a groups of tuples that are accessed together in one transaction. Hypergraph partitioning is then applied to partition the database. After the partitioning, each part is assigned to one unique physical computer node. The application manages tuple-level replication in the partitioned hypergraph by exploding a vertex (that represents a tuple) into a star-shaped configuration. The tuple is replaced by a star with $n + 1$ nodes where n is the vertex degree (the vertex degree is the number of transactions that access the tuple). A hyperedge is added to the hypergraph which contains the vertex and all its replicas; the size of the new hyperedge is $n + 1$. The new hyperedge has exactly one pin in each of the hyperedges that are incident to the vertex. The $n + 1^{th}$ tuple only belongs to the new hyperedge. Finally the *k-way* partitioning on the hypergraph is calculated to distribute the hypergraph on k compute nodes. The evaluation results shows up to 30% improvement on a set of benchmarks including Yahoo! Cloud Serving Benchmark [CST+10] and a set of random generated benchmarks.

Agent-based simulation is a new technique for simulating dynamic complex systems and their behaviour. It is composed of a set of autonomous inter-acting agents with the ability to adapt and modify their behaviours. It has lot of applications. Examples are: social simulation for social behaviour investigation and human movement

[15]OnLine Transaction Processing (OLTP).

patterns, biological science for cellular and sub-cellular molecular behaviour, stock market and supply chains for trade networks and supply chains, and traffic modelling for traffic flow management [NM09]. The simulation space is divided into smaller subspaces and each agent is responsible for processing a subspace. While processing the subspaces, communication occurs between the agents. In parallel agent-based simulation, subspaces are assigned to distributed processors. The aim is to assign subspaces to processing nodes in such a way that the load on processors is balanced and the communications between agents are minimised. In order to solve the problem with hypergraph partitioning, the problem is modelled with a hypergraph in which vertices are subspaces and hyperedges show inter-dependency among those subspaces. A partitioning on the hypergraph provides a distribution with aforementioned specifications. Examples are the work by Márquez et al. [MCS15] that uses *Zoltan* dynamic hypergraph partitioner for dynamic migration of the agents between processor nodes in biological simulations, and the work by Xu et al. [XCAL14] that uses dynamic graph and hypergraph partitioning for road network simulation.

In data classification, Zhou et al. [ZHS06] argue that the inter-dependency between objects in the real world applications cannot be always illustrated as pair-wise relationships (this is case in graphs). Pair-wise dependency causes loss of information when it is used for representing complex relationships. Consequently, they prefer to model those application with a hypergraph rather than a graph. Each object stands for a vertex and each hyperedge shows a relationship between a group of objects. The comparison result of their approach to graph methods on machine learning and web text categorisation datasets shows that hypergraph models can capture relationships much better than graph models and provides better data classification. Dickenson [Dic86] tries to cluster data points using hypergraph partitioning such that objects inside a cluster share common characteristics and those in different clusters share less common characteristics. Other examples in the field are the work by Yu et al. [YTW12] in image classification and the work by Han et al. [HKKM98].

Other examples of the application of hypergraph partitioning are:

- Biology

  - Classifying gene expression data [THK09].
  - The haplotype assembly problem for study of genetic variations among individuals and gene disease diagnoses [CPH+14].

- Constructing *mRNA* and *miRNA* interaction networks [Kim13].
- Identify clusters of pixels which form an image such as colour image segmentation for managing complex relationships [DRBL09,Rit09]
- Managing multi-label data for information storage [TKV10] in database systems.
- High dimensional data clustering: [HLT+14].
- Social Networks analysis [Was94,HC14].

3.3.1 Comments on the Applications of Hypergraph Partitioning

Although there are lots of applications for hypergraph partitioning in scientific computing, the problem of modelling the application itself with a hypergraph is a challenging task. The problem arises in representing group relationships that define the hyperedges in the hypergraph. Heintz and Chandra [HC14] investigate the challenges exist for modelling social network graphs with hypergraphs. In social networks, a group is the fundamental building block for representing interactions between entities such as groups of peoples interested in a movie, groups of friends, and football club members [LPA+09]. Graph modelling, which only captures one-to-one relationships, fails to provide a fair representation of group-level relations and it highlights the need for using hypergraph modelling for this purpose [Was94].

Managing these groups and their interactions is a challenging task. Social networks like Facebook and Twitter have become very popular recently and they are dealing with billions of users and their groups interactions. This means that analysing algorithms and tools need to be scalable in terms of computations and memory usage. Heintz and Chandra [HC14] show that there are some redundancies in the hypergraph

modelling that can be removed from the hypergraph representation without losing important information. They investigate two hypergraph benchmarks as use cases: 1) DBLP authorship database (in which vertices are authors and hyperedges are authors who contributed in writing a paper) and 2) Apache Subversion repository (in which vertices are committers and a hyperedge represents a group of committers who committed a file on the repository). They try to transform the hypergraphs into a bipartite graph representation in order to save memory, simplify computations, and remove redundant information.

The second issue arises in the structure of the hypergraphs. Hypergraphs have a skew in vertex degree (like graphs) and a skew in edge cardinality (unlike graphs). This causes more processing for some vertices and hyperedges because of different degrees and sizes, respectively. The arbitrary size of hyperedges makes the partitioning problem more challenging. In the same way that vertex-based partitioning heuristics have shown to be inefficient in graphs with highly-skewed vertex degree [GLG+12], high skew in hyperedge sizes is even a more challenging issue in the hypergraph partitioning problem. In this situation, there may be a need for vertex-based and hyperedge-based heuristics to deal with both skews.

Furthermore, average vertex degrees may be smaller than average hyperedge sizes in some hypergraphs which means they need less processing. Similarly, the opposite maybe true in other hypergraphs. Depending on the situation, we may decide to switch between two problems and reduce one to the other in order to improve the performance and overcome the problem easier. The problem type and its specifications need careful analysis.

The other problem is about the characterisation of hypergraphs. The structure of graphs such as those modelling social networks and internet topology is well understood and there is an extensive research to study their distinctive structure. For example, many natural graphs follow power-law degree distributions [FFF99]. In addition, Kang [KTF09] shows that modelling social networks with graphs tend to have small diameters that shrinks as new vertices or edges are added to the graph. The problem is that we do not know how these facts are extended to hypergraph models; the predictions are unknown and very different between practical hypergraphs [HC14].

For instance, the vertices in a social network-based hypergraph may have limited degree due to the limited capacity of humans to engage in social interactions, even though some hyperedges are very large [HC14]. As another example, the reduction of vertex degree is much higher in Apache SUV compared to DBLP when transforming the hypergraph into bipartite representation, but the increase in hyperedge size is much higher. On the other hand, the transformation needs more memory space for saving in Apache SUV (because a larger number of subset relations) but lacks a clear hierarchy structure. The transformation is intended to save memory. This is achieved for DBLP while the characteristics of Apache SUV do not allow such optimisations.

Finally, in case of programming models, there is no efficient and unified programming model for hypergraphs like the one exists for graph models such as the MapReduce [DG08] framework for large data processing and the Pregel [MAB+10] for large graph processing. A similar programming model should be developed for hypergraphs. Heintz and Chandra [HC14] believe that the model should be restricted. Restricted models leave some room for optimisation while provide sufficient expressibility to the programmers. This a difficult task, because of two reasons. First, the area of hypergraph algorithms is much smaller than graphs. Second, the are some issues for modelling applications with hypergraphs such as those described above.

3.4 HPC in the Cloud

The interest in moving scientific applications into the cloud has been increasing in recent years. The reason lies in the advantages that the cloud offers to HPC applications such as elasticity, small startup and maintenance costs, dynamic resource allocation, and economies of scale and use. On the other hand, some characteristics of the cloud are becoming performance bottlenecks for running HPC in the cloud such as hardware virtualisation, hardware heterogeneity, and multi-tenancy. Despite having challenges on the way, there are lots of works that investigate the problem and solutions have been proposed to ease the way for HPC in the cloud. This section studies the related work about moving scientific applications into the cloud.

Gupta et al. [GKG+13] address the questions of why and who should use the cloud for HPC applications? What type of applications should be used in the cloud and how? The answer to these questions are unclear. The cloud has been originally designed for web and business applications that have different specifications and requirements than HPC applications. The restricted network resources and the overhead of virtualisation on network and storage are major performance hurdles on the way of deploying the cloud for HPC [YCD+11,Wal08,MDH+12]. They propose that the cost/performance-optimal execution platform varies from HPC clusters to the cloud depending on the characteristics of the application. They test various HPC applications with different characteristics on a number of testbeds including HPC clusters (Ranger cluster that is an old HPC cluster, and new clusters such as Open Cirrus, and Taub), and the private and public clouds. The selected benchmarks are written in two different parallel programming environments: MPI and CHARM++ and they are as follows:

- NASA Parallel Benchmarks (NPB) class B (with MPI version, NPB3.3- MPI) [Div15].
- Jacobi2D: a kernel which performs 5-point stencil computation to average values in a 2D grid.
- NAMD: A highly scalable molecular dynamic application.
- ChaNGa: A highly scalable Charm N-body GrAvity solver that is used to perform collisionless N-body simulation.
- Sweep3D: a particle in transport code which is widely used for evaluating high performance parallel architectures.
- NQueens: a backtracking state space search problem.
- NPB: a small set of programs designed to help evaluate the performance of parallel supercomputers and includes benchmarks for Integer Sort (IS), Lower-Upper Gauss-Seidel solver (EU). Most of applications are communication-intensive applications.

The scalability test shows that even Ranger HPC cluster that uses old processors and network interface outperforms both the private and public clouds for some of the applications at around 32 processor cores. In addition, the performance of communication-intensive applications usually start to degrade in the cloud (both private and public) as soon as the number of cores exceeds the number of processing cores per Virtual Machine (VM). This is more evident on the public cloud which has VMs with 4 virtual cores. In this situation, there is no inter-VM communication up to 4 cores and applications show good performance and scalability. The performance suddenly degrades as soon as the communication starts across VMs. Most of performance degradation comes from network and virtualisation overheads.

Furthermore, the authors notice a big variability in the execution runtime of applications in the cloud compared to supercomputers. The variability increases with increasing the number of processor cores, partially due to the decrease in computational granularity. The major reason for variable runtimes is using shared resources in the cloud. In another observation, they report that CPU is under-utilised for almost half of time in the cloud. CPUs spend most of their time waiting for receiving data from other processors and the network overhead plays an important role in this. The magnitude of latencies and bandwidth in the cloud are worse compared to supercomputers and this makes it very challenging for communication-intensive applications such as IS, LU, NAMD and ChaNGa to scale up. Finally, OS and the hypervisor interference in the cloud are very high and bring considerable overhead in the amount of work done by processors in the cloud.

Gupta et al. [GKG+13] identify the network virtualisation as the primary bottleneck of the cloud. They argue that using light weight VMs provides performance improvement in the cloud and imposes significantly lower communication overhead. It also decreases shared resource utilisation and provides better network utilisation. In addition, CPU affinity, which instructs the operating system to bind a process to a specific CPU core, reduces the competition for shared resources among processes on the same VM, increases cache locality, and prevents the operating systems for inadvertently migrating a process. For the tested applications, they noticed that virtualisation introduces a small amount of computation overhead and removing

unnecessary I/O operations helps in achieving maximum performance.

The authors also investigate the cloud economies of use. A HPC cloud user ideally wants dedicated resources. This means that the cloud provider can not use multi-tenancy execution for HPC applications; therefore, the pricing should be increased for HPC users to compensate the loss of dedicated resources. In addition, the performance of most of HPC applications is very sensitive to network overhead which makes it very difficult for those applications to benefit from the cloud. Virtualisation overhead is another bottleneck. These facts restrict the number of HPC applications to fit into the cloud. Consequently, if too many VM instances are needed to meet a specific performance and depending on the pricing model, then the usage of HPC in the cloud becomes uneconomical. In general, running HPC application in the cloud for small and medium sized enterprises may provide savings to the company [GKG+13]. Furthermore, they use different pricing models and they estimate that the cloud can provides around 2-3 times more economical benefit than using on-premise supercomputing resources.

They conclude that, identifying the characteristics of HPC application to estimate the benefit of running the application in the cloud is crucial. These characteristics are necessary for mapping HPC application into the cloud in order to achieve cost-performance benefits. However, identifying them is not a trivial task for complex applications. In general, they suggest that using a hybrid environment (the cloud and on-premise supercomputing resources together) is the most beneficial paradigm for both performance and economical reasons.

Juve et al. [JDV+09] study the use of Amazon EC2 cloud for running scientific workflows and the performance is compared to HPC clusters. In their study, the workflow is defined as "*loosely-coupled parallel application that consist of a series of computational tasks connected by a data- and control-flow dependencies*". They describe the benefits of the cloud for workflow applications as follows:

- **Infinite resource provisioning:** Resources in the cloud are unlimited.
- **Leases:** Users are saving time in allocating resources to batch schedulers. In the cloud, users directly allocate resources as needed which removes the time needed for scheduling and results in increased performance.

- **Elasticity:** Resources can be acquired and released on demand allowing the workflow to easily scale up and down based on the computational needs of the workflow.

They select three types of applications with different I/O, memory and CPU resource usages: Montage from astronomy (high I/O, low memory and CPU requirement), Broadband which is a seismology application (medium CPU and I/O requirement and high memory needs), and Epigenomics from bioinformatics (low I/O, medium memory, and high CPU requirement). All of them are loosely-coupled applications with tasks communicate via the file system instead of directly through the network. Amazon EC2 resource types are selected with different specifications and they offer different price/hour values. Comparing the runtime of the applications shows that HPC cluster gives the highest performance in all cases. By identifying the characteristics of workflows, one can choose the best Amazon EC2 configuration for running each application in order to achieve the best performance. For example, *m.xlarge* configuration offers the highest memory per node. Using this configuration, they they achieve the best performance for Montage in which the extra memory is used for file system buffer cache to reduce the waiting time of tasks for I/O operations in Linux. They find that the virtualisation overhead has the highest impact on the performance of CPU bound applications. In addition, lack of high-performance parallel file systems on Amazon EC2 can provide a major performance bottleneck for I/O-intensive applications. Installing high performance file system is prohibitive due to the need for high speed network interconnection in the cloud. The primary costs, which are the costs of running resources and storage costs, are relatively small in the cloud.

Napper and Bientinesi [NB09] investigate the cloud computing for numerical applications and explore the cloud for HPC applications. They report similar results as previous research that is the performance on Amazon EC2 using high end computing nodes suffers from limited network inter-connectivity. They find that using smaller cluster sizes for numerical application gives better cost/performance benefit. The cost for solving a linear system increases exponentially with the problem size which is in contrast to scalable HPC clusters. They conclude that the cloud

computing is not yet ready to run HPC applications and they suggest better network interconnection and more physical memory in the cloud to be suitable for running scientific applications.

Jackson et al. [JRM+10] compare conventional HPC platforms to Amazon EC2 using real scientific applications and investigate how the communication pattern of HPC applications can affect the performance. The performance on Amazon EC2 is evaluated compared to three HPC clusters with different specifications from modern HPC clusters (Carver system) to mid-range Linux clusters (Lawrencium system, which is the slowest HPC cluster, and Franklin system, which has faster network interconnection than Lawrencium but slower than Carver system). Applications are chosen with different characteristics as follows:

- Community Atmosphere Model (CAM): A MPI application [CES].
- General Atomic and Molecular Electronic Structure System (Gamess): It needs considerable memory access operations and there are two implementations using MPI and socket communication.
- GTC [Lee87]: A fully self-consistent, gyrokinetic 3-D Particle-in-cell (PIC) code with a non-spectral Poisson solver which is MPI-based. Communications are dominated by the nearest neighbour exchange operations and it utilises indirect address that stresses random access to memory.
- Integrated Map and Particle Accelerator Tracking Time (IMPACT-T): An object-oriented Fortran90 code from computational tools. Its performance is very sensitive to memory bandwidth and MPI collective performance.
- MAESTRO: Used for simulating astrophysical flows that has a very unusual communication topology pattern that stresses simple topology interconnects; furthermore, it stresses global communications and memory performance with very low computation intensity.
- MILC: Represents Lattice Computations which are extremely dependent on memory bandwidth and pre-fetching. It exhibits a high computational intensity.

- ARAllel Total Energy Code (PARATEC): It is a MPI-based code from quantum mechanics and stresses global communication bandwidth by relatively short length point-to-point messages.

- High Performance Computing Challenge (HPCC) benchmark [LDK+05]: It has seven synthetic benchmarks which combine computation and communication. These benchmarks can be considered as very simple proxy applications.

Simulations show that HPCC benchmark runs significantly faster on EC2 than Lawrencium system. The reason is that the AMD Opteron based systems in Amazon EC2 are known to have better memory performance than Intel Harpertown-based systems used in Lawrencium. Both system are significantly slower than the Carver system. In order to test the network latency in the evaluated systems, they compare the average ping-pong latency and bandwidth, and the randomly-ordered ring latency and bandwidth on the systems. Evaluations show that the latency and bandwidth measurements of the EC2 gigabit Ethernet interconnect are more than 20×worse than the Lawrencium system. Using self-contention in ping-pong latency, EC2 is 13× slower than the Lawrencium and 400× slower than the Carver. The low performance network interconnection in EC2 has major impact on the performance on the very simple application proxies for the HPCC benchmark.

In case of other applications, GAMESS and PARATEC run 2.7× and more than 50× slower on EC2 than Craver, respectively. The difference in performance simply reflects the degree of dependency of these applications on the network. Applications that stress global communications such as PARATEC (52×), MILC (20×), and MAESTRO (17×) give the worst performance on EC2 compared to Carver. Other applications that mostly include point-to-point and local communications, which do not induce quite as much contention as global communication, are not highly impacted by performance degradation; examples are CAM (11×), IMPACT (9×) and GTC (6×).

In another evaluation, the performance of EC2 compared to other HCP systems is evaluated using Integrated Performance Monitoring (IPM) framework [WPS09]. This framework is a profiling tool which uses MPI profiling interface to measure

the time taken by an application in MPI operations on a task-by-task basis. This tool is used to identify the amount of time taken by the application on computation and communication as well as the type of MPI communication calls. This is useful to determine which cloud configuration is mostly responsible for restricting the performance[16]. Their results shows the more the time application spend on MPI communications, the worse the performance is on EC2. An interesting result that is found is about one application, which is denoted as CAM, that gets good relative performance on EC2 despite spending more than 45% of its time on communication. Analysing CAM shows that it is the only application that uses large MPI messages for both point-to-point and collective communications. The reason relies in the difference between the latency and bandwidth of communications in the cloud. Evaluating HPCC benchmarks shows that EC2 ping-pong latency is 35× worse than Lawrencium while its bandwidth is only 20× worse. Generally speaking, they found that any application that spends time on communicating large messages via point-to-point messages in the latency limit context would run slower in the cloud compared to the one that performs collective communications in the bandwidth limit. CAM application falls in the second category.

Furthermore, authors report significant variation of runtime in EC2. This mostly comes from the heterogeneity of resources in EC2. In each run of the application, they noticed different types of allocated resources, and the variability of network congestion. This heterogeneity interferes with the load balancing and performance tuning strategies. The other heterogeneity comes from shared virtualised hardware such that the user has no control on discovering weather s/he uses a non-virtualised hardware or not. For PARATEC application they see 42% runtime variability[17].

Finally, the following important lessons are learned from this study:

1. An application communication pattern affects how it uses the cloud network interconnect and, consequently, affects the performance. For example, those applications that mostly perform collective communications and global commu-

[16]The overhead of running IPM framework is insignificant. Other studies show that IPM adds only 2% overhead [WPS09].

[17]As we discussed above, Gupta et al. [GKG+13] also report runtime variation in the cloud. They argue that the major reason for the runtime variation in the cloud comes from resource sharing.

nications in the cloud get worse performance than those applications performing local communications. This emphasises on the importance of the resource locality in the cloud.

2. Communicating large messages via collective MPI messages gives better performance than point-to-point MPI communications. The reason lies in the difference between the latency and bandwidth of network communications in the cloud.

3. Heterogeneity of resources in the cloud is one of the main sources of performance degradation. It also causes large variation of application runtime in the cloud.

Evangelinos and Hill [EH08] study the applications of the HPC standard benchmark tests on Amazon EC2 and they find that EC2 is suitable for running small sized HPC applications. According to their study, on-demand capability of the cloud is an interesting characteristic for batch processing queue-based systems in which virtualised resources can be sequestered and customised for a specific scenario and target. Investigating the network bandwidth in EC2 shows the availability of high bandwidth among EC2 instances. In addition, I/O performance is tested with some benchmarks that generate large read and write requests on both local disk and remotely mounted home directory. The results show that there is big difference in the performance of read and write operations to/from local disk. They report the performance of EC2 to be comparable to low-cost HPC clusters by simulating a atmosphere-ocean climate model.

Gupta and Milojicic [GM11] report that the cloud could be a suitable platform for computation-intensive applications and communication-intensive applications can get a certain level of speedup and scalability on the cloud; up to low processor counts. For the evaluation, they have selected two benchmarks as follows:

- NAMD: A highly scalable molecular dynamics application that is used ubiquitously on Supercomputers.

- NQueens: A backtracking search problem to place N queens on a $N \times N$ chessboard so that they do not attack each other.

The first is a computation-intensive application while the latter is a tree structured computation where communication only happens for load balancing (the load balancing is done through work stealing). The results show that NAMD stops scaling after 64 processor cores while NQueens offers high scalability in the cloud. Evaluating cost effectiveness of both applications on EC2 compared to HPC cluster shows that NAMD is better to run in the HPC cluster, while EC2 is a better environment for NQueens.

Gupta et al. [GSKM13] propose a dynamic load balancer for improving the performance of tightly-coupled iterative HPC applications in the cloud. They report that heterogeneity of hardware resources and multi-tenancy characteristics of the cloud are two problematic challenges. They try to resolve the issue by proposing an HPC-aware cloud environment. Their method continuously monitors the cloud and detects load imbalances among CPU cores. When the system is imbalanced, some of the load is migrated from overloaded processors to underloaded cores. Using this strategy, they achieve up to 45% performance improvement for HPC applications in the cloud.

According to the above discussions, transferring HPC applications into the cloud is very challenging. It requires understanding of the underlying structure of the application and its specifications. In addition, one can not always get the required standards and the expected performance by moving into the cloud. According to the above discussion, the scalability issue is an interesting topic in the cloud. We refer to a complete reference, "*The Magellan Report on Cloud Computing for Science*", proposed by U.S department of energy [YCD+11] that investigates the potential role of the cloud computing for scientific applications. The scalability of parallel hypergraph partitioning algorithms, considering the structure of the hypergraph and they the algorithm does network communications, can suffer a lot in the cloud. Consequently, the problem is twofold: the parallel hypergraph partitioning algorithm itself and the structure of the hypergraph.

Chapter 4

Serial Hypergraph Partitioning Algorithm

In this chapter, we propose our serial multi-level *Feature Extraction Hypergraph Partitioning (FEHG)* algorithm. The algorithm makes novel use of the technique of rough set clustering in categorising the vertices of the hypergraph in the coarsening phase. *FEHG* considers hyperedges as attributes, which is also called features, of the hypergraph (according to rough set clustering definitions in Chapter 2.3) and tries to discard unimportant attributes to make better clustering decisions. It also focuses on the trade-off to be made between local vertex matching decisions (which have low cost in terms of the space required and time taken) and global decisions (which can be of better quality but have greater costs). The emphasis of our algorithm is mostly on the coarsening phase of the multi-level paradigm as it is the most important phase such that better vertex clustering decisions can provide better partitioning results [Kar02]. Furthermore, we evaluate our algorithm in comparison to the state-of-the-art hypergraph partitioning algorithms on a range of benchmarks from practical applications.

In the first section, we describe some of the problems of multi-level partitioning that motivate our study. Then we define the concept of *Hyperedge Connectivity Graph (HCG)* in the second section. The details of our algorithm are proposed in the third section. Finally, the last section provides simulation results and compares our algorithm to the state-of-the-art sequential hypergraph partitioning algorithms.

4.1 Introduction and Motivations

As discussed in the last chapter, heuristics proposed for multi-level hypergraph partitioning algorithms focus on finding clusters of vertices and merge vertices in each cluster to form coarse vertices in the coarser hypergraph. This requires a metric of similarity (which is also discussed as the connectivity metric in the last chapter), the evaluation of which requires the recognition of *similar* vertices[1]. Some method are described in Chapter 3 in Section 3.1.2 including Edge Coarsening (EC), Hyperedge Coarsening, First Choice (FC), or the connectivity metrics proposed by Alpert et al. [AHK98], Caldwell et al. [AAI06], and Çatalyürek and Aykanat [ÇA99]. These algorithms only define a similarity measure between vertices of the hypergraph and their performance highly depends on the structure of the hypergraph under investigation. The reason is that the structure of hypergraph makes it difficult to define this similarity measure in some practical applications. The reason lies in the fact that finding a good similarity measure in high-dimensional data set is very challenging when there are clusters with different sizes, shapes and densities [ESK03]. This is a case in hypergraphs which are considered to be high-dimensional datasets. While the problem is complicated in graphs with highly skewed vertex degrees [GLG+12], the hypergraph partitioning has an extra degree of complication namely variable hyperedge sizes [HC14].

The object connectivity metric is used for data classification in order to group objects together and build clusters. Objects with the highest connectivity among them are considered as the most similar objects and they are grouped into a cluster. The reason that makes it difficult to define a connectivity metric between objects in high-dimensional data sets is that similarity between objects are very non-uniform. For example, an object may be more similar to another object in different cluster than the objects in its own cluster [ESK03]. This situation happens in graphs and hypergraphs when the mean and standard deviation of vertex degrees are high. Considering *Euclidean distance* as one of those local similarity measures, Ertöz et al. [ESK03] find that it does not give good clustering results when applied to

[1]In the thesis, we use the connectivity metric and the similarity measure interchangeably.

high-dimensional datasets. Although other local similarity measures such as *Cosine measure* and *Jaccard distance* address the issue and resolve the problem to some extent, they are not completely reliable in high-dimensional datasets. For example, Steinbach et al. [SKK00] evaluate these similarity measures and find that they fail to capture similarity between text documents in document clustering techniques which are used in areas such as text mining and information retrieval. Authors report that the problem is not related to the lack of having a good similarity measure, but originates from the lack of trust when measuring similarity among objects in low similarity data space. The Cosine and Jaccard distances emphasise on the importance of existing attributes for measuring the similarity and they ignore the attributes that do not exist or are not common between two objects. Consequently, others move to other clustering techniques to resolve the problem such as *Shared Nearest Neighbour (SNN)* methods [ESK02] and global vertex clustering techniques.

On the other hand, decisions for vertex clustering are made locally and global decisions are avoided due to their high cost and complexity though they give better clustering results [Tri06]. All proposed heuristics reduce the size of the search domain and try to find the vertices to be matched using some degree of randomness [DBH+06]. This degrades the quality of partitioning by increasing the possibility of getting stuck in a local minimum. The quality of these methods are highly dependant on the order the vertices are selected for matching. A better trade-off is needed between the low cost of local decisions and the high quality of global ones.

Furthermore, there are some redundancies in modelling scientific applications with hypergraphs. Removing these redundancies can help in some optimisations such as improving clustering decisions, reducing the storage overhead, and optimising the processing time. An example is in the paper proposed by Heinz and Chandra [HC14] in which the hypergraph is transformed into a Hierarchical DAG (HDAG) representation and they reduce the storage overhead for storing some hypergraphs. Similarly, one can identify the redundancies in the coarsening phase for making better vertex clustering decisions and achieving better partitioning on the hypergraph.

In the algorithm proposed in this chapter, we identify and remove redundancies by using rough set clustering techniques. The algorithm provides a trade-off between

global and local vertex clustering methods. First, it calculates sets of core vertices (a global decision). Then it traverses these cores one at a time and find best matches between the vertices inside each core (local decisions). The proposed algorithm is called the *Feature Extraction Hypergraph Partitioning (FEHG)* algorithm. It is a multi-level partitioning algorithm that obtains *k-way* partitioning through recursive bipartitioning. This is the serial version of the algorithm that is proposed in this chapter. The parallel version is proposed and discussed in the next chapter.

4.2 The Hyperedge Connectivity Graph

Before going through the details of our algorithm, we define the *Hyperedge Connectivity Graph (HCG)* of a hypergraph. HCG is used as our main tool for reducing superfluous and redundant information and making better clustering decisions in the coarsening phase. For this purpose, we need to quantify the similarity between a pair of hyperedges in the hypergraph. The similarity between two hyperedges is represented as $sim(\cdot)$.

Definition 4.1 (Hyperedge Connectivity Graph (HCG)) *For a given similarity threshold* $s \in (0, 1)$*, the* **Hyperedge Connectivity Graph (HCG)** *of a hypergraph* $H = (V, E)$ *is a graph* $\mathcal{G}^s(\mathcal{V}, \mathcal{E})$ *where* $\mathcal{V} = E$ *and two vertices* $v_i, v_j \in \mathcal{V}$ *are adjacent if, for the corresponding hyperedges* $e_i, e_j \in E$ *we have* $sim(e_i, e_j) \geqslant s$*.*

The definition is similar to the definition of the **intersection graph** [EGP66], which is a graph representing the pattern of intersections of a family of sets, and the **line graph** of a hypergraph, which is the graph whose vertex set is the set of the hyperedges of the hypergraph and two hyperedges are adjacent when their intersection is non-empty. The difference is the presence of the similarity function that reduces the number of edges in HCG. Different similarity functions, such as *Jaccard Index* or *Cosine Measure*, can be used for quantifying hyperedge similarity. As the hyperedges of the hypergraph are weighted, similarity between two hyperedges is scaled according to the weight of hyperedges. For two $e_i, e_j \in E$, the scaling factor is $\frac{\gamma(e_i)+\gamma(e_j)}{2\times\max_{e\in E}(\gamma(e))}$. One of the characteristics of the HCG is that it partitions hyperedges into non-overlapping clusters.

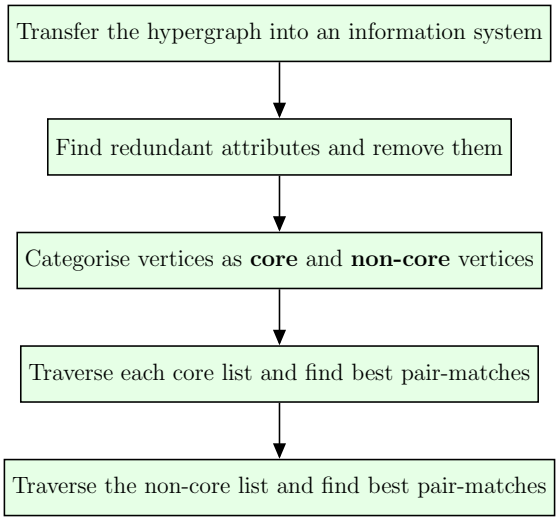


Figure 4.1: The coarsening phase at a glance. The non-core vertex list is processed after all core vertices have been processed.

4.3 The Serial Partitioning Algorithm

As mentioned earlier, *FEHG* is a recursive multi-level serial bipartitioning algorithm that is composed of three distinct phases: coarsening, initial partitioning, and uncoarsening. The emphasis of *FEHG* is on the coarsening phase as it is the most important phases of the multi-level paradigm [Kar02].

The algorithm works as follows. In the coarsening phase, *FEHG* transforms the hypergraph into an information system and uses rough set clustering techniques to find pair-matches of vertices (refer to Chapter 2.3 for rough set theory definitions). This is done in a few steps. First, it finds the **reduct** of the information system which reduces the size of the system and removes superfluous attributes. After the reduction, vertices of the hypergraph are categorised into *core* and *non-core* vertices using the definitions of the rough clustering. Cores are built using global vertex information. Then, cores are traversed one at a time and searched locally to find pair-matches of vertices inside each core. Vertices that are neither assigned to any core nor find a pair-match in core traversals are stored in the non-core vertex list. The non-core vertex list is processed later using a randomised algorithm. The whole coarsening procedure is depicted in Fig. 4.1. The last step of the coarsening phase is the hypergraph contraction in which vertices are merged with their pair-matches and build coarser vertices in the coarser hypergraph.

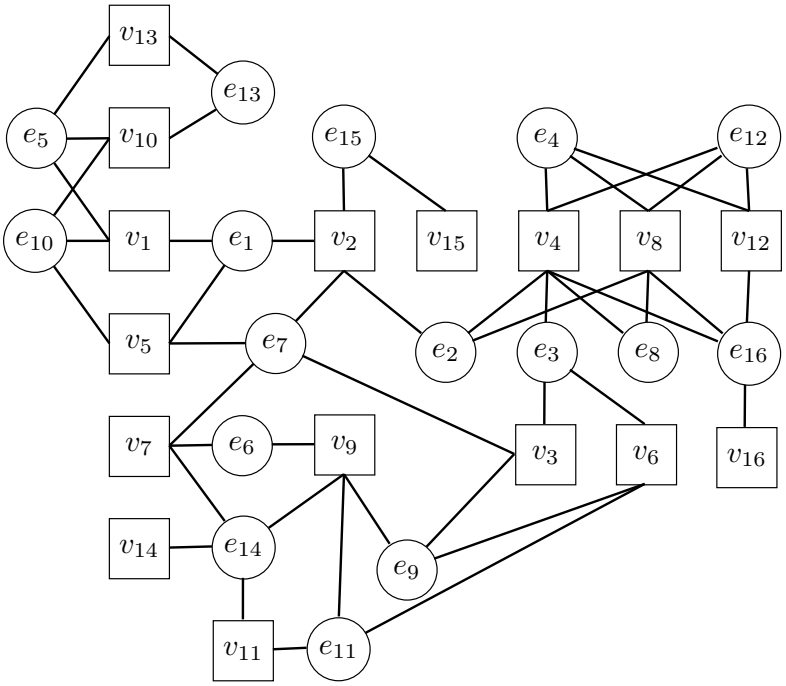


Figure 4.2: A sample hypergraph with 16 vertices and hyperedges. Vertices and hyperedges are represented as square and circular nodes, respectively. The weight of the vertices and hyperedges are assumed to be unit.

The operation is followed by the initial partitioning phase. *FEHG* applies a series of randomised algorithms to obtains the final initial partitioning. This partitioning is then projected back to the original hypergraph through the uncoarsening phase. A variation of the FM algorithm is used for the refinement phase.

We go through the details of each phase in the rest of this section. The hypergraph to be partitioned is denoted as $H(V, E)$ in our formulations for this section.

4.3.1 The Coarsening

The first step of the coarsening phase is to transform the hypergraph H into an information system. The information system representing the hypergraph is denoted as $\mathcal{I}_H = (V, E, \mathbf{V}, \mathcal{F})$ where V is the vertex set or objects, E is the hyperedge set or attributes (also called features in our *FEHG* algorithm), $\mathbf{V}$ is the set of values, and $\mathcal{F}$ is the mapping function. We define the set of values as $\mathbf{V} \in [0, 1]$ and the mapping function is defined as follows:

$$\mathcal{F}(v,e) = \frac{f(e)}{\sum_{\forall e' \triangleright v} \gamma(e')}, \text{ where } f(e) = \gamma(e) \text{ if } e \triangleright v \text{ and is otherwise } 0.$$

Table 4.1: The transformation of the hypergraph depicted in Fig. 4.2 into an information system. The values are rounded to two decimal places.

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	e_{10}	e_{11}	e_{12}	e_{13}	e_{14}	e_{15}	e_{16}
$\mathbf{v_1}$	0.33	0	0	0	0.33	0	0	0	0	0.33	0	0	0	0	0	0
$\mathbf{v_2}$	0.25	0.25	0	0	0	0	0.25	0	0	0	0	0	0	0	0.25	0
$\mathbf{v_3}$	0	0	0.33	0	0	0	0.33	0	0.33	0	0	0	0	0	0	0
$\mathbf{v_4}$	0	0.16	0.16	0.16	0	0	0	0.16	0	0	0	0.16	0	0	0	0.16
$\mathbf{v_5}$	0.33	0	0	0	0	0	0.33	0	0	0.33	0	0	0	0	0	0
$\mathbf{v_6}$	0	0	0.33	0	0	0	0	0	0.33	0	0.33	0	0	0	0	0
$\mathbf{v_7}$	0	0	0	0	0	0.33	0.33	0	0	0	0	0	0	0.33	0	0
$\mathbf{v_8}$	0	0.2	0	0.2	0	0	0	0.2	0	0	0	0.2	0	0	0	0.2
$\mathbf{v_9}$	0	0	0	0	0	0.25	0	0	0.25	0	0.25	0	0	0.25	0	0
$\mathbf{v_{10}}$	0	0	0	0	0.33	0	0	0	0	0.33	0	0	0.33	0	0	0
$\mathbf{v_{11}}$	0	0	0	0	0	0	0	0	0	0	0.5	0	0	0.5	0	0
$\mathbf{v_{12}}$	0	0	0	0.33	0	0	0	0	0	0	0	0.33	0	0	0	0.33
$\mathbf{v_{13}}$	0	0	0	0	0.5	0	0	0	0	0	0	0	0.5	0	0	0
$\mathbf{v_{14}}$	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0	0
$\mathbf{v_{15}}$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1.0	0
$\mathbf{v_{16}}$	0	0	0	0.25	0	0	0	0.25	0	0	0	0.25	0	0	0	0.25

An example of the sample hypergraph depicted in Fig. 4.2 that is transformed into an information system is depicted in Table 4.1. As explained in Chapter 2, the set of attributes in any information system can contain some redundancies and removing these redundancies could lead us to better clustering decisions and data categorisation. The remaining attributes after the reduction is called the **reduct** attribute set as defined in Definition 2.18. The reduct of an information system is not unique. Finding a minimal reduct of an information system is proved to be a NP-hard problem [SR92b]. This is one of the computational bottlenecks of the rough set theory. A number of algorithms have been proposed for problems in which the number of attributes is not high such as the work proposed by Wroblewski [Wró98], and Zirako and Shan [ZS95]. These methods are not applicable on hypergraphs for two reasons. First, the number of hyperedges in hypergraphs representing practical applications can be very high and may contain millions or even billions of hyperedges. Second, finding the reduct set has to be done in the beginning of every coarsening level. As we proceed with the coarsening phase and build approximations of the original hypergraph in each level, the structure of the hypergraph changes and needs recalculation of the reduct set in each coarser hypergraph. Consequently, calculating

the reduct set is very expensive and computationally non-affordable for hypergraphs. As a result, we have to rely on calculating an approximation of the reduct set and this is resolved by calculating the Hyperedge Connectivity Graph (HCG) of the hypergraph.

In order to calculate the HCG, we traverse hyperedges using the Breadth First Search (BFS) algorithm. It builds clusters around a randomly chosen hyperedge. In the beginning of the algorithm, hyperedges are not assigned to any cluster. A queue is built and a random hyperedge is selected and assigned to a new cluster and pushed into the queue. Algorithm runs in iterations and in each iteration of the algorithm the first hyperedge from the head of the queue is extracted. Then, the similarity of the current hyperedge with all adjacent hyperedges, which are not assigned to any cluster, are calculated. The similarity measure is the one which is defined in Definition 4.1. If the similarity of the current hyperedge with an adjacent hyperedge is more than a pre-defined similarity threshold, the adjacent hyperedge is added into the same cluster (as the current hyperedge) and it is added to the end of the queue.

If the queue is empty in an iteration and there are still hyperedges that are not assigned to any cluster, one of them is chosen randomly and it is assigned to a new cluster. This this hyperedge is pushed into the queue and the above operations are repeated. In the end of the HCG calculation algorithm, each hyperedge is uniquely assigned to one cluster. We refer the cluster set as **edge partitions** and denoted as E^{R}. The size and weight of each $e_{\mathrm{R}} \in E^{\mathrm{R}}$ is the number of hyperedges it contains and the sum of their weights, respectively. An example of the HCG for the sample hypergraph in Fig. 4.2 and similarity threshold $s = 0.5$ is depicted in Fig. 4.3.

Hyperedges belonging to the same edge partition are considered to be dependant or similar. All hyperedges belong to the same edge partition are removed from the information system $\mathcal{I}_H$ and replaced by their corresponding edge partition. This operation, builds a new information system which has smaller size compared to $\mathcal{I}_H$. The new information system is represented as $\mathcal{I}_H^{\mathrm{R}}\left(V, E^{\mathrm{R}}, \mathbf{V}^{\mathrm{R}}, \mathcal{F}^{\mathrm{R}}\right)$ in which the attribute set is replaced by E^{R}. In the new information system, the set of values is $\mathbf{V}_{e_{\mathrm{R}}}^{\mathrm{R}} \subseteq \mathbb{N}$. Furthermore, the mapping function for $\forall e_{\mathrm{R}} \in E^{\mathrm{R}}$ is redefined as follows:

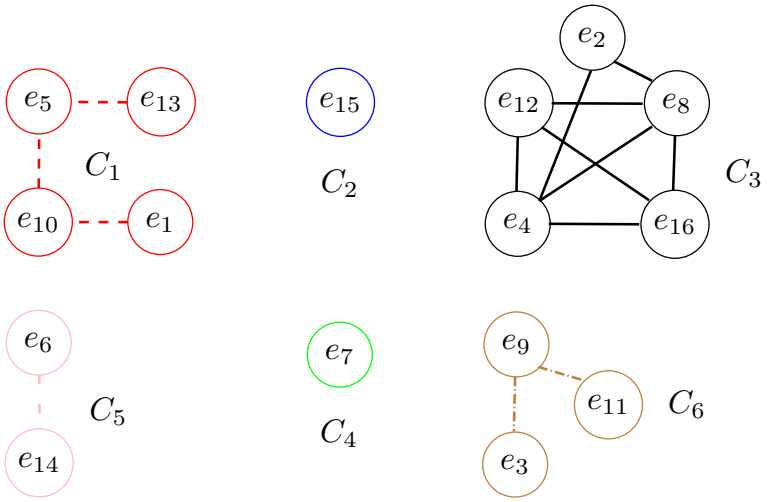


Figure 4.3: An example of Hyperedge Connectivity Graph (HCG) of the hypergraph depicted in Fig. 4.2. The similarity threshold is $s = 0.5$.

$$\mathcal{F}^{\mathrm{R}}(v, e_{\mathrm{R}}) = |\{e \triangleright v \ \wedge \ e \in e_{\mathrm{R}}, \forall e \in E \ \}| \, . \tag{4.1}$$

We can further reduce the set of attributes by going through the second phase of size reduction. For this purpose, we define a **clustering threshold** $c \in [0, 1]$ and the mapping function is changed respectively and according to Eq.(4.2) below to construct the final information system $\mathcal{I}^{\mathrm{f}}$.

$$\mathcal{F}^{f}(v, e_{\mathrm{R}}) = \begin{cases} 1, & \text{if } \frac{\mathcal{F}^{\mathrm{R}}(v, e_{\mathrm{R}})}{|\{e \triangleright v, \forall e \in E\}|} \geqslant c \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

An example of the reduced information system and the final table using the clustering threshold $c = 0.5$ for the sample hypergraph, which is proposed in Fig. 4.2, is depicted in Table 4.2. The final table is very sparse compared to the original table. At this point, we use the definitions of rough set clustering proposed in Chapter 2.3. For every vertex, we calculate its equivalence class as proposed in Definition 2.15. Then, a partitioning $\mathbb{U}/\mathrm{IND}(E^{\mathrm{R}})$ on the vertex set is obtained using the equivalence relations. We refer to parts in $\mathbb{U}/\mathrm{IND}(E^{\mathrm{R}})$ as **cores** such that each vertex belongs to a unique core. For some of the vertices in the hypergraph, the mapping function gives zero output for all attributes that is $\mathcal{F}^{\mathrm{f}}(v, e_{\mathrm{R}}) = 0$, $\forall e_{\mathrm{R}} \in E^{\mathrm{R}}$. These vertices are assigned to a list denoted as **non-core** vertex list.

Table 4.2: The reduced information system that is built based on the HCG in Fig. 4.3 (left) and the final information system when the clustering threshold is set to $c = 0.5$ (right).

