PARPATOH: A 2D-PARALLEL HYPERGRAPH PARTITIONING TOOL

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Evren Karaca
January, 2006

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Cevdet Aykanat (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Prof. Dr. Ayhan Altıntaş

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. İbrahim Körpeoğlu

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

ABSTRACT

PARPATOH: A 2D-PARALLEL HYPERGRAPH PARTITIONING TOOL

Evren Karaca
M.S. in Computer Engineering
Supervisor: Prof. Dr. Cevdet Aykanat
January, 2006

Hypergraph partitioning is a process that is being used to find solutions for optimization problems in various areas, including parallel volume rendering, parallel information retrieval and VLSI circuit design. While the current partitioning methods are adequate for hypergraphs up to certain size, these methods start to fail once the problem size exceeds this threshold.

In this thesis we introduce ParPaToH, a parallel p-way hypergraph partitioning tool that makes use of a 2-D decomposition to reduce the communication overhead and implements a parallel-computing friendly version of the accepted multi-level partitioning paradigm to generate its partitioning. We present new concepts in hypergraph partitioning that lead to a coarse-grain parallel solution. Finally, we discuss the implementation of the tool in detail and present experimental results to demonstrate its effectiveness.

Keywords: Multilevel hypergraph partitioning, parallel computing.

ÖZET

PARPATOH: BIR 2-BOYUTLU PARALEL HİPERÇİZGE BÖLÜMLEME ARACI

Evren Karaca
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Cevdet Aykanat
Ocak, 2006

Hiperçizge bölümleme, hacim boyama, bilgi getirme ve VLSİ devre tasarımı gibi, değişik alanlardaki en iyileme sorunlarını çözmek için kullanılan bir işlemdir. Varolan bölümleme yöntemleri bell irli bir boya kadar olan hiperçigeler için çalışıyor olsa da, bu boyu aşan hiperçizgeler için yetersiz kalmaktadır.

Bu tezde, iletişim yükünü azaltmak için 2-boyutlu bir veri bölümlemesi kullanan ve kabul görmüş çok-düzeyli bölümleme paradigmasını parallel işlemeye uygun hale getirmiş bir p-yön parallel hiperçizge bölümleme aracı olan ParPaToH'u takdim ediyoruz. Büyük ölçekli iletişime izin veren yeni hiperçizge bölümleme kavramlarını açıkladıktan sonra, aracın yapısını detaylı bir şekilde anlatıp, etkililiğini gözler önüne seren deney sonuçlarını sunuyoruz.

Anahtar sözcükler: Çok-düzeyli hiper çizge bölümleme, paralel işleme.

Acknowledgement

I would like to express my gratitude to Prof. Dr Cevdet Aykanat for his supervision, guidance, encouragement and patience during the development of this thesis.

I would to thank my thesis committee members, Prof. Dr. Ayhan Altıntaş and Asst. Prof. Dr. İbrahim Körpeoğlu, for reading my thesis and providing valuable comments.

This thesis would not be what it is without the environment of mirth and thought provided by the "parallel guys", Ali, Ata, Barla and Tayfun. I would like to especially thank Bora for his ideas that contributed much to this thesis.

Finally, I would like to thank my family for their invaluable support and trust in me.

Contents

1 Introduction **1**

1.1 Organization of the Thesis 2

2 Background **3**

2.1 Preliminaries 3

2.1.1 The hypergraph 3

2.1.2 Hypergraph Partitioning 4

2.1.3 State of the art in sequential partitioning 5

2.2 Applications of Hypergraph partitioning 9

2.3 Related work 10

2.3.1 Sequential Hypergraph Partitioning 10

2.3.2 Parallel Hypergraph Partitioning 11

3 Parallel Hypergraph Partitioning **12**

3.1 Reasons for parallelizing 12

3.2 Difficulties in parallelization 12

3.3 Data distribution 13
3.3.1 One-dimensional decomposition 14
3.3.2 Two-dimensional decomposition 14
3.4 Parallel partitioning methodology 17
3.4.1 The elongated V-cycle 17
3.4.2 Parallel algorithm design considerations 17
3.4.3 Restrictions of ParPaToH 19

4 Implementation **20**
4.1 Coarsening 20
4.1.1 Matching 21
4.1.2 Merging 23
4.1.3 Crossing-over 23
4.1.4 The termination of coarsening phase 24
4.2 Initial Partitioning 24
4.2.1 Parallel initial partitioning 25
4.3 Refinement 26
4.3.1 Uncoarsening 26
4.3.2 Improving the cut 27

