Optimal Hypergraph Partitioning

by. Baran Usta

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

Sabancı University

August 10, 2018

OPTIMAL HYPERGRAPH PARTITIONING

APPROVED BY:

Asst. Prof. Dr. Kamer Kaya
(Thesis Supervisor) .....

Assoc. Prof. Dr. Hüsnü Yenigün .....

Asst. Prof. Dr. Zafeirakis Zafeirakopoulos .....

DATE OF APPROVAL: 31/07/2018

Optimal Hypergraph Partitioning

Baran Usta
Computer Science and Engineering, Master's Thesis, 2018
Thesis Supervisor: Kamer Kaya

Keywords: K-way hypergraph partitioning, Parallel branch and bound, Branch and bound reordering, Combinatorial algorithms

Abstract

Hypergraph partitioning into K parts has many applications in practice such as distributed algorithms and very large scale integrated circuit (VLSI) design. There are various tools proposed in the literature which can partition a given hypergraph very fast. However, since the problem is NP-Hard and the traditional approaches heavily use heuristics, these tools do not provide an optimal partition. There is limited research on partitioning hypergraphs optimally. In this thesis, we propose *PHaraoh*, a parallel hypergraph partitioner that can provide optimal partitions for many metrics used in the literature. Such a partitioner is important in practice since it enables us to evaluate the true performance of the existing tools. Furthermore, *PHaraoh* can be started with an initial partition. Thanks to that, even an optimal solution is not found within the given time limit, *PHaraoh* improves the cost of the provided initial partition. Experimental results on hypergraphs obtained from real-life matrices show that the quality of the partitions of existing tools can be improved significantly for most of the hypergraphs. In order to increase the speed up the search-space exploration, we experimented with both master-slave and work-stealing parallelization. It also has been shown that the runtime of the algorithm highly depends on the order of the items in the branch and bound tree. In this study, we propose different ordering strategies which can offer great speed ups depending on the characteristics of the hypergraph.

Optimal Hiperçizge Parçalama

Baran Usta

Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans Tezi, 2018

Tez Danışmanı: Kamer Kaya

Anahtar Kelimeler: K parçalı hiperçizge parçalama, Dal ve sınır, Paralel hesaplama, Dal ve sınır sıralaması, Kombinatoryal algoritmalar

Özet

Hiperçizge parçalama literatürde oldukça popüler bir problemdir. Dağıtık algoritmalar ve VLSI devre tasarımı gibi uygulamaların performasi bu yöntemle büyük ölçüde arttırılabilir. Son 20 yılda, hiperçizgeyi hızlı bir şekilde parçalayabilen araçlar geliştirilmiştir. Ancak bu problem NP-Zor olduğu için, bu araçlar sezgisel yöntemlere dayanmaktadır. Bu yüzden genelde optimal olmayan sonuçları bulmaktadırlar. Optimal hiperçizge problemi üzerine sezgisel yöntemlere dayalı çalışmalara nazaran çok daha az sayıda araştırma bulunmaktadır. Bu tezde, literatürdeki bir çok metriğe göre optimal parçalamayi bulan, *PHaraoh* isimli koşut hiperçizge parçalama aracı sunulmaktadır. Optimal sonuçların bulunması sayesinde, böyle bir araç, daha önceden geliştirilen hiperçizge parçalama araçlarının gerçek performansını, olası en iyi sonuç ile karşılaştırarak ölçmemizi sağlar. *PHaraoh* herhangi bir parçalama ile başlatılabilir ve bu sonucu sürekli olarak geliştirmeye çalışır. Bu sayede, istenen optimal hiperçizge parçalama islemini verilen sürede tamamlayamasa bile, baslangıçta verilen parçalamanın kalitesini iyileştirebilir. Gerçek hayattaki uygulamalarda karşılaşılan hiperçizge modelleri üzerinde yaptığımız deneylere göre, *PHaraoh* pratikte en çok kullanılan araçların ürettiği parçalamaların kalitesini dal ve sınır arama yöntemini kullanarak büyük ölçüde iyileştirmektedir. Arama uzayında bulunan optimal parçalamanın bulunma süresini kısaltmak icin, "usta yamak" ve "iş çalma" yöntemlerine dayalı koşut algoritmalardan faydalandik. Daha önceki çalışmalarımızda ve bu tezde yaptığımız deneylere göre hiperçizge parçalama problemi icin dal ve sınır ağacındaki öğelerin sırası *PHaraoh*'ın performansını önemli oranda etkilemektedir. Bu tezde, değişik sıralama yöntemleri önerip, bunların *PHaraoh*'ın çalışma süresini nasıl değiştirdiğini ve bu değişikliğin nedenlerini hiperçizgelerin özelliklerine bağlayarak açıkladık.

Acknowledgements

I would like to thank to my supervisor Kamer Kaya for everything he has done for me, especially for his invaluable guidance, limitless support and understanding. He constantly encouraged me to develop my own ideas and apply them, but steered me in the right direction whenever he thought I needed it... And I needed a lot.

I would also like to express my very profound gratitude to my family for the love, and faith. Without the inspiration, drive, and support that they have given throughout my life, I would not be the person I am today.

Finally, my sincere thanks go to Melis Maravent, for all her love, patience, and support. I am grateful for her help with keeping my sanity during the unpleasant times, and endless support for chasing my dreams.

TABLE OF CONTENTS

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

A hypergraph is a general combinatorial data structure where the nets are allowed to connect more than two vertices, in contrast to graphs. This generalization enables researchers to model the complex relationships among objects without losing the information which can be valuable for a given task. For many applications, e.g. [65], one cannot model the exact relationships information by using graphs which are only capable of representing pairwise relationships.

Hypergraphs are used for various practical problems such as circuit layout design, parallelization of linear algebra operations, machine learning, etc. In order to solve these problems efficiently, their corresponding hypergraphs have to be partitioned in a smart way which optimizes an objective function that corresponds to some form of overhead in practice. Since multi-core processors, many-core accelerators and compute clusters are commodity resources today, partitioning a hypergraph in parallel is a straightforward strategy. However, this comes with its own challenges such as the scalability. Therefore, a lot of research has been conducted on this challenging problem in both shared and distributed memory setup.

The overhead induced by a partition is usually related to the communication among the entities in different parts. For instance, in parallel execution of an application, the tasks of the application, which correspond to the vertices of the hypergraph can be assigned to different processors. A net that connects a vertex set represents some kind of information exchange among this set of vertices. If two vertices sharing a net are assigned to different processors, the net that connects these two incurs a communication overhead. While assigning all the vertices into a single part nullifies all the overhead, this is devastating for

a parallel execution since only a single processor is responsible for all the tasks. That is why hypergraphs are partitioned minimizing the communication cost and balancing the number of vertices among processors.

K-way hypergraph partitioning problem is known to be an **NP-Hard** problem [41]. Many practical partitioners such as PaToH [12], hMETIS [34], Zoltan [22, 23], ParK-way [56], Mondriaan [59] use heuristics. They find a partition very fast but sacrifice the quality of the partition. However, one can not evaluate the quality of the partition found since the optimal partition is unknown. In the literature, their performance of partition quality is usually compared with each other. Knowing the best partition not only gives us the true performance of a partitioner but also enables us to identify its weaknesses which one could focus on to alleviate the problems and improve the tool.

In this work, we developed *PHaraoh* that can partition a hypergraph into K parts optimally. It supports various partitioning metrics that are introduced to represent the communication overhead [41]. Each metric models a type of applications better, and some metrics require direction information of the nets whereas some do not. In *PHaraoh*, we tried to cover all widely-used metrics.

In order to solve the hypergraph partitioning problem optimally, we adopted the branch and bound strategy and developed bounds for each metric. Since the branch and bound approach explores all the possible solutions that it can not prune, it is a very time-consuming approach. As we have shown in Chapter 6, it is possible to make the optimal partitioning process faster providing a good (but obviously not optimal) initial partition. This also resulted in another practical use case. Since a branch and bound technique always has a solution which is continuously improved during the search, *PHaraoh* can be used to improve the partitions that are provided by the existing tools within a given time limit.

We also investigated parallelization opportunities for optimal hypergraph partitioning. We employed master-slave paradigm and obtained improvements in the runtime. However, parallelization of branch and bound technique is very challenging due to the unbalanced and unknown nature of the search space. In order to overcome these challenges which we will discuss later, we also implemented another parallelization strategy based on work stealing.

The rest of the thesis is organized as follows: The hypergraph partitioning problem

and the mathematical background is introduced at the beginning of Chapter 2. In the rest of Chapter 2, the use cases of hypergraph partitioning and the existing tools are described. Moreover, the branch and bound strategy and its parallelization techniques are presented in the same chapter. In Chapter 3, we discuss how our branch and bound algorithm works and how we compute the bounds for each metric. The parallelization techniques we employed are given in Chapter 4. In Chapter 5, we explain the reordering techniques we applied in detail. The experiments and the discussions of their results are presented in Chapter 6. Chapter 7 concludes thesis and discusses the possible directions for future work.

Chapter 2

Background and Related Work

2.1 Hypergraph Partitioning

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ is a combinatorial data structure more generalized than a graph where the nets (hyperedges) $\mathcal{N}$ can connect any number of vertices $\mathcal{V}$. Vertices that are connected by a net $n \in \mathcal{N}$ are called its *pins*. $\mathsf{pins}[n]$ and $\mathsf{nets}[v]$ are used to represent the pin set of a net $n \in \mathcal{N}$ and the set of nets having $v \in \mathcal{V}$ as a pin, respectively. In a hypergraph, vertices can have weights and the hyperedges can have costs. The weight of v is denoted by $\mathsf{w}[v]$ and the cost of a net n is denoted by $\mathsf{c}[\mathrm{n}]$.

A K-way partition of a hypergraph $\mathcal{H}$ is obtained by distributing the vertices into K parts. A partition $\Pi=\{\mathcal{V}_1,\mathcal{V}_2,\ldots,\mathcal{V}_K\}$ has the following properties:

- parts are pairwise disjoint, i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$,
- the union of K parts is equal to $\mathcal{V}$, i.e., $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$.

A K-way partition is *ε-balanced* if it satisfies the *balance constraint*

$$W_k \leq W_{avg}(1+\varepsilon), \quad \text{for } k=1,2,\ldots,K. \tag{2.1}$$

In equation (2.1), W_k denotes the total vertex weight in $\mathcal{V}_k$, that is $W_k=\sum_{v\in\mathcal{V}_k}\mathsf{w}[v]$, and W_{avg} is the average weight for a part, which is $W_{avg}=(\sum_{v\in\mathcal{V}}\mathsf{w}[v])/K$. Here, ε represents the maximum allowed imbalance ratio.

It's worth to mention that there is also another definition of the *balance constraint* which is given as;

$$W_{avg}(1-\varepsilon) \leq W_k \leq W_{avg}(1+\varepsilon), \quad \text{for } k = 1, 2, \ldots, K. \tag{2.2}$$

In *PHaraoh*, we used equation (2.1) for the *balance constraint*.

For a K-way partition Π, a net that has at least one pin in a part is said to *connect* that part. The number of parts connected by a net n is called its *connectivity* and denoted as λ_n. A net n is said to be *uncut* (*internal*) if it connects exactly one part (i.e., $\lambda_n = 1$), and *cut* (*external*), otherwise (i.e., $\lambda_n > 1$).

In the text, the part that contains a vertex v is denoted by $\mathsf{part}[v]$. $\mathsf{P}[n]$ denotes the set of parts a net n is connected to. Let $\Lambda(n,k) = |\mathsf{pins}[n] \cap \mathcal{V}_k|$ be the number of pins of net n in part k. Hence, $\Lambda(n,k) > 0$ if and only if $k \in \mathsf{P}[n]$.

The goal of the K-way partition is minimizing an objective function $\chi(.)$. There are various objective functions proposed for practical applications in the literature. These objective functions will be introduced later in this section. Before that let us give the formal definition of K-way hypergraph partitioning problem

Definition 1 (K-way Hypergraph Partition) *Given a hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$, number of parts K, a maximum allowed imbalance ratio ε, and an objective function $\chi(.)$, where $\mathcal{V}$ is the set of vertices and $\mathcal{N}$ is the set of nets, find a K-way partition Π of $\mathcal{H}$ where the objective function $\chi(\Pi)$ is optimized and the balance constraint (2.1) is satisfied.*

It is shown that K-way hypergraph partitioning is an **NP-Hard** problem [41]. However, it is critical for a variety of real world problems which we will describe at the end of Section 2.1. That's why a great deal of effort has been put into developing nice heuristics that can partition a given hypergraph in polynomial time. An overview of these heuristics will be given after the metrics are introduced below. For a more detailed one, the reader is referred to [38].

2.1.1 Metrics for Hypergraph Partitioning

The objective function $\chi(\Pi)$ represents the communication overhead of an application when the given partition Π is applied. This communication overhead can be the cost of sending packages in a distributed environment [15, 35, 36, 18] or the amount of database tables accesses for complex queries [17, 8, 64] or another form of inter-part interaction depending on the application. There are a number of metric definitions for hypergraph

partitioning that can model different applications more accurate than others as stated earlier. Here, we will introduce the metrics implemented in *PHaraoh*.

A common metric is the *cutnet* defined as:

$$\chi(\Pi) = \sum_{n \in \mathcal{N}_{cut}} \mathsf{c}[n] \ . \tag{2.3}$$

where $\mathcal{N}_{cut}$ denotes the set of external nets that are in the cut. It gives the total cost of cut nets.

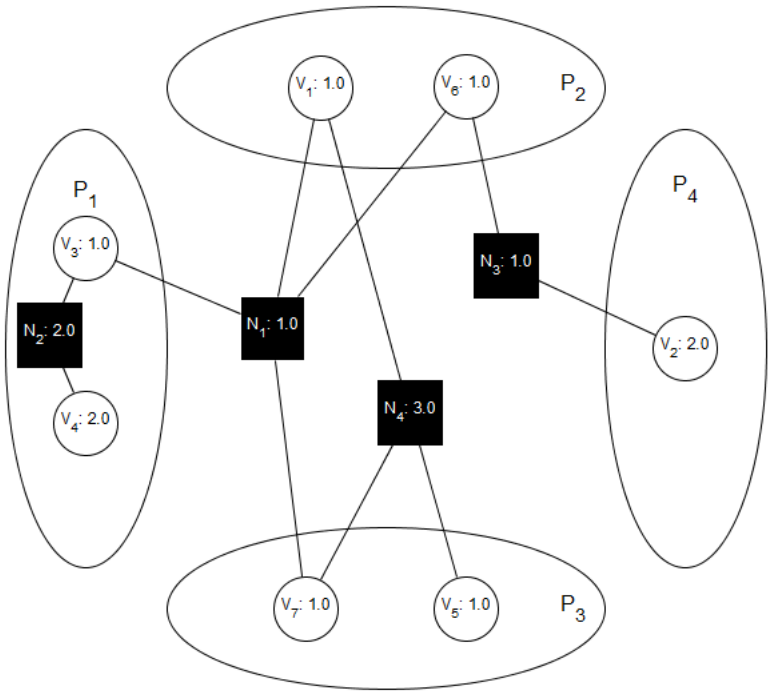
Another widely used metric in the literature is *connectivity-1* metric, referred as *total volume* (TV), since it can exactly model the total communication volume of parallel sparse matrix-vector multiplication [13]. The objective function for this metric is defined as:

$$\chi(\Pi) = \sum_{n \in \mathcal{N}} \mathsf{c}[n](\lambda_n - 1) \ . \tag{2.4}$$

For many applications, each net $n \in \mathcal{N}$ represents an item, message, information, etc. to be sent among the parts. One can assume that each such net has a single source (pin) and $(\lambda_n - 1)$ is the number of communication operations, e.g. messages, a cut net incurs. Since $\mathsf{c}[n]$ is the cost overhead of each such operation, $\mathsf{c}[n](\lambda_n - 1)$ is the total volume for this net regardless of the source part. Hence the direction of the communication is not important for this metric.

These two metrics do not use the source-target information of the communication. Thus they are well suited for undirected hypergraph models. However, such hypergraph models may not be suitable for some applications which can be modeled best with directed hypergraphs [28].

Figure 2.1: A toy undirected hypergraph with four nets and seven vertices partitioned into four parts. Three nets are in the cut. The *cutnet* metric is equal to five and the *connectivity-1* metric is equal to six.



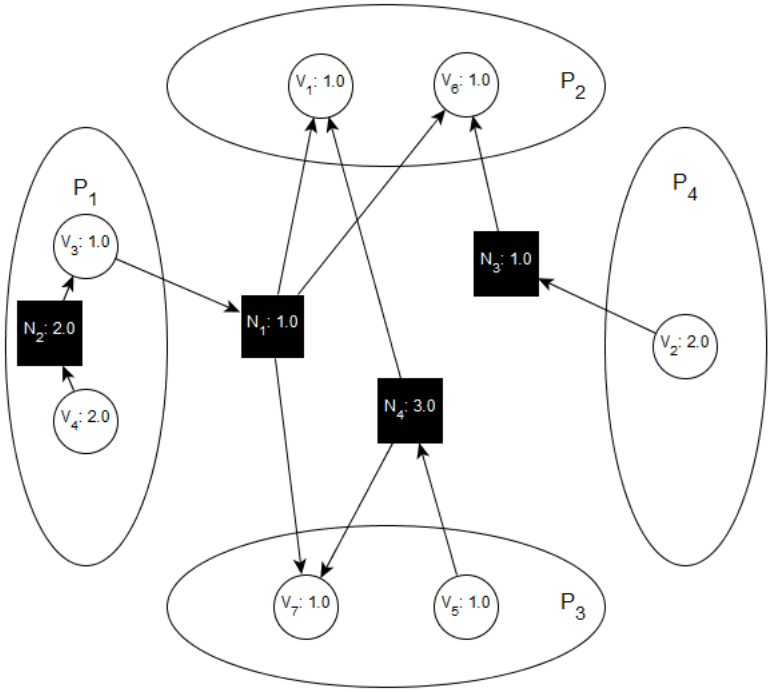
An example 4-way hypergraph partitioning of a hypergraph with 4 nets and 7 vertices is given in Figure 2.1. Since $W_{avg} = 2.25$ and $W_{max} = 3$, it satisfies the balance constraint given in (2.1) when $\varepsilon = 0.35$. One can evaluate the *cutnet* cost of the given partition by considering the cut nets which are N_1, N_3, N_4 whose costs are $1, 1$ and 3 respectively. Summing all the costs gives us the *cutnet* cost which is 5. If we want to compute *total volume* (i.e., connectivity-1), then we need to take into account not only the weight of the cut nets but also the number of parts they are connected to. N_1 has pins assigned to P_1, P_2, P_3. Based on (2.4), N_1 incurs a cost of 2. N_3 has pins assigned to P_2, P_4 which incurs 1, and finally, N_4 has pins assigned to P_2, P_3 incurring a cost of 3. Hence, the TV metric is 6 for the given partition in Figure 2.1.