	C_1	C_2	C_3	C_4	C_5	C_6
$\mathbf{v_1}$	3	0	0	0	0	0
$\mathbf{v_2}$	1	1	1	1	0	0
$\mathbf{v_3}$	0	0	1	1	0	2
$\mathbf{v_4}$	1	0	3	0	0	0
$\mathbf{v_5}$	2	0	0	1	0	0
$\mathbf{v_6}$	0	0	0	0	0	3
$\mathbf{v_7}$	0	0	0	1	2	0
$\mathbf{v_8}$	0	0	3	0	0	0
$\mathbf{v_9}$	0	0	0	0	2	2
$\mathbf{v_{10}}$	3	0	0	0	0	0
$\mathbf{v_{11}}$	0	0	0	0	1	1
$\mathbf{v_{12}}$	0	0	3	0	0	0
$\mathbf{v_{13}}$	2	0	0	0	0	0
$\mathbf{v_{14}}$	0	0	0	0	1	0
$\mathbf{v_{15}}$	0	1	0	0	0	0
$\mathbf{v_{16}}$	0	0	1	0	0	0

	C_1	C_2	C_3	C_4	C_5	C_6	
$\mathbf{v_1}$	1	0	0	0	0	0	Core 1
$\mathbf{v_5}$	1	0	0	0	0	0	
$\mathbf{v_{10}}$	1	0	0	0	0	0	
$\mathbf{v_{13}}$	1	0	0	0	0	0	
$\mathbf{v_2}$	0	0	0	0	0	0	
$\mathbf{v_3}$	0	0	0	0	0	1	Core 2
$\mathbf{v_6}$	0	0	0	0	0	1	
$\mathbf{v_4}$	0	0	1	0	0	0	Core 3
$\mathbf{v_8}$	0	0	1	0	0	0	
$\mathbf{v_{12}}$	0	0	1	0	0	0	
$\mathbf{v_{16}}$	0	0	1	0	0	0	
$\mathbf{v_9}$	0	0	0	0	1	1	Core 4
$\mathbf{v_{11}}$	0	0	0	0	1	1	
$\mathbf{v_7}$	0	0	0	0	1	0	Core 5
$\mathbf{v_{14}}$	0	0	0	0	1	0	
$\mathbf{v_{15}}$	0	1	0	0	0	0	Core 6

Cores are built using global clustering information. The final operation is to find pair-matches of vertices. Cores are visited one after the other and they are searched locally to find pair-matches. Inside each core, vertices are selected randomly one at a time and the best pair-match among its adjacent vertex list is selected; only adjacent vertices that belong to the same core is considered for finding the best match. As a result, we need a local vertex connectivity metric for finding pair-matches. We use the *Weighted Jaccard Index* defined as follows:

$$J(u,v) = \frac{\sum_{\{e \triangleright v \wedge e \triangleright u\}} \gamma(e)}{\sum_{\{e \triangleright v \vee e \triangleright u\}} \gamma(e)}, \quad v,u \in V, \text{ and } \forall e \in E. \tag{4.3}$$

This is similar to *Non-weighted Jaccard Index* in *PaToH*, which is called *Scaled Heavy Connectivity Matching*. As we mentioned in Section 4.1, this measure captures similarities in high-dimensional datasets better than Euclidean-based similarity measures. Vertices that do not find any pair-matches in core searches are transferred into the non-core vertex list, such as v_{15} in Table 4.2.

One of the issues that may happen is when the number of vertices in cores constitute a small percentage of the whole number of vertices of a hypergraph. This situation happens, for example, when the average vertex degree of vertices in

a hypergraph is high such that we end up with big dominator in Eq. (4.2)[2]. As explained in Chapter 3.1.2 regarding the multi-level partitioning algorithm, we can provide a trade-off between the quality and runtime of the algorithm by controlling the compression ratio in Eq. (3.1) between two successive coarsening levels. Referring to that discussion, and to satisfy a certain compression ratio, we process the non-core vertex list in the next step. The non-core list is traversed and vertices are selected randomly one at a time. For every selected vertex, the algorithm finds a pair-match among its unmatched adjacent vertices in the non-core list.

When pair-matches are found, the hypergraph is contracted to build a coarser hypergraph for the next coarsening level. This is done by merging matched vertices. The weight of a coarser vertex is the sum of the weight of two merged vertices and its incident hyperedges are the union of the hyperedges incident on both merged vertices. After building the coarser hypergraph, we perform two final operations on the hyperedge list. First, hyperedges of unit size are removed as they do not have any impact on the partitioning cut. Second, identical hyperedges, i.e. those having the same vertex set, are identified and removed from the coarser hypergraph. The is similar to the same strategy which is applied in both *Parkway* and *Zoltan*. Similarly, this is done using hash functions. Hyperedge are hashed to integer values based on their vertex list. Two hyperedges with the equal hash values are considered as identical. If conflicts occur, the whole content of hyperedges are compared.

4.3.2 Initial Partitioning and Refinement

The coarsening phase stops when the number of the vertices in the hypergraph is small enough. We stop coarsening when the coarser hypergraph has fewer than 100 vertices. Any partitioning on the coarsest hypergraph can be calculated very quickly in much less time compared to the original hypergraph. We use a series of algorithms for this purpose. The partitioning that gives the balance constraint and gives the minimum partitioning cost is selected and it will be projected back to the original hypergraph. Among the algorithms depicted in Chapter 3.1.2, the *FEHG* algorithm

[2]Or when the similarity between vertices is very low and we end up with a very small nominator in Eq. (4.2 for most of vertices

Table 4.3: Evaluated hypergraphs for sequential algorithm simulation and their specifications

Hypergraph	Description	Rows	Columns	Non-Zeros
CNR-2000	Small web crawl of Italian CNR domain	325,557	325,557	3,216,152
AS-22JULY06	Internet routers	22,963	22,963	96,872
CELEGANSNEURAL	Neural Network of Nematode C. Elegans	297	297	2,345
NETSCIENCE	Co-authorship of scientists in Network Theory	1,589	1,589	5,484
PGPGIANTCOMPO	Largest connected component in graph of PGP users	10,680	10,680	48,632
GUPTA1	Linear Programming matrix ($A \times A^T$)	31,802	31,802	2,164,210
MARK3JAC120	Jacobian from MULTIMOD Mark3	54,929	54,929	322,483
NOTREDAME	Barabasi's web page network of nd.edu	325,729	325,729	929,849
PATENTS_MAIN	Pajek network: mainNBER US Patent Citations	240,547	240,547	560,943
STD1_JAC3	Chemical process simulation	21,982	21,982	1,455,374
COND-MAT-2005	Collaboration network, www.arxiv.org	40,421	40,421	351,382

uses Random assignment, Linear assignment, and FM-based approaches.

As explained previously, partitioning algorithms refine the partitioning cut as the hypergraph is projected back in the uncoarsening phase. Due to the success of the FM algorithm in practice, we use a variation of the FM algorithm known as Early-Exit FM (FM-EE) [Kar02] and Boundary FM (BFM)[3] [ÇA11]. Furthermore, the v-cycle refinement is avoided because of its high cost. It is also unnecessary. As stated by Karypis [Kar02], a good coarsening algorithm needs less effort in the refinement phase which is a case for the *FEHG* algorithm.

4.4 Experimental Evaluations

In this section, we provide the evaluation of our algorithm compared to the state-of-the-art partitioning algorithms[4] including *PHG* [San14b] which is *Zoltan* hypergraph partitioner, *PaToH* [ÇA11] and *hMetis* [Kar07]. All of these algorithms are multi-level recursive bipartitioning algorithms. Except *PHG*, which is a parallel hypergraph partitioner, the other two are serial partitioning tools.

For the evaluation, we have selected a number of test hypergraphs from a variety of scientific applications with different specifications. The hypergraphs are obtained from the University of Florida Sparse Matrix Collection [DH11]. It is a large database of sparse matrices from real applications. Each sparse matrix from the database is assumed as the hypergraph incident matrix with the vertices and hyperedges

[3]Reader is referred to Chapter 3.1.2 for details about these algorithms.
[4]Reader is referred to Chapter 3.2 for details about these hypergraph partitioning tools.

representing the rows and columns of the matrix, respectively. This is similar to the column-net model proposed by Çatalyürek et al. [ÇA99]. The weight of vertices and hyperedges are assumed to be unity. The list of test data used for our evaluation is depicted in Table 4.3. The reader is referred to Appendix A for the full specifications of these hypergraphs.

The simulations are done on a computer with Intel(R) Xeon(R) CPU E5-2650 2.00GHz processor, 8GB of RAM and 40GB of disk space and the operating system running on the system is 32-Bit Ubuntu 12.04 LTS. Furthermore, we set the imbalance tolerance to 2% and the number of parts are $\{2, 4, 8, 16, 32\}$. The final imbalance achieved by the algorithms are not reported because the balance constraint is always met by all algorithms.

4.4.1 Algorithm Parameters

Each algorithm in the above mentioned evaluated tools has different input parameters that can be set by the user such as those for the coarsening, initial partitioning, and refinement phases. We use default parameters for each tool. All algorithms use a variation of FM algorithm (FM-EE and BFM) in their refinement phase. *PHG* uses the *agglomerative* coarsening algorithm as the default coarsening method that is based on the *inner product* as the measure of similarity between the vertices. This also being used in *hMetis* and *Mondriaan*. This is a variation of Euclidean inner product between the vertices based on their incident hyperedges and their weights [DBH+06]. The default partitioning tools for *hMetis* is *shmetis*[5]. The default coarsening scheme is *Hybrid First Choice (HFC)* scheme that is a combination of the *First Choice* and *Greedy First Choice* schemes[6]. *PaToH* is initialised by setting the SBProbType parameter to PATOH_SUGPARAM_DEFAULT. It uses the *Absorption Clustering* using pins as the default coarsening algorithm that is an agglomerative vertex clustering scheme. The similarity metric, which is known as *Absorption Metric*,

[5]The *hMetis* tool also has another partitioner that is called *khmetis* and it is a direct *k-way* multi-level hypergraph partitioner.

[6]This is a variation of *First Choice* described in Chapter 3.1.2. Vertices are grouped based on the *First Choice* algorithm in which the grouping is biased in favour of faster reduction in the number of the hyperedges in the coarser hypergraph.

of two vertices v and u is calculated as follows:

$$\sum_{\{\forall e \in E | v \in e \text{ and } u \in e\}} \frac{1}{|e| - 1}.$$

The algorithm accumulates the absorption metric for every pin that connects u and C_v. C_v is the cluster that vertex v is already assigned to. The reader is referred to the manuals of these tools for the full description of the parameters [San14b,ÇA11,KK98b].

The *FEHG* algorithm has two parameters that need to be set: *similarity threshold* in Definition 4.1 for building the HCG and the *clustering threshold* in Eq. (4.2). In this section, we discuss the automatic calculation of these two parameters.

Clustering Coefficient (CC) is a graph theory measure calculated by the degree to which a vertex clusters with other vertices of graph or hypergraph. Methods to calculate CC are categorised as local and global measures. The local measures are usually applied for calculating the density of neighbourhood of vertices in a graph or hypergraph and they capture local density. On the other hand, global measures are used to calculate the overall clustering tendency in the network. CC has a value in $[0, 1]$. The calculation of CC is more difficult in hypergraphs compared to graphs. There are different measures proposed for calculating the CC in hypergraphs including the works proposed by Klamt et al. [KHT09], Latapy et al. [LMDV08], and Gavin et al. [GBK+02]. In these works, the calculation of CC between two vertices is based on the intersection and union of their incident hyperedges. In addition, the weight of the hyperedges is not a case in those applications and the weight of the all hyperedges are assumed to be unit. In our hypergraph partitioning problem, the CC of the hypergraph needs to be calculated for hyperedges instead of vertices. Second, the weight of the hyperedges should be considered [7]. Given a hypergraph $H = (V, E)$, we define CC for a hyperedge $e \in E$ as follow.

[7] Although this is not a concern for designing our serial algorithm, the third requirement is that we need a fast and scalable calculation of CC in our parallel *FEHG* algorithm that is proposed in Chapter 5. Therefore, we consider this fact in our calculations.

$$\mathrm{CC}(e) = \begin{cases} \frac{\sum_{\{e' \cap e \neq \emptyset\}} \left(\left(\frac{|e \cap e'|}{|e|-1} \right) \cdot \gamma(e') \right)}{\sum_{\{v \in e\}} \sum_{\{e'' \rhd v\}} \gamma(e'')}, \forall e', e'' \in E \backslash e, & \text{if } |e| > 1 \\ 0, & \text{otherwise.} \end{cases} \tag{4.4}$$

The CC of the hypergraph is calculated as the average of CC over all its hyperedges as follows.

$$CC_H = \sum_{e \in E} \frac{CC(e)}{|E|}. \tag{4.5}$$

As we proceed to the next coarsening level, the structure of the hypergraph changes and it changes the value of CC in the coarser hypergraph. Therefore, we need to recalculate the CC in in the beginning of each coarsening level, which could be a costly operation. To avoid this, we are interested in re-adjusting CC values. Foudalis et al. [FJPS11] study the structure of social network graphs and they identify several characteristic metrics including the clustering coefficient. Beside the fact that social networks present a high clustering coefficient compared to random networks, they report that the CC is inversely related to the degree of vertices. Two vertices with low vertex degrees are more likely to cluster to each other than two vertices with higher vertex degrees. In addition, Bloznelis [Blo13] theoretically investigates random intersection graphs[8] and they show that the clustering coefficient is inversely related to the average vertex degree in the graph. Based on these results, we readjust the value of CC from one coarsening level to the next successive level based on the inverse of the average vertex degree. Finally, the similarity threshold is set to be the CC of the hypergraph at the beginning of the coarsening phase. In the next section, where we propose our evaluation results, we investigate how our algorithm performs, in terms of partition quality and running time, when the similarity threshold is calculate using different methods: readjusting the CC in each coarsening level versus recalculating it in the beginning of each coarsening level.

[8]Random intersection graphs can be obtained from randomly generated bipartite graphs with vertex set $V \cup W$. Each vertex v_i in $V = \{v_1, v_2, \cdots v_n\}$ selects a set $D_i \subset W$ as its neighbours randomly and independently such that the elements of W have equal probability to be selected. Considering the fact that a hypergraph is bipartite graph, the results of the paper are applicable on hypergraphs.

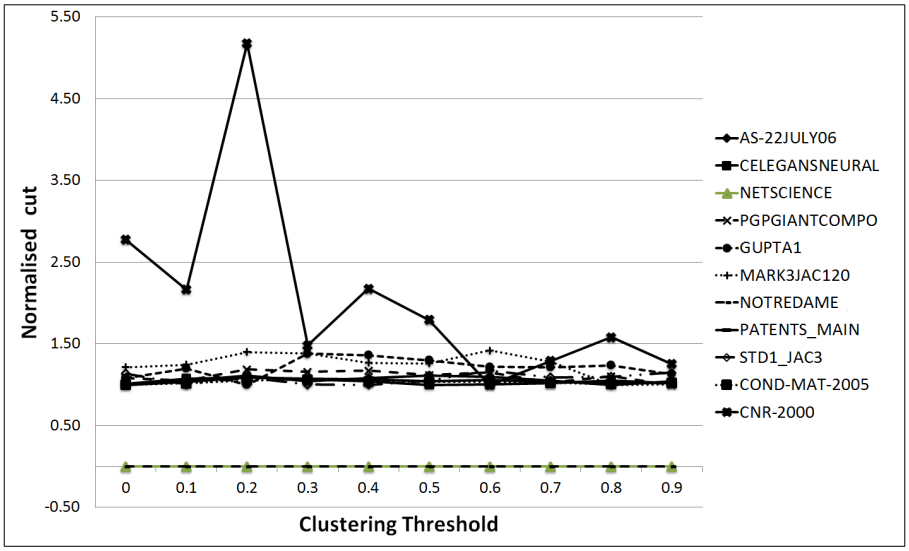


Figure 4.4: The variation of bipartitioning cut based on the clustering threshold for some of the tested hypergraphs. Values are normalised with the best cut for each hypergraph. According to the results, there i weak correlation between the partitioning quality and the clustering threshold.

Regarding the automatic calculation of the clustering threshold, we first check how the clustering threshold affect the quality of the partitioning. Figure 4.4 depicts the quality of the *2-way* partitioning of the hypergraphs in Table 4.3 and variable clustering threshold values. The cut is normalised based on the best partitioning cut for each hypergraph. The average correlation between the partitioning cut and the clustering threshold over all hypergraphs is 0.2442; the average correlation value decreases to 0.2096 when we exclude `CNR-2000`. This shows a weak correlation. The standard deviation of the changes with respect to the average cut values is also less than 4.2% over all hypergraphs. Therefore, changing the clustering threshold has a very small effect on the partitioning quality. An exception occurs for the `CNR-2000` hypergraph such that the variation of change is very high. Hyperedges with larger CC values (closer to 1) are those that more probably cluster with other hyperedges and those with small CC values (close to 0) do not form any cluster and build edge partitions in the HCG of size unity. Consequently, we can remove every edge partition in HCG of unit size and set the value of clustering threshold to zero in Eq. (4.2). As an example, edge partitions C_2 and C_4 can be removed from the reduced information system in Table 4.2 without causing any changes in the cores. Using this strategy, the partitioning cut for `CNR-2000` is 29.9% better than the best

bipartitioning cut reported in Fig. 4.4 that is achieved for the $c = 0.6$.

Table 4.4: Quality comparison of the algorithms for different part sizes and 2% imbalance tolerance. The values are normalised according to the minimum partitioning cut for each hypergraph; therefore, the algorithm that gives 1.0 cut value is considered to be the best. Unit weights are assumed for both vertices and hyperedges.

		Number of Parts									
		2		4		8		16		32	
		AVE	**BEST**	**AVE**	**BEST**	**AVE**	**BEST**	**AVE**	**BEST**	**AVE**	**BEST**
AS-22JULY06	FEHG	1.11	1.00	1.02	1.00	1.04	1.01	1.01	1.00	1.01	1.03
	PHG	2.90	2.46	1.77	1.56	1.64	1.36	1.43	1.34	1.37	1.32
	hMetis	1.34	1.95	1.19	1.30	1.16	1.18	1.04	1.06	1.09	1.04
	PaToH	1.00	1.43	1.00	1.03	1.00	1.00	1.00	1.00	1.00	1.00
	Min Value	136	93	355	319	629	599	1051	995	1591	1529
CELEGANSNEURAL	FEHG	1.00	1.00	1.09	1.00	1.10	1.06	1.11	1.08	1.07	1.03
	PHG	1.07	1.00	1.04	1.03	1.02	1.00	1.06	1.00	1.00	1.00
	hMetis	1.17	1.21	1.00	1.05	1.00	1.04	1.00	1.02	1.00	1.00
	PaToH	1.01	1.04	1.00	1.06	1.03	1.07	1.03	1.06	1.05	1.05
	Min Value	79	77	195	184	354	342	548	536	773	769
CNR-2000	FEHG	1.37	1.00	1.71	1.07	1.59	1.41	1.53	1.45	1.63	1.51
	PHG	35.88	45.62	12.48	9.17	5.73	4.84	3.54	2.98	2.42	2.02
	hMetis	12.19	18.82	8.24	8.43	5.08	4.71	3.46	3.29	2.66	2.50
	PaToH	1.00	1.71	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Min Value	81	45	244	202	569	509	1014	911	1927	1830
COND-MAT-2005	FEHG	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.02	1.01	1.00
	PHG	1.17	1.17	1.11	1.10	1.05	1.05	1.03	1.03	1.02	1.01
	hMetis	1.05	1.07	1.11	1.12	1.11	1.12	1.11	1.10	1.01	1.01
	PaToH	1.02	1.02	1.03	1.03	1.00	1.00	1.00	1.10	1.00	1.00
	Min Value	2134	2087	5057	4951	8609	8485	12370	12150	16270	16150
NETSCIENCE*	FEHG	0	0	0	0	2	1.50	1.50	1.00	2.08	1.81
	PHG	0	0	0	0	1.50	1.00	1.40	1.00	1.87	1.5
	hMetis	2.0	2.0	5.0	5.0	4.22	3.50	1.75	1.75	1.99	1.87
	PaToH	0	0	0	0	1.00	1.00	1.00	1.00	1.00	1.00
	Min Value	0	0	0	0	2	2	8	8	16	16
PGPGIANTCOMPO	FEHG	2.12	1.27	1.00	1.00	1.04	1.00	1.00	1.08	1.00	1.00
	PHG	13.23	1.83	1.44	1.04	1.25	1.04	1.02	1.00	1.08	1.00
	hMetis	9.7	9.61	1.46	1.71	1.04	1.40	1.31	1.40	1.26	1.27
	PaToH	1.00	1.00	1.04	1.27	1.00	1.04	1.02	1.15	1.08	1.06
	Min Value	18	18	242	200	419	400	695	617	956	930
GUPTA1	FEHG	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	PHG	1.58	1.45	1.31	1.24	1.15	1.04	1.07	1.04	1.09	1.05
	hMetis	1.73	1.82	1.61	1.69	1.58	1.64	1.60	1.57	1.51	1.48
	PaToH	1.22	1.17	1.08	1.09	1.04	1.05	1.05	1.07	1.08	1.09
	Min Value	486	462	1466	1384	3077	2893	5342	5134	8965	8519
MARK3JAC120	FEHG	1.01	1.01	1.02	1.01	1.01	1.00	1.00	1.00	1.06	1.07
	PHG	1.00	1.01	1.02	1.02	1.02	1.00	1.00	1.00	1.72	1.78
	hMetis	1.00	1.00	1.00	1.02	1.00	1.00	1.30	1.00	4.20	1.78
	PaToH	1.00	1.02	1.00	1.00	1.00	1.00	1.26	1.20	1.00	1.00
	Min Value	408	400	1229	1202	2856	2835	6317	6245	3142	2944
NOTREDAME*	FEHG	0	0	1.00	1.00	1.12	1.12	1.09	1.03	1.06	1.07
	PHG	4326	4326	158.56	288.69	13.82	16.78	2.09	3.06	1.72	1.78
	hMetis	880	707	67.92	129.92	10.98	12.65	3.36	3.37	2.23	2.30
	Patoh	24	22	1.90	3.31	1.00	1.00	1.00	1.00	1.00	1.00
	Min Value	0	0	27	13	316	259	1577	1484	3142	2944
PATENTS-MAIN	FEHG	1.20	1.00	1.03	1.01	1.05	1.03	1.00	1.00	1.00	1.00
	PHG	12.49	13.19	2.52	2.30	1.79	1.65	1.42	1.38	1.23	1.18
	hMetis	2.38	2.77	1.16	1.24	1.26	1.43	1.26	1.31	1.21	1.22
	PaToH	1.00	1.02	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00
	Min Value	643	528	3490	3198	6451	6096	11322	10640	16927	16460
STD1-JAC3	FEHG	1.01	1.00	1.00	1.03	1.00	1.00	1.00	1.00	1.00	1.00
	PHG	1.15	1.08	1.16	1.10	1.18	1.13	1.28	1.35	1.33	1.29
	hMetis	1.05	1.00	1.52	1.03	1.54	1.23	1.70	1.53	1.71	1.51
	Patoh	1.00	1.00	1.08	1.00	1.16	1.14	1.00	1.26	1.30	1.29
	Min Value	1490	1371	3735	3333	7616	6167	13254	11710	22242	21200

* When the minimum cut for the average or best cases are zero, the values shown are actual cut values rather than normalised values.

Table 4.5: Comparing the Standard Deviation (STD) of the partitioning cut for algorithms reported in Table 4.4. Unit weights are assumed for both vertices and hyperedges. The values are reported for 20 runs for each algorithm.

		Number of Parts				
		2	**4**	**8**	**16**	**32**
AS-22JULY06	FEHG	34	32	25	30	28
	PHG	86	92	78	87	90
	hMetis	0	7	12	23	27
	PaToH	4	16	20	37	43
CELEGANSNEURAL	FEHG	2	9	15	16	17
	PHG	6	8	9	12	18
	hMetis	0	5	0	2	6
	PaToH	0	0	0	0	0
CNR-2000	FEHG	63	131	226	218	217
	PHG	552	760	569	477	530
	hMetis	74	163	240	238	231
	PaToH	3	37	48	62	85
COND-MAT-2005	FEHG	28	58	87	88	82
	PHG	37	84	94	112	105
	hMetis	14	75	81	129	122
	PaToH	39	193	98	153	178
NETSCIENCE	FEHG	0	0	1	2	2
	PHG	0	0	1	2	2
	hMetis	0	0	1	0	2
	PaToH	0	0	0	0	0
PGPGIANTCOMPO	FEHG	8	23	18	16	18
	PHG	48	65	45	53	46
	hMetis	3	11	13	24	25
	PaToH	0	0	7	2	5
GUPTA1	FEHG	60	55	80	115	15
	PHG	67	146	204	253	58
	hMetis	2	10	58	137	643
	PaToH	32	43	84	95	120
MARK3JAC120	FEHG	6	18	23	83	132
	PHG	4	15	27	53	106
	hMetis	13	15	29	217	214
	PaToH	0	11	17	248	267
NOTREDAME	FEHG	0	9	40	116	119
	PHG	0	124	67	75	78
	hMetis	84	65	108	143	129
	Patoh	1	8	27	52	62
PATENTS-MAIN	FEHG	180	275	270	327	342
	PHG	1286	1736	1749	1575	1602
	hMetis	36	70	115	161	231
	PaToH	70	145	217	220	306
STD1-JAC3	FEHG	260	246	424	549	557
	PHG	227	377	748	768	801
	hMetis	105	1649	2057	2330	2995
	Patoh	125	506	700	827	945

4.4.2 Comparison Results

In the first evaluation, we assume unit weight for both vertices and hyperedges of the hypergraph. In this situation, a partitioning algorithm performs well if it can capture strongly connected components of the hypergraph. A strongly connected component is a group of the vertices that are tightly coupled together. In graphs, a strongly connected component is a clique. Because the weight of all hyperedges are unit, vertex connectivity is an important factor for generating high quality partitionings. The aim of the partitioning algorithm is to take these strongly connected components out of the cut as they are the major cause of increasing the partitioning cut. A partitioning algorithm that identifies those components and merges their vertices to build a coarser vertex is the one that will give better partitioning quality. In addition, the clustering algorithm that captures those strongly connected components in the first few levels of coarsening would likely obtain competitive partitioning qualities.

Each algorithm is run 20 times and the average cut is reported in Table 4.4 as well as the best partitioning cut among all runs. The results in the table are normalised with respect to the minimum partitioning cut among all algorithms. For example, *FEHG* gives the minimum average cut for a bipartitioning on `CELEGANSNEURAL` hypergraph that is 79. *PHG*, *hMetis* and *PaToH* give 1.07, 1.17, and 1.01 times worse average bipartitioning cut, respectively. The results shows that *FEHG* performs very well compared to *PHG* and *hMetis* and it is competitive with *PaToH*. As it can be seen from the results, all algorithms give close partitioning cut when the hypergraph has only few number of strongly connected components. In this situation, even the local clustering algorithms can capture strongly connected components and merge their vertices; therefore, the differences in partitioning cut for different algorithms that are using different clustering methods (either global or local) is very small.

As the number of strongly connected components increases, it gets hard to identify them especially when the boundary between these components are not clear and have overlaps. This situation happens in hypergraphs such as `Notredame`, `Patents-Main`, and `CNR-2000`. As it is shown, *FEHG* achieves a superior quality improvement compared to *PHG* and *hMetis*, but *PaToH* still generates very good partitioning

with absorption clustering using pins. One reason that may explain this situation is that *PaToH* allows matching between a group of vertices instead of pair-matching. Therefore, the algorithm can merge strongly connected components of vertices in the very first few levels of coarsening. Although *hMetis* allows multiple matches, but its seems that the agglomerative clustering strategy of *PaToH* is doing much better compared to the hybrid first choice algorithm in *hMetis*.

The standard deviation (STD) of the average cut is reported in Table 4.5. The standard deviation shows the reliability of a partitioning algorithm when it is used in practical applications. As each of the evaluated tools has a degree of randomness, there are variations of the partitioning cut in each run of algorithms. We are interested in a partitioning algorithm that generates partitioning cuts with small variations among runs. The results show that *FEHG* and *PaToH* are the most reliable algorithms with small variations among others while the worst values are reported for *hMetis* and *PHG*. The standard deviation of the cut increases as we increase the number of parts. This is due to the recursive bipartitioning nature of the algorithms. The standard deviation of i^{th} recursion is the sum of standard deviation values for all previous $(i-1)^{th}$ recursions plus the standard deviation of the current recursion.

In some practical applications such as parallel distributed systems, the hypergraph partitioning is employed to reduce the communication volume between processors. In this situation, the weight of hyperedges represent the volume of communications between a group of vertices. For these prolems, the objective of the hypergraph partitioning is to reduce the number of messages communicated between processors as well as the volume of messages[9]. In order to model this scenario, we set the weight of the hyperedges to be their sizes. Despite all, the main reason for this simulation is that we want to investigate the performance of the clustering algorithms in multi-level

[9]If we assume vertices as tasks of a parallel application, the weight of a vertex shows the amount of computational effort each processor spend for the vertex (in a homogeneous system, while the processing time for each vertex might be different on each processor in a heterogeneous system). In our scenario, we assume that the computation time spent for processing all vertices is the same (unit vertex weights) and the aim is to reduce both the number and the volume of communications. An example of this situation is in large scale vertex-centric graph processing tools such as Pregel [MAB+10].

hypergraph partitioning tools when there are weights on the hyperedges.

When hyperedges have different weights, vertex connectivity is no longer the only measure for making clustering decisions. Compared to the previous scenario, taking a group of strongly connected components of vertices can not always result in cut reduction. The vertex connectivity as well as how tightly vertices are connected to each other are both important for making good clustering decisions. The simulation results for this scenario are depicted in Fig. 4.5. In the evaluation results for *FEHG*, we calculate the similarity threshold in the beginning of bipartitioning recursion and its value is readjusted according to the inverse of the average vertex degrees.

According to the results, the *FEHG* algorithm gives the best partitioning cut on most of evaluated hypergraphs. We identify three different types of hypergraphs in our dataset and they are categorised into three groups. The first group includes hypergraphs with very irregular structure and high variations of vertex degrees and hyperedges sizes such as `CNR-2000`, `GUPTA1`, `Notredame`, `AS-22JULY06`, and `STD-JAC3`. For this group, the *FEHG* algorithm gives the best partitioning qualities compared to other algorithms. This shows that the *FEHG* algorithm with its rough set clustering technique is the best candidate for these types of hypergraphs (those with very irregular structure). Hypergraphs, which represent social networks, are of this type.

The second group contains hypergraphs with less irregularity such as `COND-MAT-2005`, `PGPGIANTCOMPO`, and `CELEGANSNEURAL`. These hypergraphs have less variable vertex degrees or hyperedge sizes than the first group. Again, *FEHG* gives the best partitioning results on these types of hypergraphs, but the difference between partitioners, except *hMetis*, are small. On these types of hypergraphs, we can get reasonable partitioning quality using local partitioners and the performance of the algorithm is highly dependent on the vertex similarity measure, for example, the one proposed by *hMetis* gives the worst quality.

The third group are those with regular structure and much smaller variability of vertex degrees and hyperedge sizes than the other two groups. We have three hypergraphs of this type: `NETSCIENCE`, `PATENTS-MAIN`, and `MARK3JAC120`. The evaluations show that the quality of *FEHG* is worse than the other algorithms.

In case of NETSCIENCE that has a very small size, all algorithms go through only one level of coarsening. The difference between the cuts is less than 50 which is a very small number. Due to regular structure of the hypergraphs in this group, local vertex matching decisions give much better results than the global vertex clustering algorithms. We notice that our rough set clustering algorithm identifies very small cores for these hypergraphs. The overall number of vertices assigned to cores constitutes a small fraction of the whole number of vertices of hypergraphs. As most of the vertices end up in the non-core vertex list, the quality of the *FEHG* algorithm mostly depends on the random local matching decisions which are based on the *Jaccard Index* similarity measure. It seems that the *Jaccard Index* similarity measure does not perform well compared to the other partitioners. The agglomerative vertex matching algorithm of *PHG* gives the best quality results on this group of hypergraphs.

The results show that *PaToH*, which was very competitive with our algorithm in the first scenario with unit weight on all hyperedges, generates much worse partitioning results compared to *FEHG* in this evaluation scenario (with non-unit weights on hyperedges). This shows that *FEHG* is better than *PaToH* and it is more reliable because it produces very good partitioning results than the other algorithms in both evaluation scenarios. The evaluations show that *PaToH* and ,then, *hMetis* generates the worst partitioning qualities. Some of the partitioning results are not reported for *hMetis* because the algorithm terminates with an internal error on some of the hypergraphs and for specific values of partition numbers. We could not figure out where is the problem, but maybe the reason is that *hMetis* suits, more or the less, to the partitioning on unit hyperedge size like the one in the VLSI circuit partitioning. Assuming non-unit hyperedge weights generates an unexpected errors in the algorithm.

The running times of the algorithms are reported in Table 4.6. The ranking of algorithms in decreasing order of their running time is *hMetis*, *FEHG*, *PHG*, and *PaToH*. We should consider the fact that *PHG* and *FEHG*, which is designed as a part of *Zoltan*, are parallel algorithms and the reported times include some overhead of the parallel code while *hMetis* and *PaToH* are serial hypergraph partitioner and

do not include any parallel code overhead.

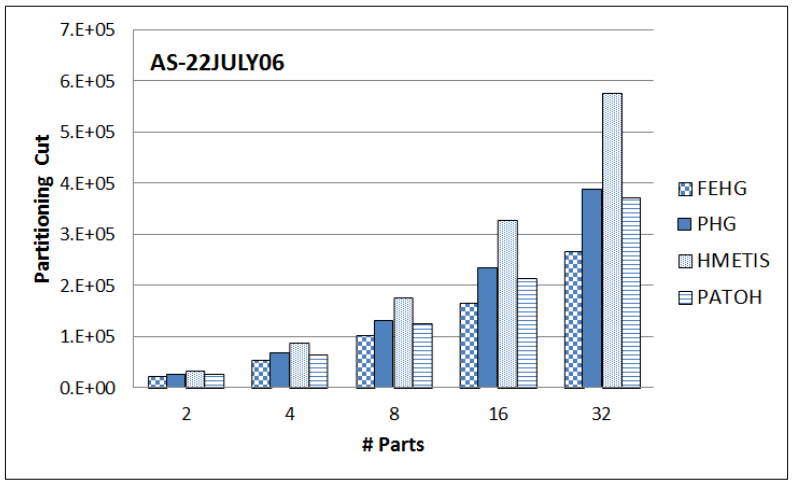
As the *FEHG* algorithm does give good runtime results compared to *PHG* and *PaToH*, we were investigating some solutions to improve the runtime. We found that allowing multiple matches of the vertices can provide better partitioning quality compared to the pair-matching. In addition, it can improve the runtime of the *FEHG* algorithm. The reason is that it provides a faster reduction in hypergraph sizes as we go through the coarsening levels. We have tested our algorithm to see the effects of multiple matching on the performance of our algorithm. For our evaluations, we assume non-unit hyperedge weights in which the weight of each hyperedge is set to its size. In our multi-match strategy, all vertices of a core are merged together to form a vertex in the coarser hypergraph (as rough set clustering generates non-overlapping clusters, this multi-match strategy does not generate any conflicts). The only limitation is that we do not allow the weight of a coarser vertex to exceed the size of a part because it makes it difficult to maintain the balance constraint. The evaluation shows that using multi-match in our algorithm can improve the runtime by up to 7% and the maximum improvement is observed for `CNR-2000` that is up to 30% improvement in runtime.

In another evaluation, we evaluate the effect of the clustering coefficient calculations on the *FEHG* algorithm. We investigate how readjusting the clustering coefficient in each coarsening level affects the quality and performance of our algorithm. Two situations are considered: in the first situation, the CC of the hypergraph is calculated in every coarsening level, and in the second situation, the CC of the hypergraph is readjusted based on the inverse of the average vertex degrees. Evaluations show that the first situation does not improve the partitioning quality on all hypergraph; for example, the quality of the third types of the hypergraphs described above are diminished by 1% in average. The best quality improvement is for `CNR-2000` that is 6% and it was between 0.2% to 1.5% on other hypergraphs. On the other hand, the runtime of the algorithms is up to 16% higher on average for the first situation compared to the second. This shows that readjusting the similarity threshold works better in term of quality and runtime compared to recalculating it in the beginning of each coarsening level.

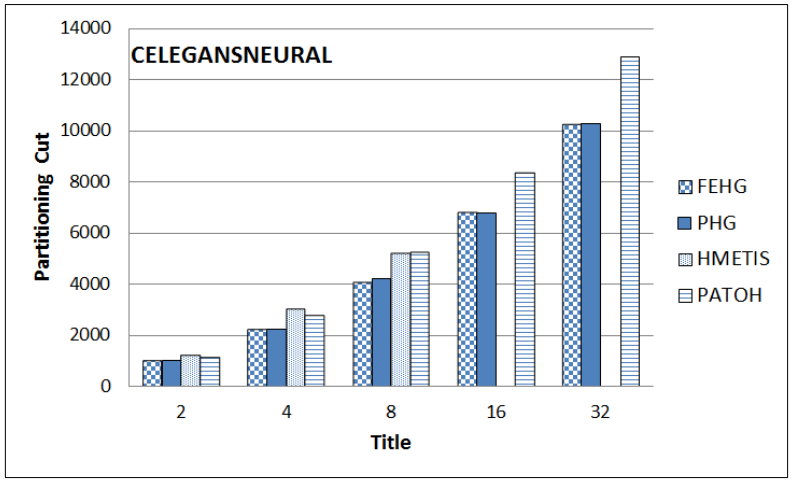
Finally, the detailed running time of the *FEHG* and the amount to time the algorithm spends in each step of the algorithm is depicted in Table 4.7. The results are reports for *2-way*, *8-way*, and *32-way* partitioning on some of the hypergraphs. In the table, the whole runtime is depicted in the first row. *Build* is the time for building data structures and preparation time, *recursion* is recursive bipartitioning time, *Vcycle* is the amount of time for reduction and hypergraph projection in the multi-level paradigm[10], *HCG* is for building HCG, *matching* includes the time for rough set matching algorithm. Finally, *coarsening, initPart* and *refinement* are the time taken for building the coarser hypergraph in the coarsening phase, initial partitioning and uncoarsening phases of *FEHG*.

The most time consuming operation in the *FEHG* algorithm is building HCG that is around 27% of the whole partitioning time. The rough set clustering takes only 13% of the runtime. Building the coarser hypergraph, and the initial partitioning each takes around 20% of the whole runtime. One can reduce the initial partitioning time by decreasing the number of algorithms in this phase. If the number of hyperedges is much higher than the number of vertices, HCG build runtime constitute a large fraction of the overall runtime. According to the data, we can optimise and reduce the runtime of building HCG by applying faster algorithms and using better data structures; this is planned as a future work. The refinement phase takes at most 6% of the whole running time. As the *FEHG* algorithm puts most of its effort in the coarsening phase in order to generate high quality partitionings, it needs less effort in the refinement phase. Increasing the number of passes of the FM algorithm does not make considerable improvement to the partitioning cut. Furthermore, as the time for the refinement phase is small compared to the whole partitioning time, increasing the number of passes of the FM algorithm does not provide considerable runtime improvement.

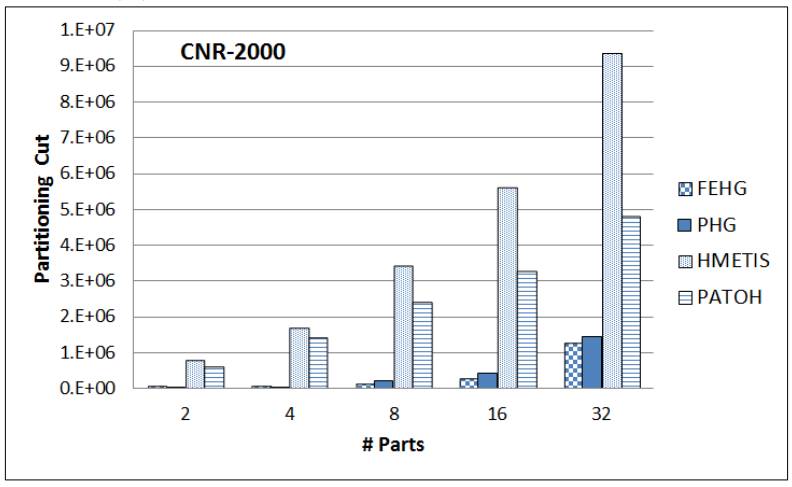
[10]The time stands for *V-cycle* with capital V letter.



(a)

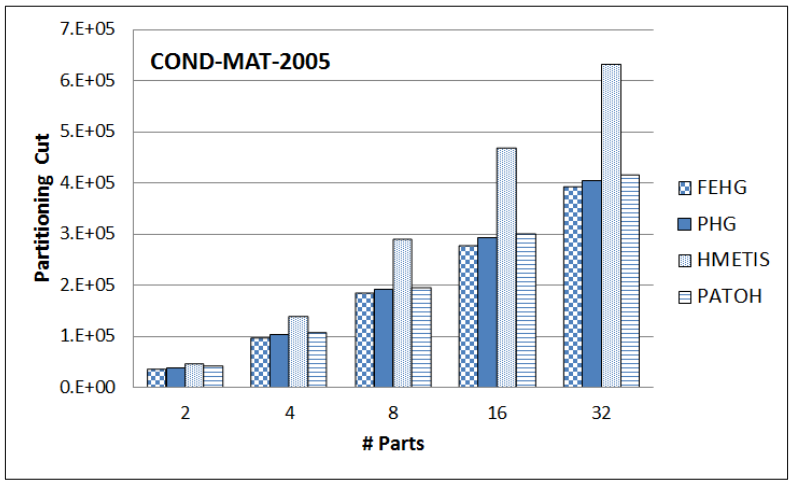


(b) Error on *hMetis* for *16-way* and *32-way*.

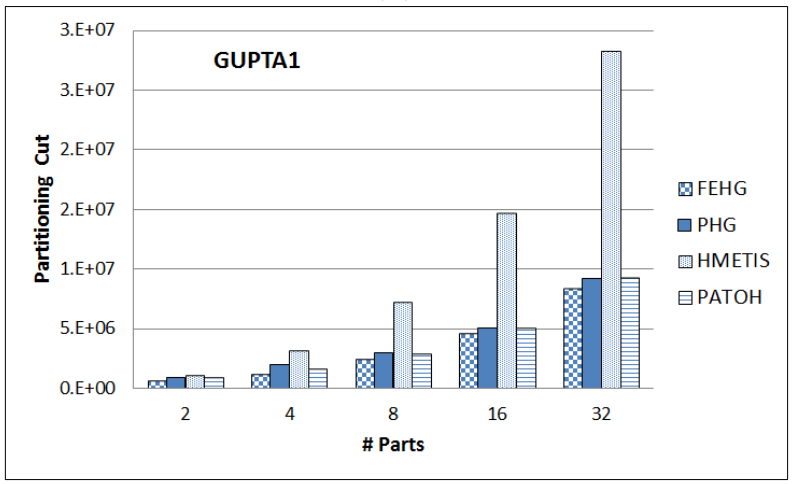


(c)

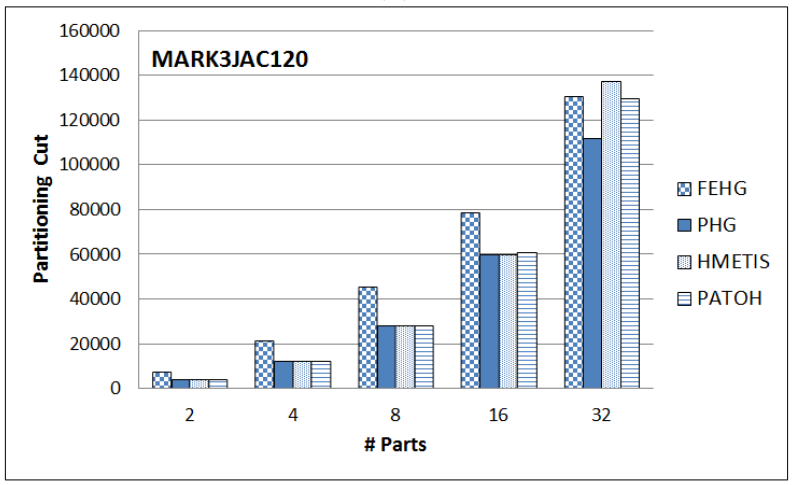
Figure 4.5: Comparing the cut variation for different partitioning numbers. The weight of vertices are unit and the weight of hyperedges are their sizes.