5 Experimental Results **35**
5.1 Cut vs. time 35

5.1.1 Tunable parameters 36
5.1.2 Time dissection 39
5.2 Scaling 39

6 Discussions & Conclusions 43

6.1 Evaluation of the results 43
6.2 Ideas for future work 43
6.2.1 Generalization and improvement of the current code-base . 44
6.2.2 Merging identical nets 44
6.2.3 Different refinement algorithm 44
6.2.4 Other features 44
6.3 Conclusion 45

List of Figures

2.1 A hypergraph and its matrix representation 4

2.2 Sequential multi-level hypergraph partitioning 7

3.1 2D data distribution 16

3.2 Parallel hypergraph partitioning 18

4.1 Explanation of terms 21

4.2 Crossing-over in action 24

4.3 Rotating the refinement schedule 30

5.1 The time dissection of a V-cycle 42

List of Tables

5.1 Properties of the partitoned hypergraphs 36

5.2 Effects of the refinement count on the cut size 37

5.3 The effects of the refinement count on the partitioning time . . . 38

5.4 Effects of coarsening depth on the cut 39

5.5 Effects of coarsening depth on the execution times 40

5.6 Effects of crossing-over on the cut 41

5.7 Effects of crossing-over on the execution times 41

5.8 Scaling to multiple processors 42

Chapter 1

Introduction

Hypergraph partitioning, is used to find solutions to many problems in various fields. Among these are parallel computing problems such as parallel sparse-matrix vector multiplication [6], sparse matrix permutation for parallel LU and QR factorization [2], performance analysis [14], and parallel volume rendering [5] as well as other research fields including VLSI design [16], software design [4], and spatial databases [12].

Although several sequential partitioning tools exist (such as PaToH [7] and hMeTiS [16]) that find good quality solutions to the problem, the nature of the algorithms used in these restrict the partitionable hypergraph size to a certain value. Larger hypergraphs cannot be partitioned successfully with these as they no longer fit into the main memory of the system and the tools either fail to work completely or work extremely slowly.

One way to partition large hypergraphs is to parallelize the hypergraph partitioning process and exploit the resources of multiple computers. The research community is actively working on this issue and has recently sucessfully demonstrated the viability of parallel hypergraph partitioning tools [25],[13]. These tools, however, are still in their infancy and, unlike its serial counterpart, there are no well studied and accepted methods is parallel hypergraph partitioning.

In this thesis, we present a parallel hypergraph partitioning implementation that is being developed concurrently, yet independently, with these other tools. It makes use of a 2-D partitioning of the hypergraph to reduce the communication overhead and introduces several novel concepts that allows a coarse-grain parallelization of the multi-level hypergraph partitioning algorithm.

1.1 Organization of the Thesis

The organization of thesis is as follows: Chapter 2 gives background information about the hypergraph partitioning problem, describes current solutions and discusses the currently available tools for hypergraph partitioning. Chapter 3 re-examines the hypergraph partitioning problem from a parallel implementation viewpoint and presents our solutions to the appearing issues. Chapter 4 explains the implementation of the tool in detail. Chapter 5 presents experimental results. Chapter 6 discusses that various strengths and weaknesses of the tool, discusses the plans for future improvement and presents conclusions that can be drawn from the current implementation.

A small note regarding the word “processor” in this document. It is used exclusively to mean a node in a cluster, a single processing entity that has its own computational capabilities and memory. It is not meant to be interpreted as a single processor in a shared memory system. (In fact this paper was not written with these systems in mind)

Chapter 2

Background

2.1 Preliminaries

2.1.1 The hypergraph

A hypergraph is a generalized form of the graph data structure. A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ consists of a set of vertices $\mathcal{V}$ and a set of nets $\mathcal{N}$ [3]. Each net n_i in $\mathcal{N}$ connects a subset of vertices in $\mathcal{V}$, which are said to be the pins of n_i. Each net n_i has a cost c_i and each vertex v_j has a weight w_j.

2.1.1.1 The matrix form

Another way of representing a hypergraph, $\mathcal{H}$, is a sparse, binary matrix, A, of dimensions $V \times N$, where $V = |\mathcal{V}|$ and $N = |\mathcal{N}|$ denote the number of nets and vertices of $\mathcal{H}$ respectively. In this matrix, a_{ij} is 1 if vertex v_i is a pin of net n_j, and 0 if this is not the case. An example illustrating the two forms is given in Figure 2.1. In this representation, a hypergraph would be a graph if number of ones in each column in its matrix representation is exactly two.