Metrics for Directed Hypergraphs: A directed hypergraph is a hypergraph whose nets have source and target pins. There are various applications in the literature which are best modeled with directed hypergraphs. Various metrics have been suggested in order to model the partitioning cost for these applications [57, 9]. These metrics can be classified into two categories; volume based and message based metrics. For parallel and distributed computing, the volume-based metrics in the first category imply a practical

overhead based on the bandwidth consumption, and the message-based metrics in the second category imply costs due to the the latency of a single communication regardless of the size of the message.

In the simple directed model, which we also follow in this study, each net has a single source which corresponds to one of the pins. The remaining pins are the target/receiver pins. If a message is sent from a part k to part k' it means that a hyperedge has its source pin in part k and at least one target pin in part k'.

Figure 2.2: A toy directed hypergraph with four nets and seven vertices partitioned into four parts.



The first category, the set of volume-based metrics, contains *max sent volume* (MSV), *max received volume* (MRV) and *max sent-received volume* (MSRV) that take the source information into account and model the volume of a specific directed communication, i.e for a single part. Let $\mathsf{SV}[k]$ and $\mathsf{RV}[k]$ be the communication volume sent from and received into part k, respectively. That is

$$\mathsf{SV}[k] = \sum_{\substack{n \in \mathcal{N} \\ \mathsf{part}[\mathsf{src}[n]]=k}} \mathsf{c}[n](\lambda_n - 1) \ , \tag{2.5}$$

$$\mathsf{RV}[k] = \sum_{\substack{n \in \mathcal{N} \\ \mathsf{part}[\mathsf{src}[n]]\neq k \\ k \in \mathsf{P}[n]}} \mathsf{c}[n], \tag{2.6}$$

where $\mathsf{src}[n]$ denotes the source vertex of net n. Hence, the TV metric defined before is equal to $\sum_{k=1}^{K} \mathsf{SV}[k] = \sum_{k=1}^{K} \mathsf{RV}[k]$. Let $\mathsf{SRV}[k] = \mathsf{SV}[k] + \mathsf{RV}[k]$ be the total communication volume sent/received from/into the part k. The MSV, MRV, and MSRV metrics are defined as:

$$\mathsf{MSV} = \max_{1 \leq k \leq K} \{\mathsf{SV}[k]\} \ , \tag{2.7}$$

$$\mathsf{MRV} = \max_{1 \leq k \leq K} \{\mathsf{RV}[k]\} \ , \tag{2.8}$$

$$\mathsf{MSRV} = \max_{1 \leq k \leq K} \{\mathsf{SRV}[k]\} \ . \tag{2.9}$$

Figure 2.2 illustrates a 4-way partitioning of a directed hypergraph which is obtained from the undirected one in Figure 2.1. As before, the balance constraint is satisfied when $\varepsilon = 0.1$. In order to calculate the $\mathsf{MSV}, \mathsf{MRV}$ and MSRV, one needs to calculate the SV and RV values for each part. There is only one net, i.e. N_1, whose source pin in part P_1. The SV of P_1 is 2 since N_1 has 2 target parts and $\mathsf{c}[N_1]$ is 1. For P_2, it is 0 since there is no net whose source pin is assigned to P_2. For P_3, this value is equal to 3 since only N_4, with cost 3, has its source pin in P_3 and it has only one target part. Lastly for P_4, the SV value is 1. In short, SV values of P_1, P_2, P_3, P_4 are 2, 0, 3 and 1, respectively.

Calculating RV requires a similar approach. There is no net whose target pins are in P_1 and P_4 so their RV values are 0. All the cut nets, i.e., N_1, N_3 and N_4, have target pins in P_2 which incur an aggregated cost of 5. Lastly, although N_1 and N_4 have target pins in P_3, since N_4's source pin is also in P_3, only N_1 incur a receiving cost. Thus, the RV value of P_3 is 1. So, the RV values of P_1, P_2, P_3, P_4 are 0, 5, 1 and 0, respectively. Hence, the directed partitioning metrics are $\mathsf{MSV} = 3$, $\mathsf{MRV} = 5$ and $\mathsf{MSRV} = 5$ based on the SV and RV values computed above.

The second category, message-based metrics, contains *max sent message* (MSM), *max received message*(MRM) and *max sent-received message* (MSRM). Given a partition, let

$\mathsf{SM}[k]$ and $\mathsf{RM}[k]$ be the number of messages sent and received, respectively, by part k. That is

$$\mathsf{SM}[k] = |\{k' : \exists n \in \mathcal{N} \; s.t. \; \mathsf{part}[\mathsf{src}[n]] = k \text{ and } k' \in \mathsf{P}[n] \setminus \{k\}\}|, \tag{2.10}$$

$$\mathsf{RM}[k] = |\{k' : \exists n \in \mathcal{N} \; s.t. \; k' = \mathsf{part}[\mathsf{src}[n]] \text{ and } k \in \mathsf{P}[n] \setminus \{k'\}\}|. \tag{2.11}$$

Based on these definitions:

$$\mathsf{TM} = \sum_{1 \leq k \leq K} \mathsf{SM}[k] = \sum_{1 \leq k \leq K} \mathsf{RM}[k] \; , \tag{2.12}$$

$$\mathsf{MSM} = \max_{1 \leq k \leq K} \{\mathsf{SM}[k]\} \; , \tag{2.13}$$

$$\mathsf{MRM} = \max_{1 \leq k \leq K} \{\mathsf{RM}[k]\} \; , \tag{2.14}$$

$$\mathsf{MSRM} = \max_{1 \leq k \leq K} \{\mathsf{RM}[k] + \mathsf{SM}[k]\} \; . \tag{2.15}$$

Let us calculate the MSM, MRM and MSRM values for the partition given in Figure 2.2. We need to calculate the SM and RM values first. For this metric, the net costs are irrelevant and we just need to consider the target part number for a net. The SM of P_1 is 2 since N_1, whose source pin is in P_1, has 2 target parts. For P_2, the SM value is 0, since there is no source pin of any net connected to P_2. For P_3, it is 1 since we only need to consider N_4 and it has only one target pin assigned to a different part. Lastly, the SM value is 1 for P_4. Thus, the SM values of P_1, P_2, P_3 and P_4 are 2, 0, 1, 1, respectively. Similarly, the RM values are 0, 3, 1, 0. One should notice that total of SM values is equal to total of RM values. With the values above, the metrics are computed as $\mathsf{MSM} = 2$, $\mathsf{MRM} = 3$ and $\mathsf{MSRM} = 3$.

2.1.2 Algorithms and Tools

Since the hypergraph partitioning problem is **NP-Hard**, various approaches and heuristics which can efficiently partition a hypergraph into K parts have been proposed. The two most popular techniques in the literature are recursive bisection [10, 13, 50, 54, 63] and direct K-way partitioning [30]. Most of the tools which implement these approaches first coarsen the graph, obtain an initial solution on this smaller hypergraph, and employ

local refinement algorithms [37, 27] during uncoarsening. There exist serial, e.g., PaToH [12], hMETIS [34], Mondriaan [59], and parallel, e.g., Zoltan [22, 23], ParKway [56], tools in the literature. Although these tools have been shown to provide sufficiently good partitions that can be used in practical applications, it is not possible to evaluate their true performance with respect to the best solution since the best partitioning is usually unknown.

Besides the heuristics and practical tools designed for large-scale hypergraphs, the literature on optimal hypergraph partitioning is very limited. Knowing optimal solutions, even for much smaller hypergraphs, can be crucial for some applications such as circuit design [11]. For VLSI design, the hypergraphs are relatively small. Once an optimal solution is found, it can be repetitively used during the production stage. Optimally solving the K-way partitioning problem also enables us to evaluate the real performance of the partitioners used in practice. Hence, we can understand what they are missing and why they are missing it.

Caldwell et al. investigated the branch and bound approach for K-way hypergraph partitioning and introduced bounds for *cutnet* metric [11]. Pelt and Bisseling applied the branch and bound strategy to hypergraph bi-partitioning and also developed several bounds for *total volume* metric [49]. Moreover, speed up opportunities such as parallelization [47], reordering the branch and bound levels [46, 47] has been investigated. In addition to these, Kucar et al. has modeled the hypergraph partitioning problem as an Integer Linear Program (ILP) and solved it with integer linear programming principles [38].

2.1.3 Applications and Variations

There is a variety of applications for which modeling the problem as a hypergraph is a better fit. Although such an application can usually be modeled as a graph as well, it is shown that especially for directed representations, hypergraph models can perform better by an order of magnitude [51]. In some cases, converting the problem to a hypergraph partitioning problem and in some cases, representing the relationships among the data as a hypergraph makes the difference. In general, if the relationships among the entities are not restricted to pairwise interactions, a hypergraph, instead of a graph, is a better fit. Some of these practical cases are discussed below.

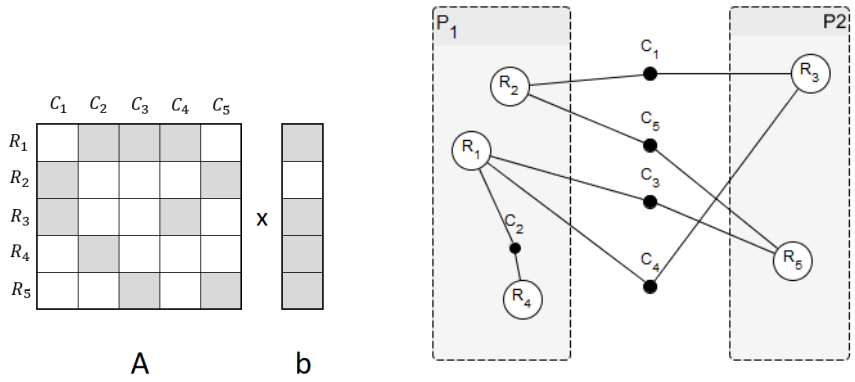


Figure 2.3: SpMV example and its column-net hypergraph modeling. White, bigger circles are the pins which stand for the rows of the matrix, and black, small circles are the nets that represent the columns.

Sparse Linear Algebra Hypergraphs can model matrices directly. There are three widely-used models to represent matrices as hypergraphs [13].

- In the row-net model, the hyperedges correspond to the rows and the vertices correspond to the rows of the matrix.
- The second model is the column-net model where the hyperedges correspond to the columns and the vertices correspond to the columns of the matrix.
- The third one, fine-grain representation, results in a special hypergraph where each vertex is connected to exactly two hyperedges. In this model, vertices correspond to nonzeros and hyperedges correspond to rows and columns.

Sparse matrix-vector multiplication (SpMV) is an extensively used kernel in many engineering applications such as solving linear systems and computing eigenvalues. A parallel and efficient implementation of SpMV can speed up all these applications in which, the SpMV is repeated multiple times with the previous iteration's result vector until a form of convergence. The parallel execution within the repeated SpMV is generally modeled as a hypergraph since the partitioning objective exactly models the communication cost of the parallel execution [13, 29, 48]. For example, let matrix **A** and vector **b** be given as in the left part of Figure 2.3. The column-net model is given on the right side of the figure. Suppose, this SpMV is performed in a distributed environment and each dot-product, i.e., the multiplication of a row with **b**, is assigned to a different processor as shown in the right side of the Figure 2.3.

At each iteration, the assigned dot products require some entries in the vector computed in the previous iteration. A required entry may not be available in the processor that the dot product is assigned to since that entry was computed by another processor. After all such entries are obtained from their owner processors, the dot product can be computed. Let's assume the rows are assigned to parts as in the right side of the Figure 2.3. As the figure shows, C_1, C_3, C_4 and C_5 are cut nets and the corresponding columns incur data transfers among the processors.

VLSI Layout Design. A circuit layout is a combination of logic gates and wires in between them. Very-large-scale integration, VLSI, is a complicated and large-scale circuit layout. An electric signal which is signaled after being processed on a component travels via the wires to other components on the circuit. This communication among different components can be very costly. Thus the related components which need each others output should be located close to each other on the circuit to reduce wire congestion. This problem is a direct application of hypergraph partitioning. Actually, this is the area that drove hypergraph partitioning until the end of 90's. The reader is referred to [3] for a comprehensive survey on the techniques that are applied in layout design. Caldwell et al. suggested a combinatorial partitioning technique in order to partition small sized circuits optimally [11].

Additionally, net distribution affects the quality of a given VLSI design. Thus Dong, Zhou, Cai and Hong [25] studied partitioning satisfying a dual partitioning constraint on both vertex weights and edge costs. They also showed that this model can be implemented in any hypergraph partitioning context.

Data/Compute Intensive Applications on Distributed Environments. When an application with multiple modules/tasks run on a distributed environment, the tasks are distributed to different processors and a communication overhead occurs among processors due to the interaction between the modules. This overhead can be directly modeled by the cut nets of a hypergraph partition [35].

In the literature, multi-constraint hypergraph partitioning has been introduced [14]. This model is also applicable to task assignment [36]. For example, a limitation on the disk space can be a constraint in addition to the balance on the workload. It is also

possible to consider a set of initially fixed vertices which means that these vertices have to reside in a fixed partition [5].

The computational structure of parallel applications may change over time which results in an uneven load assignment among partitions. To rebalance the load, repartitioning is needed and Catalyurek et al. studied this problem while considering the transfer cost of the load [15].

The problem has also been investigated in heterogeneous environments, e.g., a cluster equipped with computers/processors with varied computation power. In this scenario, a user wants to distribute the workload based on the CPU powers. Thus, the partitioning should take these into account to keep them busy [18]. Having multiple and/or hybrid partitioning objectives is another interesting model for data/compute intensive applications [16, 18].

2.2 The Branch and Bound Approach

Branch and bound is a powerful and fundamental technique that is commonly employed for solving large scale **NP-Hard** problems. It is well studied in the optimization of combinatorial problems [1, 7]. Branch and bound algorithms are able to produce one or all the exact and optimal solutions via searching all the possible solution space. In that sense, one can compare it with the brute force technique. However, the use of bounds for the objective function combined with the current best solution eliminates the unpromising branches and reduces the search space. This elimination operation is called as *pruning*.

The algorithm operates on a dynamically created search tree whose nodes represent the unexplored subspace and initially only the root exists in the tree which is the original, untouched problem. Internal nodes, which correspond to partial decisions on the problem, are generated dynamically. During the traversal of the search tree, three operations are performed; node selection, bound calculation, and branching. The sequence may vary for different implementations.

A branch is pruned if its bound, i.e. the best possible value after the corresponding partial decisions, exceeds the cost of the current best solution at hand, called as *incumbent*. Note that regardless of the later decisions, this branch for sure can not result in a better value. If the bounds are tight, they can result in fewer branches. In other words, more

and earlier pruning will result in a smaller search space exploration which yields, overall, a much faster algorithm [31].

Branch and bound algorithms offer two sources of optimization; the search strategy and bound computation. However, their performance gains differ based on the application. There are two popular search strategies for branch and bound algorithms. If the selection of the next branch to be processed is based on its lower bound value it is called as *eager strategy* [19] and follows the *Best first search* paradigm. An alternative one is called as *lazy strategy* which follows the *Depth first search* paradigm and the lower bound is calculated after the node is selected. They both have their advantages and disadvantages and the reader is referred to [19] for details. The other way of optimizing the algorithm is to develop "tighter" bounds which enable the algorithm to prune more and consequently shrink the search space.

The branch and bound technique is very popular for solving **NP-Hard** problems in the literature such as TSP [21, 52, 60] and K-SAT [39, 44]. The reader is referred to [26] for a relatively recent overview of branch and bound algorithms and applications.

Although K-way hypergraph partitioning problem is an interesting problem that can be solved by branch and bound, only a few studies have been conducted on that topic. Caldwell et al. attempted to apply this technique for partitioning the cells of electronic circuit [11]. They stated that the existing methods were not capable of generating a partitioning good enough or fails to generate one. They have developed bounds based on the cost of already assigned pins. We have investigated the similar bounds for each metric. Pelt and Bisseling investigated the bounds for bi-partitioning of a hypergraph [49]. They also introduced two additional bounds which are calculated based on the unassigned pins whose nets are partially assigned. The first bound calculates the minimum possible cut placing these unassigned pins under the balance restriction (2.1). The second one calculates the minimum possible cut taking the unassigned pins, whose nets are partially assigned to different parts, into account. In this thesis, we investigated the adaptation of these bounds for K-way partitioning.

2.2.1 Parallelization of Branch and Bound

Parallelization of branch and bound algorithms is a well studied area [6, 20, 55]. Moreover, many parallel branch and bound frameworks have been developed such as PUBB [53],

Mallba [2], and Bob++ [24].

There are two main techniques used for parallelization of branch and bound algorithms; namely low-level and high-level.

- Low-level parallelization is also referred as node based parallelism. All the studies reveal that one of the most time-consuming operation of a branch and bound algorithm is the computation of specific bounds which is amenable to the parallelization. Another parallelization option is the selection of the next node to be processed. Since these are application dependent computations, the parallelization of each algorithm has to be investigated carefully for each individual problem.