(d)

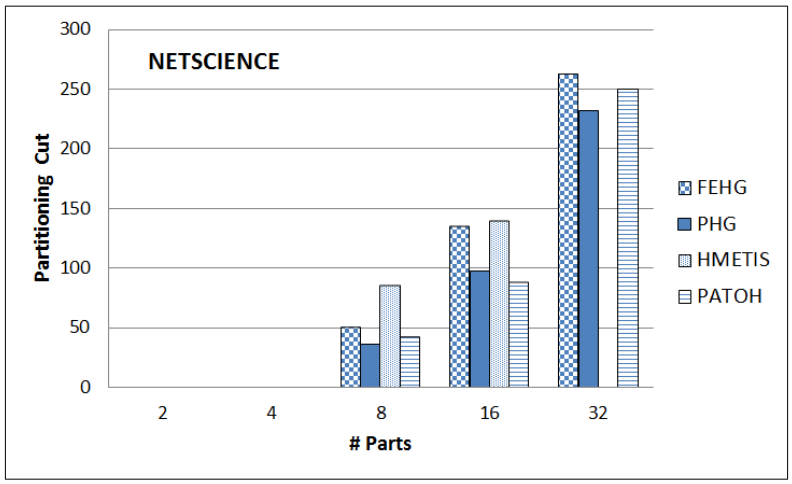


(e)

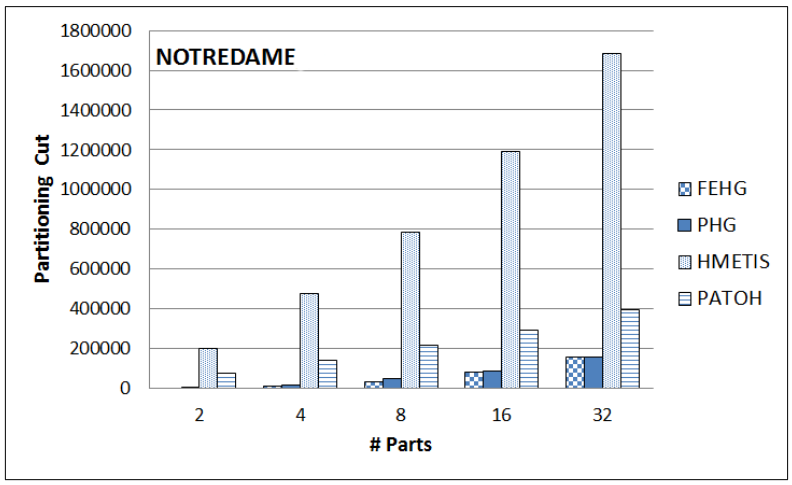


(f)

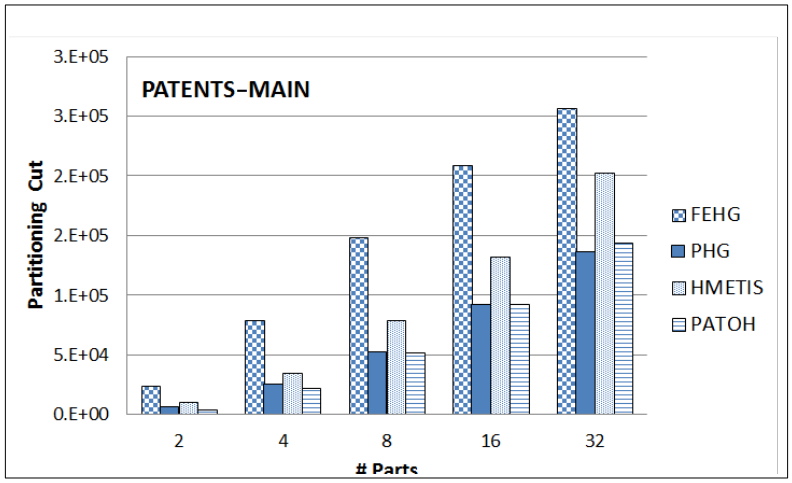
Figure 4.5: (Continued) Comparing the cut variation for different partitioning numbers. The weight of vertices are unit and the weight of hyperedges are their sizes.



(g) Zero cut on *2-way* and *4-way* partitioning and error on *hMetis* on *32-way*.

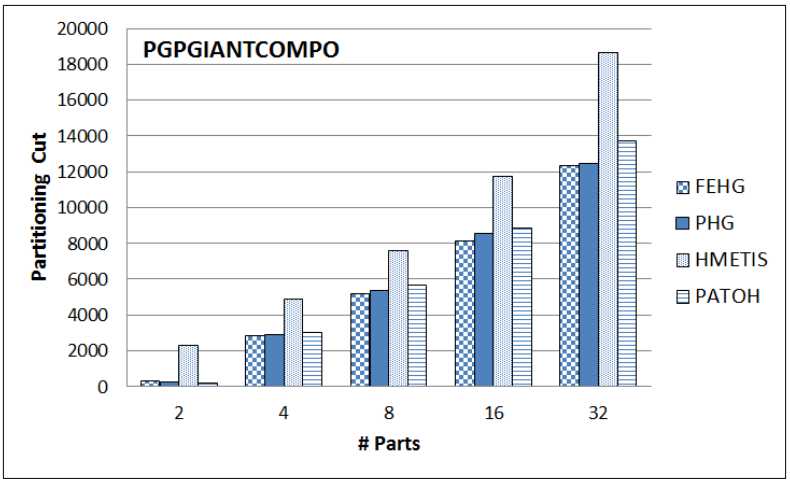


(h)

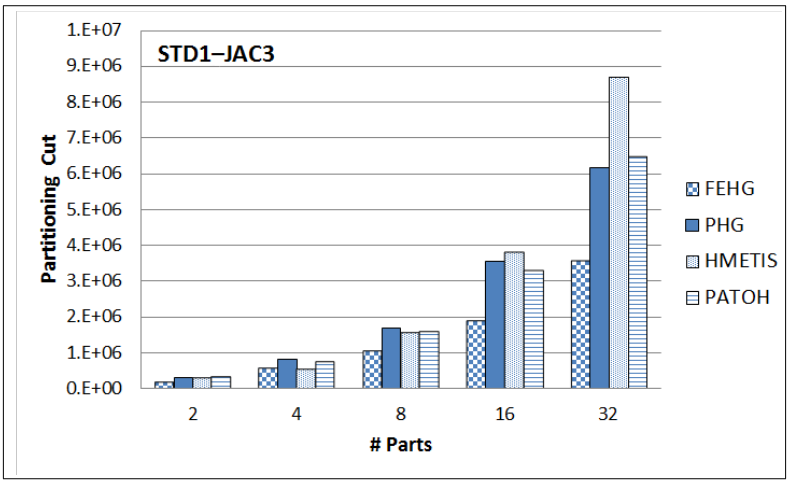


(i)

Figure 4.5: (Continued) Comparing the cut variation for different partitioning numbers. The weight of vertices are unit and the weight of hyperedges are their sizes.



(j)



(k)

Figure 4.5: (Continued) Comparing the cut variation for different partitioning numbers. The weight of vertices are unit and the weight of hyperedges are their sizes.

Table 4.6: Comparing the running time of the partitioning algorithms for variable number of parts. Vertices have unit weights and hyperedge weights are equal to their size. The times are reported in milliseconds.

		Number of Parts				
		2	**4**	**8**	**16**	**32**
AS-22JULY06	FEHG-ADJ	109	210	308	412	523
	PHG	157	274	413	522	634
	hMetis	126	344	803	1370	5902
	PaToH	82	212	336	422	514
CELEGANSNEURAL	FEHG-ADJ	8	15	21	27	33
	PHG	4	7	19	25	22
	HMETIS	12	18	32	–	–
	PATOH	4	4	6	8	12
CNR-2000	FEHG-ADJ	19480	30570	39720	50140	57560
	PHG	3035	5202	7317	9267	11060
	HMETIS	22590	41680	50990	61190	68850
	PATOH	2004	3960	6000	8084	10390
COND-MAT-2005	FEHG-ADJ	643	1137	1612	2210	2772
	PHG	318	535	750	954	1178
	HMETIS	3800	7038	9930	13740	20020
	PATOH	162	284	370	500	584
NETSCIENCE	FEHG-ADJ	5	10	17	27	34
	PHG	4	6	10	22	32
	HMETIS	–	–	14	20	–
	PATOH	2	2	4	4	8
PGPGIANTCOMPO	FEHG-ADJ	114	224	325	408	491
	PHG	44	57	89	114	147
	HMETIS	170	234	354	452	544
	PATOH	12	20	32	46	62
GUPTA1	FEHG-ADJ	1843	3014	4020	4918	6095
	PHG	937	1853	2648	3453	4285
	HMETIS	994	4066	11990	43000	331000
	PATOH	914	2140	3544	5370	7298
MARK3JAC120	FEHG-ADJ	708	1304	1913	2546	3192
	PHG	318	588	891	1204	1592
	HMETIS	1748	4570	7010	9410	11130
	PATOH	128	272	416	604	796
NOTREDAME	FEHG-ADJ	1588	4071	6487	9095	11130
	PHG	2129	3673	5054	6203	7207
	HMETIS	5442	12770	17190	23270	28060
	PATOH	632	1262	1950	2626	3316
PATENTS-MAIN	FEHG-ADJ	1933	3187	4430	5860	7514
	PHG	1274	2156	2919	3610	4251
	HMETIS	11850	24080	32860	38580	42630
	PATOH	396	734	1024	1340	1648
STD1-JAC3	FEHG-ADJ	4970	12270	19610	26710	32630
	PHG	1116	2005	2775	3451	4033
	HMETIS	4086	11480	19610	57300	175500
	PATOH	1720	3884	5372	8380	10830

Table 4.7: The time that the *FEHG* algorithm spends in different partitioning steps. Times are reported in seconds.

Parts		*AS-22JULY06*	*CELEGANSNEURAL*	*NETSCIENCE*	*PGPGIANTCOMPO*	*NOTREDAME*	*PATENTS_MAIN*	*STD1_JAC3*	*COND-MAT-2005*
2	Overall	0.1461	0.0080	0.0059	0.0831	1.5562	1.9312	7.3650	0.6155
	Build	0.0230	0.0003	0.0014	0.0118	0.4568	0.3496	0.4181	0.0697
	Recursion	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
	Vcycle	0.0036	0.0000	0.0003	0.0025	0.0048	0.0604	1.7775	0.0203
	HCG	0.0224	0.0034	0.0007	0.0257	0.0000	0.4512	3.6833	0.2475
	Matching	0.0352	0.0000	0.0009	0.0071	0.0000	0.2275	0.1238	0.0638
	Coarsening	0.0332	0.0000	0.0013	0.0181	0.0000	0.4377	1.1945	0.1108
	InitPart	0.0190	0.0040	0.0003	0.0124	1.0467	0.3019	0.0466	0.0748
	Refinement	0.0086	0.0000	0.0007	0.0051	0.0286	0.0868	0.1072	0.0260
8	Overall	0.3722	0.0218	0.0167	0.2456	6.2414	4.9619	18.2084	1.7036
	Build	0.0235	0.0009	0.0006	0.0113	0.5750	0.3474	0.4182	0.0700
	Recursion	0.0161	0.0008	0.0026	0.0081	0.2712	0.2091	0.4072	0.0562
	Vcycle	0.0095	0.0009	0.0004	0.0088	0.1453	0.1625	4.8115	0.0604
	HCG	0.0514	0.0023	0.0049	0.0727	1.7675	1.1641	8.7172	0.6657
	Matching	0.0903	0.0005	0.0012	0.0184	0.7289	0.5581	0.3300	0.1749
	Coarsening	0.0838	0.0013	0.0043	0.0470	1.3447	1.0932	3.0456	0.3349
	InitPart	0.0650	0.0116	0.0002	0.0553	1.1754	1.1883	0.1084	0.2346
	Refinement	0.0309	0.0033	0.0024	0.0230	0.2041	0.2150	0.3463	0.1017
32	Overall	0.6258	0.0360	0.0363	0.4007	9.8867	7.6629	28.5887	2.7925
	Build	0.0233	0.0009	0.0010	0.0110	0.4578	0.3456	0.4173	0.0699
	Recursion	0.0331	0.002	0.0027	0.0157	0.5302	0.3908	0.8082	0.112
	Vcycle	0.0156	0.0013	0.0024	0.0159	0.2789	0.2547	9.6070	0.0992
	HCG	0.0776	0.0023	0.0029	0.1059	2.9710	1.8239	11.8124	0.9619
	Matching	0.1317	0.0004	0.0028	0.0292	1.1691	0.8517	0.4540	0.2597
	Coarsening	0.1267	0.0012	0.0054	0.0771	2.5044	1.7197	4.3973	0.5536
	InitPart	0.1372	0.0230	0.0133	0.0905	1.4581	1.8681	0.3006	0.4734
	Refinement	0.0772	0.0045	0.0053	0.0535	0.4734	0.3724	0.7559	0.2545

Chapter 5

Parallel Hypergraph Partitioning Algorithm

The rapid growth of size and complexity of scientific applications makes these applications unsuitable for standalone computer systems. For example, graphs and hypergraphs generated by social networks such as Facebook and Twitter have billions of vertices and interconnections among them [HC14] and it is not possible to fit them into the computational capacity and memory of one computer. This fact highlights the need for a parallel and scalable hypergraph partitioning algorithm. There are two objectives for parallel partitioners: 1) the quality that should not worsen as the number of processors increases. No parallel algorithm generates partitioning qualities better than serial algorithms. The reason lies behind the data locality such that processors have less local data to access as the system scales and, consequently, the quality suffers; 2) the scalability, which means that the algorithm should get improved speedup as the number of processors increases. In order to measure scalability in the thesis, we compare the speedup of algorithms. The speedup is defined as the ratio of partitioning time on one processor to the time required to solve the partitioning on the parallel system. An algorithm is considered to be more scalable if the speedup improvement lasts longer as we increase the number of processors.

In this chapter, we propose our *Parallel Feature Extraction Hypergraph Partitioning (PFEHG) algorithm*. The algorithm is the parallel version of the *FEHG* algorithm proposed in Chapter 4. Similar to the *FEHG* algorithm, our parallel

algorithm follows the multi-level paradigm and obtains a *k-way* partitioning by recursive bipartitioning of the hypergraph. As discussed in Chapter 3.1.4, the two phases that are difficult to parallelise in the multi-level paradigm are the coarsening and uncoarsening phases. The performance of these two phases depend highly on the way the hypergraph is distributed among processors such that a bad distribution can generate high network traffic and limit the scalability and quality by interfering with vertex matching decisions.

In our parallel algorithm, we propose a parallel vertex matching algorithm for the coarsening phase based on parallel rough set clustering techniques. It defines a similarity measure for finding similar vertices (the same similarity measure as the *FEHG* algorithm, which is discussed in Chapter 4.3.1, is used) and provides a trade-off between local and global vertex matching. Following the discussion in Chapter 3.1.4, the refinement phase is the most challenging phase to parallelise as the proposed refinement algorithms are inherently serial. We propose a modified parallel FM refinement algorithm that is based on processor synchronisation.

In the rest of this chapter, we go through the details of the *PFEHG* algorithm. The algorithm starts by distributing the hypergraph on the processor set which is discussed in Section 5.1. Section 5.2 proposes our parallel coarsening, initial partitioning, and parallel FM refinement algorithms. The *PFEHG* algorithm uses two special processor reconfiguration in each bipartitioning recursion that are also discussed in this section. The simulation results and the performance comparison to the state-of-the-art hypergraph partitioner in the *Zoltan* tool, which is called *PHG*, are reported in Section 5.3. The *PHG* algorithm is the most recent parallel hypergraph partitioning algorithm that shows good scalability in HPC clusters. Evaluations are done in our HPC cluster in Durham University.

Finally, the application of the hypergraph partitioning goes beyond scientific applications and there are lots of cloud applications that can considerably benefit from hypergraph partitioning and load balancing. Partitioning these applications before running them in the cloud can provide better utilisation of the limited network resources of the cloud and will provide performance improvement and scalability. Consequently, Section 5.4 investigates the parallel hypergraph partitioning problem

in the cloud and we evaluate the scalability and speedup of our *PFEHG* algorithm compared to the *PHG* algorithm in the *Zoltan* tool.

In this chapter, we represent the input hypergraph to be partitioned as $H(V, E)$ and the number of processors of the parallel system is p. Furthermore, we assume a *k-way* partitioning problem on the input hypergraph.

5.1 Hypergraph Distribution

Deciding on the initial distribution of the hypergraph $H = (V, E)$ on p processors is a significant challenge. A bad distribution may impose excessive inter-processor communication overhead in the coarsening and uncoarsening phases which makes the partitioning algorithm non-scalable. Following the discussion in Chapter 3.1.4, the previously proposed parallel hypergraph partitioning algorithms use two common ways for the initial distribution of the input hypergraph on p processors as follows:

- The one-dimensional (1D) approach is to equally distribute both vertices and hyperedges on processors so that each processor stores $|V|/p$ vertices and $|E|/p$ hyperedges. *Parkway* [TK08] applies this strategy.

- The two-dimensional (2D) approach, which is used by *Zoltan* [DBH+06], arranges processors in a logical grid with p_x *rows* and p_y *columns* and $p = p_x \times p_y$. Typically $p_x \approx \sqrt{p}$. The vertex set is distributed on p_x processors and each processor holds $|V|/p_x$ vertices. Similarly, hyperedges are distributed on p_y processor and each processor holds $|E|/p_y$ hyperedges. Each processor holds a subblock of the hypergraph that provides a Cartesian distribution of the hypergraph on the processor set. Most communications are reduced to row or column communications.

In the 1D distribution, hyperedges with vertices on different processors are called *frontier* hyperedges. This method is liable to excessive communication overheads as processors may regularly access data stored on other processors. *Parkway* resolves the issue by replicating frontier hyperedges at the start of each coarsening phase on the processor set. An issue is that even if there is no frontier hyperedge, communications

may occur because there is no guarantee that all hyperedges incident on a vertex are present locally and they might be stored on other processors; therefore, some hyperedges still need to be communicated in the beginning of the coarsening phase. Because the number of frontier hyperedges can dramatically increase when the number of processors increases and depending on the structure of the hypergraph, the performance of this method is limited by its memory requirements [DBH+06].

The idea for the 2D distribution emerged from the parallel graph partitioning algorithm proposed by Karypis and Kumar [KK98c]. In their algorithm, the matching decision is made by $\sqrt{p}$ diagonal processors and this creates a bottleneck and restricts the scalability of the algorithm to $O\left(\sqrt{p}\right)$. The developers of *Zoltan* believe that the 2D distribution fits more into the hypergraph partitioning context than the graph partitioning context. Although the 2D distribution can alleviate the communication overhead by limiting most communication to row or column processors, some calculations need the whole vertex or hyperedge set to be communicated among column or row processors. This may incur considerable network overhead. In addition, some global decisions, such as finalizing matching decisions for the vertices, are delegated to the processors on the master row (row 0). This may create a bottleneck. Although evaluating the scalability of *Zoltan* is shown to be much better compared to the 1D distribution, this might not scale well on all types of hypergraphs. On the other hand, this distribution does not fit into the cloud. The reason is that that network resources of the cloud are very limited compared to HPC clusters. We evaluate the performance of the 2D distribution in the cloud in Section 5.4.

There is also a problem with unbalanced distribution of the input hypergraph on the processors in both distributions. Although vertices and hyperedges are distributed evenly among the processors in the 1D distribution, the replication of the frontier hyperedges creates an imbalanced distribution of hypergraph pins on the processor set; this causes imbalanced load among processors. The degree of imbalance depends on the structure of the hypergraph and how hyperedge contents are distributed among processors. This imbalance is not only the computation imbalance but also creates communication imbalance because processors with more pins may participate in more communications when interchanging its vertex and

hyperedge sets. In case of 2D distribution, which uses Cartesian distribution of the hypergraph on the processor set, some subblocks might be denser compared to others and have more pins. Considering the fact that the communication pattern in 2D distribution is somehow independent of the pin numbers on the processor set (a vertex/hyperedge should participate in row/column communication even it has no pin on some processors), the 2D distribution suffers from imbalanced computations rather than imbalanced communications.

PFEHG follows a different 1D strategy, a combination of the above two distribution, and distributes the input hypergraph in such a way that the number of pins assigned to processors is equal. A *redistribution imbalance* $\phi_d \in (0,1)$ is defined and the number of pins assigned to each processor is limited to $\left[\overline{pins} \cdot (1-\phi_d), \overline{pins} \cdot (1+\phi_d)\right]$ where $\overline{pins} = \left(\frac{pins(H)}{p}\right)$. The algorithm is provided with a hypergraph in any arbitrary distribution as an input. After reading the input, *PFEHG* redistributes the hypergraph using a collection of hash functions. Every vertex and hyperedge is assigned a globally unique *ID*. The processors concurrently hash vertices to the processor set using their global *IDs*. The hash function that satisfies the *redistribution imbalance* criteria and gives maximum number of internal hyperedges (a hyperedge is internal if all of its vertices are assigned to a processor) is selected for the initial distribution of the input hypergraph.

The hash functions used for the distribution are controlled via a user defined input parameter. One can choose a specific hash function or a collection of them. Examples are: 1) RS: Robert Sedgwick's Algorithm [Sed02], 2) JS: bitwise hash function written by Justin Sobel, 3) PJW: Peter J. Weinberger of AT&T Bell Labs [ASU86]. The complete list of the hash functions supported by the *PFEHG* algorithm is proposed in Appendix B. We do not use ghosting (replication of the frontier hyperedges on several processors like *Parkway*) and the only additional data saved for hyperedges are their global *ID* and size, a list that shows the processors on which a hyperedge is distributed (each list is a bit vector of size p. If a hyperedge has pins on processor i, the i'th bit of the vector is one; otherwise, it is zero), and a lookup table for the fast lookup of hyperedges present on each processor.

Consequently, each processor has a sub-hypergraph of the original hypergraph which none of them share a pin. This means that each pin of the hypergraph is uniquely owned by one processor. The processor with rank i holds a sub-hypergraph $H_i(V_i, E_i)$ where V_i's are non-overlapping vertex sets and $V = \cup_{i=0}^{p-1} V_i$. E_i is the hyperedges incident on all vertices in V_i such that no two processors share a pin that is $E_i \cap E_j = \emptyset, \forall 1 \leqslant i \neq j < p$.

According to our distribution, we define some terms that will be used in the rest of the chapter. In our terminology, a hyperedge all of whose vertices are assigned to one processor is called *internal*; otherwise it is *external* (or frontier as in the terminology of *Parkway*). A hyperedge and a processor are said to be connected if the hyperedge has at least one vertex on the processor. The internal $d_i(v)$ and external $d_e(v)$ degree of a vertex $v \in V$ is the number of internal and external hyperedges incident to v, respectively. We have $d(v) = d_i(v) + d_e(v)$. If $d(v) = d_i(v)$, the vertex is called *internal*; otherwise it is *external*. Furthermore, internal and external connectivity of a vertex is the sum of the weight of the internal and external hyperedges incident to the vertex, respectively.

5.2 The Parallel Algorithm

Coarsening provides a sequence of successively smaller hypergraphs $H_i = (V_i, E_i), 0 \leqslant i \leqslant c$, where the original hypergraph $H = H_0$ and $|V_i| < |V_j|$ when $i > j$. The size of H_c is small enough that it can fit into the memory and computation capacity of a single processor. Hypergraph reduction is done by means of vertex matching. The coarsening finds matching candidates for every vertex in the hypergraph H_i at the i'th coarsening level. Then matched vertices are merged into coarser vertices in H_{i+1}.

Similar to the coarsening procedure proposed for *FEHG* in Chapter 4.3, the *PFEHG* coarsening algorithm works as follows. We transform the hypergraph into an information system and use rough set clustering techniques to find pair-matches of vertices. This is done in several steps. First, we find the reduct of attributes to reduce the size of the information system. After reduction, vertices of the hypergraph are categorised as *core* and *non-core* vertices using rough set clustering

techniques. A nice property of the rough set clustering is that this categorisation can be done by processors for their local vertices independently. Cores are built based on global information and distributed among the processors such that each processor is responsible for a number of cores. Processors traverse their core vertex lists one at a time and pair-matches are found for each vertex. Vertices that are neither assigned to a core nor find a match are assigned to the non-core list and they are processed later using a parallel randomised algorithm. Processors decide about non-core vertex matchings together by going through a number of iterations. Match conflicts are resolved by a collective all-to-all processor communication. In the end of the coarsening phase, the hypergraph is contracted. Matched vertices are merged to construct the coarser hypergraph and the coarsening proceeds to the next level. We go through the details of each step in the rest of this section.

5.2.1 Parallel Attribute Reduction

Following the discussion proposed for *FEHG* and the rough set clustering definitions, we need to calculate the *reduct* on the information system representing the hypergraph (Definition 2.18). The set of attributes in any information system can contain some redundancies. Removing these redundancies can provide better clustering decisions. Finding the reduct of an information system is a NP-hard problem and it is a bottleneck of the rough set clustering [SR92b]. Following the discussion in Chapter 4.3.1, the problem is resolved by introducing the Hyperedge Connectivity Graph (HCG) in Definition 4.1. HCG provides a partitioning of the hyperedges, which are called *edge partitions*, such that hyperedges in the same edge partition are considered to be dependant or similar. Replacing dependant hyperedges with their representatives reduces the size of the information system and there are less information than the original hypergraph to decide about vertex matchings.

Sequential calculation of $\mathcal{G}^s(\mathcal{V}, \mathcal{E})$ for hypergraph $H(V, E)$ can be done using Breadth-First search. In each traversal of BFS, we need to calculate the intersection between two hyperedges in order to calculate their similarity. Consequently, a parallel BFS algorithm does not solve our problem because some of hyperedges information are not available locally (some hyperedges are split among several processors and

Algorithm 2 Parallel Hyperedge Connectivity Graph (HCG) algorithm

Require: Processor rank r, sub-hypergraph $H_r(V_r, E_r)$, number of processors p, and the number of rounds *rounds*

1: **procedure** ParallelHCG(r, $H_r(V_r, E_r)$,p,*rounds*)
2: $EP_r \leftarrow$ LocalAggClustering($H_r(V_r, E_r)$) ▷ call local agglomerative clustering function on hyperedges
3: $EP = \bigcup_{i=0}^{p-1} EP_i$
4: Build bipartite graph $B(X, Y, \mathcal{U})$ with $X = E$, $Y = EP$.
5: Assign $X_r = |X|/p$ and $Y_r = |Y|/p$ vertices to processor r
6: Assign global IDs to X vertices in lexical order.
7: **for all** $y \in Y_i$ **do** ▷ build a spanning forest on Y
8: Put y in a unique tree and set y as the root
9: **for all** $x \in X_r$ **do**
10: $priority(x) \leftarrow$ the global *ID* of x
11: $n(x) \leftarrow$ sorted list of neighbours of x in Y
12: **for** $i \leftarrow 1$ **to** *rounds* **do**
13: **for all** $x \in X_i$, $|n(x)| > 1$ **do**
14: $rep \leftarrow n(x)[0]$
15: **for** $j \leftarrow 0$ **to** $|n(x)| - 1$ **do**
16: Build tuple $\langle x, priority(x), rep, n(x)[j] \rangle$ ▷ $\langle target, \cdots, destination \rangle$
17: Send tuples to processors that hold the destination vertices as in Figure 5.1
18: **for all** $y \in Y_i$ **do**
19: Collect received tuples
20: *representative* $\leftarrow$ *rep* field of the tuple with the highest priority
21: Update the *rep* field of each tuple to the *representative*
22: Merge trees containing y and the *representative*
23: Return tuples to the processors holding the source vertex
24: **for all** $x \in X_i$, $|n(x)| > 1$ **do**
25: Update $n(x)$ list and remove duplications
26: **if** $|n(x)| = 1, \forall x \in X$ **then**
27: Break
28: Call FindRoots(Y): Find roots in the spanning forest on Y
29: $E^R \leftarrow$ neighbours of vertex set X
30: Return E^R

the *PFEHG* algorithm does not use ghosting or replication of hyperedges). We rather need a parallel fast intersection algorithm for this purpose. Our parallel algorithm is proposed in Algorithm 2. Processors calculate local *HCG* for their own sub-hypergraph (refer to Section 5.1 for hypergraph distribution) based solely on their local information and using either an agglomerative clustering algorithm or a local BFS traversal. In each iteration of the local algorithm, we need to calculate the intersection and union between a set of hyperedges. In our implementation, we

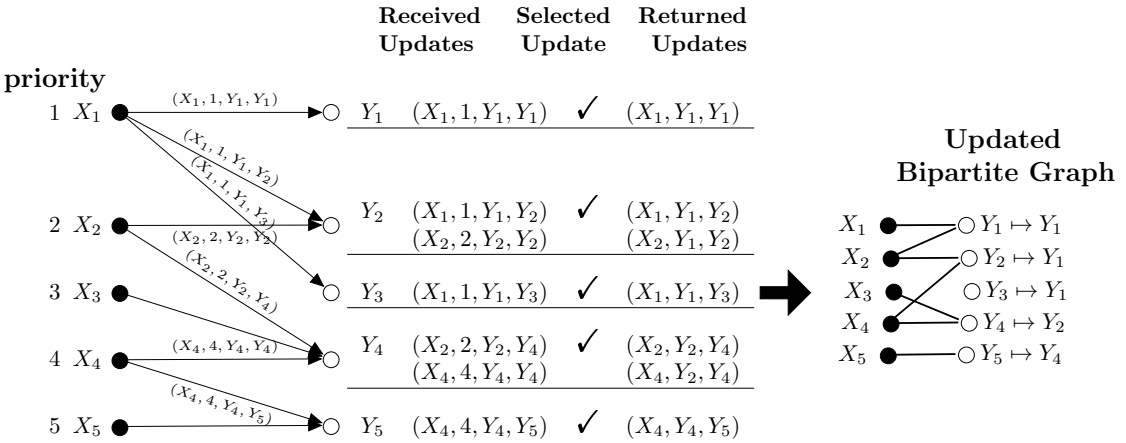


Figure 5.1: An example of the first round of parallel *HCG* algorithm.

have used the agglomerative clustering algorithm as it gives better quality results compared to the BFS algorithm. The algorithm starts by placing each hyperedge into a unique edge partition. The hypergraph is then traversed, one hyperedge at a time, and the similarity between the hyperedge and all adjacent hyperedges is calculated. If the similarity between a pair is less than the given similarity threshold, their corresponding edge partitions are merged and the new representative is appointed for the new edge partition.

The similarity between hyperedge pairs is calculated using *weighted Jaccard Index*, that is based on set intersection. According to the hypergraph distribution, the intersection of an internal hyperedge with either internal or external hyperedges can be calculated using only local data. A problem arises when calculating the intersection of two external hyperedges for which processors need data stored on other processors. For this purpose, we use a fast intersection method by hashing hyperedges to a bit vector and instead of calculating the actual intersection, the hash values are intersected. To calculate the hash values of the external hyperedges, processors calculate the hash values of their external hyperedges locally and the global hash values are calculated using a customised all-to-all communication, that is a simple union operation over the bit vector. A 2-universal hash function is used in our implementation.

In the end of the local phase, we end up with an edge partition set EP_i of size $|EP_i| = q_i$ on processor i such that $EP = \bigcup EP_i$ and $|EP| = q = \sum_{0 \leq i < p} q_i$.

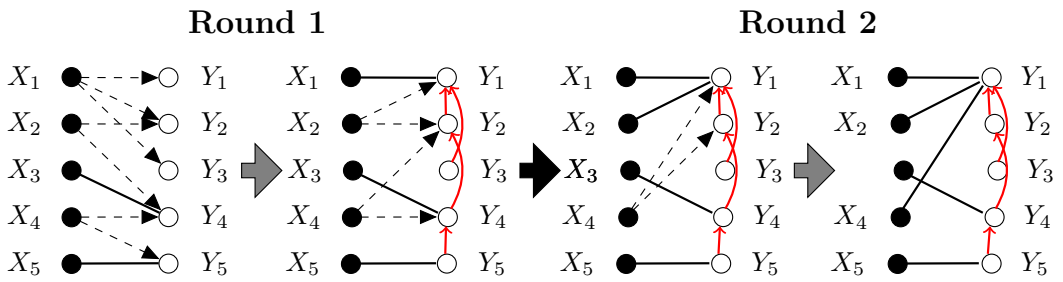


Figure 5.2: The two rounds of parallel HCG example in Fig. 5.1. Dashed, solid, and red lines show network communications, stabilised X to Y partitions, and representative dependency in Y, respectively. The algorithm stops after two rounds.

Local edge partitions are assigned a globally unique *ID*. Hyperedges straddling more than one processor are assigned to different edge partitions on each processor. According to the HCG definition, each hyperedge should be uniquely assigned to one edge partition. Therefore, for those hyperedges that are assigned to different edge partitions, the corresponding edge partitions should be unified to provide a globally unique hyperedge-to-edge partition assignment. The unification is dealt with in Algorithm 2 from line 4.

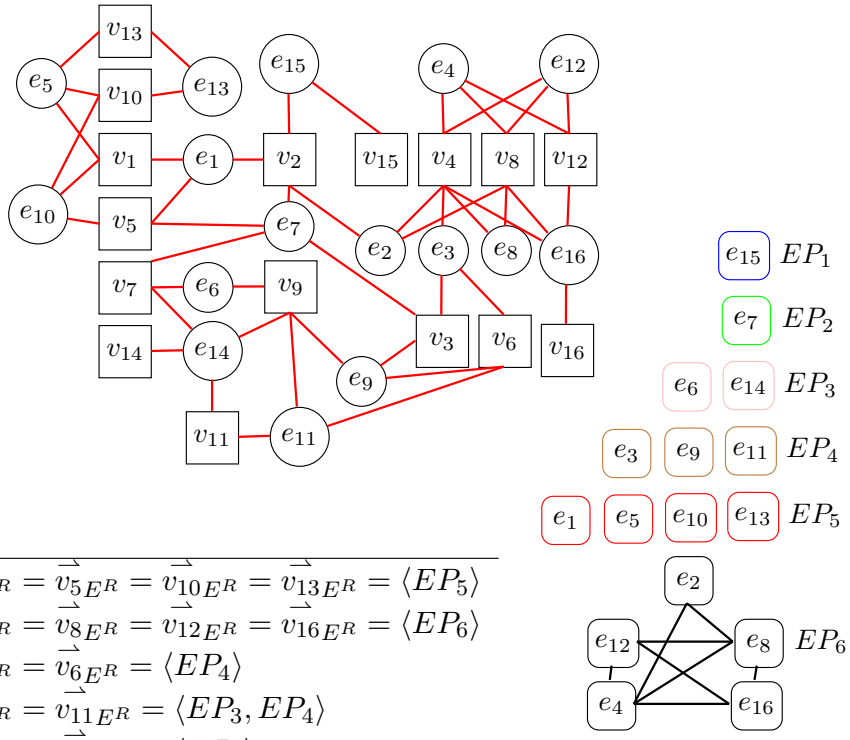
For a given hypergraph $H = (V, E)$ and edge partition set EP we define a bipartite graph $B(X, Y, \mathcal{U})$ with vertex sets $X = E$ of size $|X| = m$ and $Y = EP$ of size $|Y| = q$, and edge set $\mathcal{U}$ where a hyperedge is adjacent to an edge partition if it is assigned to the edge partition on any of the processors. The maximum degree of the vertices in X is p. Algorithm 2 works as follows. Both vertex sets X and Y are redistributed independently on the processors such that processor i holds $|X_i| \approx m/p$ and $|Y_i| \approx q/p$ of each, respectively, and new global *ID*s are assigned to them. Every vertex in $x \in X$ is assigned a priority. The priority is set to be the the global *ID* of the vertex. A smaller global *ID* means higher priority. In addition, the neighbour list of x in Y is stored in a list denoted as $n(x)$. The algorithm builds a spanning forest on Y vertices. Initially, each vertex in Y points to itself as the root (a vertex is in a unique tree initially). The algorithm runs in rounds. In each round if the degree of a vertex $x \in X$ is more than one, it chooses the first vertex in its adjacency list as representative and generates a tuple $\langle x, \mathit{priority}, \mathit{rep}, t\rangle$ per neighbour $t \in n(x)$. The *priority* is the priority assigned to x and *rep* is the representative among its adjacency list (which is selected to the first vertex in $n(x)$ list). Then, tuples are

sent to the target processors who hold the target vertices in Y.

Processors collect tuples for their local Y vertices. For each $y \in Y$, it chooses the representative with the highest priority amongst them and updates the root of y to point to the representative in the spanning forest that is built on Y. The updates are returned back to the vertices in X and the adjacency lists are updated accordingly for the next round. The operations repeat until all X vertices are adjacent to only one vertex in Y or a specified number of rounds is completed. Finally, each subtree in the spanning forest on Y represents a unique edge partition. The root of each tree stands for the representative of the edge partition. The function `FindRoots()` in line 28 finds the root for each tree in the spanning forest. The roots of the trees are returned to X vertices and finally, the neighbour of X vertices are updated.

An example of the first round of the algorithm is depicted in Fig. 5.1 that shows how the spanning forest on Y vertices is built. The complete operations, the whole two rounds, are depicted in Fig. 5.2. Dashed arrows show the network communications while assigning hyperedges to the edge partitions and the thick lines show finalised assignments (that means no communication is further done for these links). The dependency between Y vertices are shown as red arrows. After the first iteration, the connection between Y nodes does not change. The maximum degree of the vertices in X is p, which is the number of processors. In our implementation, we have an early exit condition such that we perform maximum p iterations. If the degree of a $x \in X$ vertex is still more than one after p iterations, we choose one of its neighbour in $n(x)$ randomly as the final representative. In the end, each vertex in X uniquely points to one vertex in Y. For the example given in Fig. 5.2, all vertices in X point to Y_1 as their neighbour after calling the `FindRoots()` function. This means that we end up with all hyperedges in one edge partition when the algorithm finishes.

An example of the final edge partition set for a sample hypergraph is depicted in Fig. 5.3. The edge partitions are depicted as $EP_1, \cdots, EP_6$ and their hyperedges elements are represented as $u_i, 1 \leq i \leq 16$.



Cores

Core 1:	$\overrightarrow{v_1}_{E^R} = \overrightarrow{v_5}_{E^R} = \overrightarrow{v_{10}}_{E^R} = \overrightarrow{v_{13}}_{E^R} = \langle EP_5 \rangle$
Core 2:	$\overrightarrow{v_4}_{E^R} = \overrightarrow{v_8}_{E^R} = \overrightarrow{v_{12}}_{E^R} = \overrightarrow{v_{16}}_{E^R} = \langle EP_6 \rangle$
Core 3:	$\overrightarrow{v_3}_{E^R} = \overrightarrow{v_6}_{E^R} = \langle EP_4 \rangle$
Core 4:	$\overrightarrow{v_9}_{E^R} = \overrightarrow{v_{11}}_{E^R} = \langle EP_3, EP_4 \rangle$
Core 5:	$\overrightarrow{v_7}_{E^R} = \overrightarrow{v_{14}}_{E^R} = \langle EP_3 \rangle$

Non-cores

$\{v_2, v_{15}\}$

Figure 5.3: An example of the matching algorithm. The similarity threshold is set to $s = 0.5$. Vertices are categorised into cores according to edge partitions EP_i and rough set clustering definitions.

5.2.2 Parallel Matching Algorithm

In this section, we find pair-matches of vertices. As proposed in the previous section, *HCG* builds edge partitions E^{R} on the hyperedges of $H = (V, E)$ such that the size of an edge partition is the number of hyperedges in it and its weight is sum of the weights of its hyperedges. This information is used to build the *reduced* information system $\mathcal{I}_H^{\mathrm{R}} = \left(V, E^{\mathrm{R}}, \mathbf{V}^{\mathrm{R}}, \mathcal{F}^{\mathrm{R}}\right)$ by replacing hyperedges with their edge partitions. In $\mathcal{I}_H^{\mathrm{R}}$, a vertex is incident to $e_{\mathrm{R}} \in E^{\mathrm{R}}$ if at least one of its incident hyperedges is in e_{R} and the mapping function is redefined as

$$\mathcal{F}^{\mathrm{R}}(v, e_{\mathrm{R}}) = \left|\{e \triangleright v \ \wedge \ e \in e_{\mathrm{R}}, \forall e \in E \ \}\right|. \tag{5.1}$$

where $\mathbf{V}_{e_{\mathrm{R}}}^{\mathrm{R}} \subseteq \mathbb{N}, \forall e_{\mathrm{R}} \in E^{\mathrm{R}}$.

Algorithm 3 Parallel Matching algorithm

Require: processor rank r, local sub-hypergraph $(H_r(V_r, E_r))$, number of candidates $nCand$

procedure ParallelMatching(r,$H_r(V_r, E_r)$,p,$nCand$)
  $me \leftarrow r$
  $EP \leftarrow$ParallelHCG($H_r(V_r, E_r)$,p,p)

Phase 1: *Core Matching*
  Build the final information system $\mathcal{I}^{\mathrm{f}}$ for H
  Categorise vertices into *cores* and *non-core* lists
  for all $c \in$ *core* vertex list **do**
    Assign c to a processor by hashing $\vec{c}_{\mathbf{E^R}}$
  for all $c \in$ *cores* that is assigned to *me* **do**
    pair vertices in c randomly ▷ Randomly pair-match vertices

Phase 2: *Global Vertex Matching*
  $candidates \leftarrow \{\}$
  $pl \leftarrow$ the sequence $\{1, 2, \cdots, p\}$ except *me*
  for all $v \in$ *non-core* list **do**
    if $externalConn(v) \geqslant internalConn(v)$ **then**
      $candidates \leftarrow candidates \cup \{v\}$
      Randomly shuffle pl and move *me* to the end of list
      $extDest[v][i] \leftarrow pl[i], \forall i = \{1, 2, \cdots, p-1\}$
  $rounds \leftarrow 0$
  repeat
    $lst \leftarrow$ maximum $nCand$ vertices from $candidates$
    Send each $v \in lst$ to processor $extDest[v][rounds]$
    for every received vertex u on *me* **do**
      Find best local match for u
      Update $\langle inmatch, outmatch \rangle$ for u
    Resolve *mutual* and *length-2* matches and remove the matched vertices from lst
    $bestConn \leftarrow 0$
    for all $i \in \{1, 2, \cdots, p\}$ **do**
      $state \leftarrow$ randomly choose a state in $\{inproc, outproc\}$
      if $state$ is $inproc$ **then**
        $cnn \leftarrow \sum_{\forall v \in V} J(v, inmatch(v)) \mid state(proc(inmatch(v)))$ is $outproc$
      else
        $cnn \leftarrow \sum_{\forall v \in V} J(v, outmatch(v)) \mid state(proc(outmatch(v)))$ is $inproc$
      $cnn_{\text{total}} \leftarrow$ sum of cnn on all processes
      if $cnn_{\text{total}} > bestConn$ **then**
        $bestState[me] \leftarrow state$
    MPI_Allgather($bestState$) ▷ Collect $bestState$ on all processors
    if $bestState[me]$ is $inproc$ **then**
      Match every vertex in lst with its $outproc$
    else
      Match every vertex in lst with its $inproc$
    remove all matched vertices from $candidates$
  until $(rounds \leftarrow round + 1) < p - 1$

Phase 3: *Local Vertex Matching*
  for every unmatched vertex u on *me* **do**
    Find best match for u locally

Following the discussion in Chapter 4.4.1, edge partitions of unit size contain less important information and they can be removed from the E^{R}. The reason is that unit size edge partitions are hyperedges with less similarity to other hyperedges in *HCG* (as in Definition 4.1), so they do not provide important vertex matching information. Thus the size of $\mathcal{I}_H^{\mathrm{R}}$ is reduced by removing unit size edge partitions and the final information system $\mathcal{I}^{\mathrm{f}} = \left(V, E^{\mathrm{R}}, \mathbf{V}^{\mathrm{R}}, \mathcal{F}^{\mathrm{f}}\right)$ is obtained by recalculating the mapping function as follows:

$$\mathcal{F}^{f}(v, e_{\mathrm{R}}) = \begin{cases} 1, & \text{if } \frac{\mathcal{F}^{\mathrm{R}}(v, e_{\mathrm{R}})}{Deg(v)} > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{5.2}$$

At this point, rough set clustering techniques can be employed to calculate vertex equivalence classes in $\mathcal{I}^{\mathrm{f}}$ as described in Section 2.3. Vertices are then categorised as *core* and *non-core* vertices such that those in the same V/E^{R} partition belong to the same core as they are in the same equivalence class. Vertices for which $\mathcal{F}^{\mathrm{f}}(v, e_{\mathrm{R}}) = 0$, $\forall e_{\mathrm{R}} \in E^{\mathrm{R}}$ are non-core vertices. In order to prevent the algorithm to build very big cores, we put a limit on the size of the cores in our implementation and we do not allow a core size to grow bigger than a user specified size[1]. An example of the vertex-core assignment is depicted in Fig. 5.3 for our sample hypergraph. In the example, two vertices v_2 and v_{15} are assigned to cores of size unity; therefore, we put them in the non-core list.