The degree of a net n_i is defined as the number of its pins. The degree of

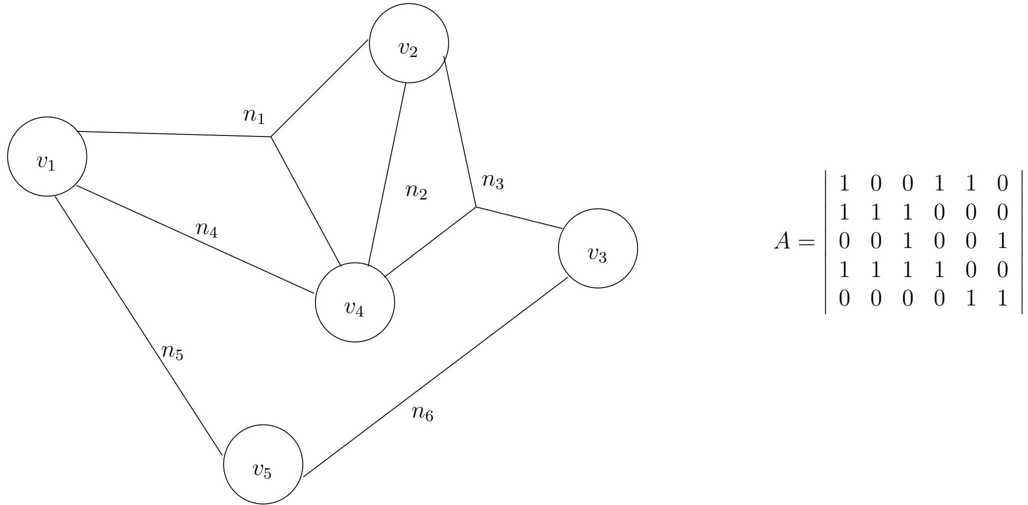

Figure 2.1: A hypergraph and its matrix representation

vertex v_j is defined as the number of nets v_j is a pin of.

2.1.2 Hypergraph Partitioning

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is a K-way vertex partition if each part $\mathcal{V}_k$ is non-empty, parts are pairwise disjoint, and the union of all $\mathcal{V}_k$ forms $\mathcal{V}$. The *weight* W_k of part V_k is the sum of the weights of the vertices forming $\mathcal{V}_k$. In Π, a net is said to connect a part if it has at least one pin in that part. The connectivity set Λ_j of a net n_j is the set of parts connected by n_j. The connectivity $\lambda_j = |\Lambda_j|$ of a net n_j is equal to the number of parts connected by n_j. If $\lambda_j = 1$, then n_j is an internal net. If $\lambda_j > 1$, then n_j is an external net and is said to be at cut. In Π, the weight W_k of a part $\mathcal{V}_k$ is equal to the sum of the weights of vertices in $\mathcal{V}_k$, i.e., $W_k = \sum_{v_i \in \mathcal{V}_k} w_i$.

The K-way hypergraph partitioning problem [1] is defined as finding a vertex partition Π for a given hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ such that a partitioning constraint is maintained while a partitioning objective is optimized. Although other options are possible, typically, the partitioning constraint is to maintain a given balance criterion on the part weights, and the partitioning objective is to minimize an objective function defined over the cut nets. The frequently used objective functions

include the cut-net metric

$$cut(\Pi) = \sum_{n_j \in \mathcal{N}_{\text{cut}}} c_j, \tag{2.1}$$

where $\mathcal{N}_{\text{cut}}$ is the set of cut nets and the connectivity-1 metric [21]

$$cut(\Pi) = \sum_{n_j \in \mathcal{N}_{\text{cut}}} c_j(\lambda_j - 1), \tag{2.2}$$

in which each cut net n_j contributes $c_j(\lambda_j - 1)$ to the cost $cut(\Pi)$ of partition Π. In this work, the connectivity-1 metric is used.

2.1.3 State of the art in sequential partitioning

The hypergraph partitioning problem is unfortunately a difficult problem in terms of time complexity. Finding the optimal solution is known to be NP-hard; hence, heuristic algorithms that run in polynomial time and generate sub-optimal solutions are utilized.