- The high-level parallelization is more interesting since it is less dependent on the application. That is why most of the research effort has been put into high-level (or tree-based) parallelism. It consists in distributing the subproblems to processors and exploring the search tree in parallel. Each processor works on a different part of the tree sharing only the incumbent term which is globally used for pruning.

Although the high-level approach seems a promising parallelism source, because of numerous reasons, it is very hard to achieve a consistent and scalable branch and bound algorithm. First, it is not known whether a branch is going to be pruned before its bounds are calculated. In other words, we do not know the size of the subproblems. Because of the irregular nature of the search tree, it is very difficult to distribute the tasks in a balanced way to the processors. Secondly, the processors have to share the incumbent term. When a better solution is found, it has to be announced to the other processors immediately. Depending on the problem, it may cause a large communication cost and a solid obstacle for scalability. Lastly and most importantly, since the traversal order of the search space is changed, it may take the same or even more time to find the best solution compared to its serial execution [4, 40, 43]. To illustrate, in the case the best solution is in the leftmost leaf, a serial version would be able to find it very fast. And the parallel version will not be able to parallelize the exploration of this path. Furthermore, due to the parallelization overhead and other performance degrading possibilities such as cache trashing, the parallel approach can take longer. All these said, the parallel approach can also yield superlinear speedups based on the location of the optimal solution(s) in the search tree.

In order to alleviate these issues, various methods are suggested and for a comprehensive survey, the reader is referred to [11]. In the master-worker paradigm, a master thread generates the tasks and distributes them among the workers. In work-stealing, there is a task pool for every processor. If one finishes all its tasks, it steals some tasks from other processors. This is more suitable when the size of the tasks are unknown beforehand which is the case for hypergraph partitioning. Work stealing based algorithms scale better since they usually keep all the threads busy [62]. However, allowing an idle thread to steal work from a busy thread, the synchronization overhead due to locks on the task pools becomes a problem during the runtime.

In this thesis, we implement both approaches and compare their performance. We explain the details of our implementations in Chapter 4 .

Chapter 3

Optimal Hypergraph Partitioning

PHaraoh can partition a hypergraph into K parts optimally based on various metrics. It adopts the branch and bound strategy for finding the best solution via investigating each possible solution. Since the search space is very large even for a small hypergraph, we need to prune the branches as early as possible. For this purpose we developed bounds and techniques for the metrics introduced in Section 2.1.1.

In this section, the branch and bound technique is discussed for the specific problem at hand and then, the simple bounds that we developed for each metric will be introduced.

3.1 Branch and Bound

The search space of the optimal K-way partitioning can be represented as an N-ary tree whose nodes correspond to a decision given for an item (vertex or net). Hence, N depends on the assignment strategy which will be discussed later. Let $(x_1, x_2, \ldots, x_d)$ denote the assignments of the first d items where $x_i \in \{1, 2, \ldots, N\}$ for $1 \leq i \leq d$ is the decision. Hence, each such d-tuple corresponds to a path from the root to an internal node in our tree. We considered two branching strategies within the scope of this thesis and adopted vertex-based branching because of the reasons discussed below.

3.1.1 Vertex-based branching

In vertex-based branching $N = K$ and the height of the K-ary search tree is $|\mathcal{V}|$, i.e., the number of vertices/pins. Each vertex corresponds to a level in the tree and assigned to one of parts after the branch arrives to its level in the tree. After a decision is taken, it is

tested whether the bounds do not exceed the current best cost and the balance constraint is satisfied. If the test fails, the branch is pruned. At any given time,we have a partial partition Π_d where the first d vertices have been assigned to the parts and the rest are undecided. This approach enables us to traverse all possible $K^{|\mathcal{V}|}$ partitionings easily.

3.1.2 Net-based branching

Net-based branching is employed by Pelt and Bisselling [49] for hypergraph bipartitioning with same vertex weights and net costs. To their approach, the decision on the assignment of a net is made in three ways: $\mathcal{V}_1$, $\mathcal{V}_2$ and *cut*. In fact, an assignment of a net n to $\mathcal{V}_k$ implies the assignment of all its pins to $\mathcal{V}_k$. With these decisions, they produce the vertex partitioning obeying the net-based decisions given during the search. There are two possible cases for a vertex to be assigned when finding corresponding vertex assignment. A vertex;

- is assigned to $\mathcal{V}_k$, $k \in \{1, 2\}$, if at least one of its nets is assigned to $\mathcal{V}_k$,
- is assigned to any part if all of its nets are assigned to *cut*.

Since the number of nets in a real-life hypergraphs is usually much smaller than the number of vertices, this approach which assigns multiple pins at once is a promising approach in terms of efficiency. However, having $K = 2$ and unit weights, one can produce a feasible partitioning obeying the net-based decisions. On the other hand, it is not practical to produce a feasible solution when $K > 2$ except for the *cutnet* metric when the hypergraph has unit weight vertices.

For *cutnet*. As defined before, the *cutnet* metric is only interested in the number of nets that cause communication among parts. Moreover, this metric does not discriminate the nets connected to 2 parts or 5 parts. That is why, for this metric, there are $N = K+1$ options for a net to be assigned; $\{1, 2, ..., K, cut\}$. Furthermore, since the vertices have unit weights the pins of the cut nets can always be placed to the parts. The only thing we always need to check is whether the internal nets do not violate the load balancing constraint which is a simple task.

For other metrics. The net-based branching fails to produce a valid solution for other metrics efficiently. Because, for these metrics, the nets that are in the cut can incur

different costs based on the parts they are connected to. For example, for TV, a net which has pins in part 1 and 2 incurs different cost than a net that has pins in parts $1, 2$ and 3. That is why the decision to be *on the cut* is not sufficient. We have to decide on the parts as well. Hence, the net-based branching introduces $N = 2^K - 1$ options for each node. Although it enlarges the search space significantly, this is not the only reason why this strategy fails to produce an efficient solution. After each decision, we have to both calculate the bounds and check the satisfiability of the balance constraint. It is impractical to check the latter efficiently since it actually is also an **NP-Complete** problem. The problem is whether a given set of net assignment can produce a feasible partition.

Giving a formal proof of NP-Completeness of this problem is out of the scope of this thesis but an argumentative proof is provided. Assume some of the nets are assigned to cut along with part ids. The pins of these nets have to be assigned in a way that each decided part has at least one pin. Let there be a polynomial time algorithm L that can find a feasible solution to that problem. This algorithm can also solve the generalized scheduling problem which is shown to be **NP-Complete** [58].

Definition 2 (Generalized Scheduling Problem) *Given a set of n jobs, the time t_i required to process job J_i, number of processors k and a time limit T, find a scheduling that all the jobs are processed within the time limit by at most k processors.*

We can reduce this problem in polynomial time to our problem such that the jobs and processors correspond to the vertices and the parts, respectively. The time t_i corresponds to the weight of the i'th vertex. We can create our hypergraph by connecting all these vertices with a single net. Additionally, each job is allowed to be scheduled in any processor; thus, we can assume the decision for this net is the whole processor set. The value of ε can be obtained from $T = (1 + \varepsilon) \sum_{i=1}^{n} t_i/k$. Running algorithm L on this hypergraph will tell us if a feasible partitioning exists which also implies the existence of a feasible schedule for the original problem. However, since the generalized scheduling problem is proven to be **NP-Complete**, there is not known an algorithm L. On top of that, it is possible to verify whether a given partition satisfies the balance constraint, i.e., it is feasible or not. These prove that our problem is also an **NP-Complete** problem.

To conclude, net-based branching can perform better than vertex-based branching when partitioning is done for $K = 2$ parts or considering the *cutnet* metric with unit-

weight vertices. However, since we want *PHaraoh* to work on any metric and any K, we adopt the vertex-based branching approach as shown in Algorithm 1.

3.1.3 Algorithm

As the algorithm presents, at any given time, there is an incumbent term denoted by $\Pi_{incumbent}$ which can be obtained from the $bestParts$ variable maintained during the execution. In the algorithm, the *stack* is recruited for search space traversal. It has the items' ids that are in the path to the current node from the root.

Couple of methods are used in Algorithm 1. Metric related methods are going to be explained under Section 3.3 and FindNextPart method that gives a part id to assign will be discussed at Section 3.2.

Assuming the root is at level 0, to reach a node in i'th level, i decisions must have been taken for i items which defines a partial partitioning. If the cost of a partial partitioning with respect to the bounds of the selected metric exceeds the minimum cost at hand for a partitioning, the subtree of the corresponding node is pruned.

Algorithm 1: BRANCH AND BOUND

input : METRIC: contains bounds, auxiliary data structures to compute the incurred cost of a new assignment.

input : $\mathcal{H}$: hypergraph data, K: number of parts, W_{total}: total vertex weight, ε part weight error tolerance

output: $bestParts$: element at each index i determines the assigned part id of the vertex with id i

```
1  W_avg ← W_total/K;
2  bestCost ← ∞;
3  foreach i ∈ {1, ··· , |V|} do currentParts[i] ← −1;
4  METRIC.INITIALIZE(H);
   // all internal data of metric is initialized.
5  STACK← ∅;
6  STACK.PUSH(0);
7  while ¬STACK.ISEMPTY() do
8      v ← STACK.TOP();
9      nextPart ← FINDNEXTPART(Π_v, v);
10     if nextPart ≠ −1 then ;          // if there are more parts to assign
11
12         if W_nextPart + w[v] ≤ W_avg(1 + ε) then
13             W_nextPart ← W_nextPart + w[v];
14             currentParts[v] ← nextPart;
15             if METRIC.CALCULATECOST(currentParts, v) +
                METRIC.CURRENTCOST < bestCost then
16                 METRIC.UPDATEBOUNDSADD(currentParts, v);
17                 if v = |V| then
18                     bestParts ← currentParts;
19                     W_nextPart ← W_nextPart − w[v];
20                     METRIC.UPDATEBOUNDSREMOVE(currentParts, v);
21                 else
22                     STACK.PUSH(v + 1);
23     else
24         currentParts[v] ← −1;
25         W_currentParts[v−1] ← W_currentParts[v−1] − w[v − 1];
26         STACK.POP();
27         METRIC.UPDATEBOUNDSREMOVE(currentParts, v − 1);
28 return bestParts
```

3.2 Symmetry based Task Elimination

In Algorithm 1, the function FindNextPart chooses the next branch to follow. Some of these branches are equivalent in terms of the objective function. Hence, a careful path selection mechanism is required. For instance, with $K = 2$, one can exchange the part ids and this will not change anything on the objective function value.

We investigate every possible solution traversing every tree node that have costs not exceeding the current optimal one. Since we have K options for any vertex, there are $K^{|\mathcal{V}|}$ leaves, and hence the same number of possible paths, in the search tree. For instance, when $|\mathcal{V}| = 3$ and $K = 2$, the possible partitionings are;

$$\{(0,0,0),(0,0,1),(0,1,0),(1,0,0),(0,1,1),(1,1,0),(1,0,1),(1,1,1)\}.$$

However, since the parts are identical, some of these partitions are equivalent. Only one partition from each equivalence class is sufficient to find the optimal one. For instance,

$$(0,0,0) \equiv (1,1,1),$$
$$(0,0,1) \equiv (1,1,0),$$
$$(0,1,0) \equiv (1,0,1),$$
$$(1,0,0) \equiv (0,1,1).$$

This observation leads us to eliminate a significant portion of possible solutions. In *PHaraoh*, after a solution is investigated, its symmetrical paths are not processed further.

Let d be the number of decisions given to visit a leaf node in the search tree. The *symmetry-based elimination* only traverses a path with the decisions $(x_1, x_2, ..., x_d)$ if and only if for all $1 \leq i \leq d$

$$x_i \leq max(\{x_j : j \leq i\}) + 1. \tag{3.1}$$

The given equation is also correct where the max of an empty set is 0 which is the case for first cells decision. So the unique partitions are

$$\{000, 001, 010, 011\}.$$

We consider a partitioning, the sequence of decisions for all vertices, as a word in the language

$$L = \{a_1 a_2 \dots a_{d+1} \in \{0,1,\dots,K-1\}^d : a_i \leq \texttt{max}(\{a_j : j < i\}) + 1\}$$

with K symbols and length d words. Moreira and Reis [45] show that for this language L,

$$|L| = \sum_{i=1}^{K} S(d,i)$$

where $S(d,i)$ are the Stirling numbers of the second kind. Open forms of the right hand side can be found in [45] for small K and d values. For instance, for $K = 4$ and word wÄśth length c, the number of tasks to be processed after d steps where $c > d$ is equal to

$$\frac{1}{24}4^d + \frac{1}{4}2^d + \frac{1}{3}.$$

Moreira and Reis [45] developed the formulas for a language representing partitions of a set of $N_d = \{1, ..., d\}$ in less or equal to K parts considering equation (3.1). So, a closed form for the number of partitions, $\rho_K(d)$, is given by

$$\rho_K(d) = \frac{c^d}{c!} + \sum_{i=3}^{c} \sum_{j}^{i-1} S^j(d,i), \quad c > 2 \tag{3.2}$$

where $S^j(d,i)$ denotes the summation of jth term in the summation of Stirling number $S(d,i)$, i.e,

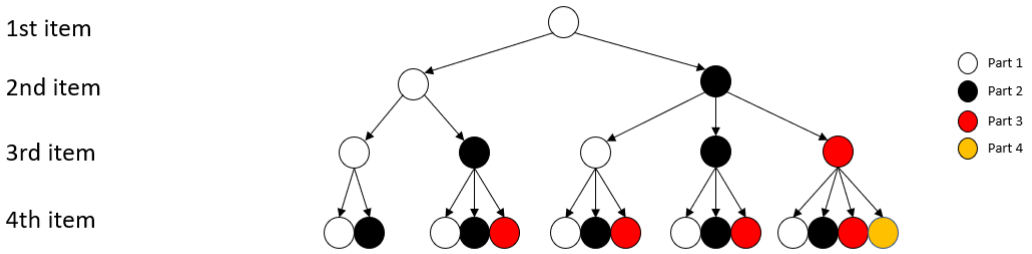
$$S^j(d,i) = \frac{1}{i!}(-1)^j \binom{i}{j} (i-j)^n \tag{3.3}$$

Open forms of the right hand side for small K and d values can be found in [45].

We applied the same logic in our problem to find the number of unique solutions. To illustrate the number of eliminated partitions, the case where $K = 4$ and $|\mathcal{V}| = 10$ has $K^{|\mathcal{V}|} = 1480576$ available partitions. However, to the formula given, only 43947 solution is unique and will exist in our search space. In other words, we have eliminated approximately %99.98 of the possible partitions thanks to symmetry. Figure 3.1 shows an exploration of a tree until the fourth level after eliminating symmetric paths.

The exploration process starts with picking the next unprocessed child of the tree. In our case, this is the next part that a vertex is assigned to. The pseudo code of FindNextPart is given below.

Figure 3.1: An exploration of the search space where the part number $K > 4$ and vertex count $|\mathcal{V}| = 4$. A path from root to a node shows a partial partition from first to that level. Node color shows an assignment decision for that vertex.



Algorithm 2: FINDNEXTPART

input : $parts[\cdot]$: the array of assigned part ids for vertices. If a vertex has not been assigned to any part, its initial value is -1. For these vertices, incrementing the value means assigning it to part 0.
v: the vertex id whose next part id will be computed

output: the part id of v is set to a value in $\{0, \cdots, K-1\}$ for v. Returning -1 implies that all the parts are tried.

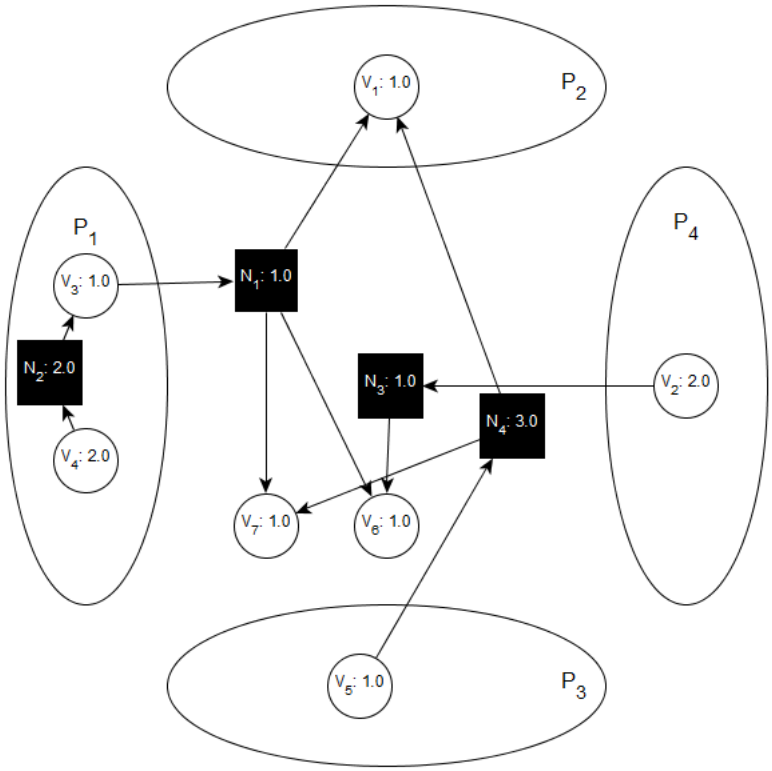
```
1 max ← 0;
2 for i from 1 to v − 1 do
3     if parts[i] > max then
4         max ← parts[i]
5 if parts[v] < MIN(max + 1, K − 1) then
6     return parts[v] + 1;
7 return −1
```

3.3 Metrics and bounds

The branch and bound strategy aims to prune the branches as early as possible utilizing the information at hand. As explained earlier, at each node in our search tree, we have a partial partition Π_d where the decisions have been made for the first d items. We can calculate the possible minimum cost of a given Π_d after the rest of the vertices are assigned to a part. At any node in our search tree, bounds can be calculated considering both assigned vertices and "partially" assigned vertices.