A nice property of the rough set clustering is that the vertex equivalence class calculation can be done in parallel and independently on all processors. Zhang et al. [ZLR+12] propose a parallel framework for calculating rough set approximations using *MapReduce* [DG08]. Following the definitions in their work, vertex to *core* assignment can be done independently by all processors.

Definition 5.1 *Given an information system* $\mathfrak{I} = (\mathbb{U}, \mathbf{A}, \mathbf{V}, \mathcal{F})$, *let* $S = \bigcup_{i=1}^{b} S_i$ *where* $S_i = (\mathbb{U}_i, \mathbf{A}, \mathbf{V}, \mathcal{F})$ *such that* $\mathbb{U} = \bigcup_{i=1}^{b} \mathbb{U}_i$ *and* $\mathbb{U}_j \cap \mathbb{U}_k = \emptyset, \forall j, k \in \{1, 2, \cdots, b\}, j \neq k$, *then we say that* S *is divided into* b *information subsystems.* S_i *is called an information subsystem of* S [ZLR+12].

[1]This value is chosen to be 500 by default and this is also used in our evaluations.

Definition 5.2 *Let* $\mathbf{B} \subseteq \mathbf{A}$ *be a subset of attributes. The information set with respect to* $\mathbf{B}$ *for any* $C \in \mathbb{U}/\mathbf{B}$ *is denoted by* $\vec{C}_{\mathbf{B}} = \vec{c}_{\mathbf{B}}, \forall c \in C$ [ZLR+12].

Lemma 5.1 [ZLR+12] *Let* $\mathbf{B} \subset \mathbf{A}$ *be a subset of attributes, and* C, D *be two equivalence classes with respect to* $\mathbf{B}$ *from two different information subsystems of* S*. One of the following holds:*

1. *If* $\vec{C}_{\mathbf{B}} = \vec{D}_{\mathbf{D}}$*, then the two equivalence classes* C *and* D *can be combined into one equivalence class* G *with respect to* $\mathbf{B}$*, where* $G = C \cup D$ *and* $\vec{G}_{\mathbf{B}} = \vec{C}_{\mathbf{B}} = \vec{D}_{\mathbf{B}}$*.*
2. *If* $\vec{C}_{\mathbf{B}} \neq \vec{D}_{\mathbf{D}}$*, the two equivalence classes* C *and* D *cannot be combined as one equivalence with respect to* $\mathbf{B}$*.*

The parallel matching algorithm is proposed in Algorithm 3. The algorithm has three main phases and begins with the processing of the core vertices in phase one. Cores are hashed to the processor set using their equivalence classes. The hash function uniformly distributes cores among processors such that each processor is assigned a number of cores. As mentioned earlier (and in order to prevent load imbalance among processors) we restrict the size of V/E^{R} partitions and we do not allow core sizes to grow freely. Then processors traverse cores one at a time and randomly pair-match vertices inside each core. Vertices which do not find a match are added to the non-core vertex list to be processed in the next step.

In Phase two, the algorithm proceeds with the processing of non-core vertices. *PFEHG* finds a pair-match for a vertex using *Weighted Jaccard Index* as the serial *FEHG* algorithm and restated here as follows:

$$J(u,v) = \frac{\sum_{e \rhd v \wedge e \rhd u} \gamma(e)}{\sum_{e \rhd v \vee e \rhd u} \gamma(e)}, \quad v, u \in V \text{ , and } \forall e \in E. \tag{5.3}$$

The matching of non-core vertices runs in rounds (line 18). In each round, processors randomly select a subset of their unmatched external non-core vertices. The selection criterion is based on the external connectivity of the vertices such that a vertex is selected if its external connectivity is greater than its internal connectivity. These vertices are sent to a remote processor to find a pair-match. The algorithm provides a list of destination processors for each vertex such that the processors in

the list are connected to at least one of the hyperedges incident on the vertex. In each round, a processor is selected randomly from this list and the vertex is sent to the processor to find a remote match.

After a customised all-to-all communication, processors traverse their received vertices and find pair-matches for them locally. Then matched candidates are sent back to the requesting processors. We only allow pair-matches and multiple matches are not supported in our algorithm. Cyclic dependency conflicts may occur; a vertex that is found as a match candidate for a remote vertex has been already been sent to another processor and has found a pair-match there. We break conflicts as follows. A pair $\langle inmatch, outmatch \rangle$ is built for every vertex (line 28). If a vertex v is sent to another processor and it finds a match u on that processor, u is saved in *outmatch* of v, and v is saved in *inmatch* of u. First, two cases are resolved before processing with the iterations.

1. *Mutual Matches*: If the *inmatch* and *outmatch* of a vertex v point to the same vertex u, then vertex v is definitely on the the *inmatch* and *outmatch* of vertex u. The match is allowed without creating any conflicts.

2. *Length-2 Matches*: If two vertices u and v form a path of length two, the match is allowed without creating any conflicts. For two vertices u and v, this situation happens when either *inmatch* or *outmatch* of the vertices are set and v is on the *inmatch* of u and u is on the *outmatch* of v or vice versa. This situation is identified with a customised all-to-all communication between the processors.

Then processors go through p iterations. In each iteration, a state is assigned to each processor randomly. The state can be either *inproc* or *outproc*. Processor state *inproc* means that the vertices (those have been sent to remote processors in order to find an external match) will select *inmatch* candidates as the final match candidates if the processor that contains the *inmatch* vertex has the *outproc* state (line 28). The same applies to *outproc* processors. In each iteration, the sum of *Jaccard Index* for the matched vertices based on processor states are calculated. If the sum is bigger than *bestConn*, the *bestConn* is updated to the new value and processor states are

Algorithm 4 Calculating vertex move gain to decide which processor should hold the coarser vertex when two vertices are merged.

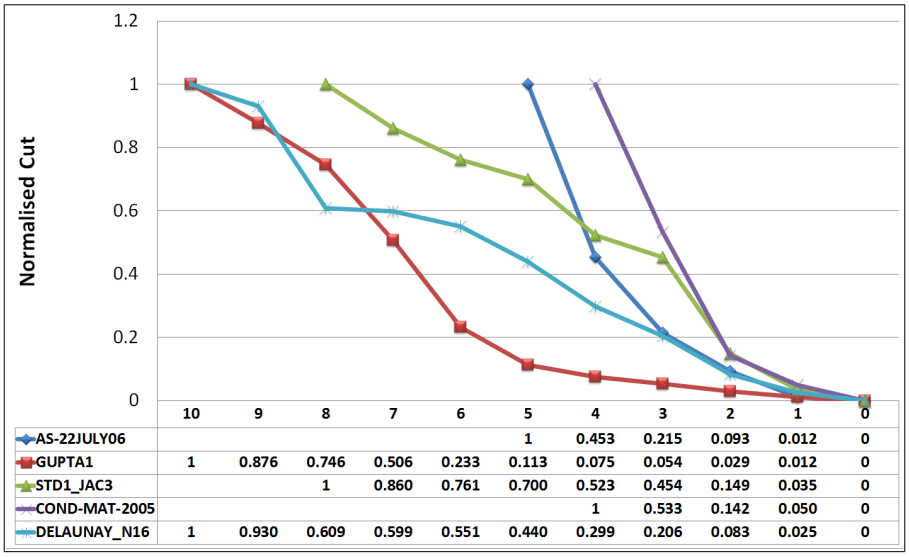
Require: Input is the vertex v matched with m_v, the minimum number of vertices on each processor *redl*

```
1: procedure VertexMoveGain(H(V, E),v,m_v)
2:     g_1 ← 0 and g_2 ← 0                                   ▷ 2-level gain
3:     p(v) ← the processor containing v
4:     p(m_v) ← the processor containing m_v
5:     if p(v) == p(m_v) then                                ▷ local match
6:         Return
7:     else if there is less than redl vertices on the local processor then
8:         g_1 ← ∞ and g_2 ← ∞
9:         Return
10:    for all hyperedge e incident on v do
11:        s_g(e) ← the global size of e
12:        s_l(e) ← the local size of e
13:        if e is local then
14:            g_1 ← g_1 − 1
15:            g_2 ← g_2 − 1
16:        else if p(m_v) is connected to e then      ▷ e has at least one vertex on p(m_v)
17:            if (s_g(e) > 2) and (s_l(e) == 1)) then
18:                g_1 ← g_1 + 1
19:            if (s_g(e) > 4) and (s_l(e) == 2)) then
20:                g_2 ← g_2 + 1
21:    Return g_1 and g_2
```

saved as the best processor states among all iterations in the *bestState* array. In the end, the state of processors are set to the best states in *bestState* array.

Phase three finds pair-matches for the remaining unmatched vertices locally. Each processor simply traverses its local unmatched vertices in random order and finds pair-matches according to the Eq. (5.3).

When pair-matches are found, vertices are merged and new coarsened vertices are built. The weight of a coarsened vertex is the sum of the weights of the two merged vertices and its incident hyperedges are the union of the hyperedges incident to both vertices. When a match is external, one of the processors should hold the coarsened vertex. We follow Algorithm 4 for making this decision. According to the algorithm, the coarsened vertex is saved on the processor that results in fewer number of external hyperedges. The algorithm calculates move gains for both vertices. If a vertex gets a higher gain, it is most likely to be moved because moving the vertex



	10	9	8	7	6	5	4	3	2	1	0
AS-22JULY06						1	0.453	0.215	0.093	0.012	0
GUPTA1	1	0.876	0.746	0.506	0.233	0.113	0.075	0.054	0.029	0.012	0
STD1_JAC3			1	0.860	0.761	0.700	0.523	0.454	0.149	0.035	0
COND-MAT-2005							1	0.533	0.142	0.050	0
DELAUNAY_N16	1	0.930	0.609	0.599	0.551	0.440	0.299	0.206	0.083	0.025	0

Figure 5.4: The variation of bipartitioning cut in different levels of uncoarsening. The values are normalised in $[0, 1]$ based on the minimum and the maximum cut.

gives fewer external hyperedges. The algorithm uses two level gains. The second gain is for the tie breaking condition and it is denoted as g_2. The algorithm does not allow the number of vertices on any processor goes bellow a threshold (which is denoted as *redl*). If the number of vertices on a processor is less than the threshold, then the processor is selected to hold the coarsened vertex; if both processors have this situation, the target processor is selected randomly. In our implementation, the value of the threshold is set to be the same as the minimum number of vertices in the coarsest hypergraph (refer to the next section for more information).

When the coarser hypergraph is built, three operations are done. First, the global size of the hyperedges and their processor adjacency lists are calculated. Second, hyperedges of unit size are removed as they do not contribute to the partitioning cut. Third, identical hyperedges are identified and collapsed in the coarsened hypergraph. The third operation is the same as the operation which is done in our serial algorithm and in *Zoltan*'s hypergraph partitioner. This is done using hash functions. Hyperedge are hashed to integer values based on their vertex list. Two hyperedges with the same hash value are considered as identical. If conflicts occur, the whole content of hyperedges are compared.

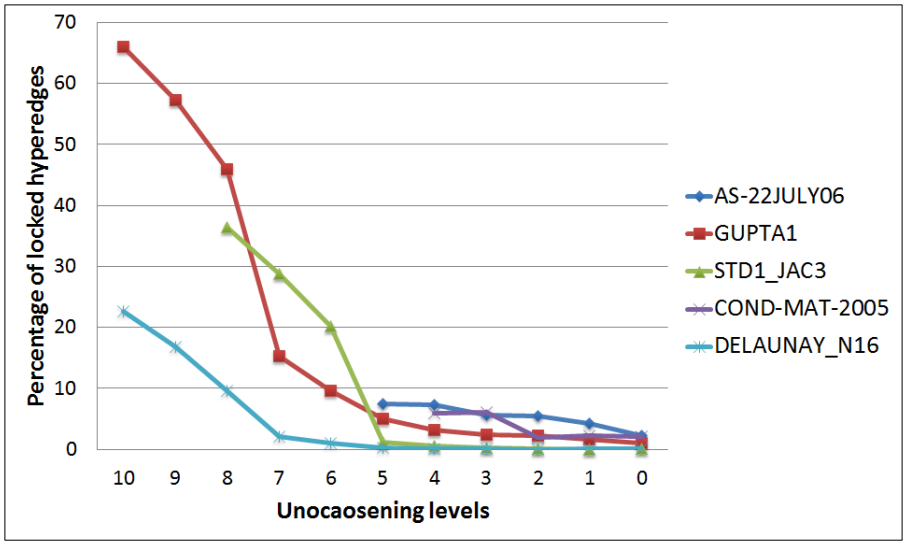


Figure 5.5: The percentage of the locked hyperedges in different levels of uncoarsening. The values are the average values over all passes of the FM algorithm in each coarsening level.

5.2.3 Initial Partitioning and Uncoarsening

The coarsening phase ends when the coarsest hypergraph H_c contains a few tens of vertices such that its bipartitioning can be done very quickly (this happens in our implementation when H_c has fewer than 100 vertices). H_c is replicated on a subset of the processors and each processor calculates a bipartitioning using a set of randomised greedy algorithms. Among the algorithms depicted in Chapter 3.1.2, *PFEHG* uses Random assignment, Linear assignment, and FM-based algorithms (the same category of algorithms are selected as the serial *FEHG* algorithm). The partitioning that preserves the balance constraint and gives the minimum partitioning cut among all runs is selected as the best partitioning result and it is projected back to the original hypergraph.

While uncoarsening, the partitioning quality is further improved. We use a modified version of the FM algorithm known as Early-Exit FM and Boundary FM which has been shown to be good in practice[2] [Kar02,ÇA11]. As mentioned earlier, the FM algorithm calculates a gain for each vertex. The gain simply shows the improvement in the cost function if the vertex is moved from its part to the opposite partition (in bipartitioning). When a vertex is moved, the gain of all adjacent vertices

[2]The reader is referred to Chapter 3.1.2 for details about these algorithms.

should be updated accordingly. This is a challenging task because it may produce lots of network communications if the algorithm runs in parallel. This phase is the most difficult phase of multi-level paradigm to parallelise [LK13,DBH+06,TK08]. Consequently, the parallel bipartitioning algorithms use a modified version of the FM Algorithm[3]. First, they split the refinement process into passes and only single direction vertex move is allowed in each pass (that is, vertices are only allowed to move from one part to the other). The direction of move is alternately changing between the passes. Second, only the gain of local vertices are updated with each vertex move. The strategy guarantees that parallel vertex moves by the processors do not adversely affect the gain of vertices on other processors which is known as *parallel move conflict* [Kar02]. Because processors locally decide about vertex moves without any synchronisation, they need to translate the global balance constraint to a local balance constraint.

This strategy, that is brought from parallel graph refinement algorithms, has some disadvantages as follows:

1. Local balance constraints put tight restrictions on vertex moves and the partitioner has limited space for optimisation. As we have mentioned in Chapter 3.1.4, the global balance constraint is translated into a local balance constraint on each processor. Local balance constraints are tighter than the global balance constraint and, consequently, limit vertex moves between partition boundaries and leave little space for optimisation.

2. Global balance constraint is sometimes violated locally and fixing this requires synchronisation among processors [DBH+06], so it is not completely synchronisation-free. Some of processors have to undo their vertex moves in order to meet the global balance constraint.

3. Increasing the number of processors adversely affects the quality of the partitioning, especially in 2D hypergraph distribution. The reason is that the processors have less local data, or fewer pins, as the number of processors

[3]The reader is referred to Chapter 3.1.4 for the details of the parallel FM refinement algorithm.

increases. In 2D distribution, we have a smaller average number of pins on each processors compared to the 1D distribution; therefore, increasing the number of processors has greater negative impact on the quality.

4. Parallel move conflicts, which proposed in Chapter 3.1.4, are more challenging on graphs and the problem on hypergraphs, although being more complicated, is not as severe as in graphs.

There are two points to be made regarding the last case. First, because the cardinality of hyperedges are not limited in hypergraphs, it is more difficult to take a hyperedge out of a cut if its cardinality increases [Kar02]. Second, Shibuya et al. [SNK95] report that almost 80% of hyperedges remain in the cut for the whole run of an the FM algorithm. According to this research, there is only a limited number of vertex moves between partition boundaries, and then most hyperedges are locked into the cut. In hypergraphs, we have two variations: vertex degrees and hyperedges sizes. We assume the first level gain for the FM algorithm in a bipartitioning problem. A cut hyperedge is considered for calculating vertex gains only if all but one of its vertices reside in one part. On the other hand, all edges of a graph participate in vertex gain updates when the edge is cut[4]. This means that the hyperedge would not participate in vertex gain update as long as it has more than one vertex in either of the parts. Furthermore, if we know that a hyperedge is locked into the cut while we move its vertices on other processors, we can make some decisions to decrease the parallel move conflict effects.

We have observed the following when investigating multi-level serial hypergraph partitioning algorithms:

1. Most improvements in the cut are done early in the uncoarsening phase when there are clusters of vertices and most vertices have large positive gains. In these early stages, moving a vertex gives a considerable improvement in the cut.

[4]The cardinality of all edges are two in graphs. When a vertex is moved, all edges that have their other end-point on another processor participate in vertex gain updates.

2. Going through coarsening levels generates hypergraphs with larger $|E|/|V|$ ratio, higher vertex degrees, and smaller hyperedge sizes [Kar02]. This means that the structure of the hypergraph is getting more closer to the graph model. In this situation, vertex move conflicts create more problems because moving a vertex on a processor would definitely impact the gain of its connected pairs on other processors.

3. The necessary condition for two consecutive vertex moves to get a hyperedge into the cut is that both vertices must belong to the same hyperedge. In high dimensional data structures such as a hypergraph, the probability of the hyperedges to get locked is inversely proportional to the average hyperedge size and directly related to the average vertex degree [ESK03]. According to case 2, we are dealing with low hyperedge sizes and high vertex degrees earlier in the uncoarsening phase, Consequently, there is a higher probability for hyperedges to get locked in the cut.

Figure 5.4 depicts the cut change in different uncoarsening levels when projecting back the hypergraph. The results are proposed for some of the hypergraphs in Appendix A. The cuts are normalised in $[0, 1]$ based on the maximum (which is the cut on the coarsest hypergraph) and the minimum (the final cut at level zero of coarsening) partitioning cuts. According to the results, most of the cut change happens in the first few levels of the uncoarsening and the reduction speed is higher. After 50% of the uncoarsening levels, the cut reduction is between 48% and 89% for the evaluated hypergraphs. This fact emphasises that more efforts should be invested early in the uncoarsening phase.

Figure. 5.5 shows the percentage of the locked hyperedges for different uncoarsening levels. According to the figure, the highest values are for the first few levels of coarsening where we are dealing with large vertex degrees and smaller hyperedge sizes; therefore, moving two consecutive vertices more likely locks some hyperedges. For hypergraphs such as `AS-22JULY06` and `COND-MAT-2005`, the [*mean vertex degree, mean hyperedge size*] in the coarsest hypergraph is $[6.27, 11.21]$ and $[27.76, 6.54]$, respectively while in the original hypergraph the values are $[2.03, 21.99]$ and $[4.16, 9.04]$, respectively. The ratio of mean vertex degree to the mean hyperedge size changes

from 0.09 to 0.55 in the AS-22JULY06 and from 0.46 to 4.24 in COND-MAT-2005. For these two hypergraphs we do not see big changes in the percentage of locked hyperedges. On the other hand, we see big changes in the ratio of mean vertex degree to the mean hyperedge size in DELAUNAY-N16 (0.74 to 9.53), STD1-JAC3 (0.94 to 32.37) and GUPTA1 (0.63 to 32.93). The changes in the mean vertex degrees and hyperedge sizes are very high such that the change from the original hypergraph to the coarsest hypergraph is: $[2.82, 3.80]$ to $[24.99, 2.62]$ in DELAUNAY-N16, $[66.18, 69.80]$ to $[326.03, 10.70]$ STD1-JAC3, $[31.06, 48.63]$ to $[246.36, 7.48]$ in GUPTA1. For these hypergraphs, we see big changes in the percentage of the locked hyperedges in early levels of uncoarsening.

As a result, we conclude that most improvements can be done early in the uncoarsening phase, while later (when we get closer to the original hypergraph) the percentage of negative moves dominates the positive moves. In our synchronisation-based refinement strategy, vertices are free to move to any direction in each pass. A *token* is defined and the processor that holds the token is allowed to move its vertices. The *token* is simply the number of vertex moves that processors can make. At the beginning of the pass, the ratio of positive gains to negative gains on each processor is calculated and the processor which gives the maximum value holds the *token*. The token is then rotated amongst processors in a round-robin manner such that each processor gets the same chance to move its vertices. When we move vertices, gains are only updated locally. The *token* may take negative values: if the ratio of positive gains to negative gains is less than a threshold, the *token* is negated and this means that all processors can move $|token|$ vertices concurrently. The global balance constraint might be violated during concurrent moves, but it will be managed with a synchronisation among processors when passing the token. FM passes continue until a user-given *pass limit* is met or no further moves can be done.

According to the algorithm, it moves the vertices on all processors in a sequential manner when the token is positive. While the size of hypergraph is small compared to the original hypergraph in the first few uncoarsening levels, these sequential updates do increase the runtime by a very small fraction. Furthermore, the negative token may create move conflicts between the processors and may increase the cut size.

This is against the conditions proposed by Karypis [Kar02] that says the cost of the projected back partitioning on H_i should be less than or equal to the cost of the partitioning on H_{i-1}. This is not important in our implementation because of following reasons:

1. The increase in the cut size is quite small as the percentage of the vertices with positive gains is very small and they cannot make big changes to the cut.

2. This provides a perturbation of the cut. As one of the problems of the FM algorithm is that it easily gets stuck in local minima [Kar02,CLL+97], the small perturbation might get the algorithm out of the local minima and provide partitioning cut improvements in the later passes of the FM algorithm.

Furthermore, we hold a state for every hyperedge that could be *free*, *loose*, or *locked*. At the beginning of each pass, all hyperedges are free. A vertex is locked after a move to prevent thrashing. When a hyperedge has a locked vertex in only one part, its state is updated to loose. The state is changed to locked when it gets locked vertices on both parts; that is, it is impossible to move that hyperedge out of the cut in later moves. When the token is positive, hyperedge states are updated among processors with each synchronisation. In our implementation, we only communicate the state of hyperedges with size two (or it is better to say edges according to the graph terminology). When the state of these hyperedges changes, we tell other processors about the changes (only the processors that are connected to these hyperedges). Consider two vertices v and u on the local and the remote processors, accordingly, that are connected by an edge e. If v moves between the parts, the state of e changes to *loose* and the gain of u is updated as follows:

1. If u and v are in the same part before the move, the gain of u is increase by the weight of e to push u to move to the other part and remove e out of the cut.

2. otherwise, the gain of u is decreased to prevent u from changing the part and get e into the cut.

Furthermore, and for the other hyperedges of size greater than two, the state can be used for calculating future move gains of the vertices such that a locked

hyperedge will have no role in updating vertex gains for the future vertex moves. In our evaluations, we have not used this option and only the state of the size two hyperedges are communicated.

5.2.4 Processor Reconfiguration

PFEHG is a recursive bipartitioning algorithm and performs the following two processor reconfigurations in each bipartitioning recursion:

Bisection processor splitting

At the end of each bipartitioning, the processors are split into two equally sized separate subsets. Vertices in the first part and their incident hyperedges are assigned to the first subset and the other vertices and their incident hyperedges are assigned to the other subset. Each subset continues with the partitioning of the hypergraph independently. This strategy is shown to be practically successful and generates better partitioning quality and gives better performance than alternative approaches [DBH+06,TK08,Kar13b]. The only drawback of this strategy is that preserving the global balance constraint needs some extra effort. In order to do that, we should apply stricter balance constraint which adversely affect the partitioning quality.

Multiple Bisection

Given a hypergraph $H = (V, E)$, a set of p processors, a *replication factor* ψ such that $(p \mod \psi) = 0$, and a minimum subgroup size $p_{\min}$, *PFEHG* splits the processor set into ψ subsets if $p/\psi \geqslant p_{\min}$. The hypergraph H is replicated on all ψ processor subgroups and $\mathcal{H}_i^{\psi} = \left(\mathcal{V}_i^{\psi}, \mathcal{E}_i^{\psi}\right)$ represents ith replication of H on ith processor set $i = \{1, 2, \cdots, \psi\}$. Subgroups partition their assigned replicated hypergraphs independently. In the end, the partitioning which maintains the balance constraint and gives the minimum cost is selected as the final partitioning for this bisection.

Both cases are tied with an extra reconfiguration overhead, but they increase processor locality that will result in higher performance and provide better quality.

As we will show in the evaluation section, the amount of time the algorithm spends on hypergraph reconfiguration is less than the time that is spent on coarsening the hypergraph. This strategy should be used with care as small replication factor will cause the reconfiguration time degrade the performance. This strategy is also used in our cloud evaluations because it provides better data locality. Our multiple bisection reconfiguration considers the scarcity of networking resources in the cloud and, as we will see in Section 5.4, it has a positive role in increasing the performance of the *PFEHG* algorithm in the cloud.

5.3 Experimental Evaluation

The *PFEHG* algorithm is implemented as a new hypergraph partitioning library in the *Zoltan* tool [DBH+06]. The communication is done in a BSP-like model using *Zoltan* communication layer. *PFEHG* can be called by setting both `LB_METHOD` and `HYPERGRAPH_PACKAGE` to `FEHG`. The algorithm has several parameters which can be set by calling `Zoltan_Set_Param` function. The code for calling the partitioner and the interface is exactly the same as *Zoltan*. We evaluate our algorithm against the state-of-the-art *Zoltan* parallel hypergraph partitioner, that is known as *PHG*. The algorithm is shown to have very good scalability in practice [DBR+09].

We select a set of hypergraphs from a variety of applications with different specifications. They are used for the evaluations in this section and the cloud evaluation in the next section. The benchmarks are obtained from the University of Florida Sparse Matrix Collection [DH11]. Each matrix represents a hypergraph in column-net model: rows and columns correspond to vertices and hyperedges of the hypergraph, respectively [ÇA99]. The weight of the vertices and hyperedges are assumed to be unity. The list of the test hypergraphs is depicted in Table 5.1. The reader is referred to Appendix A for complete specification of these hypergraphs.

5.3.1 System Configuration and Algorithm Initialisation

We run our evaluations on the Linux-based Hamilton HPC cluster in Durham University. The version of the operating system is CentOS release 6.5 and we use

Table 5.1: Evaluated hypergraphs for parallel simulation and their specifications.

Hypergraph	Description	Rows	Columns	Non-Zeros
AMAZON0601	Web Indexing	403,394	403,394	3,387,388
BCSSTK32	Structural Problems	44,609	44,609	2,014,701
CNR-2000	Web Crawling	325,557	325,557	3,216,152
CAGE13	DNA Electrophoresis	445,315	445,315	7,479,343
CAGE14	DNA Electrophoresis	1,505,785	1,505,785	27,130,349
CH8-8-b5	Combinatorial Problem	564,480	376,320	3,386,880
COND-MAT-2005	Collaboration Network	40,421	40,421	351,382
LANDMARK	Least Squares Problem	71,952	2,704	1,146,848
NOTREDAME	Social Networks	392,400	127,823	1,470,404
RAIL4284	Linear Programming	4,284	1,096,894	11,284,032
GL7d15	Combinatorial Problem	460,261	171,375	6,080,381
GL7d16	Combinatorial Problem	955,128	460,261	14,488,881
GL7d17	Combinatorial Problem	1,548,650	955,128	25,978,098
GL7d22	Combinatorial Problem	349,443	822,922	8,251,000

OpenMPI version 1.8.2. There are three computer clusters overall and they are represented as Hamilton 4 (**ham4**), Hamilton 5 (**ham5**), and Hamilton 6 (**ham6**). They are different in architecture and the number of nodes they provide. We run our algorithms in **ham4** cluster. It has 228 computer nodes (1824) cores. Each computer has:

1. 2 x quadcore Intel Xeon E5520/2.26 GHz Nehalem processors (total of 8 cores per node)
2. 24 GB memory (3 GB per core)
3. 1 x 160 GB disk
4. 1 x QDR InfiniBand interconnect

An important parameter for *PFEHG* is the similarity threshold that can affect both performance and the quality of the algorithm. We use the algorithm proposed in Chapter 4.4.1 to calculate the Clustering Coefficient (CC) of the hypergraph. A nice property of this calculation is that it can be easily parallelised by calculating the CC of hyperedges locally with the global values being calculated by one customised all-to-all communication. Then the similarity threshold is set to be the CC of the hypergraph. As the structure of the hypergraph changes during the coarsening (when

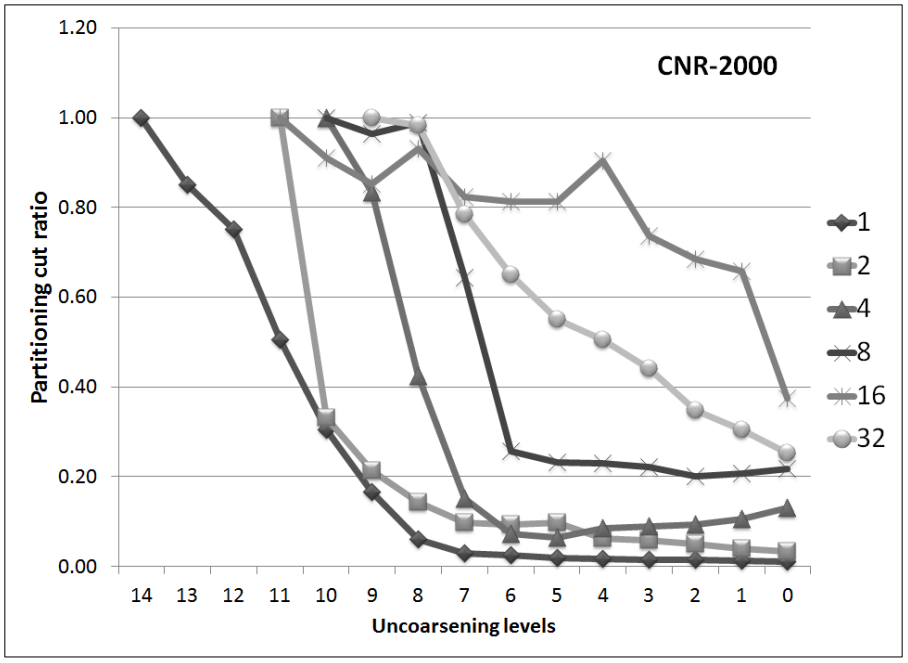


Figure 5.6: The cut reduction of our FM algorithm for a bipartitioning on `CNR-2000` with variable number of processors.

the hypergraph is coarsened and a new hypergraph is built for the next coarsening level), the CC of the hypergraph also changes. Instead of recalculating CC in each level, it is calculated once at the beginning of each multi-level recursion and its value is readjusted in each coarsening level based on the inverse of the average vertex degrees.

The redistribution imbalance in Section 5.1 to 0.1 when initially distributing each hypergraph on processors in the beginning of the algorithm. A collection of twelve hash functions are used for hypergraph initial distribution. The list of used the hash functions is depicted in Appendix B.3.2. The number of passes for the refinement function is set to two (the same is done for *PHG*) and the *token* value is set to 16. This means that processors can move a maximum of 16 vertices when holding the *token*. The imbalance tolerance is set to 5%. The reported results are the average of 10 runs for each algorithm.

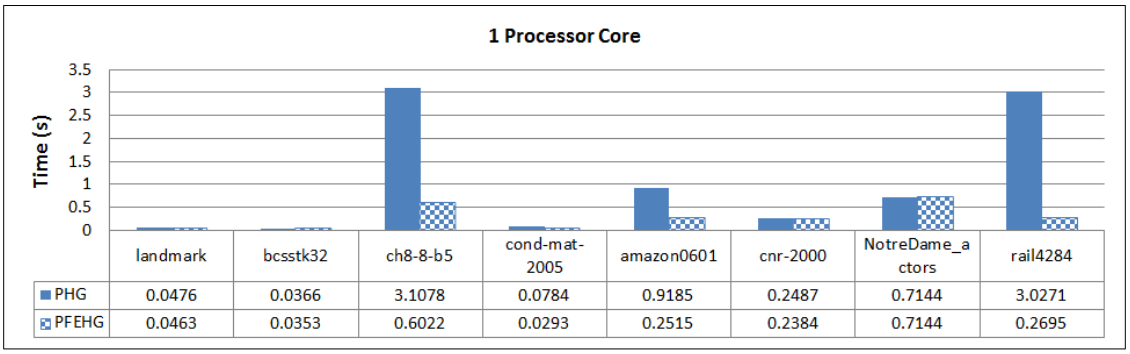
5.3.2 Parallel Refinement Performance

We have tested the performance of our synchronisation-based FM algorithm on the bipartitioning of `CNR-2000` hypergraph given in Table 5.1. The cut change at

each level of coarsening for different number of processors is depicted in Figure 5.6. Level zero contains the original hypergraph. As is shown in the figure, most of the improvement of the cut is done early in the coarsening phase and the rate of improvement decreases as we get closer to the original hypergraph. According to the figure, the perturbation of the cut (negative token) that adversely affects the quality of the partitioning is more obvious for 16 processors. The increase is observed on levels 8 and 4 of the uncoarsening, but the increase has been compensated later in the proceeding levels. In the end, we have more than 60% reduction in the partitioning cut. These cases can be simply prevented in the algorithm; that is we save the part numbers at the beginning of the FM refinement. If the cut increases in the end of the current refinement pass, we do not change vertex part numbers and restore the previous values. This option is not activated in forthcoming evaluations as the advantages of cut perturbation were more than preventing it.

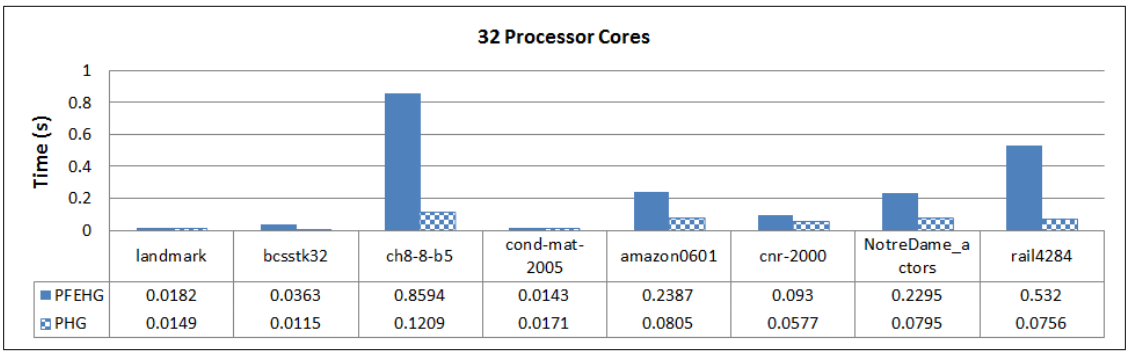
In the next evaluation, we investigate how our synchronisation-based FM algorithm performs compared to the pass-based FM algorithm that allows one-direction vertex moves in one pass. The number of processors are changed from one to 128 cores. We have not activated the multiple bisection for these evaluations. The runtime of the refinement algorithms for a bipartitioning of some of the hypergraphs are depicted in Fig. 5.7 and the percentage of the whole partitioning time that the algorithms spend on the refinement phase are proposed in Fig. 5.8. On a single processor that we do not have any conflicts and all data are local, our FM algorithm performs very well and takes less time compared to the *PHG* algorithm. This is an indication of the superiority of our rough set clustering algorithm compared to the randomised local vertex matching algorithm of *PHG*. This is in accordance to what has been reported by Karypis and Kumar [KK98a]; hence a good partitioning on the coarsest hypergraph also provides a good partitioning of the original hypergraph and we need less effort during the refinement phase. This is the reason why the refinement phase in our serial algorithm takes much less time than *PHG* refinement phase. As we have discussed in Chapter 3, the reason is the good clustering algorithm that is used in the *FEHG*.

Furthermore, as the number of processors increases, there are two extra operations



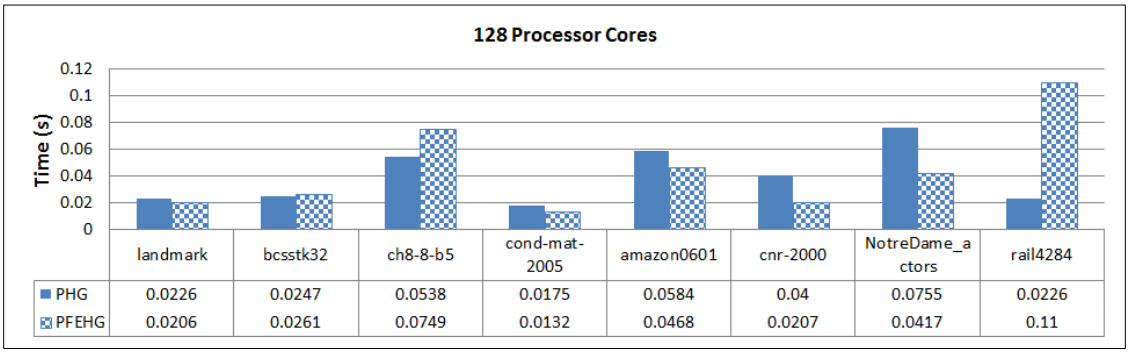
	landmark	bcsstk32	ch8-8-b5	cond-mat-2005	amazon0601	cnr-2000	NotreDame_actors	rail4284
PHG	0.0476	0.0366	3.1078	0.0784	0.9185	0.2487	0.7144	3.0271
PFEHG	0.0463	0.0353	0.6022	0.0293	0.2515	0.2384	0.7144	0.2695

(a) FM runtime on single core.



	landmark	bcsstk32	ch8-8-b5	cond-mat-2005	amazon0601	cnr-2000	NotreDame_actors	rail4284
PFEHG	0.0182	0.0363	0.8594	0.0143	0.2387	0.093	0.2295	0.532
PHG	0.0149	0.0115	0.1209	0.0171	0.0805	0.0577	0.0795	0.0756

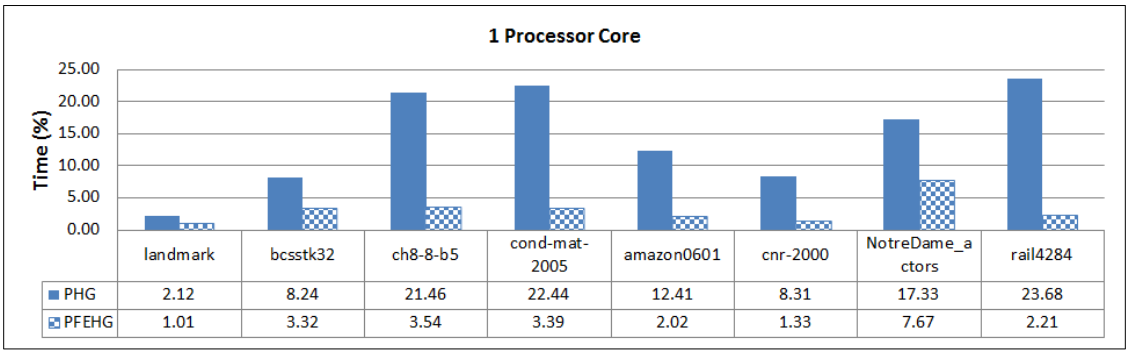
(b) FM runtime on 32 cores.



	landmark	bcsstk32	ch8-8-b5	cond-mat-2005	amazon0601	cnr-2000	NotreDame_actors	rail4284
PHG	0.0226	0.0247	0.0538	0.0175	0.0584	0.04	0.0755	0.0226
PFEHG	0.0206	0.0261	0.0749	0.0132	0.0468	0.0207	0.0417	0.11

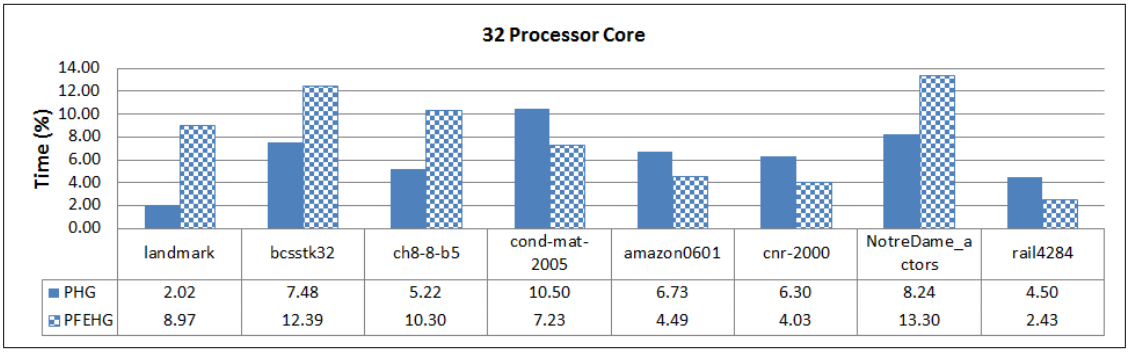
(c) FM runtime on 128 cores.

Figure 5.7: The running time of the FM algorithm on different number of processors for a bipartitioning of the hypergraphs. Two passes of FM are used for both algorithms and times are reported in seconds.



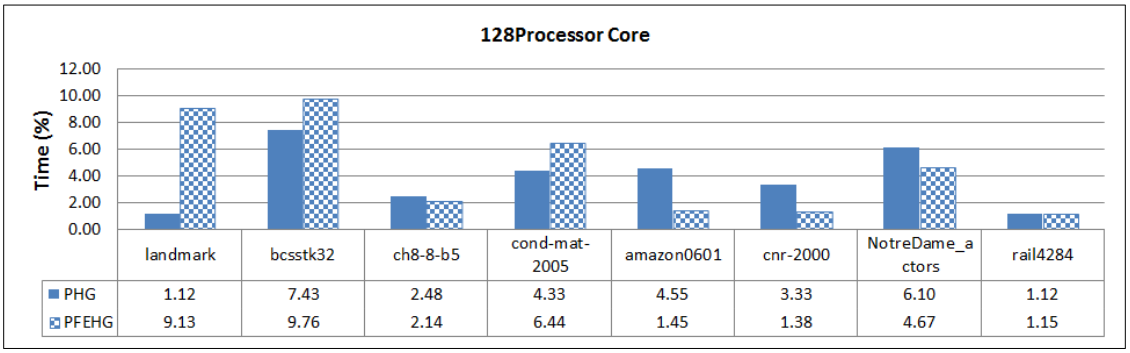
	landmark	bcsstk32	ch8-8-b5	cond-mat-2005	amazon0601	cnr-2000	NotreDame_actors	rail4284
PHG	2.12	8.24	21.46	22.44	12.41	8.31	17.33	23.68
PFEHG	1.01	3.32	3.54	3.39	2.02	1.33	7.67	2.21

(a) FM runtime on single core.



	landmark	bcsstk32	ch8-8-b5	cond-mat-2005	amazon0601	cnr-2000	NotreDame_actors	rail4284
PHG	2.02	7.48	5.22	10.50	6.73	6.30	8.24	4.50
PFEHG	8.97	12.39	10.30	7.23	4.49	4.03	13.30	2.43

(b) FM runtime on 32 cores.



	landmark	bcsstk32	ch8-8-b5	cond-mat-2005	amazon0601	cnr-2000	NotreDame_actors	rail4284
PHG	1.12	7.43	2.48	4.33	4.55	3.33	6.10	1.12
PFEHG	9.13	9.76	2.14	6.44	1.45	1.38	4.67	1.15

(c) FM runtime on 128 cores.