Initially, the problem was solved by partitioning the flat hypergraph, recursively bisecting until the number of desired partitions were reached. Kernighan-Lin (KL) based heuristics were used for the hypergraph partitioning because of their short run-times and good quality results. The KL algorithm is an iterative improvement heuristic originally proposed for graph bipartitioning [19]. The KL algorithm, starting from an initial bipartition, performs a number of passes until it finds a locally minimum partition. Each pass consists of a sequence of vertex swaps. The same swap strategy was applied to the hypergraph bipartitioning problem by Schweikert-Kernighan [23]. Fiduccia-Mattheyses (FM) [15] introduced a faster implementation of the KL algorithm for hypergraph partitioning.

In these early implementations, k-way hypergraph partitioning was performed

through *pairwise part refinement*. The 2-way FM/KL algorithms were run on the initial k-way partition multiple times, each time refining different pairs of parts to generate a final, k-way, refinement.

Later, two schools of thought on how k-way partitionings should be performed emerged. The *recursive bisection* approach is to only perform bipartitionings and to repeat the large V-cycle on the produced partitions separately to divide the hypergraph even further. The other is the *direct k-way* method, it generates k parts in the initial refinement phase and projects this to form a k-way partitioning directly.

Unfortunately, the performance of the FM algorithm applied on the flat hypergraph deteriorates for large and very sparse hypergraphs. Here, sparsity of graphs and hypergraphs refer to their average vertex degrees. To alleviate this problem, the multi-level partitioning paradigm was proposed. In this paradigm, the flat hypergraph is not partitioned straight away; instead, it goes through a three-step procedure, consisting of the *coarsening*, *initial partitioning* and *refinement* phases. This three-step procedure is also referred to as the *V-Cycle*.

Broadly speaking, the multilevel refinement paradigm operates incrementally. At each level in the coarsening phase, the input hypergraph is "coarsened"; a new, smaller hypergraph is generated from the input hypergraph through the formation of supervertices. The coarsest hypergraph undergoes initial partitioning, the formation multiple parts. This partition is projected, level-by-level, onto successively finer hypergraphs, undergoing refinement at each level to improve the cut.(Figure 2.2)

2.1.3.1 Coarsening

At each level of the coarsening phase, the vertices of the hypergraph are merged to form *supervertices*. Each supervertex has a weight equal to the sum of its constituent vertices and is connected to the union of the nets that the constituent vertices were connected to. A net that is exclusively connected to the constituent vertices, *internal nets*, of the supervertex vanishes from the next-level hypergraph.