Figure 3.2 is an example of a partially assigned hypergraph. In our scenario, V_1, V_2, V_3, V_4 and V_5 are assigned vertices to a part. These assignments cause some nets to be cut. We can compute the partial costs incurred for these nets. Moreover, we can estimate the possible additional minimum cost which will be incurred by the assignment of the currently

Figure 3.2: Partially assigned directed hypergraph



unassigned nodes. V_6 is called partially assigned since it is not assigned to a part yet but has neighbour vertices that are assigned to a part. Furthermore, it is partially assigned to P_4 through N_3, P_2 and P_1 through N_1. V_7 is also partially assigned to P_3 through N_4, P_2 through both N_1 and N_4, P_1 through N_1.

We can calculate the minimum cost that can incur after all these unassigned vertices assigned to a part. For instance, since N_1 and N_3 do not currently share a part in Figure 3.2, for the total volume metric, there will be at least 10 cost incurred due to V_6. Hence, when partially assigned and assigned vertices considered separately, we can compute an aggregated bound.

Algorithm 1 calls CalculateCost, UpdateBoundsAdd, Initialize and UpdateBoundsRemove methods which perform based on the metric we are interested in. Each metric requires different implementation of these methods. Initialize method initializes all the helper data structures used to accelerate bound calculation. CalculateCost method calculates the additional cost caused by a newly assigned vertex. UpdateBoundsAdd method updates internal data and relationships for newly assigned vertices, and UpdateBoundsRemove rollbacks the effect of assigning a vertex. Al-

though each metric requires a different implementation, the difference between each metric is small. In this section, we are going to discuss the algorithms for distinct metric groups.

3.3.1 Bounds for Assigned Vertices

A decision on a vertex may directly result in additional costs due to making its nets to be cut. Here, we will investigate bound developments considering this case for each metric.

3.3.1.1 Cutnet

The assignment decision of vertex v increases the total cost by the cost of the v's nets that has pins in more than one parts. If there is a net of v which has pins in exactly another part, this new decision causes that net to be cut. So the net's cost is added to the total cost. For example in Figure 3.2, the next decision will be taken for vertex V_6 which has two nets. Net N_1 is already a cut net since it has pins in both P_1 and P_2. That's why the decision on V_6 will not affect the cost via N_1. However, N_3 has its pins only in P_4. If V_6 is assigned to a part except P_4 the cost will be increased by one. The cost calculation for *cutnet* is given in Algorithm 3.

Algorithm 3: CALCULATECOST-CUTNET

input : *parts*[·]: the array of assigned part ids for vertices. If a vertex has not been assigned to any part, its initial value is -1. For these vertices, incrementing the value means assigning it to part 0.
v: the vertex id whose next part id will be computed

output: *additionalCost*: the cost that is incurred by the assignment of vertex v

```
1 additionalCost ← 0;
2 foreach net n ∈ nets[v] do
3     if λ_n = 1 and parts[v] ∉ P[n] then
4         additionalCost = additionalCost + c[n];
5 return additionalCost
```

The methods UPDATEBOUNDSADD and UPDATEBOUNDSREMOVE are straightforward and they directly modify the P[n] which denotes the parts that n is connected to where $n \in$ nets[v].

3.3.1.2 Volume Based Metrics

Total Volume This one is the most used objective function in literature as we discussed earlier. The difference between *cutnet* and *TV* is that the latter takes the number of different parts a net is connected to into account. Thus only the if condition differs from Algorithm 3. We increase the cost if $\lambda_n > 1$ instead of $\lambda_n = 1$. The rest is exactly the same including the UpdateBoundsAdd and UpdateBoundsRemove methods.

Max Sent Volume This metric requires the hypergraph to be directed. If an hyperedge is directed, it means that it has a source and targets. In other words, a newly assigned pin can be the source of the net or one of its targets. In the current version of *PHaraoh*, we only have bound calculation for the nets whose source pin is assigned. For this metric, it is needed to be kept the track of the parts that this nets has target pins at. Using this knowledge we can calculate the incurred cost by a newly assigned vertex. The Algorithm 4 gives the pseudo-code of *MSM* metric. It also is an example of a max-volume algorithm. Of course, the nuances among these metrics will be explained.

In this context, $P_{send}[n]$ denotes the parts that net n has target pins. UpdateBoundsAdd and UpdateBoundsRemove methods update the $P_{send}[n]$ where $n \in \mathsf{nets}[v]$ and $\mathsf{SV}[k]$ where $0 \leq k \leq K$.

Max Received Volume This operates using RV and $P_{receive}[n]$ instead of SV and $P_{send}[n]$ in Algorithm 4. All the algorithms are the same with a subtle difference. We have to update the cost if needed every time when the new assigned pin is the source of the net and its other pins are at the target part.

Max Sent-Received Volume We can obtain this metrics bound exactly merging the code for *MSV* and *MRV*.

3.3.1.3 Message Based Metrics

Total Message It is very similar to max based algorithms illustrated with Algorithm 4. The difference is that it is concerned about the message count instead of the cost of the net that is assigned to cut. Thus it only considers the interaction between the parts. It keeps the communication matrix $ComTM$ internally where the rows are sending and columns are the receiving part.

Algorithm 4: CalculateCost-MaxSendVolume

input : $parts[\cdot]$: the array of assigned part ids for vertices. If a vertex has not been assigned to any part, its initial value is -1. For these vertices, incrementing the value means assigning it to part 0.
v: the vertex id whose next part id will be computed

output: $additionalCost$: the cost that is incurred by the assignment of vertex v

```
// currentCost:  current cost of the partial partition before
// taking v into account.  It is an internal data
```

```
1  cost ← currentCost;
2  TempSV ← SV;
3  foreach net n ∈ nets[v] do
4      if parts[src[n]] = −1 then
5          return cost;
6      if parts[src[n]] ≠ parts[v] then
7          if parts[v] ∉ P_send[n] then
8              TempSV[parts[src[n]]] ← TempSV[parts[src[n]]] + c[n];
9              cost ← max(cost, TempSV[parts[src[n]]]);
10     else if src[n] = v then
11         TempUpdatedParts ← ∅;
12         foreach pin p ∈pins[n] do
13             if parts[p] ∉ P_send[n] AND parts[p] ∉ TempUpdatedParts then
14                 TempSV[parts[src[n]]] ← TempSV[parts[src[n]]] + c[n];
15                 TempUpdatedParts ← TempUpdatedParts ∪ parts[p];
16 cost ← max(cost, TempSV[parts[v]]);
17 additionalCost ← cost − currentCost;
18 return cost − currentCost;
```

UpdateBoundsAdd and UpdateBoundsRemove methods just update the $ComTM$ matrix with the new assignment.

Max Received Message The bound for this one has an advantage. We can just search for the previously max receiving part and if the new assignment causes a new message reception by that part, then cost is increased. Since a part assignment can increment any parts received message count by only 1, the max received message value can not increase more than 1. This yielded avoiding unnecessary iteration over the nets which can not affect the total cost. The algorithm is the same overall with Algortihm 5 with just the change in if conditions and its content.

UpdateBoundsAdd and UpdateBoundsRemove methods update the $ComTM$ matrix and RM with the new assignment.

Algorithm 5: CalculateCost-TotalMessage

input : $parts[\cdot]$: the array of assigned part ids for vertices. If a vertex has not been assigned to any part, its initial value is -1. For these vertices, incrementing the value means assigning it to part 0.
v: the vertex id whose next part id will be computed

output: $additionalCost$: the cost that is incurred by the assignment of vertex v

//

// `currentCost: current cost of the partial partition before taking` v `into account. It is an internal data`

```
1  additionalCost ← 0 foreach net n ∈ nets[v] do
2      if part[src[n]] = −1 then
3          return additionalCost
4      if part[src[n]] ≠ part[v] then
5          if ¬ComTM[part[src[n]]][part[v]] then
6              ComTM[part[src[n]]][part[v]] ← true;
7              additionalCost ← additionalCost + 1;
8      else if src[n] = v then
9          TempUpdatedParts ← ComTM[part[src[n]]];
10         foreach pin p ∈ pins[n] do
11             if part[p] ≠ part[v] AND TempUpdatedParts[part[p]] then
12                 TempUpdatedParts[part[p]] ← true;
13                 additionalCost ← additionalCost + 1;
14 return additionalCost;
```

Max Sent Message This is more similar to Algorithm 5 than the bound calculation for *MRM* since the contribution of an assignment might be more than 1 when the assigned vertex is the source of a net. During updates, for this metric SM needed to be updated.

Max Sent-Received Message Again for this one, if we merge two of the cost calculation of *MRM* and *MSM* metrics, we can obtain the cost calculation for this one.

3.3.2 Partially Assigned Vertex Bounds

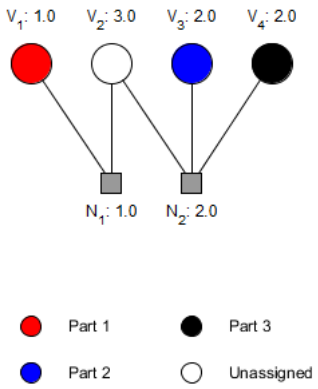
For partially assigned vertices, we only extended our work for TV metric.

3.3.2.1 Maximum Weighted Independence Set (MWIS) Bound

Conflict vertices are the pins have not been assigned yet but have neighbour pins that have been assigned to the different parts. They are indeed partially assigned vertices however, partially assigned vertices do not necessarily have neighbours assigned to different parts.

This bound is calculated considering the conflict vertices and gives the minimum cost that can be resulted after they are assigned to a part. Since this bound estimates the cost for the partially assigned pins, not the cost of the assigned pins, it does not conflict with assigned vertices bound and can be added to estimate the total cost.

Figure 3.3: Undirected hypergraph



When a conflict vertex is assigned to a part, it causes some of its nets to be cut. We need to calculate this costs minimum value by considering all of the parts to which it is partially assigned since our aim is to calculate the lower bound. For example in Figure 3.3, the V_2 is a conflict vertex since it has neighbours that are assigned to different parts. Moreover, the weight of the nets are different and it has to be taken into account as well. If $\mathcal{V}_2$ is assigned to P_2 or P_3, it causes N_1 to be cut and causes cost 1. If it is assigned to P_1, its cost would be 2 because N_2 will be cut. So we will consider 1 additional cost for $\mathcal{V}_2$ for that given partition since we are looking for the minimum cost.

During this bound calculation we picked conflict vertices that cause the maximum cost considering their minimum possible costs such that the picked vertices share no net. We can model this problem as Maximum Weighted Independent Set (MWIS). This problem can also be modeled as maximum bipartite graph matching as suggested by [49] when there are only two parts and the net weights are identical. However, this is the model of a special case of the original problem. General K-way hypergraph partitioning problem can be modeled as MWIS covering all cases.

Maximum Weighted Independent Set Problem A maximum independent set is a subset of vertices of a hypergraph satisfying a constraint based on whether it is weak or strong maximum independent set. However, regardless of its constraint, it is certain

that for given hypergraph, neither there is another vertex of $\mathcal{H}$ can be added into that set satisfying the constraint nor there can be another subset formed having larger number of vertices satisfying the property. This problem is stated among **NP-Complete** problems [33]. Before analysing the WMIS, we need to cover Independet Set Problem definitions, i.e. *Weak Maximum Independent Set* and *Strong Maximum Independent Set.*

Weak Maximum Independent Set is a subset of vertices of a hypergraph $\mathcal{H}$ that none of the net' all pins are contained. The formal definition of Weak MIS is

$$\text{a set } I_w \subset V \text{ such that } \forall e \in E : |e \cap I_w| < |e|.$$

where E is the nets in $\mathcal{H}$.

Strong Maximum Independent Set is a subset of vertices in a Hypergraph $\mathcal{H}$ that shares no net. In other words, the vertices of a Maximum Independent Set are not neighbours. The formal definition of MIS is

$$\text{a set } I_w \subset V \text{ such that } \forall e \in E : |e \cap I_w| < 1.$$

Maximum Weighted Independent Set (MWIS) is an independent set with maximum weight. In other words, the goal is maximizing the total weight of the vertices in the set instead of the number of vertices.

Our Problem In our case, we need to pick the conflict vertices with the maximum weight which prevents less number of conflict vertices from being selected for bound calculation. Because we can not pick a conflict vertex whose one of the conflict neighbours has already been selected. Hence, we modeled our problem as *Strong Maximum Weighted Independent Set* problem. To solve our problem as MWIS, we formed a subhypergraph with the conflict vertices and their nets. Then, determined new weights for vertices for this calculation by dividing the original weights $w[\cdot]$ by the number of neighbour conflict vertices. With these new determined weights we applied a greedy algorithm proposed in [32]; picked the maximum weighted vertex. Then, we removed the vertex and its neighbours from the formed subhypergraph until there is no vertex left in the subhypergraph. Algorithms analysis is given in [32]. This bound requires a lot of computation and traversal over the neighbours of all vertices. More, after each successful assignment, we

need to update the internal data structures for the conflict pins which are the neighbour of the newly assigned vertex. The vertex assignment order is determined at the beginning of the partitioning. Fortunately, this enables us to know how the neighbour data will be modified and in which order can be known after setting the order of the vertex assignment. That is why we calculated each of these internal data that accelerates finding the WMIS among conflict pins before starting partitioning.

Chapter 4

Parallel Branch and Bound for Optimal Hypergraph Partitioning

In *PHaraoh*, we implemented two parallelization approaches based on master-worker and work-stealing in order to accelerate the exploration of the branch and bound tree.

4.1 Master-Worker based Parallelization

We adopted the master-worker paradigm as in [47]. A master process divides the task into smaller tasks and then assigns them to the worker processes. In our branch and bound algorithm, a task is the exploration of the subtree of a node. Our initial task is processing the subtree of the root, i.e., the whole tree. We need to divide this task into multiple tasks.

To generate the smaller tasks, the master process explores the first d levels of the tree sequentially. The generated tasks are the subtrees whose roots are at this level. It should be noted that during this task generation, some of the branches are pruned by the bounds or by the symmetry-based elimination. The unpruned subtrees are explored by the worker processes later.

After the task generation phase, the worker processes start exploring subtrees. Since the size of a task is unknown before processing, we decided to employ dynamic scheduling for which the tasks are assigned to processors in chunks. Although there is a synchronization overhead, it is almost ensured that each processor is kept busy until there is no task left to process. However, there still is a possibility in which only one process is busy and

others are idle since the last task's subtree size is very large with respect to the others.

This approach may cause unnecessary exploration of the tree during task generation. For example in the sequential version, the left most branch may have the optimal solution and the optimal solution can be obtained at the first iteration. It may result in pruning all the other branches. However, this parallelization technique stops exploring at some depth, and the branch with optimal partition will be processed after all the tasks are generated. That is why, this technique may yield a slow down for hypergraphs that are solved very fast. This is why the value of d plays a significant role in parallelization performance. Larger d values increase the spent time on initial task generation.

4.2 Work-Stealing based Parallelization

A work-stealing based parallel algorithm is realized having multiple work pools owned by each thread. When a thread finishes all of its tasks, it acts like a thief and tries to steal the unprocessed tasks from others. In our implementation, each thread generates tasks for both itself to process later and others to steal. This property makes the approach more convenient for branch and bound algorithms. For example, suppose we have four tasks to process with four threads. Each thread acquires one task and starts processing. After some time, all of the subtrees are pruned except one. In this case 3 threads go idle and wait for the other to finish. True parallelization is not achieved in this scenario. With work stealing, the thread that processes the unpruned branch dynamically generates tasks to be stolen. Idle threads can steal some of the newly generated tasks and process them. This technique enables us to minimize the threads idle time.

There are numerous parallelization challenges of a work-stealing algorithm. First, task pool operations such as task generation or stealing, require using an efficient lock mechanism which may impact the runtime significantly. In the algorithm we implemented, there is a work pool of each thread which has public and private part. Private part is the part that can be accessed only by the owner thread and requires no use of locks. Locks are used for the operations that involve public part such as stealing. There are also acquire and release operations which manage the size of the public and private parts. These operations are performed only by the owner thread. This prevents blocking any other thief threads. In other words, when a thread adjusts its work pool sizes, thief threads can

try to steal from other working threads. Another challenge is to synchronize work unit distribution among threads, i.e. selection of the victim threads and the number of tasks to be stolen. For the first step of this problem, random work stealing scheme is employed to pick the victim thread among the working threads. For the second question, we adopted half-steal strategy which proposes stealing half of the publicly available tasks. Vu and Derbel [61] proposed using another variation of this strategy that is shown to perform better in heterogeneous distributed platforms.

The algorithm starts with assigning our initial task to a thread. After some depth d, this thread starts generating tasks until its work pool is full. In the meantime, some of these tasks are taken to public part with release operation. Other idle threads steal the tasks and continue in the same fashion. In the algorithm, we set the d value which determines the task generation speed and the pool size manually. After experiments, we decided to use a small pool size and small d value which avoid excessive task generation when there is no idle threads.

Chapter 5

Reordering

The search space of the optimal K-way partitioning can be represented as an N-ary tree as stated above. Since we have adopted vertex-based branching, we assign the vertices with id from 0 to $|V| - 1$ in increasing order.

The motivation of a branch and bound algorithm is to have tight bounds and to prune the branches as early as possible resulting in a smaller explored space. A good strategy to have early pruning is detecting potentially high cost incurring nodes and processing them early so that we can prune a larger part of the tree and avoid unnecessary work.

An ordering is a permutation of the vertex ids. Hence, the ordering determines the order of the vertices in the level of the tree. Taking a decision about a vertex puts constraints on its neighbour vertices and to the part that it is assigned to by reducing the available space. Each decision that invalidates these constraints adds costs. We can detect these potentially effective vertices who puts more constraints on others and process them first. That enables us to find high cost incurring branches earlier and prune. Another idea is finding the promising branches to contain the optimal partition earlier which enables us to find tight bounds as early as possible. For these purposes, we developed 4 different re-ordering schemes;

- *Net Cost Ordering*: sorts the vertices with respect to total costs of their nets in decreasing order.

- *Weighted Connectivity Ordering*: sorts the vertices with respect to a score which is determined based on their neighbour information in decreasing order. Each vertex contributes to its neighbour vertex's score by the nets cost which connects them.