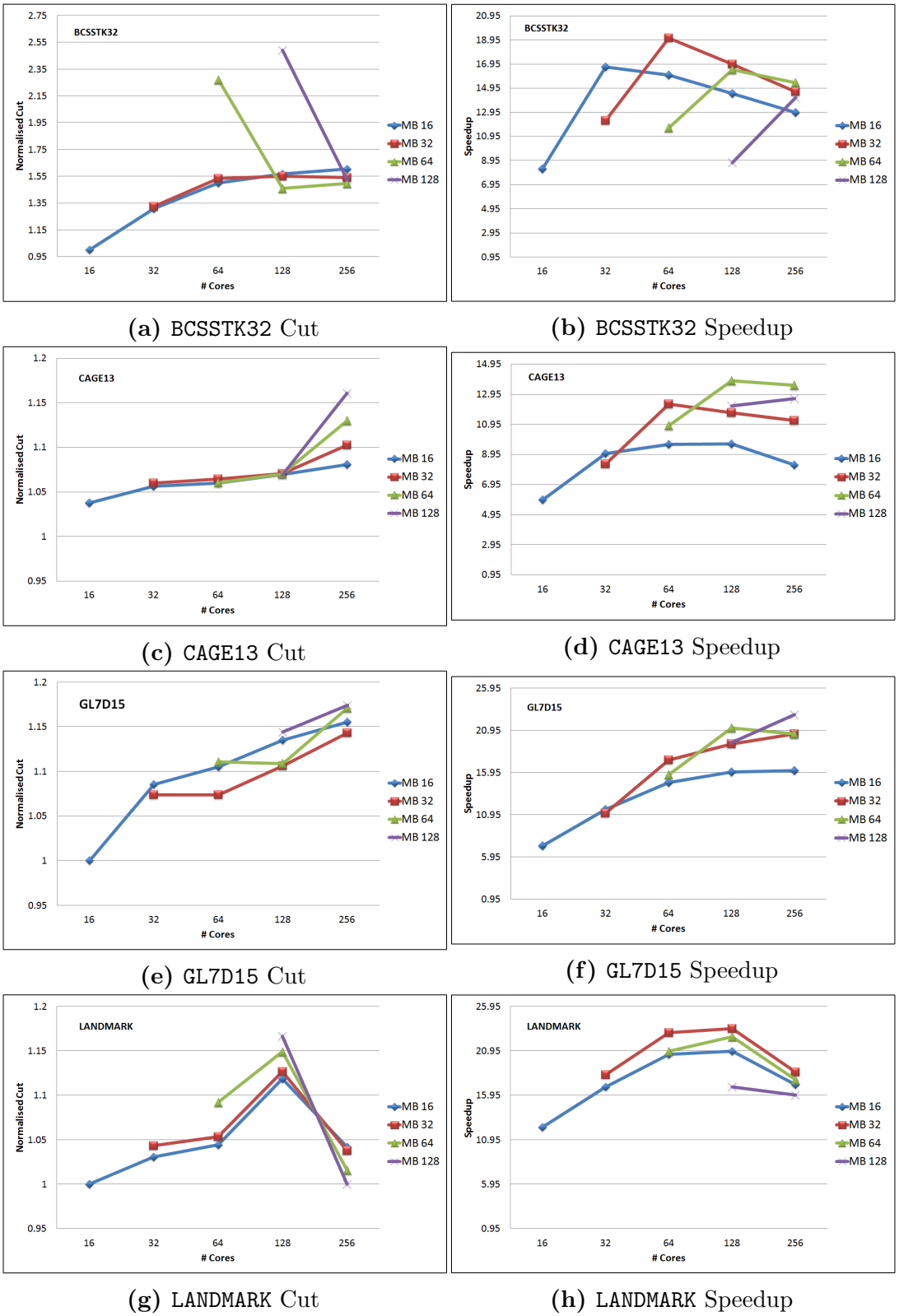
Figure 5.8: The percentage of time that algorithms spend on the FM refinement on different number of processors for a bipartitioning of the hypergraphs. Two passes of FM are used for both algorithms and times are reported in seconds.

that should be done in addition to refining the partitioning cut. First is calculating the global gain of the vertex moves as hyperedges may reside on multiple processors and the algorithm needs to know how the hyperedges spread among the parts on all processors. Second operation is the processor synchronisation during the refinement phase. Even if the refinement does not make any change to the cut, the first phase is performed in the beginning of a pass; it is the most time consuming part of the refinement phase in the *PFEHG* algorithm. According to the results, there is no advantage for one of the refinement algorithms to the other in term of running time. The runtime depends mainly on the performance of the parallel coarsening phase and the number of coarsening levels. Similar to the serial algorithm, this shows that our parallel coarsening algorithm performs very well that we do not need too much effort for the refinement phase and we do not see an overhead for our synchronised-based FM algorithm compared to the nonsynchronisation-based parallel FM algorithms.

5.3.3 Multiple Bisection Performance

In Section 5.2.4 we have introduced the multiple bisection reconfiguration and we argue that it can provide a trade-off between the quality and performance. In this section, we evaluate how the multiple bisection strategy impacts the performance and quality of the *PFEHG* algorithm. We change the number of processors from 16 to 256 and, for each evaluation, we use different values of ψ. In our tests, we select a value for the minimum subgroup size $p_{\min}$ and ψ is calculated based on the number of processors and $p_{\min}$. For example, if $p_{\min} = 32$ then ψ would be 4 and 8 on 128 and 256 processors, respectively.

The results are proposed in Fig. 5.9 for some of the hypergraphs. Evaluations show that $p_{\min} = 16$ gives the minimum partitioning cut for most of the hypergraphs. This is one of the advantages of multiple bisection such that each processor subgroup calculates a bipartitioning of the hypergraph and the best partitioning is chosen for the next bipartitioning recursion. Slightly different results are observed for `GL7D15` hypergraph, that is $p_{\min} = 32$ gives better partitioning cut compared to the $p_{\min} = 16$. This can be explained by a drawback of the recursive bipartitioning solutions compared to the direct *k-way*. Each recursion of the bipartitioning is



(a) BCSSTK32 Cut (b) BCSSTK32 Speedup

(c) CAGE13 Cut (d) CAGE13 Speedup

(e) GL7D15 Cut (f) GL7D15 Speedup

(g) LANDMARK Cut (h) LANDMARK Speedup

Figure 5.9: The 256-way partitioning cut and the speedup of *PFEHG* algorithm for variable Multiple Bisection (MB) values and different number of processors. The cut values are normalised with the partitioning cut obtained by the serial algorithm that is *FEHG*. $p_{\min}$ value is represented as MB.

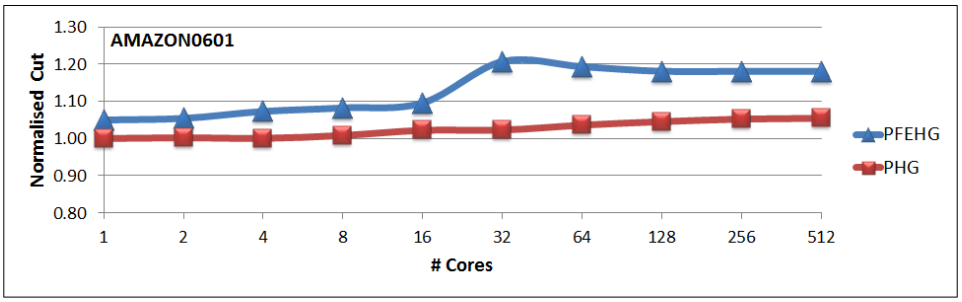
done independently without considering the forthcoming recursions and the global status of the hypergraph. This is also reported in Karypis [Kar02] as a lack of direct optimisation of cost function in recursive algorithms.

When evaluating performance, we observe the opposite results that is, generally speaking, $p_{\min} = 16$ gives the lowest speedup among others. The algorithm spends more time on hypergraph reconfiguration for smaller values of $p_{\min}$ which has a reverse effect on the performance. The idea of multiple bisection is that $p_{\min}$ processors have enough memory to hold the hypergraph, and have much less communication (the hypergraph is spread over fewer processors). On the other hand, the reconfiguration time should take less time than going through the multi-level algorithm using the whole processors in each recursion. For small $p_{\min}$, the latter is not true and the reconfiguration time overtakes the multi-level coarsening phase. In general, the evaluation shows we could get almost the highest performance for $\psi = 2$. In addition, when the number of processors goes beyond 256, choosing $\psi = 4$ gives the higher performance; with $\psi = 4$, the algorithm spend less time on processor reconfiguration. In general, this reconfiguration can provide between 2% and 48% performance improvement on average and up to 87%. For `LANDMARK` and `CH8-8-B5` we observe 7.5% and 2.1% performance degradation on 32 processors, respectively, and 10% degradation for `GL7D15` on 256 cores. In addition, the quality of the partitioning degrades from 2% to 11% on average.

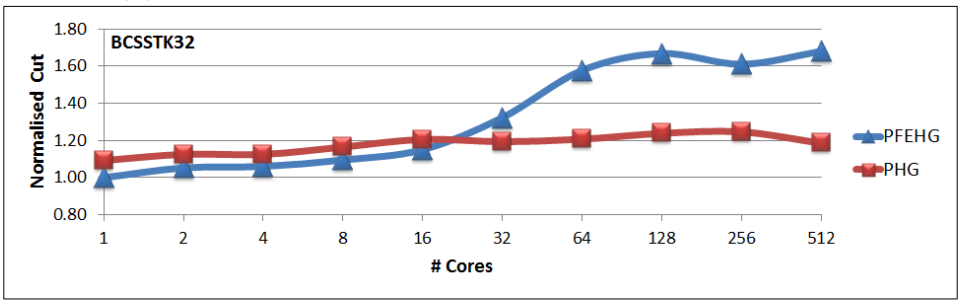
Consequently, the recursive bisection is an easy-to-apply technique for the *PFEHG* algorithm. It can provides a trade-off between the performance and quality by easily changing the replication factor. Applications that need high quality hypergraph partitioning results, should use small values for the replication factor; others that need higher performance, should use larger values of the replication factor.

5.3.4 Scalability

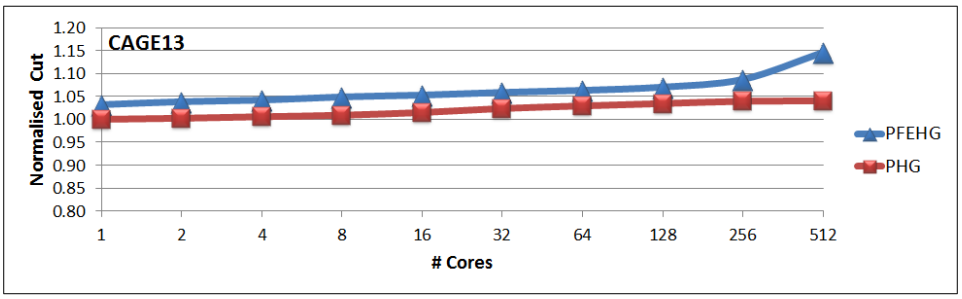
This section investigates the scalability of the algorithms on the Hamilton cluster. As mentioned earlier, an algorithm is considered to be scalable if it gives improved speedup when the number of processors increases. Among two algorithms, the one that gives its maximum speedup on larger number of processors is considered to be



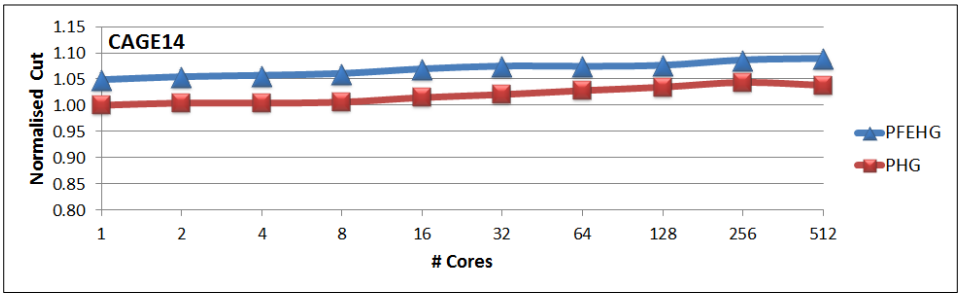
(a) AMAZON0601. Normalised to best cut that is 275,632.



(b) BCSSTK32. Normalised to best cut that is 25,558.

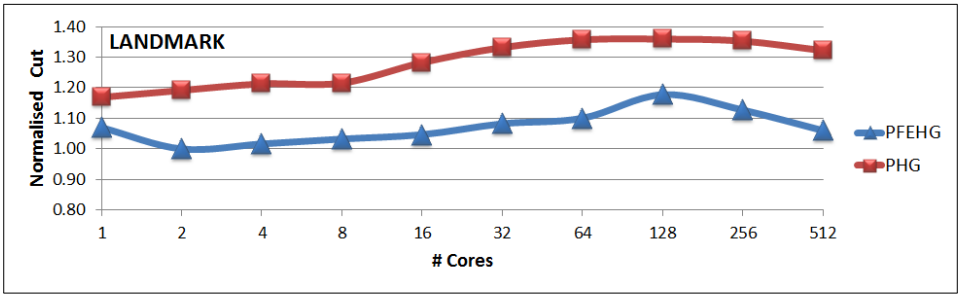


(c) CAGE13. Normalised to best cut that is 784,941.

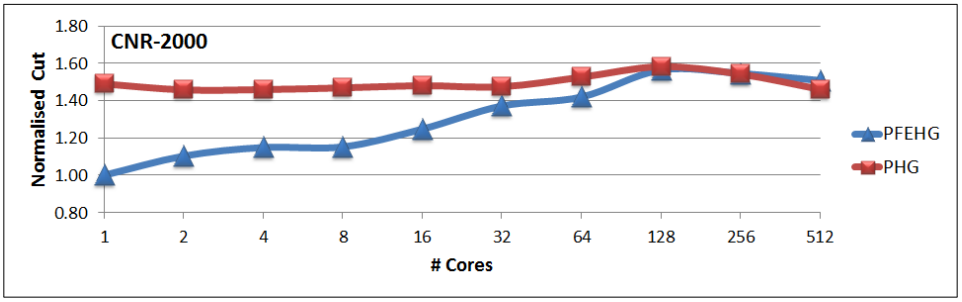


(d) CAGE14. Normalised to best cut that is 2,466,402.

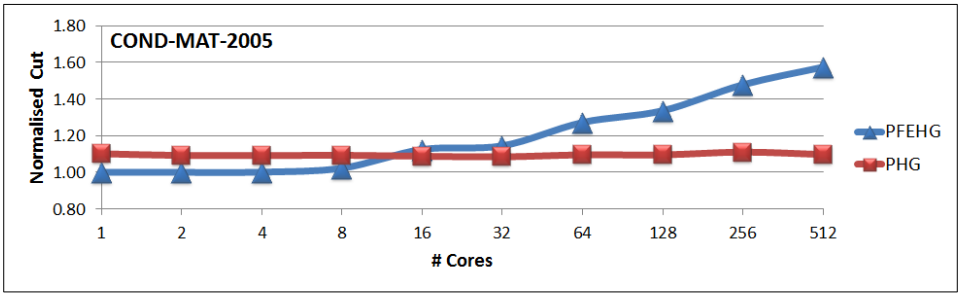
Figure 5.10: The 256-way partitioning quality comparison up to 512 processor cores. the values are normalised to the best partitioning cut among all evaluations of both algorithms (the best cut is given for each figure separately).



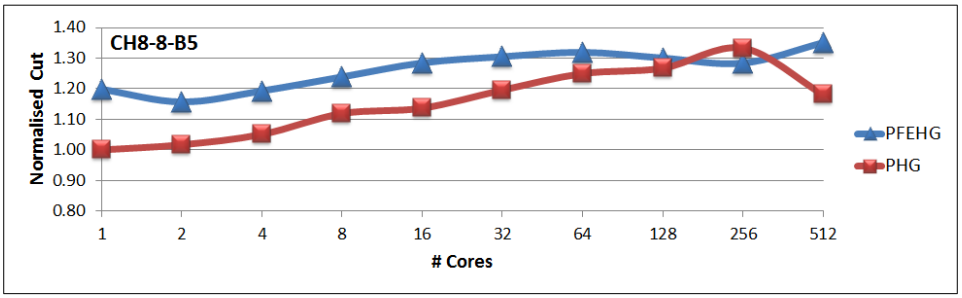
(e) LANDMARK. Normalised to best cut that is 8,076.



(f) CNR-2000. Normalised to best cut that is 16,587.

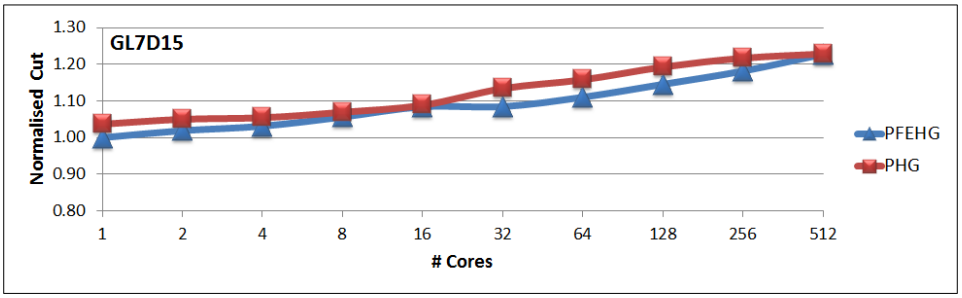


(g) COND-MAT-2005. Normalised to best cut that is 28,393.

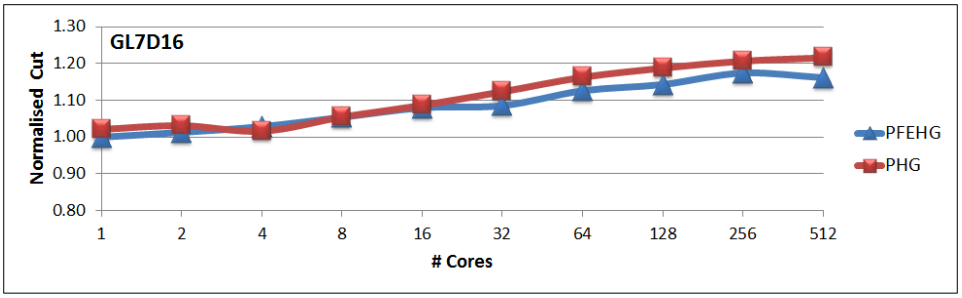


(h) CH8-8-B5. Normalised to best cut that is 831,867.

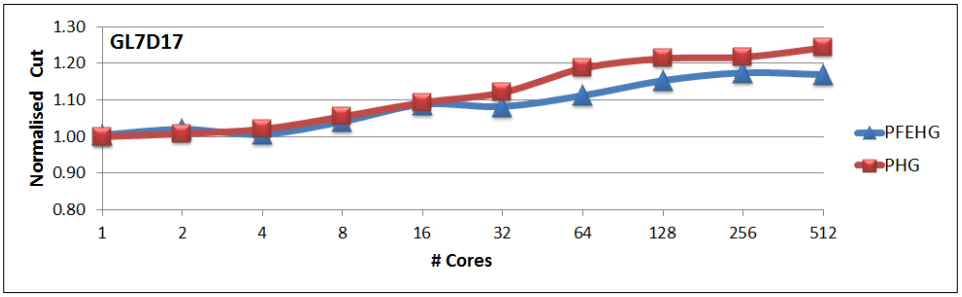
Figure 5.10: (Continued) The 256-way partitioning quality comparison up to 512 processor cores. the values are normalised to the best partitioning cut among all evaluations of both algorithms (the best cut is given for each figure separately).



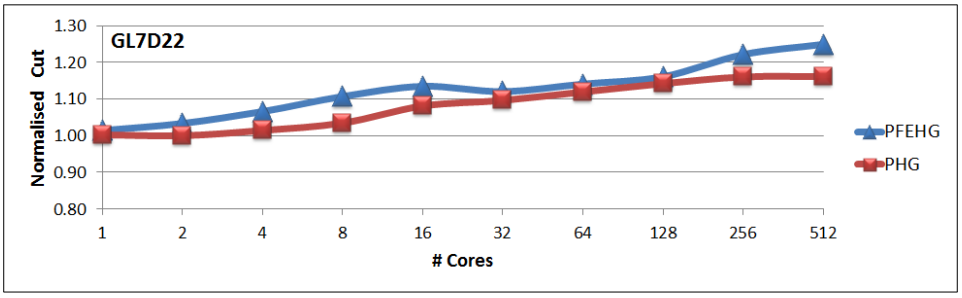
(i) GL7D15. Normalised to best cut that is 2,508,940.



(j) GL7D16. Normalised to best cut that is 10,921,171.



(k) GL7D17. Normalised to best cut that is 3,224,224.



(l) GL7D22. Normalised to best cut that is 831867.

Figure 5.10: (Continued) The 256-way partitioning quality comparison up to 512 processor cores. the values are normalised to the best partitioning cut among all evaluations of both algorithms (the best cut is given for each figure separately).

more scalable.

We run the algorithms on up to 512 processors and evaluate the quality and the speedup. When we increased the number of processors to 1024, none of algorithms give improved speedup and the performance degrades; therefore, we do not report the results on 1024 processors. The results are reported for 256-way partitioning.

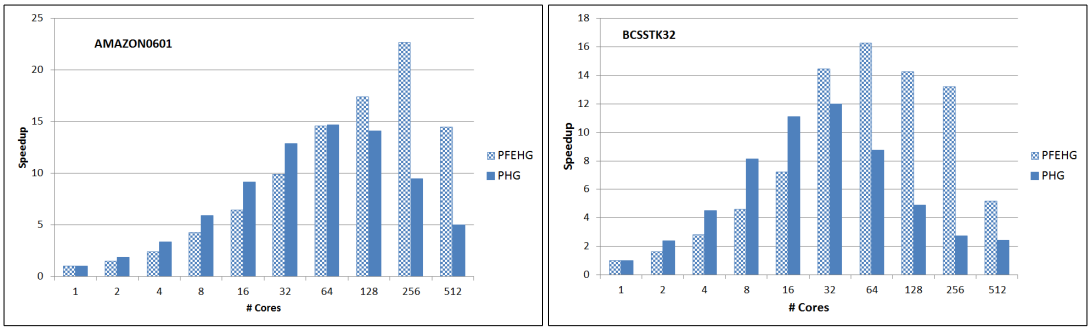
The quality of the partitioners are reported in Fig. 5.10. Both algorithms perform very competitively for the quality. The results show that the *PFEHG* algorithm performs better on the hypergraphs with more irregular structure, that is standard deviation of the vertex degree or hyperedge sizes are high. Examples of these hypergraphs are those in social networks such as hypergraphs representing friendship on Facebook and Twitter. On the other hand, when we deal with lower values of standard deviation, the *PHG* algorithm gives better quality. The same results are reported for the our serial algorithm. The *FEHG* algorithm gives better quality improvement in high dimensional data spaces with more irregularity compared to *PHG*, *PaToH*, and *hMetis*. As discussed in the previous chapter, the reason is that using local vertex similarity metric provides good clustering decisions in the coarsening phase when the irregularity in the structure of the hypergraph decreases. Using global clustering algorithms does not worth the effort; they just result in increased runtime without making any improvement to the partitioning cut compared to the local clustering algorithms. Consequently, employing algorithms such as *PHG* is preferable if an application needs higher quality partitioning results and the hypergraph to be partitioned has highly regular structure.

For example, *PFEHG* mostly generates better quality on `LANDMARK` and `CNR-2000` that have very high standard deviation for the hyperedge sizes. For the `CAGEXX` group, *PHG* generates slightly better results as these hypergraphs are more regular and they have very small deviation in the hyperedge sizes. In the `GL7DXX` group, *PFEHG* gives better quality on all except `GL7D22` that have much smaller standard deviation, which is equal to 2.2, compared to others. Furthermore, as mentioned in Section 5.3.3, a better quality can be generated for *PFEHG* by using multiple bisection if the quality has greater importance than performance. This is a solution for calculating higher quality partitions on irregular hypergraphs; on evaluated hypergraphs with regular

structure, the *PHG* algorithm still generates better partitioning cut.

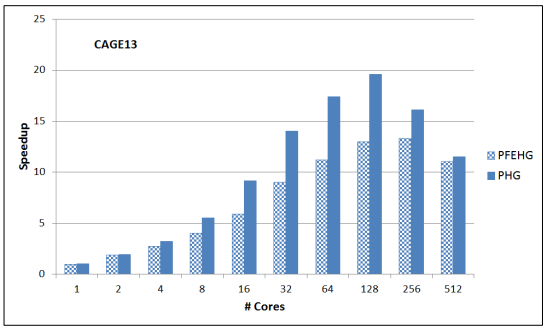
In the next evaluation, the speedup of the algorithms are tested in the cluster. The results are reported in Fig. 5.11 and Fig. 5.12. As we discussed earlier, the scalability of parallel hypergraph partitioning algorithms is limited by two factors: the structure of the hypergraph itself and the parallel partitioning algorithm. In the first case, partitioning the hypergraph imposes lots of network overhead and limits the scalability. This situation happens quite often for hypergraphs with very irregular structure and high deviation of hyperedge sizes and vertex degrees such as `CNR-2000` and `LANDMARK`. Partitioning on these types of hypergraphs are difficult to scale. According to the evaluations, *PFEHG* gives better quality and scalability for these types of hypergraphs. While the scalability of *PHG* is in all cases are limited to 32 or 64 processors, *PFEHG* gives improved speedup for up to 256 or 512 processors.

On the second set of the hypergraphs including `CAGEXX` and `GL7DXX`, our algorithm gives worse scalability compared to *PHG*. We have investigated the reasons. First, we found that these hypergraphs have very small clustering coefficient, that is 0.07 for `CAGEXX` group and 0.04 for `GL7DXX` except `GL7D22` which is 0.12. On the other hand, these hypergraphs contain only one strongly connected component of vertices. The main reason causing this issue is as follows. In the first level of the coarsening, the clustering algorithm captures vertices in the strongly connected components and pair-matches the vertices in it. Other vertices are matched using our parallel random matching algorithm. Then the hypergraph should be coarsened and the identical hyperedges should be found and collapsed. As mentioned earlier, the latter is done using the hash functions. While similarities between hyperedges are very low (this gives very small values for the CC of the hypergraph), then very few number of identical hyperedges are found. In our implementation, we have an extra condition for the coarsening phase; that is if the number of hyperedges does not decrease by a specified percentage then the algorithm stops the coarsening phase. This causes an early exit from the coarsening phase while we still have a large hypergraph. This large hypergraph should be gathered on one processor for the initial partitioning. While the size is too large, this limits the performance of the algorithm as the number of processors increases such that the initial partitioning phase take more than 40%

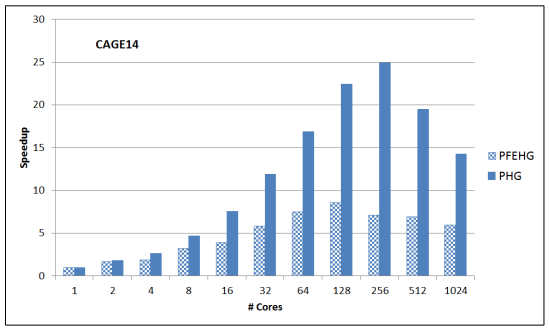


(a) AMAZON0601

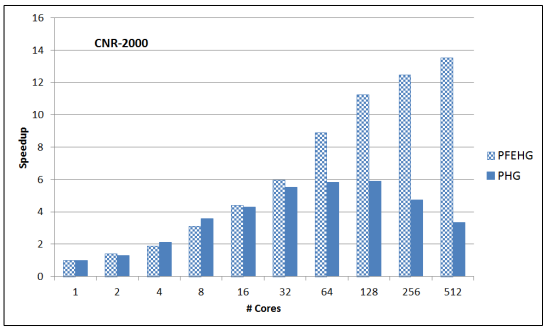
(b) BCSSTK32



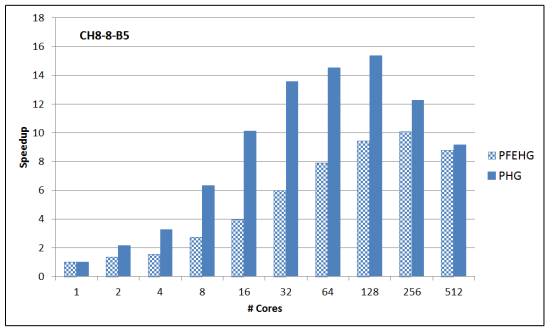
(c) CAGE13



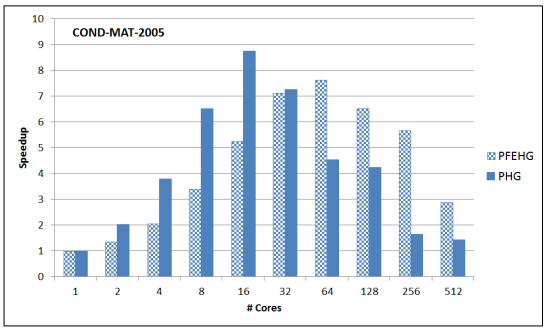
(d) CAGE14



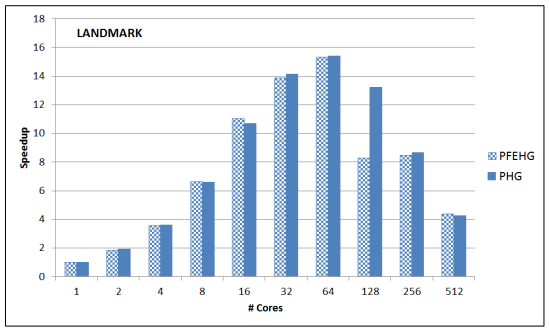
(e) CNR-2000



(f) CH8-8-B5

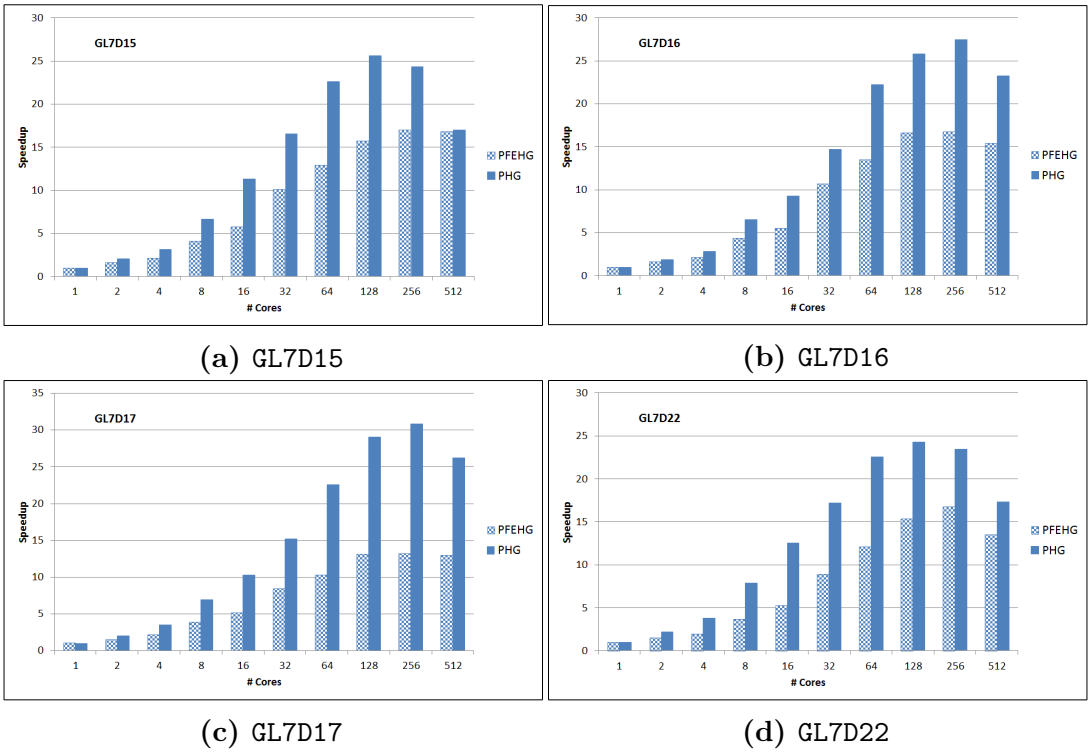


(g) COND-MAT-2005



(h) LANDMARK

Figure 5.11: Comparing the speedup of parallel algorithms on variable number of processors. The results are reported for 256-way partitioning.



(a) GL7D15 (b) GL7D16

(c) GL7D17 (d) GL7D22

Figure 5.12: Comparing the speedup of parallel algorithms on variable number of processors. The results are reported for 256-way partitioning.

of the partitioning time. For this reason, the *PFEHG* produces worse speedup than the *PHG* algorithm on the two largest hypergraphs including GL7D17 and CAGE14.

On the other hand, as we have shown, *PFEHG* performs quite well for hypergraphs with very irregular structure in term of quality and scalability. The structure of the CAGEXX hypergraphs are regular with low deviation of hyperedge sizes and vertex degrees; therefore, *PFEHG* gets less performance on this group. The last reason is due to the global vertex clustering decisions and its timing overhead. We will show later in our cloud evaluations that the performance of parallel recursive partitioning algorithms is higher on larger number of parts specially when the number of parts is higher than the number of processors. The reason is that after a few number of recursions, we have all processors concurrently partition the sub-hypergraph assigned to them and there is no network overhead. For the global clustering decisions that need more time, the performance would be much better on higher part numbers. This topic will be investigated in the next section when evaluating the algorithms in the cloud.

We have done some optimisations to resolve the above mentioned scalability issue.

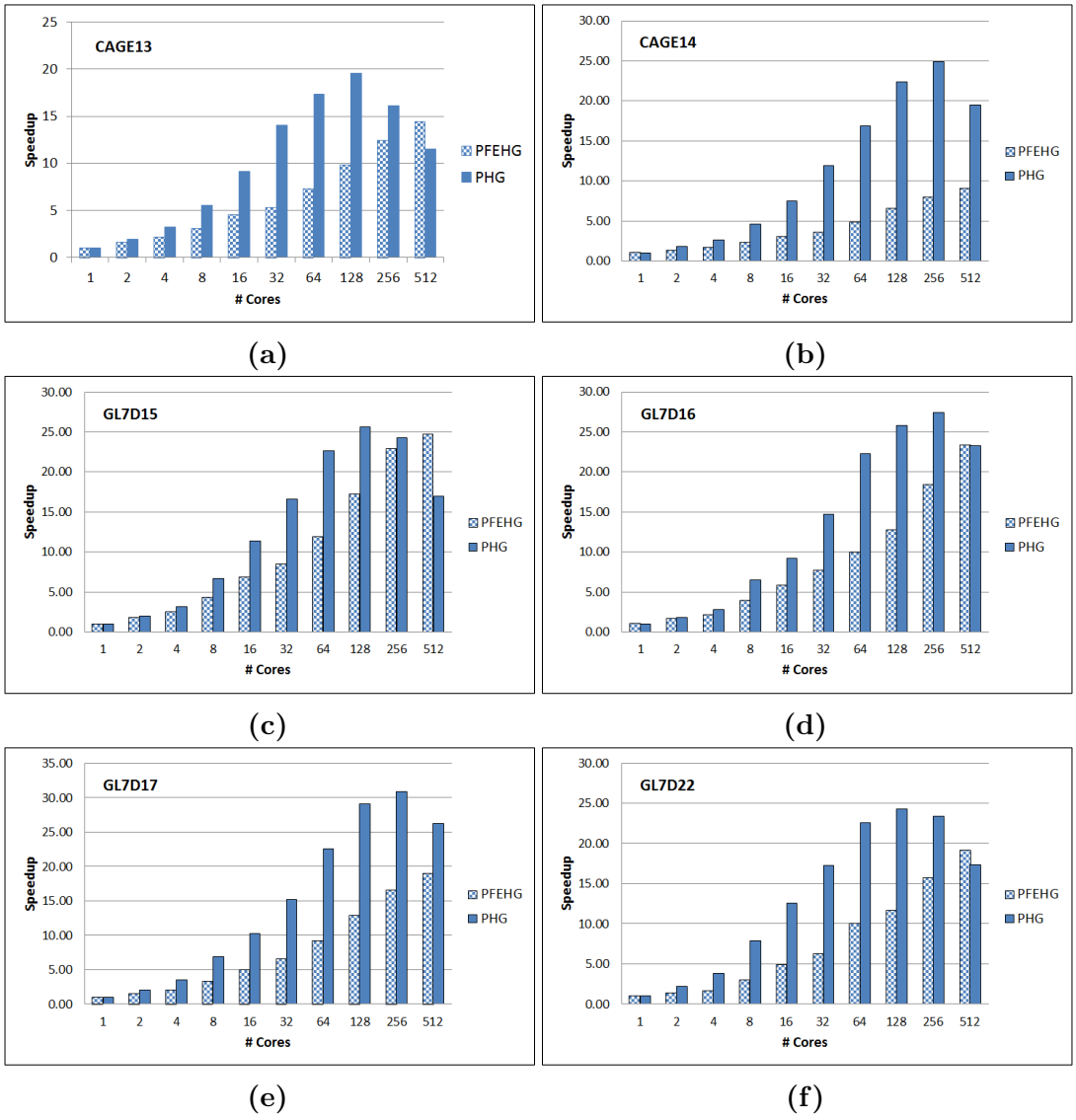


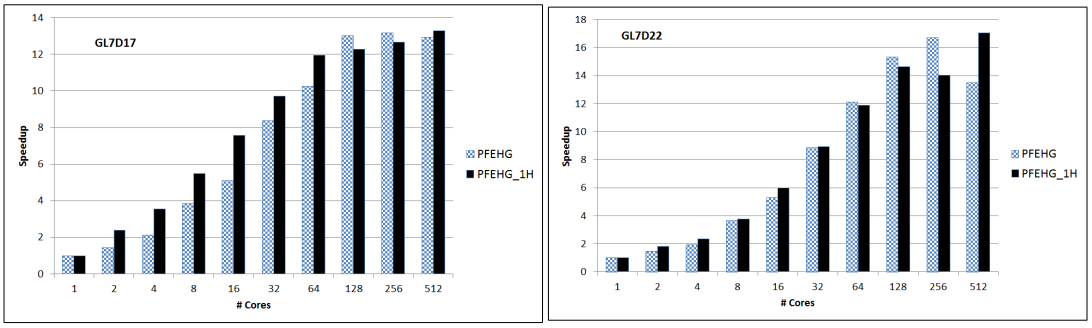
Figure 5.13: Comparing speedup of optimised *PFEHG* algorithm and the *PHG* algorithm on `CAGEXX` and `GL7DXX` hypergraphs. The results are reported for 256-way partitioning.

These optimisations are applicable when the hypergraph has very low CC such as those reported above. First, the restriction on the number of hyperedges from one coarsening level to the other is relaxed. Second, while hypergraphs are very regular with a very few number of strongly connected components of vertices, we ignore the rough set clustering and only perform local vertex matchings on processors. In this situation, we notice that a good partitioning can be found using only the parallel FM algorithm.

The results after this optimisations are reported in Fig. 5.13. For all of hypergraphs, the *PFEHG* speedup is up to 512 processors while the speedup of *PHG* is limited to 128 and 256 processors. The results show that the performance improvement is much better on `GL7DXX` hypergraphs than `CAGEXX` group such that `GL7D15` and `GL7D16` that have more irregular structure than the other two hypergraphs and *PFEHG* gives very competitive scalability compared to *PHG*. Regarding `GL7D17` and `CAGE14` hypergraphs, our evaluations show that the *PFEHG* algorithm spends most of its time in the coarsening phase. While finding similar hyperedges in this phase is based on the hashing, one solution for this could be using better hash functions and a faster data structure for looking the hyperedges on the processors using their global IDs[5].

In our evaluations, we have used a collection of twelve hash functions to distribute the hypergraphs once in the beginning of the algorithm in order to provide better data locality. We have inspected the impact of the hashing on the performance as the size of the hypergraph increases. We also evaluate the time that the *PFEHG* algorithm spends in the initial distribution of hypergraphs. We evaluate how much is the timing overhead for calculating using one hash function. For this purpose, we select the internal hash function of *Zoltan* for the initial distribution and we compare the performance to the previously proposed results that use all twelve hash functions for the initial hypergraph distribution.

[5]The current implementation uses a *Zoltan*'s internal hash function for finding similar hyperedges. Using complex hash functions is not a solution as the complexity of the hashing could be very time consuming. Furthermore, the fast lookup is using unordered map for fast lookup of hyperedges on the processors using their global IDs. This is also based on *Zoltan*'s internal hash function. We have noticed that, as the number of hyperedges increases, this lookup could be very time consuming as this is the case for the `CAGE13` and `GL7D17` hypergraphs.



(a) (b) BCSSTK32

Figure 5.14: The speedup improvement of the *PFEHG* algorithm when only one hash function is used for the initial hypergraph distribution. Improvements are only obtained for small processor counts. The results are reported for 256-way partitioning.

On small hypergraphs, no performance degradation is observed. The only differences are observed for the GL7D17 and GL7D22 hypergraphs and their performance changes are depicted in Fig. 5.14. According to the figures, the hash function has higher impact on the performance when the number of processors is small. For example, the hash time overhead is more than 130 seconds for GL7D17 hypergraph when the number of processors is two. As the number of processors increases, the average size of the sub-hypergraphs on processors decreases; therefore, calculating the hash values for the initial distribution phase is done quicker as we increase the number of processors and it gives only a tiny performance degradation if we use all twelve hash functions. The major improvement is for the GL7D17 hypergraph, that is between 2 and 64 processors and up to 40% higher speedup if we use only one hash function.

Discussions

In the following we discuss the results of our evaluations regarding the parallel algorithms.

1. Our evaluation shows that the scalability of the *PHG* algorithm is limited and it generally saturates at 64 or 128 processors and so the scalability of *PFEHG* is better on most of our test cases which is shown to give improved speedup using up to 512 processors. We have shown that *PFEHG* demonstrates the greatest advantage on hypergraphs with irregular structure such as those that

represent social networks. It calculates partitionings with better partitioning cut compared to *PHG*. On others, it is the case that *PHG* sometimes generates better quality outputs. We note that *PFEHG* can be improved by increasing the *replication factor* in our multiple bisection strategy (The quality improves by up to 11 per cent).

2. While the algorithms are recursive bipartitioning algorithms, two processor reconfiguration strategies are used in each recursion: bisection processor splitting and multiple bisection. The first is shown to provide better performance on parallel hypergraph partitioners such as *PHG*. The latter, which is our proposed strategy, executes multiple runs of the hypergraph partitioning in parallel and on subgroups of processors independently. The strategy should make sure that the hypergraph can fit into the memory of each processor subgroup. There are two advantages of this method. First, it provides an easy-to-implement trade-off between the quality and speedup (as discussed in Section 5.3.3). Using more parallel runs, provides better quality but less performance and scalability. Second, the strategy improves the scalability of parallel hypergraph partitioners on irregular hypergraphs (parallel partitioning algorithms do not scale well on these types of hypergraphs).