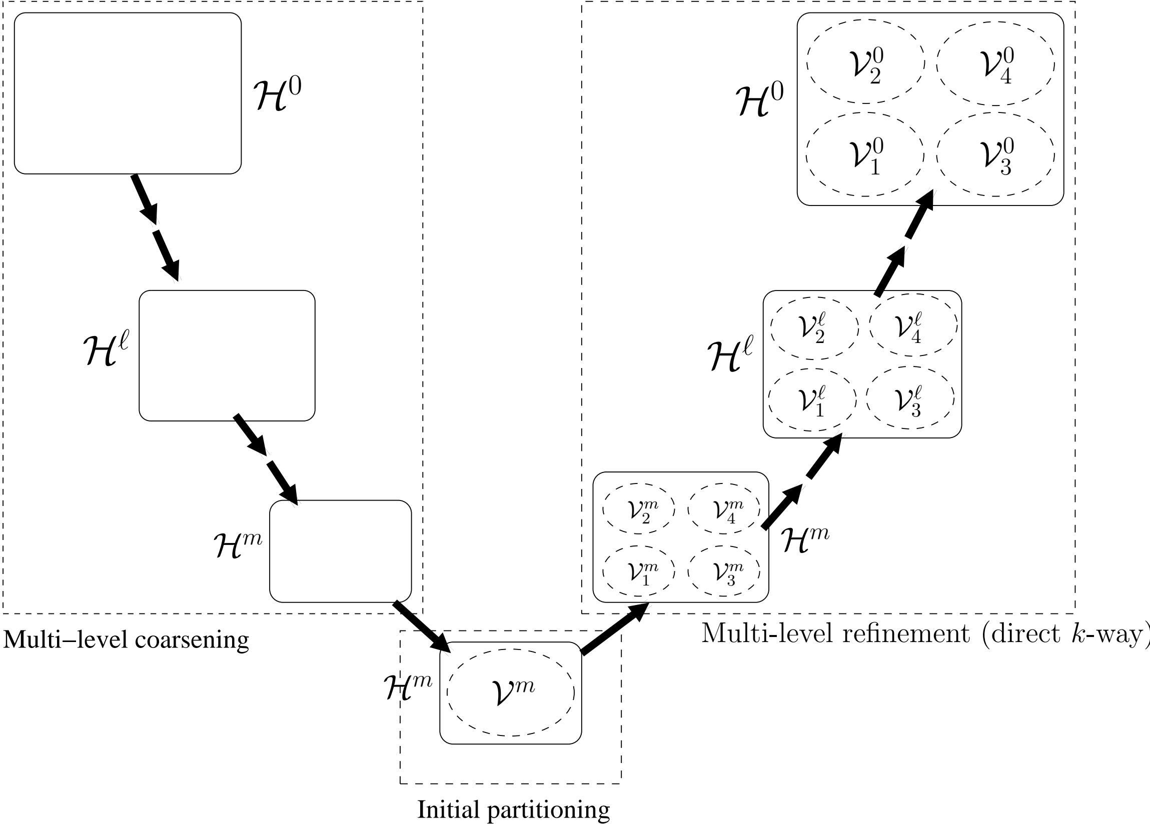

Figure 2.2: Sequential multi-level hypergraph partitioning

The idea behind this is twofold: to reduce the number of vertices that need to be processed and increase the degrees of the vertices. The first translates into a significant gain in processing time due to the nature of the heuristics; the second allows vertices to represent a more global view of the hypergraph allowing more informed vertex moves.

The decision of which vertices to merge is made using one of several different match heuristics. Heavy connectivity matching (HCM) is one that performs well. This heuristic tries to merge vertices based on the total cost of common nets between them; two vertices that share many nets with a high cost are considered good candidates to merge. In some implementations vertices are only allowed to make one merge in each pass, other implementations allow restricted multiple merges. These two approaches are termed *matching-* and *agglomerative-based* clustering, respectively.

Each level of the coarsening phase generates a new, smaller hypergraph. Coarsening terminates when the number of vertices in this hypergraph is low enough.

2.1.3.2 Initial partitioning

In the initial partitioning phase, the coarsest hypergraph is partitioned. As the coarsest hypergraph is small, methods that give good results but are infeasible with larger hypergraphs can be applied. Furthermore, as the problem size is small, algorithms can be run multiple times and the best result can be taken.

2.1.3.3 Refinement

The refinement phase projects the partition of small, coarser hypergraphs to the larger and finer hypergraphs. This projection starts with the partitioning obtained from the initial partitioning phase. Each level in the coarsening phase has a corresponding level in the refinement phase. First, the hypergraph is uncoarsened: the effects of the corresponding level's coarsening are undone and

supervertices are broken down. Internal nets are restored. The constituent vertices are assigned to the part of their supervertex. Then a refinement heuristic, such as FM [15] or KL [19], is applied to improve the cut while maintaining the balance of the parts. The produced partition is used as the initial partition in the next refinement level; unless it was the final level of the refinement, in which case it is taken as the final partitioning.

2.2 Applications of Hypergraph partitioning

Hypergraph partitioning has been used in VLSI design since 1970s [23]. The application of hypergraph partitioning in parallel computing is started by the work of Catalyurek and Aykanat [6]. This work addresses 1D partitioning of sparse matrices for efficient parallelization of matrix vector multiplies. Later, Catalyurek and Aykanat [8, 9] and Vastenhouw and Bisseling [28] proposed hypergraph partitioning models for 2D partitioning of sparse matrices. In these models, the partitioning objective is to minimize the total volume of communication incurred due to the parallelization while avoiding computational imbalance in the processors. These matrix partitioning models are utilized in different applications which involve repeated matrix-vector multiplies, such as computation of response time densities in large Markov models [14] and restoration of blurred images [27].

In the parallel computing domain, hypergraph-partitioning-based models employing objective functions other than minimizing the total volume of communication exist. For example, Aykanat, Pinar and Catalyurek [2] develop models for permuting sparse rectangular matrices into singly-bordered block diagonal form for efficient coarse-grain parallelization of linear programming, LU factorization and QP factorization problems. Their models try to minimize the size of the border, which corresponds to minimizing the overhead of the coordination task, while providing load balance over the diagonal block sizes and thus on computational loads of processors. Another example is the communication hypergraph model proposed by Ucar and Aykanat [26] for considering message latency overhead in parallel sparse matrix vector multiples based on 1D matrix partitioning.