Even if the two vertices are connected to each other through multiple nets, these vertices increase each others' score for each connecting net. With this, we can emulate an incurring volume cost due to assigning them to different parts.

- *Affected Vertices Ordering*: picks the most influential vertex whose influence is determined by the sum of its all nets multiplied by the neighbour count through that net. After that, we determined its influence over its neighbours through the net cost that connects them and we picked the most influenced one. Then it continues picking the most influenced one after adding the influence of picked one to its neighbours. In fact, this influence is the estimated value for constraints that are put by the previously assigned vertices.

- *Initial Partition Ordering*: sorts the vertices with respect to the relation obtained from the given initial partition. Initial partition is actually given as a promising solution to be the optimal one. Which is, the vertices assigned to the same part are more connected. Hence we aimed to find tighter bounds by starting the exploration from the given partition. For this purpose, we gave consecutive ids to the vertices who were assigned to same parts. We also sorted the vertices who are assigned to the same part with respect to their weights in decreasing order.

- *Initial Partition Ordering with Net Cost*: sorts the vertices as in initial partition ordering with a modification. If the vertices assigned to the same part, they are sorted as in net cost ordering.

- *Initial Partition Ordering with Weighted Connectivity*: sorts the vertices as in initial partition ordering with a modification. If the vertices assigned to the same part, they are sorted as in weighted connectivity ordering.

In *PHaraoh*, we determined the new ids of each vertex at the beginning of branch and bound algorithm. Then all the data of the hypergraph is also updated correspondingly and partition is obtained using this newly determined vertex ids. The effect of these different orderings will be discussed in Chapter 6.

Chapter 6

Results

We conducted various experiments using our framework in order to assess both the quality of a given partition and the performance improvements that we have obtained after applying the techniques discussed above. Each of these experiments will be discussed under its own subsection. We used the same set of hypergraphs for each experiment. These hypergraphs are generated using handpicked matrices which are available at [42]. We picked 20 "second" matrices which are ranging from 20x20 to 121x121 with 50 to 200 nonzeros. Then they are converted to hypergraphs using the *fine grain* and the *column-net* model. The cost of the nets and weights of the vertices are determined from normal distribution values between 10 and 20. Another type of hypergraphs are generated as both net costs and vertex weights are 10. In total we generated 80 (20 x 2 x 2) hypergraphs. We also picked the imbalance ratio ε as 0.03 for all the experiments which is a common value in the literature. A 20 minute timeout is used for each execution.

All the experiments in this chapter are performed on a single machine running on 64 bit CentOS 6.5 equipped with 512-GB RAM and a dual-socket Intel Xeon E7-4870 v2 clocked at 2.30 GHz, where each socket has 30 cores (60 in total). Each core has a 32 kB L1 and a 256 kB L2 cache, and each socket has a 30 MB L3 cache. All the codes are compiled with gcc 5.3.0.

6.1 Metrics

We partition the hypergraphs generated into 2, 4, 8 parts for each metric. Thanks to the nature of a branch and bound algorithm, although the algorithm could not complete

exploration in the time limit, we are able to obtain a partition at any time which may not be the optimal one. In the tables below, the performance of the bounds for each metric has been shown. The numbers in the tables are the number of hypergraphs on which the algorithm is completed in the given duration. We called these hypergraphs as completed hypergraphs and the rest as incompleted although it gives a partition.

Table 6.1: Number of completed tests before timeout with Cutnet. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	46	51	15	21	0	5
FG-Unit	40	7	7	0	1	0	0
FG-Normal	40	6	6	0	0	0	0
CN-Unit	40	14	19	6	10	0	3
CN-Normal	40	19	19	9	10	0	2

Table 6.2: Number of completed tests before timeout with Total Volume. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	46	51	17	24	0	8
FG-Unit	40	7	7	0	1	0	0
FG-Normal	40	6	6	0	0	0	0
CN-Unit	40	14	19	6	11	0	4
CN-Normal	40	19	19	11	12	0	4

One can conclude that hypergraphs adopting column-net models are easier to solve than the hypergraphs constructed with the fine grain model by comparing the number of completed tests. However, its main reason is that the fine grain models result in larger hypergraphs with respect to column-net. In other words, they are hypergraphs with more vertices. As a result, they have larger branch and bound trees which makes them harder to explore all in the given time limit.

We also would like to see the effect of having varying net costs and vertex weights. However, no significant pattern has been detected among unit and normal distribution version of the same hypergraphs in none of the tables.

Table 6.3: Number of completed tests before timeout with Max Sent Volume. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	45	50	16	23	3	19
FG-Unit	40	6	6	0	0	0	0
FG-Normal	40	6	6	0	0	0	0
CN-Unit	40	14	19	6	12	1	9
CN-Normal	40	19	19	10	11	2	10

Table 6.4: Number of completed tests before timeout with Max Received Volume. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	45	50	18	25	2	18
FG-Unit	40	6	6	0	0	0	0
FG-Normal	40	6	6	0	0	0	0
CN-Unit	40	14	19	7	13	2	9
CN-Normal	40	19	19	11	12	0	9

We started our algorithm with a partition having ∞ cost. It also means that any partition satisfying the balance constraint could result in a better cost. Hence, initially, none of the branches can be pruned using metric bounds. This state continues until a partition is found. In order to have a kickstart, we employed the partition that is obtained from PaToH. It increased the number of completed hypergraphs significantly. However, since PaToH works for only two metrics, *cutnet* and *total volume*, we did not have a

Table 6.5: Number of completed tests before timeout with Max Sent-Received Volume. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	45	50	18	24	2	15
FG-Unit	40	7	7	0	0	0	0
FG-Normal	40	6	6	0	0	0	0
CN-Unit	40	13	18	7	12	1	8
CN-Normal	40	19	19	11	12	1	7

partition for other metrics that *PHaraoh* supports. For these metrics, we used both of the partitions calculated the cost with that metric and picked the one which has the better cost as an initial partition. Even though the given partition was not computed for these metrics, it again helped the algorithm to complete more hypergraph partitioning. All of the message based metric bounds are the examples to observe this improvement.

Since we used PaToH results, we also had to take care of another challenge. Which is, even though PaToH tries to partition satisfying the balance constraint, it sometimes relaxes the imbalance ratio in order to have a partition with less cost. Since our hypergraph sizes are very small with respect to the sizes of the hypergraphs PaToH is useful for, this imbalance ratio relaxation resulted in a very large change for some test cases. For example, for some hypergraphs PaToH gave results with 0.1 imbalance ratio whereas 0.03 was the requested partition ratio. Moreover, this imbalance relaxation is done although there is a partition with less cost satisfying the requested imbalance ratio. However, we believe that this strategy adopted in PaToH would not yield problematic results for large hypergraphs. When using an initial partition we used its imbalance ratio if it is larger than the requested one.

Another indication of the tables is that message based metric bounds are able to finish more hypergraphs than the volume based metric bounds. This is because the bound calculations take less time with respect to volume based bounds.

Table 6.6: Number of completed tests before timeout with Total Message. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	73	78	23	33	3	18
FG-Unit	40	19	19	0	1	0	0
FG-Normal	40	19	19	1	1	0	0
CN-Unit	40	15	20	8	15	2	9
CN-Normal	40	20	20	14	16	1	9

Table 6.7: Number of completed tests before timeout with Max Sent Message. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	73	78	31	41	7	24
FG-Unit	40	19	19	4	5	0	0
FG-Normal	40	19	19	5	4	0	0
CN-Unit	40	15	20	8	16	3	12
CN-Normal	40	20	20	14	16	4	12

Table 6.8: Number of completed tests before timeout with Max Received Message. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	73	78	58	70	27	56
FG-Unit	40	19	19	14	17	3	8
FG-Normal	40	19	19	17	17	9	12
CN-Unit	40	15	20	10	18	6	18
CN-Normal	40	20	20	17	18	9	18

Table 6.9: Number of completed tests before timeout with Max Sent-Received Message. The columns "with PaToH" show the results of the experiments where the cost of the partition obtained by PaToH is given as the initial cost.

Hypergraph	**Test**	**K=2**	**K=2 with PaToH**	**K=4**	**K=4 with PaToH**	**K=8**	**K=8 with PaToH**
total	80	73	78	40	51	9	32
FG-Unit	40	19	19	7	9	0	2
FG-Normal	40	19	19	8	8	0	1
CN-Unit	40	15	20	9	17	5	15
CN-Normal	40	20	20	16	17	4	14

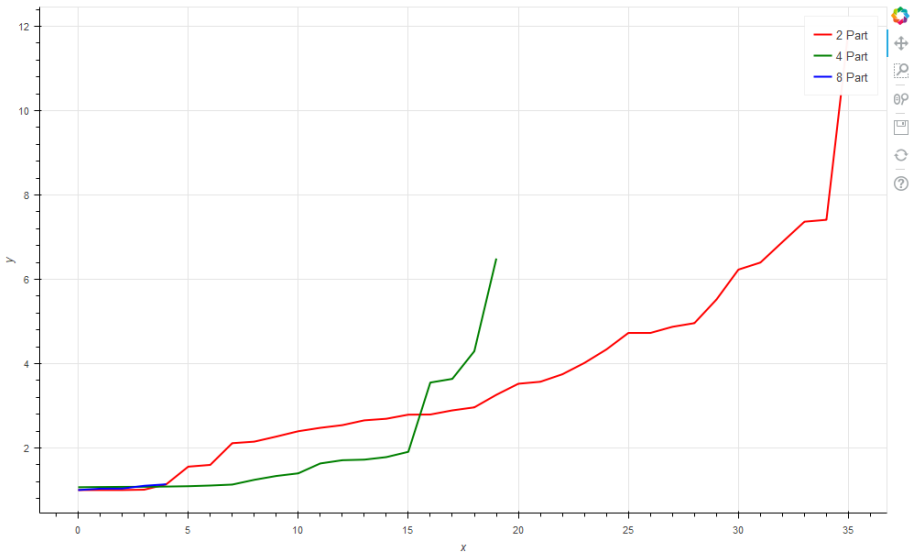
6.2 PaToH Performance Analysis

We evaluated the performance of PaToH [12] w.r.t. to the best solution. Since PaToH can partition only for *cutnet* or TV metrics we have evaluated its performance for these metrics.

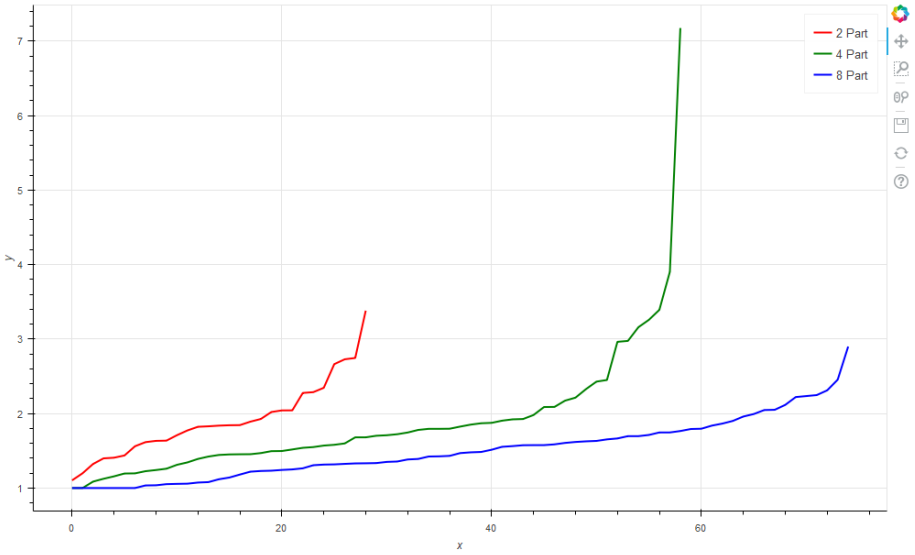
We compared the cost of the given partition with that of the partition found by *PHaraoh* in order to assess the quality of the partition found by PaToH. In Figure 6.1, the ratio between the cost of PaToH partition and optimal one is shown. This ratio is calculated as $PaToHcost/optimalcost$. The hypergraphs whose optimal costs are 0 are excluded from the figure since it distracts the graph a lot.

Figure 6.1a shows the number of hypergraphs that can be completed before timeout and their $PaToHcost/optimalcost$ ratios. Although when the part count increases the number of hypergraphs can be completed decreases, the cost ratios are significant. For example, among the completed hypergraphs for $K = 2$ (red line), there are less than 10 hypergraphs whose PaToH partition is close to its optimal value. Being close to means that the cost is no more than 2 times the optimal one's cost.

Figure 6.1b shows the same relationship among the incomplete hypergraphs in the given time duration. Since the optimal partition is not found, we used the final best partition's cost. Interestingly, it can be seen that again the ratio of the costs are significant. Although most of the tests are failed to be completed, most of their given costs are improved significantly. It can be concluded that although the *PHaraoh* as a standalone application is not a viable option for practical purposes, this tool may be used to improve

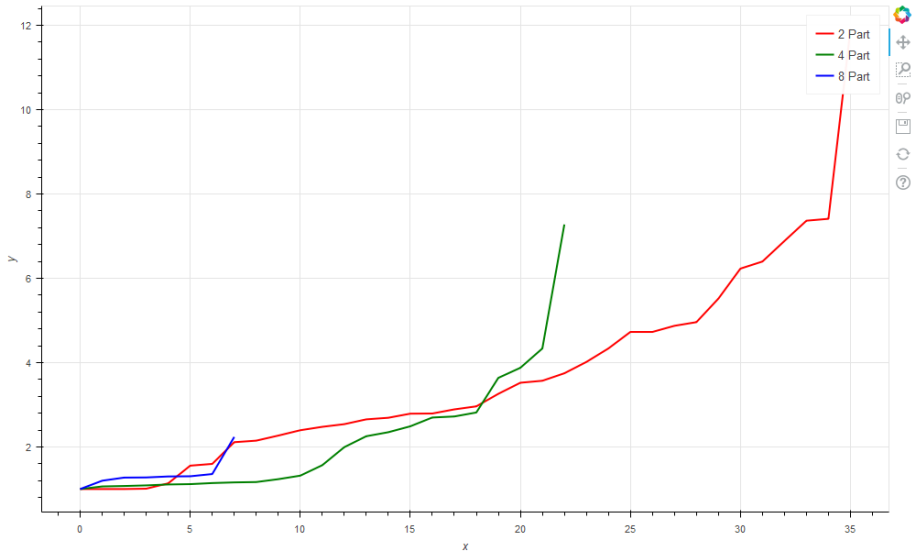


(a) Completed tests

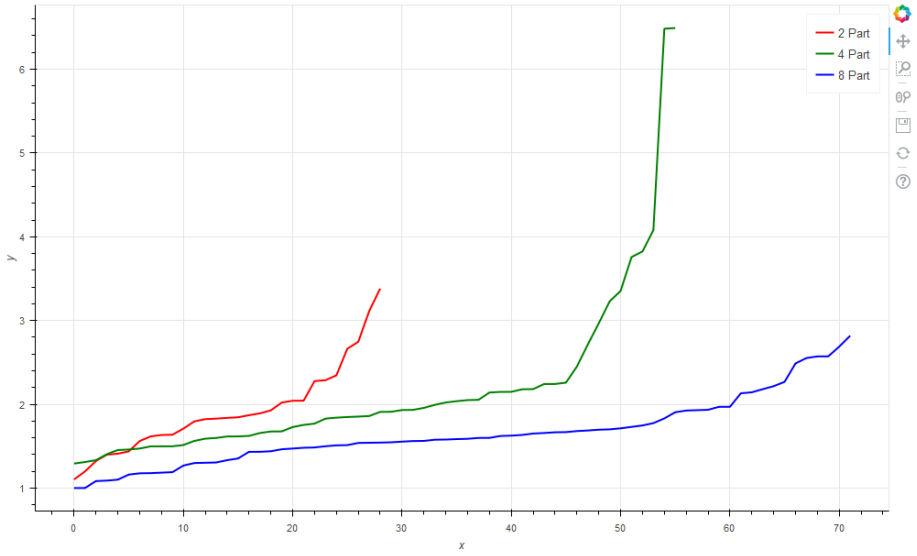


(b) Incomplete tests

Figure 6.1: Cost Ratio for *cutnet*. We calculated the ratio of the PaToH partition cost to the found best partition cost which is the optimal cost for the tests, of each test for *cutnet* metric. Figures shows the obtained ratios for 2, 4, and 8 parts. y axis is the value of $PaToHcost/optimalcost$ and x axis is for the number of tests



(a) Completed tests



(b) Incomplete tests

Figure 6.2: Cost Ratio for TV. We calculated the ratio of the PaToH partition cost to the found best partition cost which is the optimal cost for the tests, of each test for TV metric. Figures shows the obtained ratios for 2, 4, and 8 part. y axis is the value of $PaToHcost/optimalcost$ and x axis is for the number of tests

a partition provided by another tool which can give a partition fast. Figure 6.2 shows the same comparison graph for TV metric. Similar results as for *cutnet* are observed.

6.3 Parallelization

We applied the master-slave and work-stealing approaches to parallelize the branch and bound algorithm.

We investigated the performance of the parallel algorithms for only TV and TM

metrics. We partitioned into 2, 4 and 8 part using 1, 4, 8, 16 threads.

6.3.1 Master-Slave based Parallelization

For the experiments of master-slave parallelization, the depth value is selected as 6. $K = 4$ results are given in Table 6.10 and Table 6.11 and the reader can see the results for 2 and 8 parts in Appendix A.