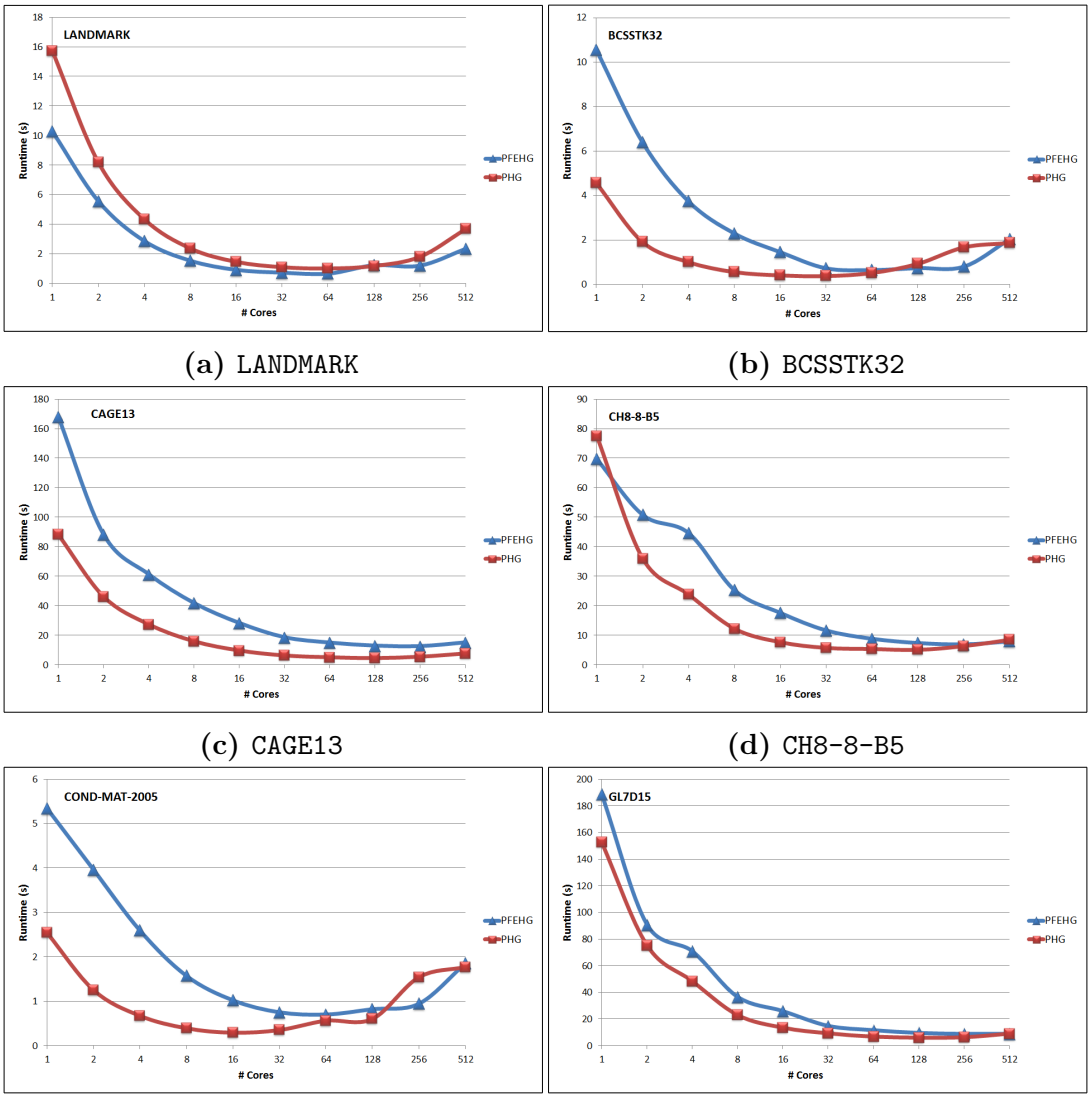
3. The only limiting factor of recursive bisectioning is the processor reconfiguration time. In order to achieve performance improvement, the reconfiguration time should be kept low. As we noticed, when we use large number of processors, choosing a larger replication factor gives better performance improvement in order to keep the reconfiguration time as low as possible. This situation happened in our evaluations when we had more than 256 processors such that choosing $\psi = 4$ gives better scalability than $\psi = 2$. In our implementations, we use simple heuristics for the replication of hypergraph on processor subgroups. Any investment in decreasing the reconfiguration time gives better performance improvement and higher scalability and this is planned as future work.

4. As discussed earlier, the parallel refinement algorithm is the most difficult phase of multi-level to parallelise as the proposed algorithms, such as FM algorithm,

are inherently serial. We showed that any investment in the coarsening phase (for making better clustering decisions) will leave less effort as well as less restrictions in the refinement phase . This is opposite to the already proposed parallel hypergraph partitioners such as *PHG* and *Parkway* that impose strict limitations on the hypergraph in order to provide more speedup in the refinement phase. For example, they allow only one-way move in each pass of the algorithm and they avoid processor synchronisations. The idea for imposing these restrictions comes from the parallel graph partitioning algorithms that have a very different structure than hypergraph. By removing the restrictions of previous algorithms (which are one-way moves and local balance constraints), we showed that the new algorithm still generates good partitioning results with comparable runtime. This idea is, in itself, something that merits further research.

5. As discussed in Chapter 4, using global clustering decisions comes at a cost that is increased running time. In the previous chapter, where we propose our serial algorithm, we showed the superiority of *FEHG* and we identified the types of the hypergraphs that can benefit the most from our serial algorithm. The same results are reported here for the parallel algorithm. In Fig. 5.15 the runtime of algorithms on different types of the hypergraphs is reported. The global clustering decision is the main bottleneck. The difference in runtime between *PFEHG* and *PHG* decreases by increasing the number of parts[6]. To solve the issue, we suggest using multiple vertex matches instead of the pair-match strategy. In Chapter 4, we showed that the multi-match strategy can improve the runtime of the algorithm up to 30% on some of the hypergraphs. We expect that the improvement should be higher in our parallel algorithm. The reason is that matching cores in the first phase of our parallel matching algorithm provides less cost than the second phase, which is global random matching.

[6]Our algorithm that is based on global vertex clustering decisions benefits more from increasing the number of parts while other algorithms that are based on local clustering decisions, such as *PHG*, benefit less.



(a) LANDMARK (b) BCSSTK32

(c) CAGE13 (d) CH8-8-B5

(e) COND-MAT-2005 (f) GL7D15

Figure 5.15: The runtime of the parallel algorithms on HPC cluster. The results are reported for 256-way partitioning. *PFEHG* gives higher runtime dues its global vertex clustering algorithm and using pair vertex matches instead of the multi-match strategy.

5.4 Hypergraph Partitioning in the Cloud

5.4.1 Why in the Cloud?

In Chapter 3.4, we discussed about the interest for moving scientific and distributed applications into the cloud. The reason for this transition is the advantages that the cloud provides for scientific applications such as elasticity, small start-up and maintenance costs, dynamic resource allocation, and economies of scale and use [MAB+10,YCD+11]. Cloud services are offered in virtualised form that is the enabling technology of the cloud [BVS13]. Virtualisation, which is built on top of a physical computing node, separates the node into one or more *virtual* instances and offered in forms such as virtualised storage, and virtualised network. Despite these advantages, the limited communication bandwidth of the cloud makes it less suitable for running communication-intensive applications and they suffer from poor scalability in the cloud [GKG+13].

The hypergraph partitioning with the load balance constraint provides an efficient approach for reducing the communication volume and increasing the performance of parallel applications [ÇA99,MAB+10,MLLS14]. These parallel applications can benefit from hypergraph partitioning because partitioning an application before running it in the cloud can lead to considerable performance improvements due to savings in hardware and network resources usage. In this section, we discuss the benefits of using hypergraph partitioning for running communication-intensive applications in the cloud. We provide use cases and discuss why we need parallel scalable hypergraph partitioners in the cloud.

The first use case is the parallel sparse matrix-vector (SpMV) multiplication, one of the kernel operation in many scientific applications such as iterative solvers. It is notorious for using a low fraction of peak processor performance [WOV+09]. The hypergraph partitioning can be employed to improve the performance of SpMV operations because the hypergraph cut metric can exactly model the communication volume between processors. It can effectively reduce the inter-processor communication volume between 30% and 38% on average [ÇA99]. As the limited network resource of the cloud is the main scalability bottleneck [JRM+10], SpMV-based

applications can gain considerable benefit from hypergraph partitioning. The same argument can be made for all HPC applications modelled with hypergraphs.

As the second use case we discuss large-scale graph processing tools. Data is often represented by large graphs that must be frequently analysed; for example, Google's PageRank calculations [BJKT05,LM11] or Facebook's processing of its friendship network (such as calculating the average number of friends for each member). The graphs are distributed and processed between multiple systems and there are scalable systems designed for processing them such as Pregel-like vertex-centric large-scale graph processing tools [MAB+10] such Apache Giraph [7], which is currently used at Facebook to analyse the social graph formed by users and their connections. Partitioning the graph is shown to provide balanced resource utilisation among processors and increases data locality with less network communication [MLLS14,WXSW14]. While the first is important due to limited network resources of the cloud, the second provides efficient resource utilisation of the shared resources in the virtualised environment.

There are several graph partitioning algorithms for this purpose. Using the graph partitioning tools such as *Metis* can provide considerable performance improvement and scalability to these system compared to the random hash partitioning such as works by Wang et al. [WXSW14], Chen et al. [CBL+14], and Ho et al. [HWL12]. These works are based on the serial graph partitioning. Two problems arise here. First, the size of current graphs is very big and contain millions (or billions) of vertices and edges and the capacity of serial algorithms is limited that emphasis the need for parallel algorithms. Consequently, we need parallel graph algorithms such as *ParMetis*. Second, as discussed in Chapter 3.3, there is a problem in modelling group relations such that graphs cannot fairly capture group relationships and we need a better representation of relations in big graph processing such as hypergraphs [HC14]. To cover these issues, algorithms such as *Spinner* [MLLS14] are introduced that is a parallel algorithm and is based on label propagation methods that provides better results than *Metis* graph partitioner, but can not still capture and represent

[7] `http://giraph.apache.org/`.

group relations as good as hypergraphs. It could be considered as a variant of the hypergraph partitioner.

We summarise some of the advantages of hypergraphs that motivates using the cloud for distributed applications as follows:

1. it provides less network utilisation while the hypergraph can exactly model the communication model within a distributed application.

2. considering the network heterogeneity of the cloud and uneven communication bandwidth, hypergraph partitioning employed with efficient mapping algorithms can provide better network resource utilisation and performance optimisation.

3. the load balancing condition of hypergraph partitioning can deal with the computational resource heterogeneity of the cloud including issues such as multi-tenancy [GSKM13].

To sum it up, using the hypergraph partitioning, as discussed to be a much better solution than the graph partitioning, provides lots of advantages for moving the HPC and distributed applications into the cloud, those that have different characteristics than web applications. The advantages can be summarised as follow.

1. Hypergraphs can exactly model the communication pattern in distributed applications. Parallel applications can make better use of the limited network resources of the cloud using hypergraph partitioning. The results are better performance and scalability than random or graphs partitioners.

2. Considering the network heterogeneity of the cloud and uneven communication bandwidth between computing nodes (for example: due to the hierarchical design of the cloud, there is higher communication bandwidth between computing nodes in the same rack than the bandwidth available between two computing nodes in two different racks [BCH13]). The hypergraph partitioning employed with efficient mapping algorithms can provide better network resource utilisation and performance optimisation. It can be used to make better utilisation of the uneven network heterogeneity in the hierarchical design. Example is the work by Chen et al. [CYW+12].

3. The load balancing condition of hypergraph partitioning can deal with the computational resource heterogeneity of the cloud including issues such as multi-tenancy [GSKM13]. This criteria provide a balanced distribution of the work among the computing instances and better resource utilisation.

Partitioning the distributed application can be done in two ways: 1) before transferring it into the cloud, and 2) in the cloud as a pre-processing step within the application. While the first solution decouples the performance of the hypergraph partitioning from the distributed application, it is not always feasible and practical because it demands extra local resources (which is not always feasible). Therefore, we focus on the second solution and investigate the challenge on the way. There is an issue for the second solution. The overall runtime of the application is the sum of the runtime of the distributed application and the hypergraph partitioning itself. While the parallel hypergraph partitioning is a communication-intensive application, running it in the cloud is challenging, suffers from poor scalability, and can increase the overall runtime.

Due to the need for parallel scalable hypergraph partitioning algorithms in the cloud, we investigate how our proposed algorithm scales up in the cloud compared to the *PHG* algorithm. We identify challenges and problems on the way and provide solutions. All of cloud's limitations should be taken into account when we transfer distributed applications into the cloud that is not a straightforward and easy task. Sometimes the structure of the applications becomes a performance bottleneck. The reader is referred to Chapter 3.4 for the details of the problems. Identifying the characteristics of the distributed application before the transfer is crucial and they are necessary in order to achieve cost-performance benefits. The same discussion stands from algorithm design point of view. We will see in the next section how these features are provisioned in our parallel algorithm.

5.4.2 System Configuration and Algorithm Parameters

We run our evaluations on a private cloud in the University of Mainz, Germany which is controlled and managed by OpenStack. The cluster has 34 compute nodes in each

rack. Each node has 2 PCPUs and each CPU has 8 cores, 64 GB of RAM and 250 GB of hard disk. The processor model of cores is Intel(R) Xeon(R) E5-2650 2.00GHz. Nodes are connected by 1 Gbps Ethernet switches and networking is controlled with OpenStack *nova* network management. Comparing to the Infiniband network in Durham Hamilton cluster, this is much slower connectivity. For our evaluations, we create a testbed with 16 Virtual Machine (VM) instances each having 4 cores, 8 GB of RAM and 40 GB of hard disk. The operating system running on instances is 32-Bit Ubuntu 12.04 LTS. The *Zoltan* library is compiled and built with OpenMPI version 1.8.2. To evaluate only the partitioning time without file system overheads, we copy test data on the local file system of VMs.

The similarity threshold is calculated similar to the approach used for the evaluations of algorithms in Durham HPC cluster with one difference; as network resources are limited in the cloud, we calculate CC in the beginning of the algorithm and its value is readjusted in each coarsening level. Furthermore, we use the history of the CC for the upcoming recursions of the algorithm such that the average of the CC over all coarsening levels is calculated. After each recursion, the CC for each sub-hypergraph is the average CC history times the density of the sub-hypergraph[8]. This way, we use less time in calculating CC in each recursion of the algorithm.

We set redistribution imbalance in Section 5.1 to 0.1 and a collection of twelve hash functions are used for the hypergraph initial distribution (the same as our configuration done in HPC cluster). The reader is referred to Appendix B.3.2 for the details of the hash functions. The number of passes for the refinement function is set to two (the same is done for *PHG*) and the *token* value is set to 16. This means that processors can move a maximum of 16 vertices when holding the *token*. Minimum subgroup $p_{\min}$ is set to 8 in multiple bisection. The value is selected because of the slow network connectivity of the cloud compared to HPC cluster and in order to have more data locality. The replication factor ψ is set according to the number of processors and $p_{\min}$. For example, for 16 and 32 processors, ψ would be 2 and 4, respectively. *PHG* is initialised with default values using agglomerative clustering for

[8]The density is simply calculated as the number of pins divided by the number of hyperedges.

Table 5.2: *PFEHG* vs *PHG* runtime in the cloud for $k = 256$ with 1, 8, and 64 cores. The values are reported in seconds.

	cores=1		cores=8		cores=64	
	PFEHG	PHG	PFEHG	PHG	PFEHG	PHG
NOTREDAME	37.25	22.84	14.53	9.26	44.49	392.3
AMAZON0601	104.6	36.27	23.3	10.67	85.83	283.9
BCSSTK32	9.54	4.05	1.97	1.21	26.41	14.95
CAGE13	134.76	80.45	32.55	18.54	78.7	209.2
CAGE14	613.1	349.72	196.34	87.74	267.34	466.95
CH8-8-B5	60.58	69.15	17.22	14.96	59.33	231.86
CNR-2000	67.81	15.05	20.77	8.5	126.42	421.13
COND-MAT-2005	4.6	2.16	1.89	0.98	17.02	12.89
GL7D15	137.88	139.74	27.3	35.94	69.51	528.84
GL7D16	431.62	411.06	96.38	94.73	136.7	575.75
GL7D17	1039.8	881.16	281.54	181.02	285.7	738.46
GL7D22	278.6	214.2	71.58	35.73	115.91	323.7
LANDMARK	9.87	12.32	1.72	4.98	8.13	124.83
RAIL4284	114.22	100.8	78.41	22.05	96.28	183.7

the vertex matching algorithm. The imbalance tolerance is set to 5%. Algorithms are tested for 256 and 1024 part numbers and the reported results are the average of 10 runs for each algorithm.

5.4.3 Scalability

The scalability of algorithms in the cloud is evaluated on the test hypergraphs depicted in Table 5.1. In order to measure scalability, we compare the speedup of the algorithms. The speedup is defined as the ratio of partitioning time on one processor to the time required to solve the partitioning on the parallel system. An algorithm is considered to be more scalable if the speedup improvement lasts longer as we increase the number of processors.

Tables 5.2 compares the runtime of *PFEHG* to *PHG* on 1, 8, and 64 processors and *256-way* partitioning. The complete evaluation results are depicted in Fig. 5.16 to Fig. 5.19. According to the results, the *PHG* speedup is achieved up to 4 and, in some cases, up to 8 virtual machine cores. It achieves very poor speedup when the number of cores exceeds the number of cores per virtual machine, that is four cores.

After eight core, the runtime of *PHG* increases dramatically such that the run time is incredibly higher than the runtime on one core. The increase in runtime for the two largest hypergraphs are 34% for `GL7D17` and the runtime increases by more than 2× for the `CAGE13` hypergraph.

On the other hand, *PFEHG* gives much better results. On small-sized hypergraphs, it gets better results than *PHG* such that it gives improved speedup up to 8 cores for all hypergraphs. The speedup starts to increase on 16 cores onward. On most of hypergraphs, the algorithm gets better runtime compared to the serial algorithms. Similar to *PHG*, the algorithm gets better performance and scalability as the size of the hypergraph increases and it can be inferred from `GL7DXX` group. The algorithm gets improved speedup for up to 32 cores on the `GL7D17` hypergraph that is 4.6×. The observed higher runtime for *PFEHG* on 8 cores compared to *PHG* is the discussed in the previous section and a solution is proposed that is making multi-match decisions instead of the using pair-matches. This strategy can improve the runtime up to 30% in the serial *FEHG* algorithm, as discussed earlier, and the improvement can be higher in the parallel algorithm.

As discussed earlier, serial hypergraph partitioning algorithms generate smaller partitioning cuts compared to parallel algorithms. The reason is that processors have less local data for making partitioning decisions as the number of processors increases. In order to evaluate the quality, we are looking for algorithms that give comparable quality as serial algorithms. The changes in partitioning cuts for different number of processors are reported in the figures. Both *PFEHG* and *PHG* algorithms give good stability of the partitioning cut with increasing the number of processors. According to the results on our HPC cluster, *PFEHG* gets better partitioning quality on irregular hypergraphs; for others, *PHG* sometimes gets better quality.

One interesting result is observed for `RAIL4284` hypergraph that has average vertex degree 2633 compared to the average hyperedge size 10 and in this situation, *PFEHG* spends more than 95% of its time in calculating HCG. In this type of hypergraph, a near optimal partitioning on the hypergraph can be calculated by simply running FM algorithm on it without going through any coarsening levels. As mentioned in Chapter 3.1.1, one of the targets of the multi-level approach is to

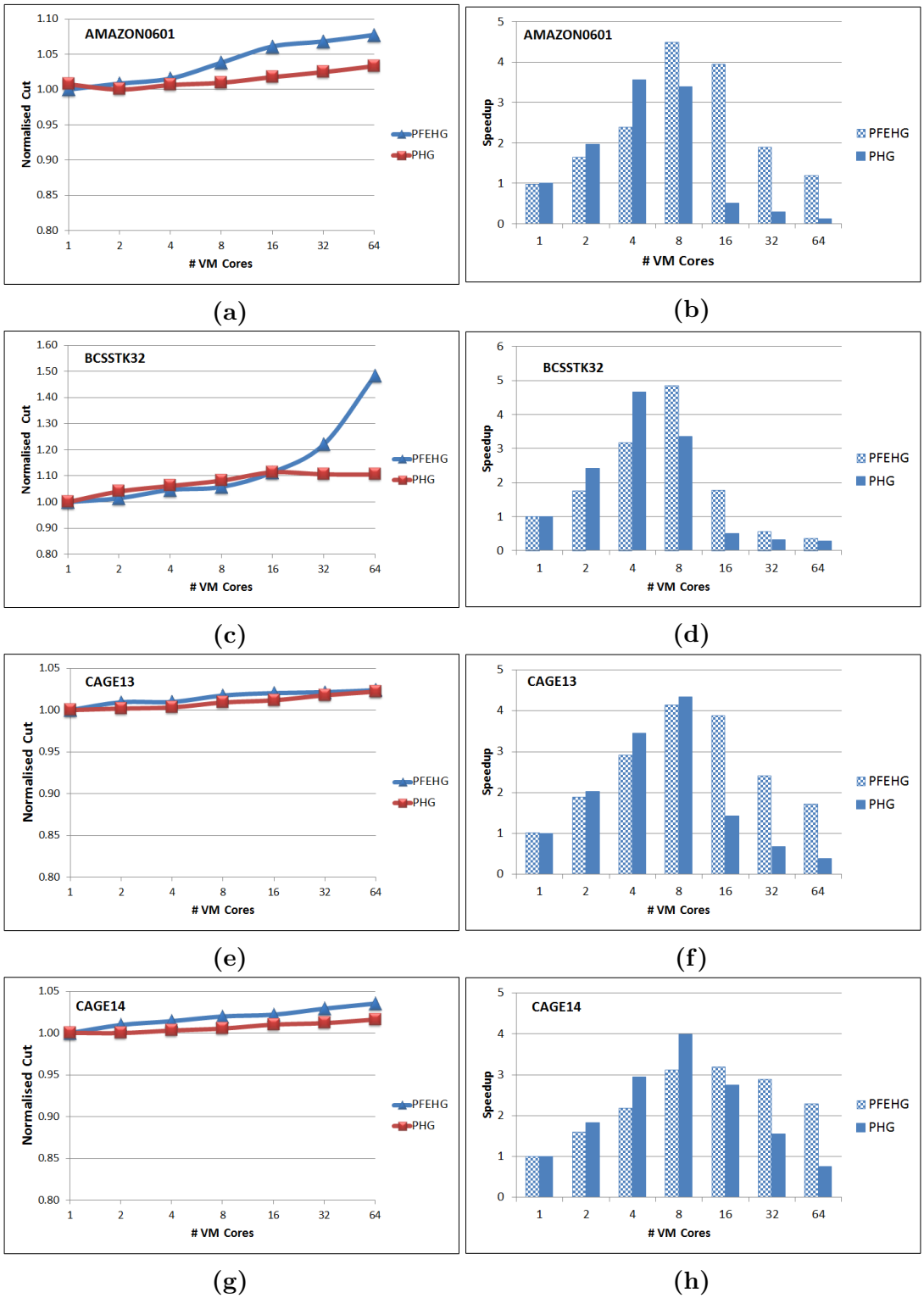


Figure 5.16: The quality and speedup of the algorithms in the cloud for $k = 256$. The partitioning cut is normalised with the average best cut obtained for each algorithm.

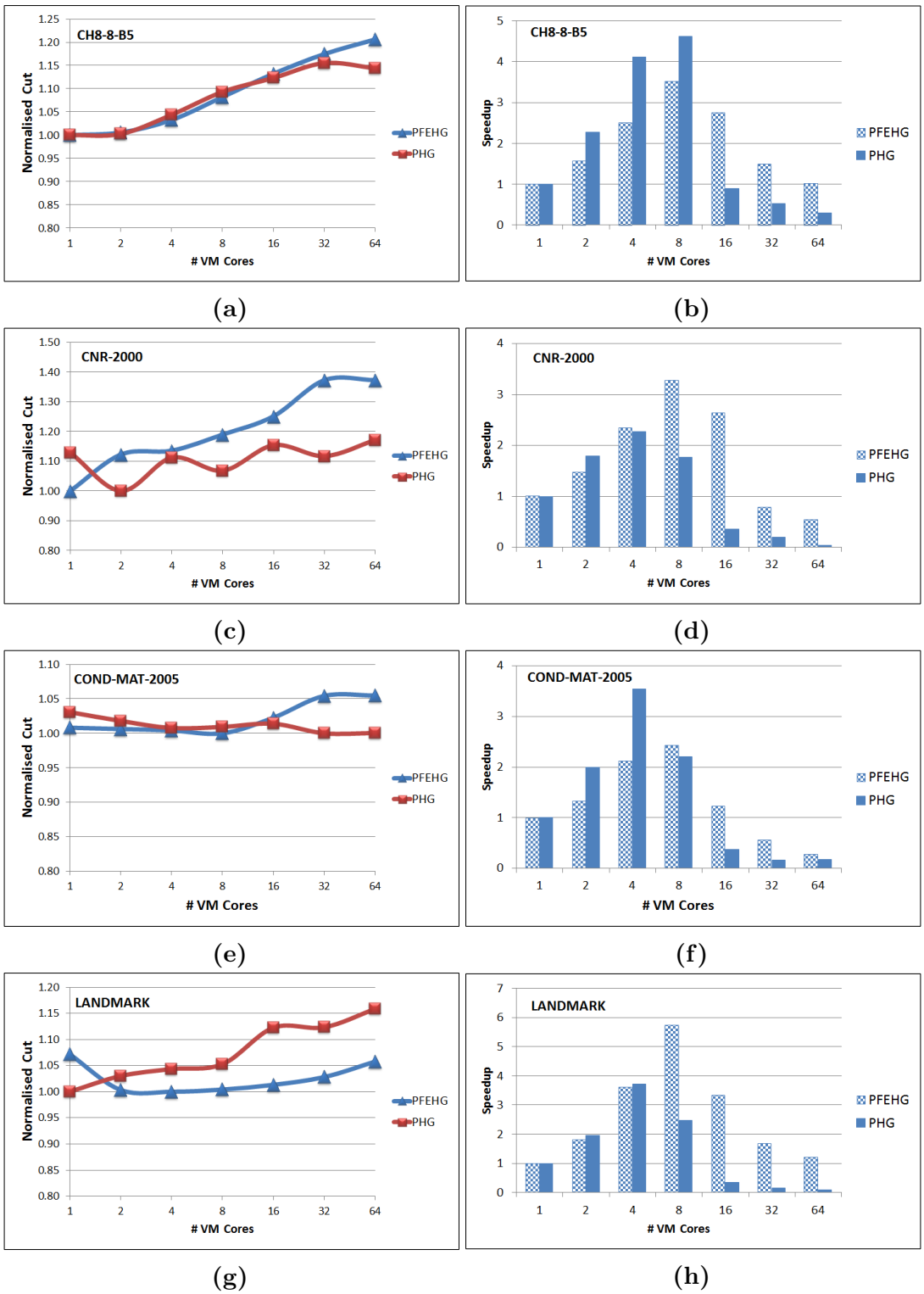


Figure 5.17: The quality and speedup of the algorithms in the cloud for $k = 256$. The partitioning cut is normalised with the average best cut obtained for each algorithm.

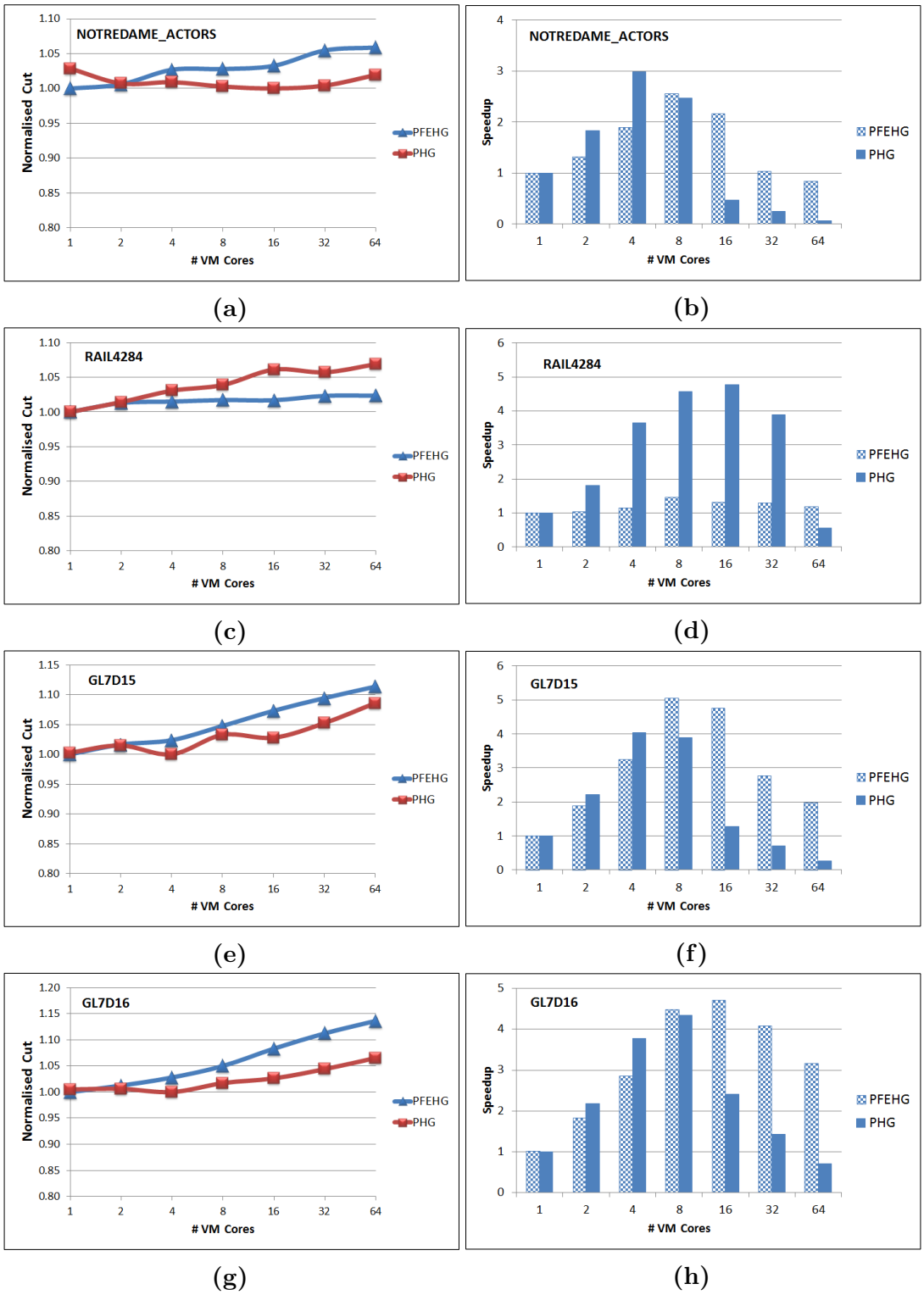


Figure 5.18: The quality and speedup of the algorithms in the cloud for $k = 256$. The partitioning cut is normalised with the average best cut obtained for each algorithm.

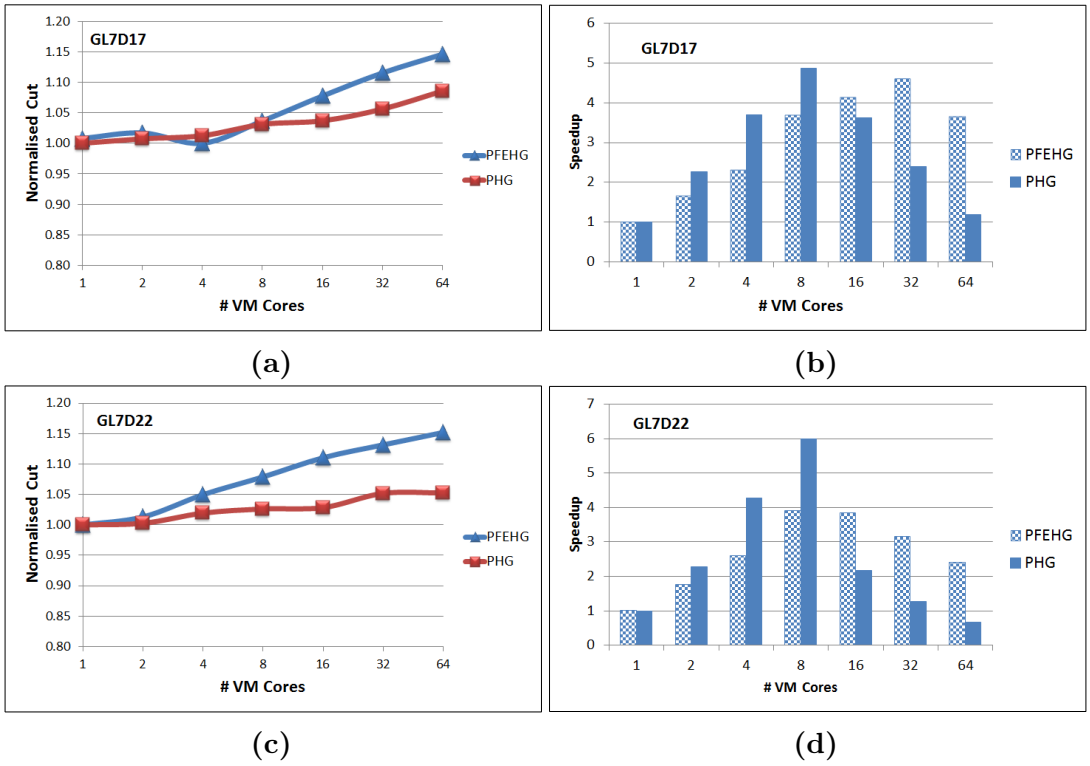


Figure 5.19: The quality and speedup of the algorithms in the cloud for $k = 256$. The partitioning cut is normalised with the average best cut obtained for each algorithm.

increase the average vertex degree in the coarsened hypergraph. The reason is that in high connectivity space, the iterative move-based algorithms can get near-optimal solutions and the possibility of getting stuck in local minima is very low. Considering this fact, we have optimised its running time by 95% from $1,111$ seconds to only 114.22 seconds on one processor. In the parallel algorithm, the runtime improves more than 50%.

The proposed recursive bipartitioning algorithms are based on the divide-and-conquer strategy for the partitioning and use bisection processor splitting in each recursion as explained in Section 5.2.4. As the number of processors goes beyond four[9], the parallel partitioners start using network resources. The network resources is only used for the first $\log(p/4)$ recursions if $p > 4$; then all communications are done locally. Consequently, the partitioning process in the cloud can be divided into two epochs. In the first epoch, which is first $\log(p/4)$ recursions for our cloud configuration, the performance degrades due to slow network bandwidth of the cloud. In the second

[9]The number of processing cores per virtual machine is four.

epoch, all communications happen locally and we expect performance improvement. Overall, the parallel partitioning algorithm gives performance improvement if the decrease in runtime in the second epoch compensates the increased runtime of the first epoch. The *PHG* algorithm is shown to be unable to overcome the bottleneck in the first epoch; therefore, the runtime starts to increase as soon as the number of processors goes beyond four.

According to this discussion, increasing the number of parts should have positive effect on the performance. To investigate this, we increase the number of parts from 256 to 1024 and we have evaluated the algorithms and their performance. We did not observe any performance improvement for the *PHG* algorithm; therefore, only results for the *PFEHG* algorithm is reported. For hypergraphs including `BCSSTK32`, `CH8-8-B5`, and `COND-MAT-2005`, *PFEHG* gets 17%, 33%, and 5.5% performance improvement on 8 cores, respectively. The other results are reported in Fig. 5.20. On all large hypergraphs, the performance improvement is between 26% to 33% on 16 cores.

Finally, we investigate the effect of the vertical scalability on the performance of the algorithms by using higher-end compute nodes in the cloud. Although previous research reports variability of application runtime in the cloud [GKG+13], we expect that this effect is smaller in the private cloud and low scale simulations. We build a second testbed using eight VMs each having eight processing cores. In this configuration, more MPI communication is managed through shared memory communications and less network resources would be used. While in 2D distribution network communications are independent, to some extent, from the vertex and hyperedge distribution[10], we can provide an approximate analysis of the performance improvement.

Assume that we have 32 processor cores. In the first evaluation testbed, VMs arrange in 2×4 mesh each having 4 processor cores. In the second testbed, VMs arrange in a 2×2 mesh each have 8 processor cores. In the first, 33% and 28% of row and column communications are local, respectively. In the latter, the percentage

[10] As we explained previously, a vertex/hyperedge still needs to take part in collective row/column communications even if it has no pins on a specific processor

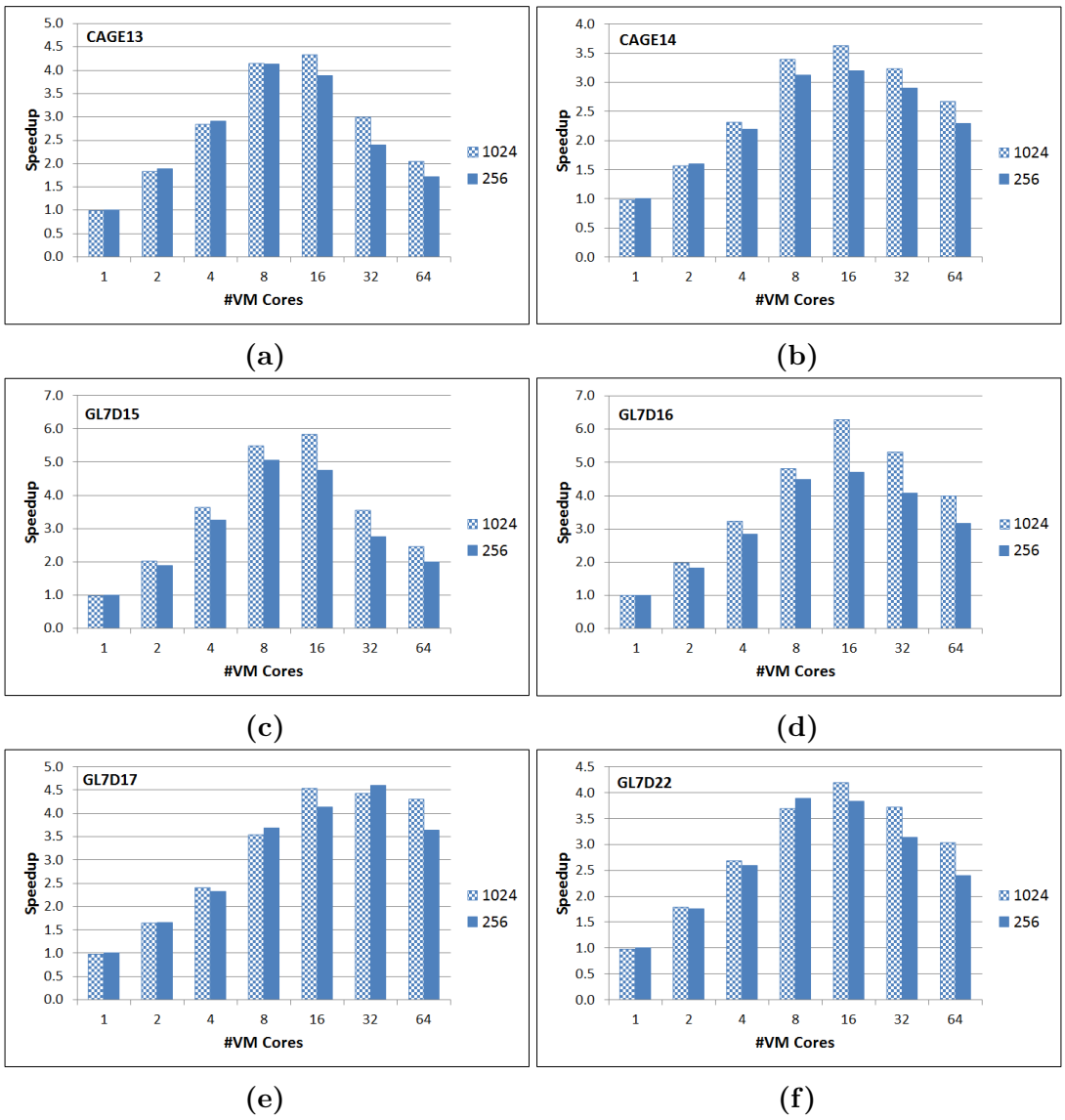


Figure 5.20: Comparing the speedup of the *PFEHG* algorithm in the cloud for *256-way* partitioning versus *1024-way* partitioning.

of local row communications does not change, but the percentage of local column communications increases to 42% because we have less network barrier on the way. It means that the column communications improves by 33.3%. Evaluating *PHG* proves this fact. The performance of the algorithms improves between 1% to 53% with the average improvement of 31.7% on all hypergraphs.

Conducting the same analysis on 1D would be difficult because the communication patters depends on distribution of the hypergraph on the processor set. To simplify the analysis, we assume that all 32 processors are arranged in one row and hyperedges have at least one pin on every processor. With this simplified assumption, every hyperedge and vertex will take part in collective communications. In this situation, the percentage of local messages changes from 9% to 21% when we change the number of VM cores from four to eight that is 57% improvement; this is the maximum improvement that can be achieved. Among the tested hypergraphs, the `CAGEXX` hypergraphs have more than 95% external hyperedges when distributed on 32 processors. We set $p_{\min} = 16$ for multiple bisection and other parameters are unchanged. The performance improvement for this group is shown to be 53% and 49% for `CAGE13` and `CAGE14` hypergraphs, respectively. The value is smaller on other hypergraphs with the average value of 18.6%. The evaluation shows that higher network usage for 2D distribution and employing higher-end processing nodes provide more benefit for 2D configuration than 1D distribution.

5.4.4 Discussions

Although *PFEHG* gives much better speedup (up to 32 cores) compared to the *PHG* algorithm (up to 8 cores), the scalability of the parallel hypergraph partitioning algorithms is still limited in the cloud. There are two aspects of the *PFEHG* algorithm that are promising: First, the scalability of the algorithm improves as the size of the hypergraph increases so hypergraph partitioning could replace graph partitioning algorithms and random load balancers in the cloud. In addition to the advantages of the hypergraph partitioning that are discussed earlier, the hypergraph partitioning provides better cost/performance ratio and better resource utilisation in the cloud for large social graphs or HPC applications. This can make cloud more suitable

for running large scale HPC applications which is previously showed to be non-advantageous as reported by Evangelinos and Hill [EH08]. Second, the algorithm's performance improves as the number of parts increases. If the number of parts is equal to the number of processors in the distributed system, the algorithm will give better performance and support in exascale systems and match the horizontal scalability of the cloud.

The biggest challenge of *PHG* is the 2D distribution. Although this can be effective in HPC clusters with high speed networks, it performs poorly in the cloud because of limited network resources.Communication patterns are independent of the structure of the hypergraph during the partitioning; for example, if a hyperedge does not have any pins on a processor block it still needs to participate in column communications. The distribution is not suitable for the cloud and the algorithm experiences increases in runtime as soon as it uses network resources.

According to our discussion in Chapter 3.4, applications that stress global communications give the worst performance in the cloud and those applications that communicate large messages (like BSP-like communication) give better performance than those that communicate more regularly with short messages [JRM+10]. Although the communication layer of *Zoltan* is done in BSP-like MPI communications, which suits the cloud, and 2D distribution breaks the processor into row and column processors, it still uses global network communications when the number of processors is large. For example for 32 processors, *PHG* arranges the processors in an 8×4 mesh. In this arrangement, the row communications have a network barrier but column communications are done locally. If the hypergraph has fewer hyperedges than vertices, then swapping the columns and rows gives better performance; therefore, better strategies are required for processor distribution and the distribution of hypergraph on the two-dimensional matrix

An advantage that *PFEHG* has over *PHG* is the multiple bisection strategy. First, it provides an easy-to-apply trade-off between performance and quality. Second, the poor scalability of hypergraph partitioning with increasing number of processors is not only related to the algorithm itself but also to the structure of the hypergraph. It breaks the problem into smaller subproblems and gives better data locality and less

network communication. The only overhead is the processor reconfiguration time and it give even better results by using superior heuristics to keep this overhead low. Third, the performance for *k-way* partitioning improves as k increases as the worst bottleneck is in the first few recursions of the algorithm. Passing this limit seems to be impossible for *PHG*, while the multiple bisection strategy helps *PFEHG* to overcome this bottleneck. Considering the characteristics of the cloud, the recursive bipartitioning algorithms get better performance than direct *k-way* partitioning algorithms as they successively break the problem into smaller subproblems and the network communication becomes more localised. In addition, the optimisations proposed for *PFEHG* are not applicable to direct *k-way* partitioning algorithms.