Besides matrix partitioning, hypergraph partitioning models have also been proposed for use in other parallel and distributed computing applications. These include workload partitioning in data aggregation [10], image-space-parallel direct volume rendering [5], and scheduling file-sharing tasks in heterogeneous master-slave computing environments [20],[18].

Finally, we should note that hypergraph partitioning also finds application in problems outside the parallel computing domain such as road network clustering for efficient query processing [12], pattern-based data clustering [22], reducing software development and maintenance costs [4], topic identification in text databases [11] and for processing spatial join operations [24].

2.3 Related work

2.3.1 Sequential Hypergraph Partitioning

Although hypergraph partitioning is widely used in both academia and industry, the number of publicly available tools is quite limited. In fact, there are only two sequential hypergraph partitioning tools that we are aware of: hMeTiS [17], PaToH [6].

hMeTiS [17] is the earliest hypergraph partitioning tool, published in 1998 by Karypis and Kumar. It contains algorithms for both recursive bisection-based and direct K-way partitioning. The objective functions that can be optimized using this tool are the cut-net and sum of external degrees metrics. The tool has support for partitioning hypergraphs with fixed vertices.

PaToH [6] is published in 1999 by Catalyurek and Aykanat. It is a multi-level, recursive bisection-based partitioning tool with support for multiple constraints and fixed vertices. The built-in objective functions are the cut-net and connectivity-1 cost metrics. A high number of heuristics for coarsening, initial partitioning and refinement phases are readily available in the tool for use by the end users.

2.3.2 Parallel Hypergraph Partitioning

Unlike sequential partitioners, parallel hypergraph partitioners are under active development. To our knowledge there are two available tools: par*k*way and the hypergraph partitioning component of the Zoltan toolkit. Both implement a parallel version of the multi-level partitioning paradigm to generate their results.

2.3.2.1 Par*k*way

Par*k*way is the hypergraph partitioning tool developed by Trifunovic and Knottenbelt [25]. It makes use of a 1D decomposition of the hypergraph and uses fine-grain approaches in generating the result. It utilizes a novel hashing technique that assures an even distribution of the data generated during the various stages of computation. Vertices are replicated at processor boundaries and communication is performed in batch format. It performs direct k-way partitionings.

2.3.2.2 Zoltan

Zoltan is an open-source library of parallel partitioning and load balancing methods [13]. A component of this library is a parallel hypergraph partitioning tool that makes use of a 2D distribution of the hypergraph. It is coarser grained when compared to Par*k*way and makes use of the 2D distribution of the data to communicate more effectively. To generate k-way partitionings Zoltan makes use of the recursive bisection paradigm, but modifies it so that each part is further processed by different processor sets.

Chapter 3

Parallel Hypergraph Partitioning

3.1 Reasons for parallelizing

As stated in previous chapter, the current methods for hypergraph partitioning are able to partition a given hypergraph up to a certain size. Beyond this limit, the partitioning costs become overwhelming as the data structures used in the partitioning no longer fit into the memory and page swapping is performed by the operating system. As the operations performed are usually dependent on the ready availability of the hypergraph, massive hits in processing speed can be observed. While today RAM is cheap and bountiful, computers with large processing capabilities that can handle large amounts of RAM are expensive and scarce. Parallel hypergraph partitioners provide a more affordable solution in partitioning larger hypergraphs.

3.2 Difficulties in parallelization

Unfortunately, hypergraph partitioning is not an easily parallelizable task. The main reason for this is the same that led to the parallelization: the traditional methods of processing require a large portion of the hypergraph to be available.

Successful coarsening algorithms need to examine individual vertices along with their entire neighbor sets; without that information the algorithms cannot make good decisions. Similarly, the refinement algorithms make decisions with the information contained in the entire inter-part boundary. All of the data that is required to run the algorithms have to be made available, if the data is in the other processor's domain it has to be fetched first.

This data distribution problem is compounded by the fact that the operations performed in traditional hypergraph partitioning make many small updates instead of few large ones. The coarsening methods look at all the vertices, one-by-one, and decide which of its neighbors it should be matched to. The refinement algorithms similarly look at all the vertices at the part boundaries separately and compute the effects of moving one vertex to the other part with respect to the cut. If these updates should occur across processors boundaries communication is required. The ideal solution to this problem would be to modify the partitioning algorithms.