Table 6.10: Total Message when $K = 4$. T stands for the Time. Times are given in seconds. Last four columns are the performance statistics of given parallelization for the tests which are completed by sequential version

						Cost Ratio		Stats for completed tests by sequential			
Hypergraph	**Test**	**Thread**	**Completed**	**Avg. T Completed**	**Avg. T All**	**Completed**	**All**	**Completed**	**Avg. T**	**Min T**	**Max T**
total	80	1	33	39.00	721.09	2.90	3.01	33	39.00	0.00	1092.24
		4	36	18.72	668.48	3.09	3.35	33	8.61	0.00	237.28
		8	38	16.09	637.81	3.28	3.51	33	5.50	0.01	151.88
		16	40	27.66	613.99	3.17	3.49	33	2.81	0.01	70.49
FG-Unit	20	1	1	1.39	1140.07	5.50	3.25	1	1.39	1.39	1.39
		4	2	1.18	1080.20	5.25	3.78	1	1.46	1.46	1.46
		8	3	17.89	1022.95	5.33	4.03	1	1.80	1.80	1.80
		16	3	8.79	1021.60	5.33	4.08	1	1.84	1.84	1.84
FG-Normal	20	1	1	11.57	1140.58	5.50	3.37	1	11.57	11.57	11.57
		4	2	2.21	1080.29	6.00	4.11	1	3.29	3.29	3.29
		8	3	18.45	1023.05	6.00	4.38	1	3.58	3.58	3.58
		16	3	11.83	1022.12	6.00	4.35	1	3.89	3.89	3.89
CN-Unit	20	1	15	3.55	302.66	2.66	2.65	15	3.55	0.00	21.53
		4	16	24.86	259.92	2.74	2.73	15	0.66	0.00	3.95
		8	16	20.80	256.69	2.73	2.72	15	0.44	0.01	2.37
		16	17	34.97	209.74	2.65	2.70	15	0.49	0.02	3.84
CN-Normal	20	1	16	76.29	301.04	2.81	2.77	16	76.29	0.01	1092.24
		4	16	16.85	253.52	2.81	2.77	16	16.85	0.00	237.28
		8	16	10.60	248.53	2.93	2.91	16	10.60	0.01	151.88
		16	17	26.46	202.50	2.80	2.84	16	4.98	0.01	70.49

Table 6.11: Total Volume when $K = 4$. T stands for the Time. Times are given in seconds. Last four columns are the performance statistics of given parallelization for the tests which are completed by sequential version

						Cost Ratio		Stats for completed tests by sequential			
Hypergraph	**Test**	**Thread**	**Completed**	**Avg. T Completed**	**Avg. T All**	**Completed**	**All**	**Completed**	**Avg. T**	**Min T**	**Max T**
total	80	1	24	74.46	862.34	2.42	2.25	24	74.46	0.00	1068.86
		4	28	108.69	818.29	2.74	2.46	24	23.22	0.01	278.24
		8	31	132.17	786.56	2.94	2.44	24	12.22	0.03	179.12
		16	31	89.05	769.86	2.94	2.49	24	7.66	0.08	106.31
FG-Unit	20	1	1	225.18	1151.26	7.27	2.28	1	225.18	225.18	225.18
		4	1	118.47	1146.25	7.27	2.31	1	118.47	118.47	118.47
		8	1	31.91	1142.02	7.27	2.32	1	31.91	31.91	31.91
		16	1	25.16	1141.81	7.27	2.35	1	25.16	25.16	25.16
FG-Normal	20	1	0	0.00	1200.00	0.00	2.06	0	-	-	-
		4	1	255.25	1153.11	8.25	2.75	0	-	-	-
		8	1	56.17	1143.41	8.25	2.65	0	-	-	-
		16	1	37.24	1142.37	8.25	2.80	0	-	-	-
CN-Unit	20	1	11	18.63	550.25	2.32	2.29	11	18.63	0.00	89.58
		4	14	164.30	475.14	2.56	2.33	11	6.30	0.02	29.46
		8	14	100.50	430.52	2.56	2.34	11	3.11	0.03	12.60
		16	14	77.40	414.40	2.56	2.35	11	2.05	0.09	7.02
CN-Normal	20	1	12	113.08	547.85	2.11	2.35	12	113.08	0.01	1068.86
		4	12	30.79	498.67	2.11	2.43	12	30.79	0.01	278.24
		8	15	173.47	430.28	2.66	2.46	12	18.92	0.04	179.12
		16	15	107.64	380.88	2.66	2.46	12	11.35	0.08	106.31

From the tables, it can be seen that having parallelization facilitate the algorithm to complete more tests in the given time span. For this purpose, statistics for the hypergraphs that are completed by the sequential implementation part of the figures, the last 4 columns should be analyzed. For some test cases, the parallelization provides linear speedups, which means increasing the thread count affects the runtime with the same ratio especially for the tests with TV metric. However, for some cases which are solvable fast with a single thread, even slowing down in the runtime is observed.

Table 6.12: 16 Thread Performance when $K = 2$. Execution times are in seconds

				Total Volume				Total Message			
Matrix	**Row**	**Column**	**NNZ**	**FG-Normal**	**FG-Unit**	**CN-Normal**	**CN-Unit**	**FG-Normal**	**FG-Unit**	**CN-Normal**	**CN-Unit**
lpi_woodinfe	35	89	140	1200.21	1200.20	0.04	0.04	1200.00	1200.00	0.00	0.00
GD01_c	33	33	135	299.42	88.89	0.02	0.02	0.00	0.00	0.00	0.00
ibm32	32	32	126	1200.12	1200.12	0.15	0.15	0.00	0.00	0.00	0.00
Hamrle1	32	32	98	5.55	6.64	0.04	0.03	0.00	0.00	0.00	0.00
football	35	35	118	1200.14	1200.14	0.06	0.06	0.00	0.00	0.00	0.00
pores_1	30	30	180	1200.17	1200.17	0.02	0.01	0.01	0.00	0.00	0.00
Trec7	11	36	147	1200.14	295.41	0.00	0.00	0.00	0.00	0.00	0.00
lp_afiro	27	51	102	1200.11	1200.11	0.05	0.04	0.60	0.44	0.00	0.00
Ragusa16	24	24	81	191.61	185.18	0.02	0.02	0.00	0.00	0.00	0.00
GD98_b	121	121	207	1200.70	1200.54	1200.17	1200.17	9.20	9.84	8.68	7.63
karate	34	34	78	0.64	0.18	0.03	0.03	0.00	0.00	0.00	0.00
p0033	15	48	113	66.28	16.42	0.01	0.01	0.02	0.02	0.00	0.00
can_24	24	24	92	1185.93	574.50	0.02	0.02	0.00	0.00	0.00	0.00
bcspwr01	39	39	85	0.09	0.10	0.02	0.02	0.00	0.00	0.00	0.00
GD95_b	73	73	96	0.86	227.77	1.62	1.10	0.02	0.02	2.39	1.06
GlossGT	72	72	122	1200.22	1200.23	59.61	31.41	0.01	0.01	0.01	0.01
Trefethen_20	20	20	89	1200.92	1200.91	0.01	0.01	0.00	0.00	0.00	0.00
d_ss	53	53	149	1200.20	1200.19	0.09	0.07	0.00	0.00	0.00	0.00
n3c4-b2	20	15	60	111.73	67.52	0.01	0.01	0.00	0.00	0.00	0.00
Trefethen_20b	19	19	83	577.52	1200.06	0.01	0.01	0.00	0.00	0.00	0.00

Table 6.12 shows the execution time of the partitioning for each hypergraph with 16 threads. You can also see the properties of each matrices from which hypergraphs are built such as the number of columns and rows. This table shows that not only the size of the hypergraph but also its internal properties plays a crucial role on execution time. For example, even though GD01_c and ibm32 matrices have similar columns, rows, and none zeros, there are huge execution time differences especially in the fine grain models.

Figure 6.3 shows the incumbent term update fashion for some of the selected test cases. These hypergraphs are the ones that *PHaraoh* gives the best cost ratios with respect to PaToH for partitioning into 4 parts with *total volume* using 8 threads. The interesting finding is that, generally, before the half of the runtime of a test, very close partitions to the optimal one is already found. In the rest of the search, the improvement of the cost is much smaller then the first half. More, in the first 10% of the runtime, a partition whose cost is less than or equal to the half of the initial partition cost is obtained. Hence we

can conclude that although an optimal partition can not be found in the given duration, *PHaraoh* can still provide a significantly better partition.

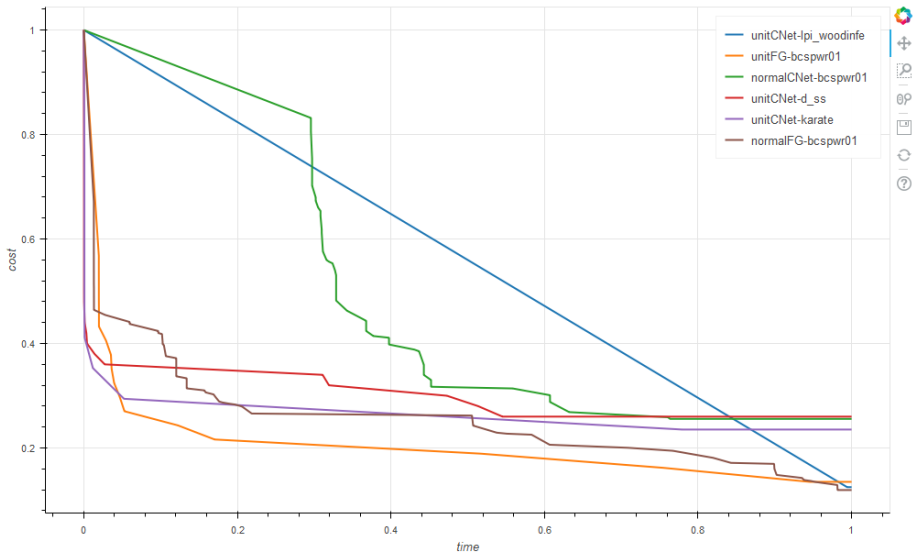


Figure 6.3: Incumbent term update times for various selected tests when K=8 and 8 threads with *total volume* metric. y axis is for the cost ratio of the updated incumbent term to the cost of PaToH partition and x axis is for the time for each update. These times are normalized by each test cases overall runtime.

6.3.2 Work-Stealing based Parallelization

We compared the performances of the two parallelization techniques we implemeted with each other, i.e. work-stealing and master-slave approaches. For the experiments, we picked 10 for the pool size and 6 for the depth size which determine the frequency of the task generation.

We expected to see more consistent speed ups and more completed tasks with work-stealing approach than the master-slave. The reason for that, as discussed, the former avoids having idle threads better and utilizes all the threads. Although for some of the cases, e.g. 2 parts and 4 parts rows in Table 6.13, work-stealing improved the runtime of the common solved matrices and is about four times faster on average. However, these results do not comply with the results of the Table 6.14 and work-stealing variant can not complete some of the tests that are completed by the master-slave algorithm. We believe that the main reason for that is the work-stealing approach changes the order of the explored branches considerably. Additionally, although there are improvements, to some extend, for 2 parts and 4 parts tests in Table 6.13 and Table 6.14, work stealing has

a substantial negative impact on the runtime when the partitioning is done for 8 parts. When the part count is increased, the numbers of explored and pruned branches increase exponentially. It also means that stealing and producing tasks happen more frequently. Because of this, work-stealing parameters has to be adjusted accordingly. Since we used same parameter set for all test cases, the effect varied significantly.

Table 6.13: Comparison of work-stealing and master-worker algorithms performances for partitioning to 2,4, and 8 parts with TM metric. The columns with title "Solved" shows the number of tests it completed among 80 tests and number of tests both of the algorithms completed in given scenario. "Avg. T" is the average completion time of the tests that both of the algorithms completed.

		4 Thread				8 Thread			
		Master-Worker		Work-Stealing		Master-Worker		Work-Stealing	
Part	**Test**	**Solved**	**Avg. T**	**Solved**	**Avg. T**	**Solved**	**Avg. T**	**Solved**	**Avg. T**
2	80	78-78	1.60	78-78	0.24	78-78	0.81	78-78	0.23
4	80	36-32	20.85	34-32	5.42	38-32	15.70	34-32	4.36
8	80	18-16	8.31	18-16	46.78	20-16	4.27	18-16	48.21

Table 6.14: Comparison of work-stealing and master-worker algorithms performances for partitioning to 2,4, and 8 parts with TV metric. The columns with title "Solved" shows the number of tests it completed among 80 tests and number of tests both of the algorithms completed in given scenario. "Avg. T" is the average completion time of the tests that both of the algorithms completed.

		4 Thread				8 Thread			
		Master-Worker		Work-Stealing		Master-Worker		Work-Stealing	
Part	**Test**	**Solved**	**Avg. T**	**Solved**	**Avg. T**	**Solved**	**Avg. T**	**Solved**	**Avg. T**
2	80	56-56	76.86	56-56	72.16	58-57	80.32	57-57	104.97
4	80	28-28	108.69	29-28	100.78	31-27	27.78	27-27	97.53
8	80	13-13	272.12	13-13	258.80	15-9	44.67	9-9	273.18

6.4 Reordering

We tested proposed reordering strategies for TV and TM metrics and partition the hypergraphs into 2, 4, 8 parts.

Table 6.15: Number of won test by each reordering. Winning criteria is either having less execution time or producing a partition with less cost. Test hypergraphs are the column-net (CN) and the fine grain (FG) models and partitioned with *total volume* (TM) and *total message* (TM) metrics into 2, 4, and 8 parts. Here, the performances of the *Net Cost Ordering*(NCO), *Weighted Connectivity Ordering*(WCO), *Affected Vertices Ordering*(AVO), *Initial Partition Ordering*(IPO), *Initial Partition Ordering with Net Cost*(IPOwNC), and *Initial Partition Ordering with Weighted Connectivity*(IPOwWC) are compared by the number of won test cases.

K	**Metric**	**Natural**	**IPO**	**IPOwNC**	**IPOwWC**	**NCO**	**WCO**	**AVO**
2	TMCN	35	16	21	23	27	26	37
4	TMCN	4	3	4	1	13	9	17
8	TMCN	4	3	0	4	12	7	14
2	TVCN	8	6	7	7	24	18	26
4	TVCN	4	0	1	2	21	4	15
8	TVCN	1	0	1	0	14	5	25
2	TMFG	33	0	0	0	2	0	14
4	TMFG	25	0	1	0	10	5	11
8	TMFG	34	2	2	2	2	2	11
2	TVFG	16	0	2	1	5	1	19
4	TVFG	19	0	0	0	2	1	25
8	TVFG	14	2	2	2	3	2	30

Table 6.15 shows the number of tests a particular reordering strategy gives the best result. If all of them enabled the algorithm to partition in a given time, we determined the one which gives best result based on its completion time. If none of them find the optimal one, we picked the one which finds better cost. As discussed in Chapter 5, we aimed to prune branches earlier adopting 2 strategies which are adding more constraints in earlier levels of the tree and finding the optimal one earlier. For the second one, we have developed the initial partitioning based reordering methods. However, this approach did not achieve to give improvements as we thought it would. Adding more constraints in earlier levels performed better than the natural order based on the properties of the hypergraph. One important finding is that the fine grain model hypergraphs are not improved that much by these net based reordering strategies. The reason for that is, the fine grain model has a specific property which each vertex has exactly 2 nets. This limited net information prevents us to obtain an efficient order.

In order to alleviate that we also introduced the *Affected Vertex Ordering*. Table 6.15 shows that it performs much better than others, especially in total volume metric.

In Table 6.16, we can see the number of completed tests for each reordering strategies. These numbers are the relative values to the natural order. One can observe that although

Net Cost Ordering and *Weighted Connectivity Ordering* completed more tests in the given duration, they have caused solving fewer tests in some test categories. *Affected Vertex Ordering* achieved consistent improvements in all cases. However, its benefit is still limited for the fine grain model tests compared to the column-net model tests.

Table 6.16: Completed Tests for different reorderings. We found the number of completed tests using natural ordering. The table shows the number of completed tests using different reordering strategies with respect to naturel order. For example, 0 means that it completed the same number of tests with Natural ordering. For each row there are 40 tests. Test hypergraphs are the column-net (CN) and the fine grain (FG) models and partitioned with *total volume* (TM) and *total message* (TM) metrics into 2, 4, and 8 parts. Here, the performances of the *Net Cost Ordering*(NCO), *Weighted Connectivity Ordering*(WCO), *Affected Vertices Ordering*(AVO), *Initial Partition Ordering*(IPO), *Initial Partition Ordering with Net Cost*(IPOwNC), and *Initial Partition Ordering with Weighted Connectivity*(IPOwWC) are compared by the number of completed test cases.

K	**Metric**	**Natural**	**IPO**	**IPOwNC**	**IPOwWC**	**NCO**	**WCO**	**AVO**
2	TMCN	40	-2	-3	-1	0	0	0
4	TMCN	34	-5	-4	-3	0	1	4
8	TMCN	15	2	2	6	12	12	17
2	TVCN	38	0	0	0	0	1	2
4	TVCN	23	3	2	4	8	9	10
8	TVCN	8	3	4	4	13	15	16
2	TMFG	38	-13	-9	-9	-9	-7	2
4	TMFG	0	1	0	0	3	2	4
8	TMFG	0	0	0	0	0	0	4
2	TVFG	13	-7	-7	-9	-4	-9	5
4	TVFG	1	-1	-1	-1	-1	-1	3
8	TVFG	0	0	0	0	0	0	2

6.5 MWIS

We tested the performance of the MWIS bound designed for total volume metric for 2, 4, 8 parts. Table 6.17 shows the impact of the MWIS bound compared to the performance with only the assigned vertex bound for $K = 2$. For $K = 4$ and $K = 8$ results, the reader is referred to Appendix A. Unfortunately, although it improves the execution time of a few tests whereas worsens most of the tests. The worsening impact is significant especially in the column-net models.

Table 6.17: Performance of MWIS bound when $K = 2$. 2nd column is the number of tests which were not being completed without the bound but became completed. 3rd column is the number of tests which were being completed without the bound but became incomplete. The rest are the statistics for the tests which are completed with and without the MWIS bound.