In addition, we argue that recursive bipartitioning algorithms could give better speedup in the cloud than the direct *k-way* partitioning solutions for two reasons. First, the problem is progressively break into smaller subproblems that runs independently in parallel by smaller sets of processors. Second, network communications are break into local messages as we proceed to the the partitioning. These optimisations are not possible for direct *k-way* algorithms.

The last discussion is that relying solely on the hypergraph partitioning cannot improve performance of distributed applications in the cloud. The structure of the application that employs the hypergraph partitioning is also important; for example, those using collective communications through large local messages gives better performance than those global and short messages. We refer to the work by Huber et al. [HBD+12] that investigates the scalability of the HYpergraph-based Distributed Response-time Analyser (HYDRA) [DHK03] in the cloud. The operation uses sparse matrix-vector multiplication as a core operation. Therefore it is a good candidate for hypergraph partitioning. The HYDRA is using hypergraph partitioning for distributing the sparse matrix on the processors. The evaluation in the HPC cluster shows improved scalability and performance, but they report very poor performance and scalability of HYDRA on the cloud. After analysing their work, we found that the reason could be the use of non-blocking MPI communications between the processors in order to decouple communication and computations. As this kind of communication is not cloud-friendly, it could be the reason for poor

performance. In contrast, the Pregel-like graph processing tools, which is based on BSP-like communication using super-steps, show major performance improvement in the cloud when the input data is partitioned using graph or hypergraph partitioners [MAB+10,WXSW14,CBL+14,HWL12].

Chapter 6

Conclusions

In this chapter, we provide a summary of the thesis and discuss the evaluations of previous chapters. The opportunities for the future work are also discussed at the end of the chapter.

6.1 Summary of Achievements

Chapter 1 introduced the partitioning and load balancing problem, the advantages of hypergraphs compared to graphs, the motivation of the thesis, and the summary of objectives and contributions.

In Chapter 2, hypergraph and the hypergraph partitioning problem were defined. In practical applications, the hypergraph partitioning problem requires two objectives: the cost objective, which measures the quality of the partitioning, and the balance constraint. Obtaining the optimum partitioning solution considering these objectives is NP-Hard and we need polynomial time heuristic algorithms for practical applications. The rough set clustering theory was also defined in this chapter. It was discussed that not all information in the rough set theory, which are used for data clustering, are important and one can remove redundancies to make better clustering decisions. Removing redundancies is known to be NP-Hard and it is one of the computational bottlenecks of the rough set clustering. The chapter finished by introducing the cloud and its architecture, the specifications, and services. It is discussed that some specifications of the cloud such as cost effectiveness, dynamic

resource allocation, and economies of scales, are the reasons that make the cloud useful for running scientific applications.

Chapter 3 was dedicated to the literature review. We categorised the hypergraph partitioning algorithms according to different aspects. The specifications of each category including their advantages and drawbacks were investigated. We used the discussions in this chapter to build the framework for our partitioning algorithms which are multi-level and recursive bipartitioning. The available hypergraph partitioning tools were also introduced. We found that there is no general framework for hypergraph partitioning and all the tools use different interfaces, various input formats, etc., and there is a need for a consensus. Furthermore, we have investigated the challenges for running scientific applications in the cloud, such as the virtualisation overheads, and we studied how the structure of the parallel application is important when run in the cloud such as the communication pattern, and data locality. The discussions in this chapter are used for designing our parallel hypergraph partitioning algorithm.

In Chapter 4, we proposed our serial multi-level hypergraph partitioning algorithm based on feature extraction and attribute reduction. The algorithms is known as *Feature Extraction Hypergraph Partitioner (FEHG)* algorithm. In the serial algorithm, the hypergraph was first transformed into an information system according to the rough set theory definition. Then the rough set clustering techniques were used for finding pair-matches of the vertices during the coarsening phase. Because the attribute reduction in the rough set clustering is a known NP-Hard problem, we overcame the problem by introducing the *Hyperedge Connectivity Graph (HCG)* that clusters the hyperedges using different similarity metrics and a similarity threshold. The HCG removed less important attributes and features from the information system representing the hypergraph. The *FEHG* algorithm provided a trade-off between global and local vertex matching decisions by categorising the vertices into core and non-core vertices.

We have evaluated *FEHG* against the state-of-the-art hypergraph partitioning algorithms including *hMetis*, *PaToH*, and *PHG* in the *Zoltan* tool. We built the test hypergraphs including different vertex and hyperedge weights (to model different

scenarios in real applications). The simulation results showed very good partitioning quality of our algorithm on the tested hypergraphs. Furthermore, *FEHG* showed incredible quality improvement for partitioning hypergraphs with irregular structure. One of the drawbacks of the local vertex matching decisions is that they perform completely differently under various problem circumstances and their behaviour can change based on the structure of the hypergraph under investigation. The worst case was observed for *PaToH* that generated very good and competitive partitioning quality compared to our algorithm when the hyperedge weights were assumed to be unit, while it gave much worse quality when the hyperedge weights were driven by the hyperedge sizes.

Evaluation of the runtime of the algorithms has shown that the *FEHG*, despite using global clustering decisions, runs slower than *PHG* and *PaToH*, but it runs faster than *hMetis*, the popular hypergraph partitioner. The runtime specification of *FEHG* is controlled by the runtime parameters that can be set by the user. The algorithm runs faster by fine tuning these parameters.

Chapter 5 proposed an extended version of the *FEHG* algorithm known as the *Parallel Feature Extraction Hypergraph Partitioner (PFEHG)*. The algorithm was designed for scalability. The algorithm applied a new one-dimensional hypergraph distribution among processors or the balanced pin distribution. The distribution overcame the high memory requirement of previous 1D distributions by avoiding hyperedge replication. It also reduced the communication overhead of collecting information about external hyperedges in each coarsening level by employing hash functions and fast intersection methods.

The parallel coarsening phase was composed of three phases. In the first phase, the core vertices were found and matched using parallel rough set clustering techniques. The rest of the vertices were matched using a randomised algorithm. First the algorithm searched for global matches. If a vertex could not find a global match, *PFEHG* would try to find a local match for it on the local processor. Furthermore, a new parallel synchronisation-based FM algorithm was proposed for the refinement phase. The algorithm was designed based on the observations, characteristics, and the evaluation of the serial FM algorithm. Our analysis showed that the previously

proposed parallel FM algorithm (which is based on parallel refinement algorithms for graph partitioning) does not perform well on hypergraphs and it has several limitations. By removing some restrictions of previous algorithms (such as one-way moves and local balance constraints), we showed that the new algorithm still generates good partitioning results with comparable runtime. This idea was, in itself, something that merits further research.

As *PFEHG* is a recursive bipartitioner, two different processor reconfigurations were provided in each recursion: bisection processor splitting and multiple bisection. These reconfigurations have increased data locality in the cloud while preserving the partitioning quality. The latter, multiple bisection, is our proposed strategy. It had two advantages: it gave an easy-to-apply method to make a trade-off between quality and scalability, and it improved the scalability of parallel partitioners on irregular hypergraphs (parallel partitioning algorithms do not scale well on these types of hypergraphs).

The algorithm was evaluated against *PHG*, the state-of-the-art parallel hypergraph partitioner in the *Zoltan* tool. The tested hypergraphs were chosen from real applications including very large hypergraphs. The algorithms were evaluated in the high speed HPC cluster using up to 1024 processing nodes. The algorithms are compared based on their partitioning qualities on different hypergraphs as well as their scalability. We have shown that *PFEHG* demonstrates the greatest advantage and generates better partitioning results on hypergraphs with irregular structure. On others, it is the case that *PHG* sometimes generates better partitioning qualities. The scalability of *PFEHG* was shown to be better on most of the evaluated hypergraphs.

We discussed that the *PFEHG* algorithm can be further improved and we have made suggestions for further work. For example, we have provided solutions in order to improve the runtime of *PFEHG* algorithm by using multi-vertex match instead of pair-match in the coarsening phase. We have also identified a problem with the hash-lookup of the hyperedges on the processors that prevents scalability for very large hypergraphs. We suggested using better data structures that allow faster lookup.

Due to the growing applications of hypergraph partitioning and the interest for

transferring HPC and distributed applications into the cloud, we have evaluated the performance of the parallel hypergraph partitioning algorithms in the cloud. The evaluations showed that the performance of parallel hypergraph partitioning algorithms suffers from the limited network resources of the cloud and the proposed algorithms give much worse scalability in the cloud than the HPC cluster. We showed that *PFEHG* can achieve much better scalability and on up to 32 processor cores compared to 8 core scalability achieved by *PHG*. The main reason was using customised one-to-one communication pattern in 1D dimensional distribution of the hyeprgraph in *PFEHG* and our proposed multiple bisection processor reconfiguration strategy. Despite the performance improvements that *PFEHG* provides, the scalability of parallel hypergraph partitioning is still limited in the cloud. In addition, it is investigated that the 2D hypergraph distribution strategy performs poorly in the cloud and some optimisations should be done to make it more cloud-friendly. In addition, the recursive bipartitioning algorithms get better performance in the cloud than direct k-way partitioning algorithms.

The algorithms were implemented as a new hypergraph partitioning package in the *Zoltan* tool. The details of the implementation are outlined in Appendix B of the thesis. The algorithms were implemented to use the same programming interface as *Zoltan* which makes it easy to compare them with other partitioning algorithms. The source code of the algorithm was made available online.

6.2 Future Work

The outline for future work is summarised as follows. Some of the ideas in this section regarding the parallel algorithm in Chapter 5 are discussed in the end of the evaluation sections in the chapter.

1. In the coarsest level of the hypergraph partitioning tool, we used a limited number of simple heuristics for the initial partitioning of the hypergraph. Extending the partitioning package to have interfaces to other serial hypergraph partitioning tools is planned.

2. Our algorithms are recursive bipartitioning algorithms. Implementing algorithms to support direct *k-way* partitioning is an ongoing work and it is under implementation.

3. The initial distribution of the hypergraph on the processor set is very important. A good distribution would save lots of network overhead in the later levels of the partitioning process. Currently, we have resolved the problem with a collection of hash functions. Giving a good distribution using hashes is hypergraph dependant and increasing the number of hash functions would not make any difference in general. We plan to find a better solution for this.

4. Similar to the previous case, the intersection of the hyperedges in the parallel coarsening phase was calculated using fast intersection methods based on hash functions. Selecting a good hash function that works well for all hypergraphs which also give better intersection precision is difficult. Using other hash-based algorithms such as Locality Sensitive Hashing (LSH) was not a good solution as it is a runtime bottleneck (even using a small number of signatures for every hyperedge would cause a major increase in the runtime). In addition, it did not make any improvements to the quality. Improving this would have a positive effect on the performance of the partitioning and it is planed as future work.

5. A 2D hypergraph distribution strategy, albeit having some drawbacks such as decreased data locality and overhead in communications that are both disadvantages in the cloud, is a promising solution for initial hypergraph distribution. An interesting work is to evaluate how the multiple bisection reconfiguration technique, which is being used in our algorithm, affects the performance of two-dimensional distribution in the cloud.

6. The evaluation of the algorithms are done in the private cloud that provides higher customisation and better performance than the public cloud. Evaluating the algorithm on public clouds such as Amazon EC2 is planned.

7. Currently, parallel hypergraph partitioning algorithms are implemented using MPI. Considering the structure of the recent cloud and HPC clusters, which

have lot of support for shared memory programming, an interesting work would be an implementation that is based on both MPI and shared memory programming model. This implementation would provide better memory usage and probably better performance as communication and computations can overlap during the coarsening phase. We believe that the main problem would be the refinement phase. The idea is worth further investigation.

8. In our evaluations in the cloud, we have copied the input hypergraph on the local file system of virtual machines. As we have evaluated the algorithms for the comparison purposes, it did not create a problem because the overhead stays the same for both algorithms. A real evaluation is to consider the file system overheads in the cloud and get a real estimation of running hypergraph partitioning in the cloud.

Appendix A

Benchmark Specification

The list of hypergraphs used for simulation purposes in the thesis is depicted in Table A.1. The data is collected from the University of Florida Sparse Matrix Collection [DH11]. It is a large database of sparse matrices from real applications. It provides a robust basis for experimental evaluations, while the simulation results proposed by using random generated matrices are not always reliable as their structure might be different from the structure of real applications. The database includes sparse matrices from a wide range of applications such as structural engineering, optimization, circuit simulation, computational fluid dynamics, network graphs, social sciences, model reduction, electromagnetics, semiconductor devices, robotics, etc. The matrices are provided in there different formats: *Matlab*, Rutherford/Boeing, and Matrix Market (MM) formats. In our simulations we have used the Matrix Market format. Each matrix includes a main file describing the matrix and it has ***.mtx** extension. Some of the matrices have xyz coordinates file, which is not being used in our work as the target of the thesis is not geometric hypergraph partitioning.

Each sparse matrix from the database is treated as the hypergraph incident matrix with the vertices and hyperedges as rows and columns of the matrix, respectively. This is similar to the column-net model proposed in [ÇA99]. The hypergraphs have different specifications and they are chosen from variable applications with different sizes, symmetrical structure and number of strongly connected components. The description of the hypergraphs is as follow.

- `AS-22JULY06` contains a symmetrized snapshot of the structure of the Internet

at the level of autonomous systems, reconstructed from BGP tables posted at `archive.routeviews.org`.

- `CELEGANSNEURAL` describes a weighted, directed network representing the neural network of C. Elegans. The data were taken from the web site of Prof. Duncan Watts at Columbia University, `http//cdg.columbia.edu/cdg/datasets`.
- `NETSCIENCE` contains a co-authorship network of scientists working on network theory and experiments.
- `PGPGIANTCOMPO` is a graph of the largest component of the network of users of the Pretty-Good-Privacy algorithm for secure information interchange.
- `GUPTA1` is a graph from optimization problem and represents a linear programming matrix $A * A^T$.
- `MARK3JAC120` is Jacobian from MULTIMOD Mark3 from economical problems.
- `NOTREDAME-actors` is a bipartite co-stardom network with nodes of two types actors and movies such that an actor and a movie are connected by an edge if the actor was in the movie. `NOTREDAME` is a similar bipartite graph shows the web page network of `nd.edu`. While the `NOTREDAME` hypergraph has smaller size than the `NOTREDAME-actors` hypergraph, the first is used for evaluating serial algorithms and the latter is used in parallel algorithm evaluations.
- `PATENTS-MAIN` is based on the The NBER U.S. Patent Citations Data File, version 2001. These data comprise detailed information on almost 3 million U.S. patents granted between January 1963 and December 1999, all citations made to these patents between 1975 and 1999, and a reasonably broad match of patents to Compustat (the data set of all firms traded in the U.S. stock market).
- `STD1-JAC3` comes from computer aided chemical simulation and shows the graph of a chemical process simulation.
- `DELAUNAY-N16` is from computational geometry and shows the triangulations of random points in the plane.

- `AMAZON0601` is a network created by crawling the Amazon website. It shows the frequency by which items and features from Amazon are bought together by customers. If an item (row number) is frequently co-purchased with a product (column number), the corresponding item in the matrix contains a non-zero value.
- `BCSSTK32` is from structural engineering and shows the stiffness matrix for automobile chassis.
- `CAGE`xx is a model for DNA electrophoresis in which a matrix element a_{ij} shows the probability that a polymer of length xx in state i moves to state j.
- `CH8-8-b5` is a linear algebra problem and it is for simplicial complexes from Homology from Volkmar Welker.
- `CNR-2000` is a very small matrix generated by crawling the Italian CNR domain which aims to gather large data sets to study the structure of the web domain.
- `COND-MAT-2005` is the collaboration network of scientists posting preprints on the condensed matter archive at `www.arxiv.org`.
- `LANDMARK` is a matrix for least square problems.
- `RAIL4284` is a set covering problem on the Italian railroad network.
- `GL7D`xx are differentials of the Voronoi complex of perfect forms of rank 7 mod `GL-7(Z)` equivalences, (related to the cohomology of `GL-7(Z)` and the K-theory of Z).

The statistical specifications of the hypergraphs are depicted in Table A.2. Hypergraphs have a skew in vertex degree (like graphs) and a skew in edge cardinality (unlike graphs). The specification includes the number of isolated vertices, the mean vertex degree and hyperedge size and their standard deviation. The number of isolated vertices are the number of vertices that are not incident on any hyperedge. The partitioning of these vertices is easy as they do not have any effect on the partitioning cut. These vertices can be used to alleviate the balancing constraint and generate better partitioning qualities.

Table A.1: The list of hypergraphs used for simulation purposes in the thesis.

Hypergraph	Description	Rows	Columns	Non-Zeros	Structure[1]	NSC[2]
CNR-2000	Small web crawl of Italian CNR domain	325,557	325,557	3,216,152	USYM	100,977
AS-22JULY06	Internet routers	22,963	22,963	96,872	SYM	1
CELEGANSNEURAL	Neural Network of Nematode C. Elegans	297	297	2,345	USYM	57
NETSCIENCE	Co-authorship of scientists in Network Theory	1,589	1,589	5,484	SYM	396
PGPGIANTCOMPO	Largest connected component in graph of PGP users	10,680	10,680	48,632	SYM	1
GUPTA1	Linear Programming matrix ($A \times A^T$)	31,802	31,802	2,164,210	SYM	1
MARK3JAC120	Jacobian from MULTIMOD Mark3	54,929	54,929	322,483	USYM	1,921
NOTREDAME	Barabasi's web page network of nd.edu	325,729	325,729	929,849	USYM	231,666
PATENTS-MAIN	Pajek network: mainNBER US Patent Citations	240,547	240,547	560,943	USYM	240,547
STD1-JAC3	Chemical process simulation	21,982	21,982	1,455,374	USYM	1
COND-MAT-2005	Collaboration network, www.arxiv.org	40,421	40,421	351,382	SYM	1,798
DELAUNAY-N16	10th DIMACS Implementation Challenge	65,536	65,536	393,150	SYM	1
AMAZON0601	Web Indexing	403,394	403,394	3,387,388	USYM	1,588
BCSSTK32	Structural Problems	44,609	44,609	2,014,701	SYM	1
CAGE13	DNA Electrophoresis	445,315	445,315	7,479,343	USYM	1
CAGE14	DNA Electrophoresis	1,505,785	1,505,785	27,130,349	USYM	1
CH8-8-b5	Combinatorial Problem	564,480	376,320	3,386,880	USYM	1
LANDMARK	Least Squares Problem	71,952	2,704	1,146,848	USYM	32
NOTREDAME-actors	Social Networks	392,400	127,823	1,470,404	USYM	11,902
RAIL4284	Linear Programming	4,284	1,096,894	11,284,032	USYM	5
GL7d15	Combinatorial Problem	460,261	171,375	6,080,381	USYM	6
GL7d16	Combinatorial Problem	955,128	460,261	14,488,881	USYM	4
GL7d17	Combinatorial Problem	1,548,650	955,128	25,978,098	USYM	4
GL7d22	Combinatorial Problem	349,443	822,922	8,251,000	USYM	17

[1] NSC stands for the number of strongly connected components.
[2] SYM stands for symmetric and USYM stands for unsymmetric.

Table A.2: The statistical specification of the hypergraphs depicted in Table A.1.

Hypergraph	$\|V\|$	ISO[1]	vDegree[2]	vDegree STD[3]	eSize[4]	eSize STD[5]
CNR-2000	325,557	82,615	9.46	18.47	16.36	287.24
AS-22JULY06	22,963	451	2.03	2.16	21.99	104.19
CELEGANSNEURAL	297	23	7.37	6.96	8.86	7.22
NETSCIENCE	1,589	556	1.48	1.88	4.13	3.42
PGPGIANTCOMPO	10,680	4,055	1.94	4.61	5.49	7.03
GUPTA1	31,802	12	31.06	13.23	48.63	403.36
MARK3JAC120	54,929	0	6.23	4.35	6.23	6.61
NOTREDAME	325,729	193,262	2.43	5.03	8.36	65.88
PATENTS-MAIN	240,547	71,743	1.97	2.77	4.98	5.15
STD1-JAC3	21,982	2	66.18	169.32	69.80	132.33
COND-MAT-2005	40,421	3,731	4.16	4.69	9.04	13.29
DELAUNAY-N16	65,536	8,760	2.82	1.82	3.80	1.64
AMAZON0601	403,394	1,002	8.26	2.81	9.56	16.13
BCSSTK32	44,609	0	23.08	10.10	23.15	10.39
CAGE13	445,315	0	16.80	5.13	16.80	5.13
CAGE14	1,505,785	0	18.02	5.37	18.02	5.37
CH8-8-b5	564,480	0	6	0	9	0
LANDMARK	71,952	0	16	0.01	431.17	140.63
NOTREDAME-actors	392,400	10,181	3.72	10.28	12.33	11.82
RAIL4284	4,284	4	2632.95	4209.25	10.33	1.79
GL7d15	460,261	2	13.21	2.37	35.48	14.24
GL7d16	955,128	1	15.17	2.11	31.48	11.37
GL7d17	1,548,650	1	16.77	1.98	27.20	8.87
GL7d22	349,443	0	23.61	9.01	10.03	2.22

[1] ISO shows the number of isolated vertices.
[2] vDegree is the mean vertex degree in the hypergraph.
[3] vDegree STD is the standard deviation of the vertex degrees.
[4] eSize is the mean hyperedge size in the hypergraph.
[5] eSize STD is the standard deviation of the hyperedge sizes.

Appendix B

Programming Interface

B.1 Introduction

Our hypergraph partitioning algorithms have been implemented as a new library package inside the *Zoltan* [San14b] tool from the Sandia National Labs. The library is implemented in ANSI C with interfaces for C++ and Fortran (They are only a wrapper around the C code). It is a toolkit developed for scientific computing and includes the following packages.

1. **Dynamic load balancing and parallel partitioning** that includes geometric, hypergraph and graph partitioning methods.
2. **Data migration tools** for moving data from old partitions to new partitioning when the partitioning is done.
3. **Parallel graph colouring** for 1-distance and 2-distance parallel graph colouring.
4. **Distributed data directories** that is scalable and distributed directory management.
5. **Unstructured communication package** for unstructured BSP-like MPI communications between processors.
6. **Dynamic memory management tool** for dynamic memory allocations and memory debugging.

The complete reference about *Zoltan* programming interface and how to build the library can be found from the developer's web page at `http://www.cs.sandia.gov/zoltan/ug_html/ug_intro.html`. The target of this appendix is not to propose *Zoltan* in detail. The main focus of this appendix is to explain the implementation details of our rough set clustering based hypergraph partitioning algorithm and the runtime parameters of our algorithms. Consequently, when proposing *Zoltan* features and functions, we depict only the function list that are necessary for our purpose. Our algorithm is implemented as part of load balancing and parallel partitioning package in *Zoltan*. The MPI communication between the processors in the parallel algorithm is done through the unstructured communication package of *Zoltan*. The other packages of *Zoltan* are not used in our implementation.

Zoltan is designed to run on parallel computers and clusters of workstations. The most common builds and installations of Zoltan needs the following.

- ANSI C or C++ compiler.
- MPI library for message passing (version 1.1 or higher), such as MPICH, OpenMPI or LAM.
- A Unix-like operating system (e.g., Linux or Mac OS X) and gmake (GNU Make) are recommended to build the library.
- A Fortran90 compatible compiler is required if you wish to use Zoltan with Fortran applications.

In the following, we provide a brief introduction on how to use *Zoltan* for hypergraph partitioning and then we describe algorithm specific parameters. Functions are proposed in C or C++ syntax.

B.2 Zoltan at a Glance

A nice feature of *Zoltan* is that it does not impose neither any restriction on the format of the data representation nor requires any specific data structure when the input hypergraph is provided to the *Zoltan*. The user provides *callback functions*

for the library. *Zoltan* queries the application for the required data; therefore, the application should provide these query functions for the *Zoltan*. We refer to these user provided functions as **query** functions.

Query functions return information about only on-processor data and they should NOT contain processor communications as each processor can call a specific query function several times; therefore, having communication inside the query functions may cause *Zoltan* to halt because not all processors may contribute in the requested MPI communication. There are two types of query functions in *Zoltan*, **general Zoltan query functions** and **migration query functions**. The latter is not explained here as we are not using the migration functions.

The query functions have a function type, describing their purpose. Function can be registered by calling either Zoltan_Set_Fn or Zoltan_Set_<zoltan_fn_type>_Fn. The first function needs another argument, represented as fn_type, that shows the function type, while in the latter, the function type is implicit in fn_ptr parameter. Furthermore, a query function, when called by a processor, can return information about a list of objects on the local processor (referred as list-based functions) or an individual object. Users can provide either version of the query function and they do not need to provide both. Zoltan calls the list-based functions with the IDs of all objects needed; this approach often provides faster performance as it eliminates the overhead of multiple function calls. List-based functions have the word **MULTI** in their function-type name. If, instead, the application provides iterator functions, Zoltan calls the iterator function once for each object whose data is needed. This approach, while slower, allows Zoltan to use less memory for some data.

Many of the functions have both global and local object identifiers (IDs) in their argument lists. The global ID is unique among all processors and used for global identification of the objects. The local IDs are for the convenience of the application and they are not used by the *Zoltan* library[1]. All of the functions have, as their first argument, a pointer to data that is passed to Zoltan through Zoltan_Set_Fn or

[1] *Zoltan* assigns its own local ID to the objects while the program is run. The local ID is solely for the application use. The user may provide both Global and local ID for each objects. The local ID may save some time while reading the input data by *Zoltan* as the program does not need to refer to a global list for accessing some information.

Zoltan_Set_<zoltan_fn_type>_Fn. This data is not used by Zoltan. A different set of data can be supplied for each registered function. For example, if the local ID is an index into an array of data structures, then the data pointer might point to the head of the data structure array.

Every function returns an error code in *Zoltan*. The error handling in *Zoltan* is local. When a processor returns an error, other processors might not be aware of the error returned by the processor. Therefore, debugging of the parallel code is not convenient and sometimes complicated. The error codes of *Zoltan* are described as follows:

ZOLTAN_OK

function returned without warnings or errors.

ZOLTAN_WARN

function returned with warnings. The application will probably be able to continue to run.

ZOLTAN_FATAL

a fatal error occurred within the Zoltan library.

ZOLTAN_MEMERR

an error occurred while allocating memory. When this error occurs, the library frees any allocated memory and returns control to the application. If the application then wants to try to use another, less memory-intensive algorithm, it can do so.

The behaviour of *Zoltan* is controlled by several parameters and debugging-output levels. These parameters can be set by calls to Zoltan_Set_Param. Reasonable default values for all parameters are specified by Zoltan. Parameters are categorised as general ad algorithm specific parameters. General parameters are used for whole algorithms in *Zoltan*. Setting an algorithm parameter for other algorithms returns an error. For example, we have a set of specific parameters for our algorithm that that controls the runtime behaviour.

B.2.1 General Functions

Functions for initialising *Zoltan* are described in this section. Only necessary function are introduced. The details about each function can be found on the developers web site at `http://www.cs.sandia.gov/zoltan/ug_html/ug_interface_init.html`. Functions are proposed in C syntax. The list of functions is described as follows:

```
int Zoltan_Initialize(int argc, char **argv, float *ver)
struct Zoltan_Struct *Zoltan_Create(MPI_Comm communicator);
struct Zoltan_Struct *Zoltan_Copy(Zoltan_Struct *from);
int Zoltan_Set_Param(struct Zoltan_Struct *zz,char *param_name,char
    *new_val);
int Zoltan_Set_Param_Vec(struct Zoltan_Struct *zz, char *param_name
   , char *new_val, int index);
int Zoltan_Set_Fn(struct Zoltan_Struct *zz, ZOLTAN_FN_TYPE fn_type,
    void (*fn_ptr)(), void *data);
int Zoltan_Set_<zoltan_fn_type>_Fn(struct Zoltan_Struct *zz, <
   zoltan_fn_type> (*fn_ptr)(), void *data);
void Zoltan_Destroy(struct Zoltan_Struct **zz);
```

The details of functions are described as follows:

Zoltan_Initialize

initializes MPI for Zoltan. If the application uses MPI, this function should be called after calling MPI_Init. If the application does not use MPI, this function calls MPI_Init for use by Zoltan. This function is called with the *argc* and *argv* command-line arguments from the main program, which are used if Zoltan_Initialize calls MPI_Init.

Zoltan_Create

allocates memory for storage of information to be used by Zoltan and sets the default values for the information. The pointer returned by this function is passed to many subsequent functions. The pointer returned by this function is referred as zz for the rest of the functions.

Zoltan_Copy

creates a new Zoltan_Struct and copies the state of the existing Zoltan_Struct, which it has been passed, to the new structure. It returns the new Zoltan_Struct.

Zoltan_Set_Param

is used to alter the value of one of the parameters used by Zoltan. All Zoltan parameters have reasonable default values, but this routine allows a user to provide alternative values if desired.

Zoltan_Set_Param_Vec

is used to alter the value of a vector parameter in Zoltan. A vector parameter is a parameter that has one name but contains multiple values. These values are referenced by their indices, usually starting at 0. Each entry (component) may have a different value. This routine sets a single entry (component) of a vector parameter.

Zoltan_Set_Fn

registers an application-supplied query function in the Zoltan structure. All types of query functions can be registered through calls to Zoltan_Set_Fn. To register functions while maintaining strict type-checking of the fn_ptr argument, use Zoltan_Set_<zoltan_fn_type>_Fn.

Zoltan_Set_<zoltan_fn_type>_Fn

where <zoltan_fn_type> is one of the query function types, register specific types of application-supplied query functions in the Zoltan structure. One interface function exists for each type of query function.

Zoltan_Destroy

frees the memory associated with a Zoltan structure and sets the structure to NULL in C or nullifies the structure in Fortran. There is no explicit Destroy method in the C++ interface. The Zoltan object is destroyed when the destructor executes. As a side effect, Zoltan_Destroy (and the C++ Zoltan destructor) frees the MPI communicator that had been allocated for the

structure. So it is important that the application does not call MPI_Finalize before it calls Zoltan_Destroy or before the destructor executes.

B.2.2 Query Functions

In this section, we provide the list of query functions that should be defined and provided by the user to *Zoltan*. As mentioned earlier, these functions tell *Zoltan* how to read the input hypergraph. Functions are proposed in C and C++ syntax, which is the same in *Zoltan*. Function specifications are taken from `http://www.cs.sandia.gov/zoltan/ug_html/ug_query_lb.html`.

A hypergraph is supplied to *Zoltan* by either **compressed edge** or **compressed vertex**[2] formats. The two compressed formats are analogous to Compressed Row Storage (CRS) and Compressed Column Storage (CCS) for matrices. The input format is provided by the following two parameters:

ZOLTAN_COMPRESSED_EDGE

a list of global hyperedge IDs is provided then a list containing the hypergraph pins is provided. A pin is identified by the global ID of the hyperedge and the vertex that construct the pin.

ZOLTAN_COMPRESSED_VERTEX

a list of global vertex IDs is provided then a list containing the hypergraph pins is provided. A pin is identified by the global ID of the vertex and the hyperedge that construct the pin.

In both formats, there is a list that shows where pins of a specific hyperedge/vertex starts. This list is referred as *hyperedge index* and *vertex index* lists in compressed edge and compressed vertex formats, respectively. For the hypergraph given in Fig. 2.1 the compressed edge and vertex formats would be as follow.

- Compressed Edge Format:

  1. $EDGE\_lst = \{e_1, e_2, e_3\}$

[2]This format is not yet supported by our algorithm. It is planned as the future work.

2. $PIN\_lst = \{v_1, v_2, v_3, v_5, v_2, v_3, v_5\}$
3. $INDEX = \{0, 4, 6\}$

- Compressed Vertex Format:

1. $VERTEX\_lst = \{v_1, v_2, v_3, v_4, v_5\}$
2. $PIN\_lst = \{e_1, e_1, e_2, e_1, e_2, e_1, e_3\}$
3. $INDEX = \{0, 1, 3, 5, 6\}$

The format should be provided to *Zoltan* by the ZOLTAN_HG_CS_FN_TYPE query function.

```
typedef void ZOLTAN_HG_SIZE_CS_FN (void *data, int *num_lists, int
    *num_pins, int *format, int *ierr);
```

The purpose of this query function is to tell Zoltan in which format the application will supply the hypergraph, how many vertices and hyperedges there will be, and how many pins. The actual hypergraph is supplied with a query function of the type ZOLTAN_HG_CS_FN_TYPE.

data

Pointer to user-defined data. num_lists) Upon return, the number of vertices (if using compressed vertex storage) or hyperedges (if using compressed hyperedge storage) that will be supplied to Zoltan by the application process.

num_pins

Upon return, the number of pins (connections between vertices and hyperedges) that will be supplied to Zoltan by the application process.

format

Upon return, the format in which the application process will provide the hypergraph to Zoltan. The options are ZOLTAN_COMPRESSED_EDGE and ZOLTAN_COMPRESSED_VERTEX.

ierr

Error code to be set by function.

In the following we describe the other query functions of *Zoltan*.

```
typedef void ZOLTAN_HG_CS_FN (void *data, int num_gid_entries, int
    num_vtx_edge, int num_pins, int format, ZOLTAN_ID_PTR
    vtxedge_GID, int *vtxedge_ptr, ZOLTAN_ID_PTR pin_GID, int *ierr)
    ;
```

the function returns a hypergraph in a compressed storage (CS) format. The size and format of the data to be returned must have been supplied to Zoltan using a ZOLTAN_HG_SIZE_CS_FN_TYPE function. When a hypergraph is distributed across multiple processes, Zoltan expects that all processes share a consistent global numbering scheme for hyperedges and vertices. Also, no two processes should return the same pin (matrix non-zero) in this query function. (Pin ownership is unique.)

data

Pointer to user-defined data.

num_gid_entries

The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.

num_vtx_edge

The number of global IDs that is expected to appear on return in vtxedge_GID. This may correspond to either vertices or (hyper-)edges.

num_pins

The number of pins that is expected to appear on return in pin_GID.

format

If format is ZOLTAN_COMPRESSED_EDGE, Zoltan expects that hyperedge global IDs will be returned in vtxedge_GID, and that vertex global IDs will be returned in pin_GIDs. If it is ZOLTAN_COMPRESSED_VERTEX, then vertex global IDs are expected to be returned in vtxedge_GID and hyperedge global IDs are expected to be returned in pin_GIDs.

vtxedge_GID

Upon return, a list of num_vtx_edge global IDs.

vtxedge_ptr

Upon return, this array contains num_vtx_edge integers such that the number of pins specified for hyperedge j (if format is ZOLTAN_COMPRESSED_EDGE) or vertex j (if format is ZOLTAN_COMPRESSED_VERTEX) is vtxedge_ptr[j+1] - vtxedge_ptr[j]. If format is ZOLTAN_COMPRESSED_EDGE, vtxedge_ptr[j] * num_gid_entries is the index into the array pin_GID where edge j's pins (vertices belonging to edge j) begin; if format is ZOLTAN_COMPRESSED_VERTEX, vtxedge_ptr[j] * num_gid_entries is the index into the array pin_GID where vertex j's pins (edges to which vertex j belongs) begin. Array indices begin at zero.

pin_GID

Upon return, a list of num_pins global IDs. This is the list of the pins contained in the hyperedges or vertices listed in vtxedge_GID.

ierr

Error code to be set by function.

```
typedef void ZOLTAN_HG_SIZE_EDGE_WTS_FN (void *data, int *num_edges
    , int *ierr);
```

the function returns the number of hyperedges for which a process will supply edge weights. The number of weights per hyperedge was supplied by the application with the EDGE_WEIGHT_DIM parameter. The actual edge weights will be supplied with a ZOLTAN_HG_EDGE_WTS_FN_TYPE function.

data

Pointer to user-defined data.

num_edges

Upon return, the number of hyperedges for which edge weights will be supplied.

ierr

Error code to be set by function.

```
typedef void ZOLTAN_HG_EDGE_WTS_FN (void *data, int num_gid_entries
    , int num_lid_entries, int num_edges, int edge_weight_dim,
    ZOLTAN_ID_PTR edge_GID, ZOLTAN_ID_PTR edge_LID, float  *
    edge_weight, int *ierr);
```

the function returns edges weights for a set of hypergraph edges. The number of weights supplied for each hyperedge should equal the value of the EDGE_WEIGHT_DIM parameter.

data

Pointer to user-defined data.

num_gid_entries

The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.

num_lid_entries

The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES. (It should be zero if local ids are not used.)

num_edges

The number of hyperedges for which edge weights should be supplied in the edge_weight array.

edge_weight_dim

The number of weights which should be supplied for each hyperedge. This is also the value of the EDGE_WEIGHT_DIM parameter.

edge_GID

Upon return, this array should contain the global IDs of the num_edges hyperedges for which the application is supplying edge weights.

edge_LID

Upon return, this array can optionally contain the local IDs of the num_edges hyperedges for which the application is supplying edge weights.

edge_weight

Upon return, this array should contain the weights for each edge listed in the edge_GID. If edge_weight_dim is greater than one, all weights for one hyperedge are listed before the weights for the next hyperedge are listed.

ierr

Error code to be set by function.

```
typedef int ZOLTAN_NUM_OBJ_FN (void *data, int *ierr);
```

the function returns the number of objects that are currently assigned to the processor.

data

Pointer to user-defined data.

ierr

Error code to be set by function.

```
typedef void ZOLTAN_OBJ_LIST_FN (void *data, int num_gid_entries,
   int num_lid_entries, ZOLTAN_ID_PTR global_ids, ZOLTAN_ID_PTR
   local_ids, int wgt_dim, float *obj_wgts, int *ierr);
```

the function fills two (three if weights are used) arrays with information about the objects currently assigned to the processor. Both arrays are allocated (and subsequently freed) by Zoltan; their size is determined by a call to a ZOLTAN_NUM_OBJ_FN query function to get the array size.

data

Pointer to user-defined data.

num_gid_entries

The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.

num_lid_entries

The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES. (It should be zero if local ids are not used.)

global_ids

Upon return, an array of unique global IDs for all objects assigned to the processor.

local_ids

Upon return, an array of local IDs, the meaning of which can be determined by the application, for all objects assigned to the processor. (Optional.)

wgt_dim

The number of weights associated with an object (typically 1), or 0 if weights are not requested. This value is set through the parameter OBJ_WEIGHT_DIM.

obj_wgts

Upon return, an array of object weights. Weights for object i are stored in obj_wgts[(i-1) * wgt_dim:i * wgt_dim-1]. If wgt_dim = 0, the return value of obj_wgts is undefined and may be NULL.

ierr

Error code to be set by function.

```
typedef void ZOLTAN_PART_MULTI_FN (void *data, int num_gid_entries,
    int num_lid_entries, int num_obj, ZOLTAN_ID_PTR global_ids,
   ZOLTAN_ID_PTR local_ids, int *parts, int *ierr);
```

the function returns a list of parts to which given objects are currently assigned. If a ZOLTAN_PART_MULTI_FN or ZOLTAN_PART_FN is not registered, Zoltan assumes

the part numbers are the processor number of the owning processor. Valid part numbers are non-negative integers. The information for one object is provided with ZOLTAN_PART_FN.

data

Pointer to user-defined data.

num_gid_entries

The number of array entries used to describe a single global ID. This value is the maximum value over all processors of the parameter NUM_GID_ENTRIES.

num_lid_entries

The number of array entries used to describe a single local ID. This value is the maximum value over all processors of the parameter NUM_LID_ENTRIES. (It should be zero if local ids are not used.)

global_ids

The global IDs of the objects for which the part numbers should be returned.

local_ids

The local IDs of the objects for which the part numbers should be returned. (Optional.) (Optional.)

parts

Upon return, an array of part numbers corresponding to the global and local IDs.

ierr

Error code to be set by function.

Finally, the main partitioning and load-balancing functions of *Zoltan* are described as follow. Refer to the developers website for more information and explanation about the function arguments at [San14b].

```
int Zoltan_LB_Partition (
      struct Zoltan_Struct *zz,
      int *changes,
      int *num_gid_entries,
      int *num_lid_entries,
      int *num_import,
      ZOLTAN_ID_PTR *import_global_ids,
      ZOLTAN_ID_PTR *import_local_ids,
      int **import_procs,
      int **import_to_part,
      int *num_export,
      ZOLTAN_ID_PTR *export_global_ids,
      ZOLTAN_ID_PTR *export_local_ids,
      int **export_procs,
      int **export_to_part);
```

The list of the arguments are.

zz

Pointer to the Zoltan structure.

changes

Set to 1 or TRUE if the decomposition was changed by the load-balancing method; 0 or FALSE otherwise.

num_gid_entries

Upon return, the number of array entries used to describe a single global ID.

num_lid_entries

Upon return, the number of array entries used to describe a single local ID.

num_import

Upon return, the number of objects that are newly assigned to this processor or to parts on this processor (i.e., the number of objects being imported from different parts to parts on this processor). If the value returned is -1, no import information has been returned and all import arrays below are NULL. The RETURN_LISTS parameter determines whether import lists are returned.

import_global_ids

Upon return, an array of num_import global IDs of objects to be imported to parts on this processor.

import_local_ids

Upon return, an array of num_import local IDs of objects to be imported to parts on this processor.

import_procs

Upon return, an array of size num_import listing the processor IDs of the processors that owned the imported objects in the previous decomposition (i.e., the source processors).

import_to_part

Upon return, an array of size **num_import** listing the parts to which the imported objects are being imported.

num_export

Upon return, this value of this count depends on the value of the RETURN_LISTS parameter. Refer to the developer's website for more information about this parameter.

export_global_ids

Upon return, an array of **num_export** global IDs of objects to be exported from parts on this processor.

export_local_ids

Upon return, an array of num_export local IDs associated with the global IDs returned in export_global_ids.

export_procs

Upon return, an array of size num_export listing the processor ID of the processor to which each object is now assigned (i.e., the destination processor).

export_to_part

Upon return, an array of size num_export listing the parts to which the objects

are assigned under the new partition.

Finally, user can set the size of the parts in *Zoltan*. These is done through the Zoltan_LB_Set_Part_Sizes function. By default, *Zoltan* assumes that all parts should be of equal size. Zoltan_LB_Free_Part function frees the memory allocated by *Zoltan* to return the results of Zoltan_LB_Partition. There are more functions that can be called for load balancing and partitioning purposes and they can be seen in `http://www.cs.sandia.gov/zoltan/ug_html/ug_interface_lb.html`. Here we described only the functions that are necessary for our purpose and how to use our hypergraph partitioner in *Zoltan*.

B.2.3 General Parameters

As mentioned previously, the behaviour of *Zoltan* is controlled with runtime parameters. The parameters are set b calling Zoltan_Set_Param. The parameters that are described here are general parameters. All of them have default values. The list of general parameters are as follow.

IMBALANCE_TOL

The partitioning imbalance tolerance.

NUM_GLOBAL_PARTS

The number of partitioning parts.

NUM_GID_ENTRIES

The number of unsigned integers that should be used to represent a global identifier (ID). Values greater than zero are accepted. The **default** is 1.

NUM_LID_ENTRIES

The number of unsigned integers that should be used to represent a local identifier (ID). Values greater than or equal to zero are accepted. The **default** is 1.

DEBUG_LEVEL

An integer indicating how much debugging information is printed by Zoltan.

Higher values of DEBUG_LEVEL produce more output and potentially slow down *Zoltan*'s computations. The least output is produced when DEBUG_LEVEL = 0. DEBUG_LEVEL primarily controls *Zoltan*'s behaviour; most algorithms have their own parameters to control their output level. Values used within *Zoltan* are listed below. The **default** is 1.

DEBUG_MEMORY

Integer indicating the amount of low-level debugging information about memory-allocation should be kept by *Zoltan*'s Memory Management utilities. Valid values are 0, 1, 2, and 3. The **default** is 1.

OBJ_WEIGHT_DIM

The number of weights (to be supplied by the user in a query function) associated with an object. If this parameter is zero, all objects have equal weight. Some algorithms may not support multiple (multidimensional) weights. The **default** is 0.