These two issues need to be resolved in a successful parallel hypergraph partitioning tool.

3.3 Data distribution

As said, one of the issues greatly affecting performance is the distribution of the hypergraph that is to be processed. The naïve solution to this would be to divide the hypergraph directly and to communicate among processors whenever a computation requires data that is not available. In practice, this would require large amounts of communication and make the partitioning time unfeasibly long. Therefore, more refined approaches to distributing the hypergraph among the processors are needed.

3.3.1 One-dimensional decomposition

A more advanced solution would be to replicate the data at the processor boundaries, thereby avoiding excessive communication when accessing the data. Communication would only be necessary if, either the properties of the elements at the boundary changes, or the data is redistributed and the boundary is no longer valid.

This is the approach taken in Par*k*way [25]. First, nets are distributed among processors; then, vertices belonging to those nets are assigned to those processors. When nets belonging to different processors compete for the ownership of a vertex, all processors are given the data required to use that vertex in calculations, but one processor is assigned the responsibility to allow access to that vertex. The algorithms used in processing the hypergraph are similar to their sequential versions, with some programming improvements to reduce the communication further.

This simplistic approach taken in the partitioning algorithms leads to other issues. When communication occurs, it takes place in the form of many short bursts aimed at the entire processor set. As the underlying architecture is message passing, a large start-up overhead is incurred at each transmission. Additionally, as the messages may be between any pair of processors, determined more or less randomly, it is hard to predict the time spent on communication. Finally, the run time of the partitioner depends heavily on the initial distribution of the hypergraph. If the distribution is bad, the number of replicated vertices will increase and consequently the time spent on coordinating network flow between the processors will increases. To avoid this the authors of Par*k*way [25] assume that the initial data is distributed contiguously.

3.3.2 Two-dimensional decomposition

We, along with Zoltan [13], make use of a two-dimensional (2D) distribution of the data. This means that, unlike Par*k*way [25], both the nets and the vertices can

span multiple processors. This has the advantage that, when using a mesh-like processor arrangement, most communication occurs either in rows or columns. This regularizes the communication pattern and thus makes the computational and communication loads of the individual processors more even. Additionally, we have adjusted our refinement algorithms so that the communication occurs less frequently but with more data, thus reducing the total time penalties due to communication start-up.

In our application, the data is distributed among the processors in a structured manner. If the processor arrangement is seen as a mesh, the nets and vertices of the hypergraph are distributed among the processor columns and rows respectively. This means that all processors in the same column of the mesh are responsible of a particular set of nets and all the computations using those nets will be performed by those processors exclusively. The same concept is applied to the mesh's rows and the hypergraph vertices. If the hypergraph is seen as a matrix as described in Section 2.1.1.1, the assignment of nets and vertices to processors can be seen as a $(\sqrt{k} \times \sqrt{k})$-way rectilinear partitioning of the hypergraph matrix, $A_{\mathcal{H}}$ into k matrix blocks. Here, each matrix block (α, β) is stored and processed by the α-th processor in the β-th row. In order to achieve balance in storage, nets and vertices are distributed among the processor columns and rows in a scatter (round-robin) manner.

Figure 3.1 shows the storage of a hypergraph, with 5 vertices and 6 nets. The initial hypergraph, A undergoes a (2×2)-way rectilinear partitioning. Each of the 4 sub-matrices is assigned to a processor. Nets in the original hypergraph may translate into multiple nets on different processors, this may lead to the formation of single pin nets. Nets with same net ids in a processor column are still considered the same net; however, these nets are virtual in the sense that each processor knows only its portion of the net.