Hypergraph	Test	Becomes Completed	Becomes Incomplete	Improved	Speed Ratio	Worsened	Speed Ratio
FG-Unit	20	0	3	0	0.00	4	0.29
FG-Normal	20	0	2	0	0.00	4	0.31
CN-Unit	20	0	0	2	1.36	16	0.30
CN-Normal	20	0	0	1	1.49	17	0.35

Table 6.18 shows the average spent time ratio on each operation during tree exploration of $K = 2$ partitioning with total volume metric including the MWIS bound. It can be seen that most of the time has been spent on the update operations during advance and retreat step. Another interesting result is that MWIS bound calculation is the least time-consuming operation on average. However, this can not imply that the MWIS bound calculation does not have a significant effect on the execution time. Its major time-consuming operation is done during updates. Additional internal data structure updates for MWIS calculation takes a lot of time.

Table 6.18: The ratio of the time that each operation takes to the overall partitioning execution time. Ratios are averaged over all the tests for $K = 2$

Hypergraph	Test	Remove T. Ratio	Update T. Ratio	Bound T. Ratio	MWIS T. Ratio
FG-Unit	20	0.44	0.46	0.10	0.00
FG-Normal	20	0.45	0.45	0.10	0.00
CN-Unit	20	0.40	0.39	0.19	0.02
CN-Normal	20	0.40	0.39	0.19	0.02

Table 6.19 shows the performance of the bounds at each level of tree exploration for total volume metric. The numbers in the table are the ratio of the number of branches that the bound is eliminated among all the branches explored at that level. We should note that MWIS bound is calculated for the branches which could not be pruned by the assigned vertex bound. So, it may reduce the success rate of the MWIS bound.

An interesting finding of Table 6.19 is that the fine grain model produces hypergraphs which are very hard to partition with the branch and bound method. Not only assigned vertex bound but also MWIS could not eliminate branches in early levels of the tree. This might be because of the lack of the constraints due to having only 2 nets for each vertex.

Table 6.19 also indicates that the MWIS and the assigned vertex bound elimination performances are close. So, we can conclude that, although MWIS worsens the majority of the tests execution time, it is a promising bound that prunes as well as the assigned vertex bound.

Table 6.19: The values are the ratio of the number of eliminated branches by the initial bound to number of all branches tested at this level | ratio of the number of branches which were not eliminated by the initial bound but eliminated by MWIS bound to number of all branches tested at this level for $K = 2$ with total volume metric.

Hypergraph	**Test**	**0-10%**	**10-20%**	**20-30%**	**30-40%**	**40-50%**	**50-60%**	**60-70%**	**70-80%**	**80-90%**	**90-100%**
FG-Unit	20	0.00|0.00	0.00|0.00	0.00|0.02	0.02|0.08	0.08|0.14	0.14|0.14	0.14|0.16	0.16|0.14	0.14|0.17	0.17|0.05
FG-Normal	20	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.01	0.01|0.01	0.01|0.01	0.01|0.01
CN-Unit	20	0.03|0.01	0.01|0.03	0.03|0.04	0.04|0.06	0.06|0.04	0.04|0.07	0.07|0.05	0.05|0.10	0.10|0.09	0.09|0.03
CN-Normal	20	0.03|0.02	0.02|0.04	0.04|0.03	0.03|0.04	0.04|0.04	0.04|0.06	0.06|0.06	0.06|0.12	0.12|0.16	0.16|0.05

Chapter 7

Conclusion and Future Work

In this thesis, we developed an optimal hypergraph partitioning tool adopting branch and bound strategy. We introduced bounds for various metrics such as *total volume* and *max send message*. We tested our tool on the hypergraphs having up to 200 vertices and 250 nets. The tool performed better for message-based metrics; for instance, it only failed to partition a few of the hypergraphs in to 2 parts optimally with 16 threads in 20 minutes which is the time limit we determined manually. In order to speed up the algorithm, we provided an initial partitioning and it greatly increased the speed and the number of solved hypergraphs.

This also introduced another practical use case for the tool in addition to enabling us to compare the current existing partitioning tools. One can use *PHaraoh* in order to improve a partitioning which is used in practice. In our experiments, it finds a partition significantly better than the given partition for most of the cases before the timeout.

We tested our tool on small hypergraphs. However, it is not useful for practical applications to use without an initial partition. More, its performance in large hypergraphs is unknown even using an initial partition. It would be interesting to investigate its applicability in practical applications and its performance.

Another fruitful direction for this study is developing tighter bounds. We introduced the bounds for assigned vertices. They also do not take into account the nets whose source pins are not assigned for directed hypergraph metrics. The bounds using these nets or unassigned vertices could increase the performance of branch and bound considerably. Additionally, the balance constraint can be taken into account while dealing with unassigned vertices for possible earlier pruning.

We have introduced a bound using unassigned vertices for *total volume*. The results show that, although it could not improve the overall algorithm speed, it offers a good pruning ratio. The main reason why it does not have a positive impact is that it takes time to update an internal data structure that facilitates computing the bound. Optimizing the bound calculation technique and used using different data structures could accelerate the process.

The operations during the exploration process has to be investigated thoroughly. Considering Table 6.19 and Table 6.18, we can also point out some promising optimization directions. For example, in early levels of the branch and bound tree for fine grain hypergraphs with different vertex weights and net costs, both of the bounds do not prune any branches. In other words, the operations that take more than 10% of the total execution time are done resulting in no benefits. This could be a source of future improvement.

We introduced various reordering strategies and tested their performances. It is seen that the performance of the algorithm greatly depends on the order of the vertices in the search tree. Adding constraint based reordering strategies performed well on hypergraphs which are constructed using the column net model. Although *Affected Vertex Ordering* improved the performance for fine grain models, there is still opportunity to have speed up more using more information earlier with reordering. It is another interesting area to investigate reordering strategies which can perform well on any kind of hypergraphs. Moreover, we only investigated reordering approaches which do not take the source pin information into account. The proposed approaches in this thesis treats each pin equally regardless of their role in the communication, i.e., source or target. Furthermore, our approaches are not metric-dependent. More advanced reordering strategies can be developed specific to each metric.

Another technique we employed to gain speedup is parallelization. We employed a simple master-slave approach and get reasonably well speed up. However, we faced scalability issue here as normal to branch and bound algorithms. Work-stealing based parallelization approaches are proven to work better on the branch and bound algorithms. Although we implemented a work-staling based algorithm we did not experimented with large number of threads and different parameters. Results show that it can hasten the algorithm considerably. However, the effect of the parameters such as the pool size, the depth value, on different tests has to be investigated in detail and optimized for optimal

hypergraph partitioning problem. Moreover, the bound calculations take long times which constitutes the large portion of the overall time. Investigating an effective strategy to assign these computations to GPU is a promising future avenue as well.

Appendix A

Table A.1: Total Volume when $K = 2$. T stands for the Time. Times are given in seconds. Last four columns are the performance statistics of given parallelization for the tests which are completed by sequential version

Hypergraph	Test	Thread	Completed	Avg. T Completed	Avg. T All	Cost Ratio Completed	Cost Ratio All	Stats for solved tests by sequential Completed	Avg. T	Min T	Max T
total	80	1	51	87.48	490.77	14.53	9.96	51	87.48	0.00	858.00
		4	56	76.86	413.86	13.55	10.03	51	34.16	0.00	331.20
		8	58	95.02	398.96	13.19	10.06	51	26.91	0.00	310.70
		16	58	68.92	380.04	13.19	10.08	51	14.81	0.00	295.41
FG-Unit	20	1	7	258.54	870.49	3.65	2.61	7	258.54	0.03	662.13
		4	9	212.47	755.69	3.56	2.63	7	132.32	0.04	331.20
		8	10	249.93	725.11	3.68	2.77	7	91.88	0.05	310.70
		16	10	146.26	673.26	3.68	2.81	7	67.88	0.10	295.41
FG-Normal	20	1	6	194.00	898.20	5.76	3.07	6	194.00	0.04	707.08
		4	9	228.71	763.08	5.11	3.31	6	80.35	0.04	270.15
		8	10	275.94	738.09	4.77	3.31	6	79.48	0.05	267.09
		16	10	243.96	722.13	4.77	3.32	6	30.86	0.09	111.73
CN-Unit	20	1	19	45.35	103.08	14.59	13.91	19	45.35	0.00	858.00
		4	19	8.93	68.48	14.59	13.92	19	8.93	0.00	166.86
		8	19	5.67	65.39	14.59	13.93	19	5.67	0.00	106.54
		16	19	1.74	61.66	14.59	13.93	19	1.74	0.00	31.41
CN-Normal	20	1	19	32.96	91.31	21.24	20.24	19	32.96	0.00	621.77
		4	19	8.63	68.20	21.24	20.24	19	8.63	0.00	161.18
		8	19	7.62	67.25	21.24	20.25	19	7.62	0.00	140.77
		16	19	3.26	63.10	21.24	20.26	19	3.26	0.00	59.61

Table A.2: Total Message when $K = 2$. T stands for the Time. Times are given in seconds. Last four columns are the performance statistics of given parallelization for the tests which are completed by sequential version

Hypergraph	Test	Thread	Completed	Avg. T Completed	Avg. T All	Cost Ratio Completed	Cost Ratio All	Stats for solved tests by sequential Completed	Avg. T	Min T	Max T
total	80	1	78	2.34	32.28	1.60	1.60	78	2.34	0.00	78.40
		4	78	1.60	31.56	1.60	1.60	78	1.60	0.00	56.51
		8	78	0.81	30.79	1.60	1.60	78	0.81	0.00	17.48
		16	78	0.51	30.50	1.60	1.60	78	0.51	0.00	9.84
FG-Unit	20	1	19	1.35	61.28	1.58	1.57	19	1.35	0.00	19.80
		4	19	0.82	60.78	1.58	1.57	19	0.82	0.00	13.61
		8	19	0.83	60.78	1.58	1.57	19	0.83	0.00	14.55
		16	19	0.54	60.52	1.58	1.57	19	0.54	0.00	9.84
FG-Normal	20	1	19	0.76	60.72	1.58	1.57	19	0.76	0.00	8.59
		4	19	0.58	60.55	1.58	1.57	19	0.58	0.00	8.92
		8	19	0.59	60.56	1.58	1.57	19	0.59	0.00	9.94
		16	19	0.52	60.49	1.58	1.57	19	0.52	0.00	9.20
CN-Unit	20	1	20	4.07	4.07	1.62	1.62	20	4.07	0.00	78.40
		4	20	2.90	2.90	1.62	1.62	20	2.90	0.00	56.51
		8	20	0.84	0.84	1.62	1.62	20	0.84	0.00	14.82
		16	20	0.44	0.44	1.62	1.62	20	0.44	0.00	7.63
CN-Normal	20	1	20	3.04	3.04	1.62	1.62	20	3.04	0.00	58.65
		4	20	2.01	2.01	1.62	1.62	20	2.01	0.00	37.98
		8	20	0.97	0.97	1.62	1.62	20	0.97	0.00	17.48
		16	20	0.55	0.55	1.62	1.62	20	0.55	0.00	8.68

Table A.3: Total Volume when $K = 8$. T stands for the Time. Times are given in seconds. Last four columns are the performance statistics of given parallelization for the tests which are completed by sequential version

Hypergraph	Test	Thread	Completed	Avg. T Completed	Avg. T All	Cost Ratio Completed	Cost Ratio All	Stats for solved tests by sequential Completed	Avg. T	Min T	Max T
total	80	1	8	85.65	1088.57	1.37	1.63	8	85.65	0.01	287.49
		4	13	272.12	1049.60	1.50	1.66	8	34.30	0.01	144.33
		8	15	249.36	1022.17	1.59	1.68	8	18.68	0.04	86.87
		16	16	208.63	1002.21	1.56	1.69	8	11.18	0.12	57.25
FG-Unit	20	1	0	0.00	1200.00	0.00	1.62	0	-	-	-
		4	0	0.00	1200.46	0.00	1.63	0	-	-	-
		8	0	0.00	1200.49	0.00	1.64	0	-	-	-
		16	0	0.00	1200.60	0.00	1.64	0	-	-	-
FG-Normal	20	1	0	0.00	1200.00	0.00	1.71	0	-	-	-
		4	0	0.00	1200.54	0.00	1.73	0	-	-	-
		8	0	0.00	1200.47	0.00	1.74	0	-	-	-
		16	0	0.00	1200.49	0.00	1.74	0	-	-	-
CN-Unit	20	1	4	74.37	974.87	1.45	1.56	4	74.37	0.01	287.49
		4	7	350.27	902.85	1.57	1.58	4	37.13	0.01	144.33
		8	7	172.72	840.81	1.57	1.60	4	22.22	0.04	86.87
		16	8	228.20	811.69	1.51	1.63	4	14.66	0.12	57.25
CN-Normal	20	1	4	96.93	979.39	1.28	1.63	4	96.93	0.01	226.23
		4	6	180.95	894.54	1.43	1.69	4	31.48	0.01	73.44
		8	8	316.41	846.90	1.60	1.73	4	15.15	0.04	35.44
		16	8	189.06	796.07	1.60	1.76	4	7.70	0.12	18.59

Table A.4: Total Message when $K = 8$. T stands for the Time. Times are given in seconds. Last four columns are the performance statistics of given parallelization for the tests which are completed by sequential version

Hypergraph	Test	Thread	Completed	Avg. T Completed	Avg. T All	Cost Ratio Completed	Cost Ratio All	Stats for solved tests by sequential Completed	Avg. T	Min T	Max T
total	80	1	18	41.74	939.39	3.59	3.12	18	41.74	0.01	302.59
		4	18	10.61	932.51	3.56	3.32	18	10.61	0.00	63.83
		8	20	73.38	918.49	3.68	3.42	18	4.04	0.01	38.05
		16	20	36.62	909.40	3.68	3.57	18	1.56	0.01	11.94
FG-Unit	20	1	0	0.00	1200.00	0.00	3.05	0	-	-	-
		4	0	0.00	1200.03	0.00	3.16	0	-	-	-
		8	0	0.00	1200.09	0.00	3.20	0	-	-	-
		16	0	0.00	1200.25	0.00	3.33	0	-	-	-
FG-Normal	20	1	0	0.00	1200.00	0.00	3.15	0	-	-	-
		4	0	0.00	1200.05	0.00	3.71	0	-	-	-
		8	0	0.00	1200.07	0.00	3.86	0	-	-	-
		16	0	0.00	1200.25	0.00	4.04	0	-	-	-
CN-Unit	20	1	9	54.12	684.36	3.50	3.03	9	54.12	0.01	302.59
		4	9	11.95	665.59	3.46	3.08	9	11.95	0.00	63.83
		8	9	6.68	663.24	3.48	3.09	9	6.68	0.01	38.05
		16	9	2.41	661.33	3.48	3.21	9	2.41	0.01	11.94
CN-Normal	20	1	9	29.35	673.21	3.67	3.25	9	29.35	0.01	90.39
		4	9	9.27	664.38	3.65	3.34	9	9.27	0.00	54.11
		8	11	127.96	610.58	3.85	3.55	9	1.40	0.01	4.31
		16	11	64.61	575.78	3.85	3.72	9	0.70	0.01	2.39

Table A.5: 16 Thread Performance when $K = 4$. Execution times are in seconds

Matrix	Row	Column	NNZ	Total Volume FG-Normal	FG-Unit	CN-Normal	CN-Unit	Total Message FG-Normal	FG-Unit	CN-Normal	CN-Unit
lpi_woodinfe	35	89	140	1200.43	1200.20	1.43	1.43	1200.12	1200.12	0.05	0.04
GD01_c	33	33	135	1200.80	1200.42	0.88	0.86	1200.24	1200.63	0.09	0.11
Trec7	11	36	147	1200.02	1200.82	0.08	0.13	1200.85	1200.59	0.01	0.02
ibm32	32	32	126	1200.88	1200.98	1200.94	1200.97	1200.12	1200.12	6.08	0.27
Hamrle1	32	32	98	1200.80	1200.49	106.31	130.47	1200.09	1200.10	70.49	129.77
pores_1	30	30	180	1200.94	1200.95	8.73	2.20	1200.17	1200.14	0.27	0.60
football	35	35	118	1200.67	1200.85	1200.84	1201.00	1200.54	1200.10	1.14	0.94
lp_afiro	27	51	102	1200.43	1200.64	624.99	1200.98	1200.91	1200.98	370.12	457.39
Ragusa16	24	24	81	1200.09	1200.26	3.67	4.23	1200.71	1200.55	0.06	0.19
GD98_b	121	121	207	1200.85	1200.21	1200.03	1200.23	1200.18	1200.16	1200.11	1200.11
karate	34	34	78	1200.81	1200.82	241.35	109.47	29.55	22.34	0.04	3.84
p0033	15	48	113	1200.92	1200.94	0.10	0.09	1200.84	1200.98	0.03	0.03
can_24	24	24	92	1200.40	1200.45	9.86	7.02	1200.82	1200.61	0.26	0.26
bcspwr01	39	39	85	37.24	25.16	1.13	0.90	3.89	1.84	0.06	0.06
GD95_b	73	73	96	1200.11	1200.21	1200.32	1200.39	2.05	2.20	1200.07	1200.07
GlossGT	72	72	122	1200.58	1200.80	1200.90	1200.81	1200.10	1200.10	1200.07	1200.07
Trefethen_20	20	20	89	1200.07	1200.08	2.29	1.35	1200.08	1200.08	0.18	0.11
d_ss	53	53	149	1200.76	1200.90	612.00	821.15	1200.13	1200.12	0.81	0.78
Trefethen_20b	19	19	83	1200.49	1200.85	0.94	0.79	1200.08	1200.27	0.05	0.04
n3c4-b2	20	15	60	1200.12	1200.17	0.77	3.49	1200.94	1200.06	0.07	0.08

Table A.6: 16 Thread Performance when $K = 8$. Execution times are in seconds

Matrix	Row	Column	NNZ	Total Volume FG-Normal	FG-Unit	CN-Normal	CN-Unit	Total Message FG-Normal	FG-Unit	CN-Normal	CN-Unit
lpi_woodinfe	35	89	140	1200.95	1200.99	1200.99	1200.97	1200.24	1200.24	1.00	1.46
GD01_c	33	33	135	1200.26	1200.47	161.56	57.25	1200.29	1200.23	0.91	2.52
Trec7	11	36	147	1200.25	1200.24	0.12	0.12	1200.27	1200.26	0.01	0.02
ibm32	32	32	126	1200.72	1200.08	1200.45	1200.50	1200.22	1200.24	1200.95	1200.94
Hamrle1	32	32	98	1200.94	1200.93	1200.67	1200.27	1200.18	1200.19	1200.89	1200.96
football	35	35	118	1200.10	1200.11	1200.89	1200.93	1200.21	1200.21	1200.10	1200.10
pores_1	30	30	180	1200.73	1200.76	495.34	1092.35	1200.33	1200.31	666.73	1200.94
lp_afiro	27	51	102	1200.07	1200.84	1200.99	1200.99	1200.19	1200.18	1200.95	1200.92
Ragusa16	24	24	81	1200.21	1200.71	1200.97	1200.98	1200.15	1200.15	2.39	11.94
GD98_b	121	121	207	1200.10	1200.27	1200.92	1200.89	1200.38	1200.41	1200.23	1200.22
p0033	15	48	113	1200.50	1200.19	0.16	0.18	1200.22	1200.22	0.01	0.01
karate	34	34	78	1200.32	1200.66	1200.54	1200.64	1200.15	1200.16	37.61	1200.09
bcspwr01	39	39	85	1200.19	1200.40	718.87	330.96	1200.98	1200.99	0.42	0.73
can_24	24	24	92	1200.58	1200.74	1200.93	1200.99	1200.18	1200.18	1200.88	1200.32
GD95_b	73	73	96	1200.73	1200.79	1200.90	1200.77	1200.18	1200.17	1200.14	1200.14
GlossGT	72	72	122	1200.74	1200.71	1200.39	1200.11	1200.21	1200.21	1200.15	1200.14
Trefethen_20	20	20	89	1200.38	1200.83	105.94	238.26	1200.16	1200.17	0.58	0.81
d_ss	53	53	149	1200.62	1200.64	1200.21	1200.23	1200.27	1200.25	1200.58	1200.12
Trefethen_20b	19	19	83	1200.91	1200.91	18.59	1.08	1200.16	1200.16	0.12	0.84
n3c4-b2	20	15	60	1200.59	1200.78	11.91	105.37	1200.12	1200.13	0.91	3.37

Table A.7: Performance of MWIS bound when $K = 4$. 2nd column is the number of tests which were not being completed without the bound but became completed. 3rd column is the number of tests which were being completed without the bound but became incomplete. The rest are the statistics for the tests which are completed with and without the MWIS bound.