EDGE_WEIGHT_DIM

The number of weights associated with an edge. If this parameter is zero, all edges have equal weight. Many algorithms do not support multiple (multidimensional) weights. The **default** is 0.

TIMER

The timer with which you wish to measure time. Valid choices are wall (based on MPI_Wtime), cpu (based on the ANSI C library function clock), and user. The resolution may be poor, as low as 1/60th of a second, depending upon your platform. The **default** is *wall*.

Furthermore, some high debugging levels use processor synchronization to force processors to write one-at-a-time. Therefore they need to be the same on all processors.

B.3 FEHG Algorithm Parameters

FEHG is a parallel multi-level hypergraph partitioning algorithms that achieves *k-way* partitioning through recursive bipartitioning. It has three main phases: coarsening, initial partitioning, and uncoarsening (refinement). In this section, we describe all algorithm specific parameters for each phase separately.

B.3.1 Partitioning Parameters

In order to use *FEHG* hypergraph partitioning algorithm, LB_METHOD and HYPERGRAPH_PACKAGE parameters should be both set to FEHG. In addition, the following parameters are defined for *FEHG*.

LB_APPROACH

defines the load balancing approach. It can be set to either PARTITION or REFINE. The first performs the hypergraph partitioning from the scratch without taking into account the current vertex distribution. REFINE, refines the current vertex partitioning without going through any multi-level partitioning. *Dynamic repartitioning* is not supported yet.

FEHG_MULTILEVEL

if the parameter is set, the algorithm does not go through any multi-level partitioning and directly performs the refinement phase. The multi-level approach generates higher quality partitioning but requires more execution time and memory.

FEHG_OUTPUT_LEVEL

sets the level of verbosity in the output. Level zero generates no output. More output about the runtime status of the algorithm can be seen by setting the output level to higher values.

FINAL_OUTPUT

if set, the final partitioning result are generated and returned to the user.

RETURN_LISTS

The lists returned by calls to Zoltan_LB_Partition. The values are:

IMPORT: returns only information about objects to be imported to a processor .

EXPORT returns only information about objects to be exported from a processor.

ALL returns both import and export information.

PARTS returns the new process and part assignment of every local object, including those not being exported.

NONE returns neither import nor export information.

CHECK_HYPERGRAPH

if set, the input format of the hypergraph is checked for the correctness.

FEHG_USE_TIMERS

The timing level of *FEHG* algorithm. The higher the value, the more detailed timing would be generated.

FEHG_CUT_OBJECTIVE

if set to **connectivity**, the *connectivity*-1 cut objective is used; setting the value to **hyperedges**, reduces the number of cutting hyperedges. The **default** is *connectivity*.

B.3.2 Coarsening Parameters

The section describes the partitioning parameters for the coarsening phase.

FEHG_HASH_FUNCTION

The hash function used for hypergraph initial redistribution among processors. It can be set to one the following values:

1. **auto:** selects the hash function automatically i.e. the one that gives the best distribution. This is the **default** value.
2. **zhash:** *Zoltan's* internal hash function.
3. **rshash:** Robert Sedgwick's Algorithm [Sed02].

4. **jshash:** Bitwise hash function written by Justin Sobel.
5. **pjwhash:** Peter J. Weinberger of AT&T Bell Labs [ASU86].
6. **elfhash:** The ELF hash function.
7. **bkdrhash:** The hash function proposed in the "The C programming language book" by Kernighan and Ritchie [KRE88].
8. **sdbmhash:** The hash function from SDBM open source database management project.
9. **djbhash:** The hash function by Daniel J. Bernstein.
10. **dekhash:** The hash function proposed by Knuth [Knu98].
11. **bphash:** The BP hash function.
12. **fnvhash:** Fowler-Noll-Vo hash function proposed by Glenn Fowler, Landon Curt Noll, and Kiem-Phong Vo.
13. **aphash:** The hash function proposed by Arash Partow.

FEHG_COARSENING_LIMIT

it tells when to stop the coarsening. if the number of vertices in the coarsest hypergraph is less than this limit, the coarsening process stops and the algorithm proceeds with the initial partitioning phase. The **default** is 100.

FEHG_COARSENING_METHOD

defines the coarsening method. There is only one option for this parameter and should be set to **rough** that is rough set clustering based algorithm.

FEHG_VERTEX_VISIT_ORDER

the order by which vertices of the hypergraph are visited for vertex matching in the rough clustering algorithm. The values can be set to *random* (randomly visit vertices), *linear* (linearly visit vertices in the order provided), and *degree* (sorts vertices based on their degrees). The **default** is *random*.

FEHG_ENABLE_MULTI_MATCH

whether to use multi-match vertex matching in the coarsening phase or only perform pair-matches. The value is only supported by the serial algorithm and

the parallel algorithm performs pair vertex matching. If the value is set to one, multi-match is supported for the vertices that are incident on exactly the same hyperedge set. If the value is set to two, *FEHG* matches all the vertices belong to the same core[3] to build a coarser vertex (as described in Chapter 4.4.2). Vertex weight limitations is considered such that the weight of a vertex is not allowed to grow more than a limit (the limit is defined as half size of a part). The **default** is 0.

FEHG_USE_RANDOM_MATCH

if set, the algorithm performs random matching as described in Chapter 4.3.1 to meet a certain reduction level between two coarsening levels. The parameters is only used in the serial algorithm. Setting this parameter gives higher quality. The **default** is 1.

FEHG_CORE_SIZE_LIMIT

an upper bound for the size of cores when categorising vertices to *core* and *non-core* vertices. The **default** is 500.

FEHG_MERGE_EXTERNAL_EDGES

if set, the algorithm identifies identical external hyperedges. Among a set of identical hyperedges, one is kept and the others are removed from the hypergraph. The weight of the kept hyperedge is the sum of the weight of all identical hyperedges. Identical internal hyperedges are found and removed from the hypergraph in either situation. The **default** is 1.

FEHG_REASSIGN_EDGE_GNOS

reassigns hyperedge global IDs in the coarser hypergraph. The algorithm is based on hash functions and uses global ID hashing for some hyperedge operations. Setting this parameter would increase the performance and quality of some hypergraph operations. The **default** is 1.

FEHG_EDGE2PROC_LOCALITY

if set, the algorithm considers hyperedge locality when performing global

[3]In our rough set clustering approach, vertices are categorised as core and non-core vertices.

hyperedge operations such as calculating global edge sizes and edge-to-processor adjacency list. A hyperedge is hashed to a processor that have a local copy of the hyperedge. This cause less network communication but more load imbalance among processors. The **default** is 1.

FEHG_EDGE_SIZE_THRESHOLD

hyperedges greater than this size are not processed while finding matches for the vertices. The **default** is 500.

FEHG_USE_FAST_INTERSECT

if set, the algorithm uses a fast intersection method using hash functions for calculating hyperedge intersection in the parallel Hyperedge Connectivity Graph (HCG) algorithm. The **default** is 0.

FEHG_HEDGE_INTERSECT

if the fast intersection is not set, *FEHG* calculates the intersection between two external hyperedges using one of the following methods set by this parameter. The parameters determines how to calculate the intersection between two hyperedges for the non-local data.

optimistic calculates the intersection locally and assumes that two external hyperedges have the same vertex set on other processors.

pessimistic calculates the intersection locally and assumes that two external hyperedges do not have any vertex in common on other processors.

approximate approximates the external intersection. This is the **default** value.

ignore ignores off-processor data and calculates the intersection using local data.

FEHG_HCG_SIMILARITY_METHOD

the hyperedge similarity function while calculating the Hyperedge Connectivity Graph (HCG). The values can be as follows:

jaccard the Jaccard similarity function. This is the **default** value.

set_cosine the set cosine similarity function.

dice the Dice similarity function.

FEHG_HCG_SIMILARITY_THRESHOLD

the similarity threshold for calculating HCG.

FEHG_SIM_THRESHOLD_AUTOADJUST

determines how to calculate the similarity threshold when the structure of the hypergraph changes. The values are as follows:

0 no re-adjustment is done. The value of the similarity threshold is given by FEHG_HCG_SIMILARITY_THRESHOLD parameter.

-1 the similarity threshold value is calculated in the beginning of each bipartitioning recursion and it is readjusted in the beginning of each coarsening level. This is the **default** value.

-2 the similarity threshold value is calculated in the beginning of the algorithm for the original hypergraph. Then it is readjusted in each coarsening level. When going through the coarsening levels, the history of the values are saved. The history is used to calculate the similarity threshold for left (sub-hypergraph assigned to part 0) and right (sub-hypergraph assigned to part 1) sub-hypergraphs at the end of each recursion.

-3 the similarity threshold value is calculated in the beginning of each bipartitioning recursion as well as in each coarsening level.

FEHG_CLUSTERING_THRESHOLD

the **clustering threshold** in the rough set clustering algorithm. The **default** is 0.0.

FEHG_HCG_LOCAL_CLUSTERING

the local hyperedge clustering algorithm while building HCG. Either of **agg** (agglomerative) or **bfs** (breadth first search) can be used. The first is slower but gives better quality. The **default** is *agg*.

B.3.3 Initial Partitioning Parameters

The section describes the partitioning parameters for the initial partitioning phase.

FEHG_COARSEPARTITION_METHOD

sets the initial partitioning algorithm. The possible values are.

random randomly assigns vertices to the parts.

linear linearly assigns vertices to the parts.

greedy in bipartitioning, it select a vertex randomly and assign it to part one and all other vertices to part 0. Then a single run of the FM algorithm calculates a bipartitioning on the hypergraph.

auto the algorithm is selected automatically. This is the **default** value.

FEHG_NUM_PROCESSOR_FOR_INIT_PART

the percentage of processor that are participating in the initial partitioning phase. The value is given in $[0, 1]$. At least one processor calculates the initial partitioning. The **default** is 0.5.

B.3.4 Uncoarsening Parameters

The section describes the partitioning parameters for the refinement phase.

FEHG_DIRECT_KWAY

if set, *FEHG* uses direct *k-way* refinement otherwise recursive bipartitioning. Direct *k-way* algorithm is under development. The **default** is 0.

FEHG_REFINEMENT_DOUBLE_GAIN

if set, *FEHG* uses double gain FM algorithm, otherwise one gain is used for every vertex. The **default** is 1.

FEHG_REFINEMENT_LOOP_LIMIT

the number of passes of the FM algorithm. The **default** is 4.

FEHG_FM_EDGE_SCALING

if set, hyperedge weights are scaled according to their **edge partition** in the

Hyperedge Connectivity Graph (HCG). The higher the density of the edge partition, the higher its hyperedges weights would be. The **default** is 0.

FEHG_REFINEMENT_MAX_NEG_MOVE

the maximum number of consecutive negative moves allowed by the FM algorithm in each pass. The pass terminates if FM exceeds this limit. The **default** is 250.

FEHG_REFINEMENT_TOKEN_HOLD

The value of **token** in the synchronised based parallel FM algorithm. The value should be positive. The **default** is 16.

FEHG_REFINEMENT_QUALITY

It is used in the parallel FM algorithm and the values are as follow.

> 1 The hyperedge status change are communicated among processors if their size is two and the token value is positive. This is the **default** value.

> 2 Saves the vertex-part number in the beginning of the refinement phase. If the cut increases at the end of the refinement phase, the part values are restored. This operation is done if the balance criteria is ok.

B.3.5 Recursive Bipartitioning Parameters

The section describes the recursive bipartitioning parameters.

FEHG_RUNS

The number of runs of the *FEHG* algorithm before proceeding to the next recursive bipartitioning level. In each recursion of the algorithm, *FEHG* runs the multi-level bipartitioning FEHG_RUNS times and the best partitioning that meets the balance constraint and gives the minimum cost is selected for the next recursion. The **default** is 1.

FEHG_RECURSIVE_PROC_SPLIT

if set, *FEHG* uses *Bisection processor splitting*. At the end of each bipartitioning, the processors are split into two equally sized separate subsets. Vertices in the

first part and their incident hyperedges are assigned to the first subset and the other vertices and their incident hyperedges are assigned to the other subset. Each subset continues with the partitioning of the hypergraph independently. The **default** is 1.

FEHG_MULTIPLE_BISECTION

if set, *FEHG* uses **Multiple Bisection** as described in Chapter 5.2.4. This option can be used when the number of processors is a power of two. Supporting arbitrary number of processors is planned as the future work. The **default** is 1.

FEHG_MULTIPLE_BISECTION_NGROUPS

the **replication factor** in the multiple bisection. If the value is zero, *FEHG* calculates the replication factor according to the FEHG_MULTIPLE_BISECTION_MIN_GSIZE parameter. The **default** is 0.

FEHG_MULTIPLE_BISECTION_MIN_GSIZE

The **minimum subgroup size** in the multiple bisection. The **default** is 8.

FEHG_BAL_TOL_ADJUSTMENT

The balance tolerance readjustment in each recursion of the *FEHG* algorithm. If the value is less than or equal to one, the balance tolerance is multiplied by FEHG_BAL_TOL_ADJUSTMENT after each recursion. Otherwise, the balance is readjusted based on the number of global partitions, the level of recursion, and the current imbalance tolerance of the partitioning that is obtained up to this level of recursion.

B.4 Partitioning Example Code

In this section, we provide an example of the partitioning code written in the following. After declaring the variables, the program starts by initialising the *MPI*. The hypergraph is read from the input file and saved in *hg* object the is defined in "Hypergraph.h". The function read_sparse_rowFormat reads the hypergraph from the input and then it is distributed among the processors by distribute function.

In the next step, *Zoltan* is created and initialised. This must be done after initialising the *MPI*. After that, the list of the query function is registered with *Zoltan*. Here is the list of all functions that are needed for hypergraph partitioning. All functions, such as get_number_of_vertices, are defined in our hypergraph class. The partitioning method is selected by setting LB_METHOD to *FEHG*.

We have two categories of parameters. The general parameters such as the imbalance tolerance and the number of partitions are set as 1.05 and 8, respectively. The *FEHG* specific parameters are set in the next lines of codes. After initialising all parameters, the partitioner function is called by invoking the Zoltan_LB_Partition function. After the partitioning, the internal memory is freed by calling Zoltan_LB_Free_Part method.

The last step releases all resources including the MPI resources and destroys the *Zoltan* data structure. The source code is proposed in the next page.

```
#include "Hypergraph.h"
#include "mpi.h"
#include "zoltan.h"

using namespace std;

int main(int argc, char** argv) {

    /******************/
    /* Define variables
        .
        .
        .
    *******************/

    Zoltan_Struct *zz=NULL;      /* zoltan data structure */
    Hypergraph* hg=NULL;         /* hypergraph to be partitioned */

    /* Define partitioning zoltan input lists */
    ZOLTAN_ID_PTR importGlobalGids=NULL, importLocalGids=NULL,
                  exportGlobalGids=NULL, exportLocalGids=NULL;
    int           *importProcs=NULL    , *importToPart=NULL,
                  *exportProcs=NULL    , *exportToPart=NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /************************/
    /* Read the hypergraph */
    /************************/
    hg = new Hypergraph(rank, size);
    hg->read_sparse_rowFormat(ingraph);
    hg->distribute(MPI_COMM_WORLD);

    /*********************/
    /* Initialise Zoltan */
    /*********************/
    zz = Zoltan_Create(MPI_COMM_WORLD);
    err_code = Zoltan_Initialize(argc, argv, &version);

    /*********************************/
    /* Register Zoltan query functions */
    /*********************************/
    Zoltan_Set_Num_Obj_Fn           (zz,get_number_of_vertices, hg);
    Zoltan_Set_Obj_List_Fn          (zz,get_vertex_list, hg);
    Zoltan_Set_HG_Size_Edge_Wts_Fn  (zz,get_number_of_hedges, hg);
```

```
Zoltan_Set_HG_Edge_Wts_Fn       (zz,get_hedge_list, hg);
Zoltan_Set_HG_Size_CS_Fn        (zz,get_hypergraph_size, hg);
Zoltan_Set_HG_CS_Fn             (zz,get_hypergraph, hg);

/*************************/
/* The partitioning method */
/*************************/
Zoltan_Set_Param(zz, "LB_METHOD"                ,"FEHG");
Zoltan_Set_Param(zz, "LB_APPROACH"              ,"PARTITION");
Zoltan_Set_Param(zz, "HYPERGRAPH_PACKAGE"       ,"FEHG");

/*************************/
/* set general parameters. */
/*************************/
Zoltan_Set_Param(zz, "IMBALANCE_TOL"            ,"1.05");
Zoltan_Set_Param(zz, "NUM_GLOBAL_PARTS"         ,"8");
Zoltan_Set_Param(zz, "RETURN_LISTS"             ,"NONE");
Zoltan_Set_Param(zz, "NUM_GID_ENTRIES"          ,"1");
Zoltan_Set_Param(zz, "AUTO_MIGRATE"             ,"FALSE");
Zoltan_Set_Param(zz, "NUM_LID_ENTRIES"          ,"1");
Zoltan_Set_Param(zz, "OBJ_WEIGHT_DIM"           ,"1");
Zoltan_Set_Param(zz, "EDGE_WEIGHT_DIM"          ,"1");
Zoltan_Set_Param(zz, "DEBUG_LEVEL"              ,"0");

/************************/
/* set FEHG parameters. */
/************************/
Zoltan_Set_Param(zz, "FEHG_USE_FAST_INTERSCT"       , "0");
Zoltan_Set_Param(zz, "FEHG_BAL_TOL_ADJUSTMENT"      , "2.0");
Zoltan_Set_Param(zz, "FEHG_COARSEPARTITION_METHOD"  , "auto");
Zoltan_Set_Param(zz, "FEHG_CLUSTERING_THRESHOLD"    , "0.0");
Zoltan_Set_Param(zz, "FEHG_REFINEMENT_TOKEN_HOLD"   , "16");
Zoltan_Set_Param(zz, "FEHG_OUTPUT_LEVEL"            , "1");
Zoltan_Set_Param(zz, "FEHG_USE_TIMERS"              , "2");
Zoltan_Set_Param(zz, "FEHG_EDGE2PROC_LOCALITY"      , "8");


/************************/
/* Call the partitioner */
/************************/
err_code = Zoltan_LB_Partition(
    zz,                 /* Zoltan data structure created by Zoltan_Create() */
    &changes,           /* 1 if partitioning was changed, 0 otherwise */
    &numGidEntries,     /* Number of integers used for a global ID */
    &numLidEntries,     /* Number of integers used for a local ID */
    &numImport,         /* Number of vertices to be sent to me */
    &importGlobalGids, /* Global IDs of vertices to be sent to me */
```

```
        &importLocalGids,  /* Local IDs of vertices to be sent to me */
        &importProcs,      /* Process rank for source of each incoming vertex */
        &importToPart,     /* New partition for each incoming vertex */
        &numExport,        /* Number of vertices I must send to other processes*/
        &exportGlobalGids, /* Global IDs of the vertices I must send */
        &exportLocalGids,  /* Local IDs of the vertices I must send */
        &exportProcs,      /* Process to which I send each of the vertices */
        &exportToPart);    /* Partition to which each vertex will belong */

    /*************************************/
    /* Free zoltan internal data structure */
    /*************************************/
    Zoltan_LB_Free_Part(&importGlobalGids, &importLocalGids,
       &importProcs, &importToPart);
    Zoltan_LB_Free_Part(&exportGlobalGids, &exportLocalGids,
       &exportProcs, &exportToPart);

    /**********************/
    /* Finalise operations
        .
        .
        .
    ***********************/
   Zoltan_Destroy(&zz);
   MPI_Finalize();
   delete hg;

   return 0;
}
```

Bibliography

[AAI06] C. Andrew, B.K. Andrew, and M. Igor. MLPart: High-performance min-cut bisector. `http://vlsicad.ucsd.edu/GSRC/bookshelf/Slots/Partitioning/MLPart/`, 2006.

[ACKM00] C.J. Alpert, A.E. Caldwell, A.B. Kahng, and I.L. Markov. Hypergraph partitioning with fixed vertices [VLSI CAD]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(2):267–272, Feb 2000.

[ACU08] C. Aykanat, B.B. Cambazoglu, and B. Uçar. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *Journal of Parallel and Distributed Computing*, 68(5):609–625, 2008.

[AFG+09] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica. Above the clouds: A Berkeley view of cloud computing. Technical report, University of California at Berkeley, 2009.

[AHK98] C.J. Alpert, J. Huang, and A.B. Kahng. Multilevel circuit partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(8):655–667, 1998.

[AK95] C.J Alpert and A.B. Kahng. Recent directions in netlist partitioning: A survey. *Elsevier Integration, the VLSI Journal*, 19(1):1–81, 1995.

[Alp96] C.J. Alpert. *Multi-way Graph and Hypergraph Partitioning*. PhD thesis, Computer Science Department, University of California, Los Angeles, 1996.

[Alp98] C.J. Alpert. The ISPD98 circuit benchmark suite. In *Proceedings of the 1998 International Symposium on Physical Design*, ISPD '98, pages 80–85. ACM, 1998.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques.* Addison wesley, 1986.

[AV00] S. Areibi and A. Vannelli. Tabu search: A meta heuristic for netlist partitioning. *VLSI Design*, 11(3):259–283, 2000.

[AV03] S. Areibi and A. Vannelli. Tabu search: Implementation and complexity analysis for netlist partitioning. *International Journal of Computers and their Applications*, 10:211–232, 2003.

[BCH13] L.A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[BCLS87] T.N. Bui, S. Chaudhuri, F.T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.

[BDH03] L.A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[BHJL89] T. Bui, C. Heigham, C. Jones, and T. Leighton. Improving the performance of the Kernighan-Lin and simulated annealing graph bisection algorithms. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 775–778. ACM, 1989.

[BJKT05] J.T. Bradley, D.V. Jager, W.J. Knottenbelt, and A. Trifunović. *Proceedings of Formal Techniques for Computer Systems and Business Processes: European Performance Engineering Workshop (EPEW 2005), and International Workshop on Web Services and Formal Methods (WS-FM 2005)*, chapter Hypergraph Partitioning for Faster Parallel PageRank Computation, pages 155–171. Springer Berlin Heidelberg, 2005.

[Blo13] M. Bloznelis. Degree and clustering coefficient in sparse random intersection graphs. *The Annals of Applied Probability*, 23(3):1254–1289, 2013.

[BVS13] R. Buyya, C. Vecchiola, and T.S. Selvi. *Mastering cloud computing: Foundations and applications programming*. Newnes, 2013.

[ÇA99] Ü.V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[ÇA11] Ü.V. Çatalyürek and C. Aykanat. PaToH (Partitioning Tool for Hypergraphs). In *Encyclopedia of Parallel Computing*, pages 1479–1487. Springer, 2011.

[ÇBD+07] Ü.V. Çatalyürek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy, and L.A. Riesen. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, 2007.

[CBL+14] Q. Chen, S. Bai, Z. Li, Z. Gou, B. Suo, and W. Pan. GraphHP: A hybrid platform for iterative graph processing. *Retrieved July*, 17:2014, 2014.

[CES] CESM: The Community Earth System Model. CCSM 3.0 Community Atmosphere Model (CAM). `http://www.cesm.ucar.edu/models/atm-cam/`.

[CJZM10] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[CKM99] A.E. Caldwell, A.B. Kahng, and I.L. Markov. Hypergraph partitioning with fixed vertices. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 355–359, 1999.

[CL98] J. Cong and S.K. Lim. Multiway partitioning with pairwise movement. In *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '98, pages 512–516, 1998.

[CLL+97] J. Cong, H.P. Li, S.K. Lim, T. Shibuya, and D. Xu. Large scale circuit partitioning with loose/stable net removal and signal flow based clustering. In *IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers.,*, pages 441–446, 1997.

[CPH+14] Xiao C., Qinke P., Libin H., Tao Z., and Tao X. An effective haplotype assembly algorithm based on hypergraph partitioning. *Journal of Theoretical Biology*, 358(0):85 – 92, 2014.

[CST+10] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[CYW+12] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. Improving large graph processing on partitioned graphs in the cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 3, 2012.

[DBH+05] K.D. Devine, E.G. Boman, R.T. Heaphy, B.A. Hendrickson, J.D. Teresco, J. Faik, J.E. Flaherty, and L.G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, February 2005.

[DBH+06] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and Ü.V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2006)*, page 10pp, April 2006.

[DBR+09] K.D. Devine, E.G. Boman, L.A. Riesen, Ü.V. Çatalyürek, and C. Chevalier. Getting started with Zoltan: A short tutorial. In *Proceeding of 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*, 2009. Also available as Sandia National Labs Tech Report SAND2009-0578C.

[DD97] S. Dutt and W. Deng. VLSI circuit partitioning by cluster-removal using iterative improvement techniques. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 194–200. IEEE Computer Society, 1997.

[DG08] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DH11] T.A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1, 2011.

[DHK03] N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. HYDRA: HYpergraph-based Distributed Response-time Analyzer. In *PDPTA*, volume 3, pages 215–219, 2003.

[Dic86] J.R. Dickinson. *The bibliography of marketing research methods*. Lexington Books, 1986.

[Div15] NASA Advanced Supercomputing Division. The NAS parallel benchmarks (NPB). `http://www.nas.nasa.gov/publications/npb.html`, 2015.

[DRBL09] A. Ducournau, S. Rital, A. Bretto, and B. Laget. A multilevel spectral hypergraph partitioning approach for color image segmentation. In *IEEE International Conference on Signal and Image Processing Applications (ICSIPA)*, pages 419–424, Nov 2009.

[EGP66] P. Erdos, A.W. Goodman, and L. Pósa. The representation of a graph by set intersections. *Canadian Journal of Mathematics*, 18(106-112):86–93, 1966.

[EH08] C. Evangelinos and C. Hill. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on Amazons EC2. *ratio*, 2(2.40):2–34, 2008.

[ESK02] L. Ertöz, M. Steinbach, and V. Kumar. A new shared nearest neighbor clustering algorithm and its applications. In *Workshop on Clustering*

High Dimensional Data and its Applications at 2nd SIAM International Conference on Data Mining, pages 105–115, 2002.

[ESK03] L. Ertöz, M. Steinbach, and V. Kumar. Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data. In *Proceedings of the SIAM International Conference on Data Mining (SDM '03)*, pages 47–58. SIAM, 2003.

[FFF99] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *SIGCOMM '99 Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, 29(4):251–262, August 1999.

[Fjä98] P. Fjällström. Algorithms for graph partitioning: A survey. *Linköping Electronic Articles in Computer and Information Science*, 3(10), 1998.

[FJPS11] I. Foudalis, K. Jain, C. Papadimitriou, and M. Sideri. Modeling social networks through user background and behavior. In *Proceedings of the 8th International Conference on Algorithms and Models for the Web Graph*, WAW'11, pages 85–102. Springer-Verlag, 2011.

[Flo14] Florida State University. The Scalable Parallel Random Number Generators Library (SPRNG). `http://www.sprng.org/`, 2014.

[FM82] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *19th Conference on Design Automation*, pages 175–181. IEEE, 1982.

[GB83] M.K. Goldberg and M. Burstein. *Heuristic improvement technique for bisection of VLSI networks*. IBM Thomas J. Watson Research Division, 1983.

[GBK+02] A. Gavin, M. Bösche, R. Krause, P. Grandi, M. Marzioch, A. Bauer, J. Schultz, J.M. Rick, A. Michon, C. Cruciat, et al. Functional organization of the yeast proteome by systematic analysis of protein complexes. *Nature*, 415(6868):141–147, 2002.

[GKG+13] A. Gupta, L.V. Kale, F. Gioachin, V. March, C.H. Suen, B. Lee, P. Faraboschi, R. Kaufmann, and D. Milojicic. The who, what, why and how of high performance computing applications in the cloud. In *Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science*, 2013.

[GL98] J. Gong and S.K. Lim. Multiway partitioning with pairwise movement. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD 98)*, pages 512–516, 1998.

[GLG+12] J.E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30. USENIX Association, 2012.

[GM11] A. Gupta and D. Milojicic. Evaluation of HPC applications on cloud. In *Sixth IEEE Open Cirrus Summit (OCS)*, pages 22–26, 2011.

[Gra03] A. Grama. *Introduction to parallel computing*. Pearson Education, 2003.

[GSKM13] A. Gupta, O. Sarood, L.V. Kale, and D. Milojicic. Improving HPC application performance in cloud through dynamic load balancing. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 402–409, 2013.

[HB97] S. Hauck and G. Borriello. An evaluation of bipartitioning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, 1997.

[HBD+12] N. Huber, F. Brosig, N. Dingle, K. Joshi, and S. Kounev. Providing dependability and performance in the cloud: Case studies. In *Resilience Assessment and Evaluation of Computing Systems*, pages 391–412. Springer, 2012.

[HC14] B. Heintz and A. Chandra. Beyond graphs: Toward scalable hypergraph analysis systems. *ACM SIGMETRICS Performance Evaluation Review*, 41(4):94–97, 2014.

[HDF11] K. Hwang, J. Dongarra, and C.G. Fox. *Distributed and cloud computing: From parallel processing to the internet of things*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[Hen98] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Solving Irregularly Structured Problems in Parallel*, pages 218–225. Springer, 1998.

[HK92] L. Hagen and A.B. Kahng. A new approach to effective circuit clustering. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD-92), Digest of Technical Papers.*,, pages 422–427, 1992.

[HKKM98] E. Han, G. Karypis, V. Kumar, and B. Mobasher. Hypergraph based clustering in high-dimensional data sets: A summary of results. *IEEE Data Engineering Bulletin*, 21(1):15–22, 1998.

[HL95] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 28. ACM, 1995.

[HLT+14] T. Hu, C. Liu, Y. Tang, J. Sun, H. Xiong, and S.Y. Sung. High-dimensional clustering: A clique-based hypergraph partitioning framework. *Knowledge and information systems*, 39(1):61–88, 2014.

[HSS10] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2010)*, pages 1–12, 2010.

[HWL12] L. Ho, J. Wu, and P. Liu. Distributed graph database for large-scale social computing. In *IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 455–462, 2012.

[IWW93] E. Ihler, D. Wagner, and F. Wagner. Modeling hypergraphs by graphs with the same mincut properties. *Information Processing Letters*, 45(4):171–175, March 1993.

[JAMS89] D.S. Johnson, C.R. Aragon, L.A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Operations research*, 37(6):865–892, 1989.

[JDV+09] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P Berman, and P. Maechling. Scientific workflow applications on Amazon EC2. In *5th IEEE International Conference on E-Science Workshops*, pages 59–66, 2009.

[JNWH04] T.B. Jeremy, J.D. Nicholas, J.K. William, and J.W. Helen. Hypergraph-based parallel computation of passage time densities in large semi-Markov models. *Linear Algebra and its Applications*, 386:311 – 334, 2004. Special Issue on the Conference on the Numerical Solution of Markov Chains 2003.

[JRM+10] K.R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H.J. Wasserman, and N.J. Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 159–168, 2010.

[KAKS99] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999.

[Kar02] G. Karypis. Multilevel hypergraph partitioning. Technical report, University of Minnesota, 2002.

[Kar07] G. Karypis. hMetis: Hypergraph and circuit partitioning - version 1.5. `http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview`, 2007.

[Kar13a] G. Karypis. Metis: Serial graph partitioning and fill-reducing matrix ordering (version 5.1.0). `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`, 2013.

[Kar13b] G. Karypis. ParMetis: Parallel graph partitioning and fill-reducing matrix ordering (version 4.0.3). `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`, 2013.

[KHT09] S. Klamt, U. Haus, and F. Theis. Hypergraphs and cellular networks. *PLoS Computational Biology*, 5(5):e1000385, 2009.

[Kim13] S.P. Kim. Mechanisms underlying restoration of hepatic insulin sensitivity with CB1 antagonism in the obese dog model. *Adipocyte*, 2(1):47–49, 2013.

[KK96] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs, department of computer science technical report 96-036. *University of Minnesota, Minneapolis, MN*, 1996.

[KK97] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[KK98a] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KK98b] G. Karypis and V. Kumar. hMetis: A hypergraph partitioning package version 1.5 user manual, 1998.

[KK98c] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[KK99] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. In *Proceedings of ACM/IEEE Design Automation Conference*, pages 343–348, 1999.

[KK00] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. *VLSI design*, 11(3):285–300, 2000.

[KL70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, 1970.

[Knu98] D.E. Knuth. *The art of computer programming: Sorting and searching*, volume 3. Pearson Education, 1998.

[KPÇA12] E. Kayaaslan, A. Pinar, Ü.V. Çatalyürek, and C. Aykanat. Partitioning hypergraphs in scientific computing applications through vertex separators on graphs. *SIAM Journal on Scientific Computing*, 34(2):A970–A992, 2012.

[KRE88] B.W. Kernighan, D.M. Ritchie, and P. Ejeklint. *The C programming language*, volume 2. Prentice-Hall Englewood Cliffs, 1988.

[Kri84] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. Comput.*, 33(5):438–446, May 1984.

[KTF09] U. Kang, C.E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Ninth IEEE International Conference on Data Mining (ICDM'09)*, pages 229–238, 2009.

[LDK+05] P. Luszczek, J.J. Dongarra, D. Koester, R. Rabenseifner, B. Lucas, J. Kepner, J. McCalpin, D. Bailey, and D. Takahashi. Introduction to the HPC challenge benchmark suite. *Lawrence Berkeley National Laboratory*, 2005.

[Lee87] W. Lee. Gyrokinetic particle simulation model. *Journal of Computational Physics*, 72(1):243–269, 1987.

[Len90] T. Lengauer. *Combinatorial algorithms for integrated circuit layout*. Wiley-Teubner, 1990.

[LK13] D. LaSalle and G. Karypis. Multi-threaded graph partitioning. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2013)*, pages 225–236, May 2013.

[LM11] A.N. Langville and C.D. Meyer. *Google's PageRank and beyond: The science of search engine rankings*. Princeton University Press, 2011.

[LMDV08] M. Latapy, C. Magnien, and N. Del Vecchio. Basic notions for the analysis of large two-mode networks. *Social Networks*, 30(1):31–48, 2008.

[LPA+09] D. Lazer, A. Pentland, L. Adamic, S. Aral, A. Barabsi, D. Brewer, N. Christakis, N. Contractor, J. Fowler, M. Gutmann, T. Jebara, G. King, M. Macy, D. Roy, and M. Van Alstyne. Computational social science. *Science*, 323(5915):721–723, 2009.

[LT02] G. Lambert-Torres. Application of rough sets in power system control center data mining. In *IEEE Power Engineering Society Winter Meeting 2002*, volume 1, pages 627–631, 2002.

[LW01] D. Liu and M. Wu. A hypergraph based approach to declustering problems. *Distributed and Parallel Databases*, 10(3):269–288, 2001.

[LW04] P. Lingras and C. West. Interval set clustering of web users with rough k-means. *Journal of Intelligent Information Systems*, 23(1):5–16, 2004.

[MAB+10] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, 2010.

[MCS15] Claudio Mrquez, Eduardo Cesar, and Joan Sorribes. Graph-based automatic dynamic load balancing for HPC agent-based simulations. *In Proceeding of 3rd Workshop on Parallel and Distributed Agent-Based Simulations (PADABS2015), Vienna, Austria*, 2015.

[MDH+12] P. Mehrotra, J. Djomehri, S. Heistand, R. Hood, H. Jin, A. Lazanoff, S. Saini, and R. Biswas. Performance evaluation of Amazon EC2 for NASA HPC applications. In *Proceedings of the 3rd workshop on Scientific Cloud Computing Date*, pages 41–50. ACM, 2012.

[MJ79] G.R. Michael and D.S. Johnson. Computers and intractability: a guide to the theory of NP-Completeness. *W. H. Freeman and Company*, 1979.

[MLLS14] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable graph partitioning in the cloud. *arXiv preprint arXiv:1404.3861*, 2014.

[NB09] J. Napper and P. Bientinesi. Can cloud computing reach the top500? In *ACM Proceedings of the combined workshops on Unconventional high performance computing workshop plus memory access workshop*, pages 17–20, 2009.

[NM09] C. Nikolai and G. Madey. Tools of the trade: A survey of various agent based modeling platforms. *Journal of Artificial Societies and Social Simulation*, 12(2):2, 2009.

[Paw91] Zdzisław Pawlak. *Rough Sets: Theoretical aspects of reasoning about data*. Kluwer Academic Publishers, Norwell, USA, 1991.

[PC10] S. Peter and S. Christian. Engineering multilevel graph partitioning algorithms. *CoRR*, abs/1012.0006, 2010.

[PM07] D.A. Papa and I.L. Markov. Hypergraph partitioning and clustering. *Approximation algorithms and metaheuristics*, 61:1–19, 2007.

[PPS05] Z. Pawłak, L. Polkowski, and A. Skowron. Rough sets: An approach to vagueness. *Encyclopedia of Database Technologies and Applications*, pages 575–580, 2005.

[PRN+11] T. Peterka, R. Ross, B. Nouanesengsy, T. Lee, H. Shen, W. Kendall, and J. Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, pages 580–591, 2011.

[PRR12] M. Prerna, K. Rekha, and V. Ritu. Rough set approach in machine learning: A review. *International Journal of Computer Applications*, 56(10):1–13, October 2012.

[PWB07] D. Parmar, T. Wu, and J. Blackhurst. MMR: An algorithm for clustering categorical data using rough set theory. *Data and Knowledge Engineering*, 63(3):879–893, 2007.

[RBT+13] B. Rob, F.A. Bas, L. Tristan, M. Wouter, P. Daan, V. Brendan, Y. Albert-Jan, Ü.V. Çatalyürek, and K. Stanley. Mondriaan for sparse matrix partitioning. `http://www.staff.science.uu.nl/~bisse101/Mondriaan/mondriaan.html`, 2013.

[Rit09] S. Rital. Hypergraph cuts and unsupervised representation for image segmentation. *Fundamenta Informaticae*, 96(1):153–179, 2009.

[RPG+13] F. Rahimian, A.H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 51–60, 2013.

[San89] L.A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, Jan 1989.

[San14a] Sandia National Laboratories. Trilinos: Open source software libraries for the development of scientific applications. `http://trilinos.org/`, 2014.

[San14b] Sandia National Laboratories. Zoltan: Parallel partitioning, load balancing and data-management services. `http://www.cs.sandia.gov/zoltan/`, 2014.

[Sch09] C. Schulz. *Scalable parallel refinement of graph partitions*. PhD thesis, Diploma thesis, Institute for Theoretical Computer Science (Algorithmics II) and Institute for Applied and Numerical Mathematics, Universität Karlsruhe, 2009.

[Sci08] Scientific Discovery through Advanced Computing (SciDAC). Partitioning, load balancing, and ordering using Zoltan. `http://cscapes.cs.purdue.edu/projects.html`, 2008.

[Sci15] SciDAC: Office of Science. Interoperable technologies for advanced petascale simulations (ITAPS). `http://www.scidac.gov/math/ITAPS.html`, 2015.

[Sed02] R. Sedgewick. *Algorithms in Java, Parts 1-4*. Addison-Wesley Professional, 2002.

[She12] N.A. Sherwani. *Algorithms for VLSI physical design automation*. Springer Science and Business Media, 2012.

[SK06] N. Selvakkumaran and G. Karypis. Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):504–517, March 2006.

[SKK99] K. Schloegel, G. Karypis, and V. Kumar. *A New Algorithm for Multi-objective Graph Partitioning*. Springer, 1999.

[SKK00] M. Steinbach, G. Karypis, and V. Kumar. A comparison of document clustering techniques. *KDD Workshop on Text Mining*, 400(1):525–526, 2000.

[SNK95] T. Shibuya, I. Nitta, and K. Kawamura. SMINCUT: VLSI placement tool using min-cut. *Fujitsu Scientific and Technical Journal*, 31(2):197–207, 1995.

[SR92a] Y. Saab and Vasant Rao. On the graph bisection problem. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 39(9):760–762, Sep 1992.

[SR92b] A. Skowron and C. Rauszer. The discernibility matrices and functions in information systems. In *Intelligent Decision Support*, pages 331–362. Springer, 1992.

[ST97] H.D. Simon and S. Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997.

[THK09] Z. Tian, T. Hwang, and R. Kuang. A hypergraph-based learning algorithm for classifying gene expression and arraycgh data with prior knowledge. *Bioinformatics*, 25(21):2831–2838, 2009.

[TK04] A. Trifunovic and W.J. Knottenbelt. A parallel algorithm for multilevel k-way hypergraph partitioning. In *Third International Workshop on Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004.*, pages 114–121, July 2004.

[TK08] A. Trifunovic and W.J. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *Journal of Parllel and Distributed Computing*, 68(5):563–581, 2008.

[TKV10] G. Tsoumakas, I. Katakis, and I. Vlahavas. Mining multi-label data. In O. Maimon and L. Rokach, editors, *Data Mining and Knowledge Discovery Handbook*, pages 667–685. Springer US, 2010.

[TP09] K. Thangavel and A. Pethalakshmi. Dimensionality reduction based on rough set theory: A review. *Applied Software Computing*, 9(1):1–12, January 2009.

[Tri06] A. Trifunovic. *Parallel algorithms for hypergraph partitioning.* PhD thesis, University of London, Imperial College of Science, Technology and Medicine, Department of Computing, 2006.

[UA04] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[VB05] B. Vastenhouw and R.H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[Wal08] E. Walker. Benchmarking Amazon EC2 for hig-performance scientific computing. *USENIX;login: magzine*, 33(5):18–23, 2008.

[Was94] S. Wasserman. *Social network analysis: Methods and applications*, volume 8. Cambridge university press, 1994.

[WC07] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).

[WLCS00] M. Wang, S. Lim, J. Cong, and M. Sarrafzadeh. Multi-way partitioning using bi-partition heuristics. In *Proceedings of the ACM Asia and South Pacific Design Automation Conference*, page 667, 2000.

[WOV+09] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[WPS09] N.J. Wright, W. Pfeiffer, and A. Snavely. Characterizing parallel scaling of scientific applications using IPM. In *The 10th LCI International Conference on High-Performance Clustered Computing*, pages 10–12, 2009.

[Wro95] J. Wroblewski. Finding minimal reducts using genetic algorithms. In *Proccedings of the second annual join conference on infromation science*, pages 186–189, 1995.

[Wró98] J. Wróblewski. Genetic algorithms in decomposition and classification problems. In *Rough Sets in Knowledge Discovery 2*, pages 471–487. Springer, 1998.

[WXSW14] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. In *IEEE 30th International Conference on Data Engineering (ICDE)*, pages 568–579, 2014.

[XCAL14] Yadong X., Wentong C., H. Aydt, and M. Lees. Efficient graph-based dynamic load-balancing for parallel large-scale agent-based traffic simulation. In *Simulation Conference (WSC)*, pages 3483–3494, Dec 2014.

[Yan88] S. Yang. Logic synthesis and optimization benchmarks. In *published at 1989 MCNC International Workshop on Logic Synthesis*, 1988.

[YCD+11] K. Yelick, S. Coghlan, B. Draney, R. S Canon, et al. The Magellan report on cloud computing for science. *US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research (ASCR)*, 2011.

[You11] K. Yousef. Building a cloud computing platform for new possibilities. *Computer*, 44(3):29–34, 2011.

[YTW12] J. Yu, D. Tao, and M. Wang. Adaptive hypergraph learning and its application in image classification. *IEEE Transactions on Image Processing*, 21(7):3262–3272, 2012.

[ZG11] P. Zaspel and M. Griebel. Massively parallel fluid simulations on Amazon's HPC cloud. In *IEEE First International Symposium on Network Cloud Computing and Applications (NCCA)*, pages 73–78, 2011.

[ZHS06] D. Zhou, J. Huang, and B. Schölkopf. Learning with hypergraphs: Clustering, classification, and embedding. In *Advances in Neural Information Processing Systems*, pages 1601–1608, 2006.

[ZLR+12] J. Zhang, T. Li, D. Ruan, Z. Gao, and C. Zhao. A parallel method for computing rough set approximations. *Information Sciences*, 194:209–223, 2012.

[ZS95] W. Ziarko and N. Shan. Discovering attribute relationships, dependencies and rules by using rough sets. In *Proceedings of the IEEE 28th Annual*

Hawaii International Conference on System Sciences, pages 293–299, 1995.