Hypergraph	Test	Becomes Completed	Becomes Incomplete	Improved	Speed Ratio	Worsened	Speed Ratio
FG-Unit	20	0	0	0	0.00	1	0.41
FG-Normal	20	0	0	0	0.00	0	0.00
CN-Unit	20	1	0	0	0.00	10	0.46
CN-Normal	20	1	0	1	1.61	11	0.46

Table A.8: Performance of MWIS bound when $K = 8$. 2nd column is the number of tests which were not being completed without the bound but became completed. 3rd column is the number of tests which were being completed without the bound but became incomplete. The rest are the statistics for the tests which are completed with and without the MWIS bound.

Hypergraph	Test	Becomes Completed	Becomes Incomplete	Improved	Speed Ratio	Worsened	Speed Ratio
FG-Unit	20	0	0	0	0.00	0	0.00
FG-Normal	20	0	0	0	0.00	0	0.00
CN-Unit	20	0	0	0	0.00	4	0.34
CN-Normal	20	0	0	0	0.00	4	0.47

Table A.9: The ratio of the time that each operation takes to the overall partitioning execution time. Ratios are averaged over all the tests for $K = 4$

Hypergraph	Test	Remove T. Ratio	Update T. Ratio	Bound T. Ratio	MWIS T. Ratio
FG-Unit	20	0.41	0.42	0.16	0.00
FG-Normal	20	0.42	0.42	0.16	0.00
CN-Unit	20	0.34	0.30	0.34	0.02
CN-Normal	20	0.33	0.29	0.35	0.02

Table A.10: The values are the ratio of the number of eliminated branches by the initial bound to number of all branches tested at this level | ratio of the number of branches which were not eliminated by the initial bound but eliminated by MWIS bound to number of all branches tested at this level for $K = 4$ with total volume metric.

Hypergraph	Test	0-10%	10-20%	20-30%	30-40%	40-50%	50-60%	60-70%	70-80%	80-90%	90-100%
FG-Unit	20	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00
FG-Normal	20	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00
CN-Unit	20	0.00|0.00	0.00|0.00	0.00|0.01	0.01|0.01	0.01|0.01	0.01|0.03	0.03|0.03	0.03|0.07	0.07|0.08	0.08|0.07
CN-Normal	20	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.01	0.01|0.02	0.02|0.02	0.02|0.06	0.06|0.05	0.05|0.06

Table A.11: The ratio of the time that each operation takes to the overall partitioning execution time. Ratios are averaged over all the tests for $K = 8$

Hypergraph	Test	Remove T. Ratio	Update T. Ratio	Bound T. Ratio	MWIS T. Ratio
FG-Unit	20	0.41	0.40	0.19	0.00
FG-Normal	20	0.40	0.39	0.20	0.00
CN-Unit	20	0.28	0.27	0.43	0.02
CN-Normal	20	0.29	0.26	0.43	0.03

Table A.12: The values are the ratio of the number of eliminated branches by the initial bound to number of all branches tested at this level | ratio of the number of branches which were not eliminated by the initial bound but eliminated by MWIS bound to number of all branches tested at this level for $K = 8$ with total volume metric.

Hypergraph	**Test**	**0-10%**	**10-20%**	**20-30%**	**30-40%**	**40-50%**	**50-60%**	**60-70%**	**70-80%**	**80-90%**	**90-100%**
FG-Unit	20	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00
FG-Normal	20	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00
CN-Unit	20	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.01	0.01|0.02	0.02|0.14	0.14|0.09	0.09|0.02
CN-Normal	20	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.00	0.00|0.01	0.01|0.11	0.11|0.05	0.05|0.02

Bibliography

[1] N. Agin, *Optimum seeking with branch and bound*, Management Science, 13 (1966), pp. B–176–B–185.

[2] E. Alba, F. Almeida, M. J. Blesa, C. Cotta, M. DÄśÌĄaz, I. Dorta, J. Gabarró, J. Gonzalez, C. Léon, L. A. Moreno, J. Petit, J. Roda, A. L. Rojas, and F. Xhafa, *Mallba: Towards a combinatorial optimization library for geographically distributed systems*, in Proceedings of the XII Jornadas de Paralelismo, 2001, pp. 105–110.

[3] C. J. Alpert and A. B. Kahng, *Recent directions in netlist partitioning: A survey*, VLSI Journal, 19 (1995), pp. 1–81.

[4] B. Archibald, P. Maier, C. McCreesh, R. Stewart, and P. Trinder, *Replicable parallel branch and bound search*, Journal of Parallel and Distributed Computing, 113 (2018), pp. 92 – 114.

[5] 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 (2008), pp. 609–625.

[6] D. A. Bader, W. E. Hart, and C. A. Phillips, *Parallel Algorithm Design for Branch and Bound*, Springer New York, New York, NY, 2005, pp. 1–44.

[7] E. Balas, *Letter to the editor-a note on the branch-and-bound principle*, Operations Research, 16 (1968), pp. 442–445.

[8] C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J. Ullman, and M. Yannakakis, *Properties of acyclic database schemes*, in Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing, STOC '81, New York, NY, USA, 1981, ACM, pp. 355–362.

[9] R. H. Bisseling and W. Meesen, *Communication balancing in parallel sparse matrix-vector multiplication*, Electronic Transactions on Numerical Analysis, 21 (2005), pp. 47–65.

[10] T. N. Bui and C. Jones, *A heuristic for reducing fill-in in sparse matrix factorization*, in Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing, SIAM, 1993, pp. 445–452.

[11] A. E. Caldwell, A. B. Kahng, and I. L. Markov, *Optimal partitioners and end-case placers for standard-cell layout*, in Proceedings of the 1999 International Symposium on Physical Design, ISPD '99, New York, NY, USA, 1999, ACM, pp. 90–96.

[12] Ü. Çatalyürek and C. Aykanat, *PaToH (partitioning tool for hypergraphs)*, Encyclopedia of Parallel Computing, 2011.

[13] Ü. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel and Distributed Systems, 10 (1999), pp. 673–693.

[14] ——, *A hypergraph-partitioning approach for coarse-grain decomposition*, in ACM/IEEE SC2001, Denver, CO, Nov 2001.

[15] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdağ, R. T. Heaphy, and L. A. Riesen, *A repartitioning hypergraph model for dynamic load balancing*, Journal of Parallel and Distributed Computing, 69 (2009), pp. 711 – 724. Best Paper Awards: 21st International Parallel and Distributed Processing Symposium (IPDPS 2007).

[16] U. V. Catalyurek, M. Deveci, K. Kaya, and B. Uçar, *UMPA: A Multi-objective, multi-level partitioner for communication minimization*, in Graph Partitioning and Graph Clustering 2012, D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, eds., vol. 588 of Contemporary Mathematics, AMS, 2013, pp. 53–66.

[17] T. Catarci and L. Tarantino, *Database querying by hypergraph manipulation*, in Interfaces to Database Systems (IDS94), P. Sawyer, ed., London, 1995, Springer London, pp. 84–103.

[18] U. V. Çatalyürek, K. Kaya, and B. Uçar, *Integrated data placement and task assignment for scientific workflows in clouds*, in Proceedings of the Fourth International Workshop on Data-intensive Distributed Computing, DIDC '11, New York, NY, USA, 2011, ACM, pp. 45–54.

[19] J. Clausen and M. Perregaard, *On the best search strategy in parallel branch-and-bound:best-first search versus lazy depth-first search*, Annals of Operations Research, 90 (1999), pp. 1–17.

[20] T. Crainic, B. Le Cun, and C. Roucairol, *Parallel Branch-and-Bound Algorithms*, John Wiley & Sons, Inc., 2006, pp. 1–28.

[21] G. Dantzig, R. Fulkerson, and S. Johnson, *Solution of a large-scale traveling-salesman problem*, Journal of the Operations Research Society of America, 2 (1954), pp. 393–410.

[22] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan, *Zoltan data management services for parallel dynamic applications*, Computing in Science Engineering, 4 (2002), pp. 90–96.

[23] K. Devine, E. Boman, L. Riesen, U. Catalyurek, and C. Chevalier, *Getting started with Zoltan: A short tutorial*, in Proc. of 2009 Dagstuhl Seminar on Combinatorial Scientific Computing, 2009. Also available as Sandia National Labs Tech Report SAND2009-0578C.

[24] A. Djerrah, B. L. Cun, V. D. Cung, and C. Roucairol, *Bob++: Framework for solving optimization problems with branch-and-bound methods*, in 2006 15th IEEE International Conference on High Performance Distributed Computing, 2006, pp. 369–370.

[25] C. Dong, Q. Zhou, Y. Cai, and X. Hong, *Hypergraph partitioning satisfying dual constraints on vertex and edge weight*, in 2008 51st Midwest Symposium on Circuits and Systems, Aug 2008, pp. 85–88.

[26] D. K. Fedor V. Fomin, *Exact Exponential Algorithms*, Springer-Verlag Berlin Heidelberg, 2010.

[27] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in Proc. 19th Design Automation Conference, 1982, pp. 175–181.

[28] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen, *Directed hypergraphs and applications*, Discrete Appl. Math., 42 (1993), pp. 177–201.

[29] B. Hendrickson, *Graph partitioning and parallel solvers: Has the emperor no clothes?*, in Solving Irregularly Structured Problems in Parallel, A. Ferreira, J. Rolim, H. Simon, and S.-H. Teng, eds., Berlin, Heidelberg, 1998, Springer Berlin Heidelberg, pp. 218–225.

[30] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, in Proceedings of the 1995 ACM/IEEE Conference on Supercomputing, Supercomputing '95, New York, NY, USA, 1995, ACM.

[31] T. Ibaraki, *Enumerative approaches to combinatorial optimization - part I*, Ann. Oper. Res., 10 (1988), pp. 3–342.

[32] A. Kako, T. Ono, T. Hirata, and M. M. Halldórsson, *Approximation algorithms for the weighted independent set problem*, in Graph-Theoretic Concepts in Computer Science, D. Kratsch, ed., Berlin, Heidelberg, 2005, Springer Berlin Heidelberg, pp. 341–350.

[33] R. M. Karp, *Reducibility among Combinatorial Problems*, Springer US, Boston, MA, 1972, pp. 85–103.

[34] 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 (1999), pp. 69–79.

[35] K. Kaya and C. Aykanat, *Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogeneous master-slave environments*, IEEE Transactions on Parallel and Distributed Systems, 17 (2006), pp. 883–896.

[36] K. Kaya, B. Uçar, and C. Aykanat, *Heuristics for scheduling file-sharing tasks on heterogeneous systems with distributed repositories*, Journal of Parallel and Distributed Computing, 67 (2007), pp. 271 – 285.

[37] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell System Technical Journal, 49 (1970), pp. 291–307.

[38] D. Kucar, S. Areibi, and A. Vannelli, *Hypergraph partitioning techniques*, Dynamics of Continuous, Discrete and Impulsive Systems Series A: Mathematical Analysis, 11 (2004), pp. 339–367. cited By 2.

[39] O. Kullmann, *New methods for 3-SAT decision and worst-case analysis*, Theoretical Computer Science, 223 (1999), pp. 1 – 72.

[40] T.-H. Lai and S. Sahni, *Anomalies in parallel branch-and-bound algorithms*, Commun. ACM, 27 (1984), pp. 594–602.

[41] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley–Teubner, Chichester, U.K., 1990.

[42] M. LLC, *Mondriaanopt for optimal sparse matrix bipartitioning*, 2018.

[43] B. Mans and C. Roucairol, *Performances of parallel branch and bound algorithms with best-first search*, Discrete Applied Mathematics, 66 (1996), pp. 57 – 74.

[44] B. Monien and E. Speckenmeyer, *Solving satisfiability in less than 2n steps*, Discrete Applied Mathematics, 10 (1985), pp. 287 – 295.

[45] N. Moreira and R. Reis, *On the density of languages representing finite set partitions*, Journal of Integer Sequences, 8 (2005).

[46] A. Mumcuyan, K. Kaya, and H. Yenigün, *Reordering matrices for optimal sparse matrix bipartitioning*, 2017.

[47] A. Mumcuyan, B. Usta, K. Kaya, and H. Yenigün, *Optimally bipartitioning sparse matrices with reordering and parallelization*, Concurrency and Computation: Practice and Experience, (2018).

[48] D. M. Pelt and R. H. Bisseling, *A medium-grain method for fast 2d bipartitioning of sparse matrices*, in 2014 IEEE 28th International Parallel and Distributed Processing Symposium, May 2014, pp. 529–539.

[49] D. M. Pelt and R. H. Bisseling, *An exact algorithm for sparse matrix bipartitioning*, J. Parallel Distrib. Comput., 85 (2015), pp. 79–90.

[50] A. Pothen, H. D. Simon, and K.-P. Liou, *Partitioning sparse matrices with eigenvectors of graphs*, SIAM J. Matrix Anal. Appl., 11 (1990), pp. 430–452.

[51] S. Rajamanickam and E. G. Boman, *Parallel partitioning with Zoltan: Is hypergraph partitioning worth it?*, 2013.

[52] J. Rambau and C. Schwarz, *On the benefits of using np-hard problems in branch & bound*, in Operations Research Proceedings 2008, B. Fleischmann, K.-H. Borgwardt, R. Klein, and A. Tuma, eds., Berlin, Heidelberg, 2009, Springer Berlin Heidelberg, pp. 463–468.

[53] Y. Shinano, M. Higaki, and R. Hirabayashi, *A generalized utility for parallel branch and bound algorithms*, in Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing, Oct 1995, pp. 392–401.

[54] H. Simon, *Partitioning of unstructured problems for parallel processing*, Computing Systems in Engineering, 2 (1991), pp. 135 – 148. Parallel Methods on Large-scale Structural Analysis and Physics Applications.

[55] H. W. Trienekens and A. de Bruin, *Towards a taxonomy of parallel branch and bound algorithms*, tech. rep., 1992.

[56] A. Trifunovic and W. J. Knottenbelt, *ParKway 2.0: A parallel multilevel hypergraph partitioning tool*, in Computer and Information Sciences - ISCIS 2004, C. Aykanat, T. Dayar, and İ. Körpeoğlu, eds., Berlin, Heidelberg, 2004, Springer Berlin Heidelberg, pp. 789–800.

[57] B. Uçar and C. Aykanat, *Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies*, SIAM J. Sci. Comput., 25 (2004), pp. 1837–1859.

[58] J. Ullman, *Np-complete scheduling problems*, Journal of Computer and System Sciences, 10 (1975), pp. 384 – 393.

[59] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Review, 47 (2005), pp. 67–95.

[60] T. Volgenant and R. Jonker, *A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation*, European Journal of Operational Research, 9 (1982), pp. 83 – 89.

[61] T.-T. Vu and B. Derbel, *Parallel branch-and-bound in multi-core multi-cpu multi-gpu heterogeneous environments*, Future Generation Computer Systems, 56 (2016), pp. 95 – 109.

[62] T.-T. Vu, B. Derbel, A. Asim, A. Bendjoudi, and N. Melab, *Overlay-Centric Load Balancing: Applications to UTS and B&B*, in CLUSTER - 14th IEEE International Conference on Cluster Computing, Beijing, China, Sept. 2012, IEEE.

[63] R. D. Williams, *Performance of dynamic load balancing algorithms for unstructured mesh calculations*, Concurrency: Practice and Experience, 3, pp. 457–481.

[64] M. Yannakakis, *Algorithms for acyclic database schemes*, in Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7, VLDB '81, VLDB Endowment, 1981, pp. 82–94.

[65] D. Zhou, J. Huang, and B. Schölkopf, *Learning with hypergraphs: Clustering, classification, and embedding*, MIT Press, September 2007, pp. 1601–1608.