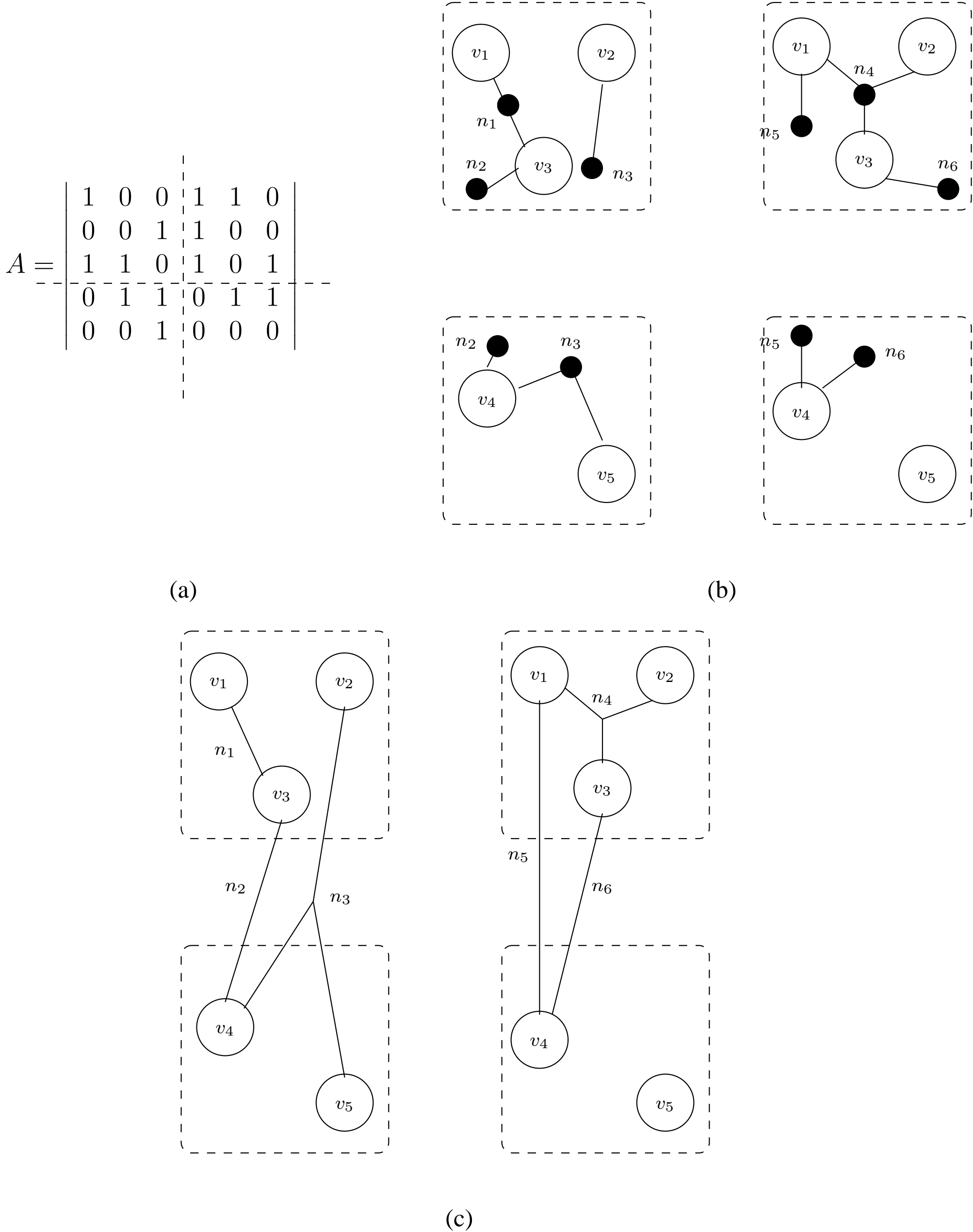

Figure 3.1: 2D data distribution: *(a)* The hypergraph matrix A is distributed to 4 processors. *(b)* The hypergraph distributed to 4 processors *(c)* The same hypergraph, with added virtual nets

3.4 Parallel partitioning methodology

Multilevel partitioning has been applied successfully in sequential hypergraph partitioning, resulting in increased partition qualities and reduced execution times. Hence, it is a good starting point for a parallel partitioner. However, care has to be taken that the applied algorithms are suitable for a parallel environment.

3.4.1 The elongated V-cycle

One way to look at the operation of ParPaToH is a way of elongating the "arms" of the V-cycle used in sequential hypergraph partitioning. Figure 3.2 shows the stages of a V-cycle. The large input hypergraph is coarsened until it reaches a size that can be partitioned comfortably by a sequential partitioner. This small hypergraph is replicated among the processors and a parallel initial partitioning procedure performed with the aid of a sequential hypergraph bipartitioner. The resulting k-way partition is used as a starting point for the distributed parallel refinement procedure.

It is interesting to note that since the sequential partitioner used during the initial partitioning phase is PaToH – a multilevel partitioner – the other parallel phases can be seen as additional coarsening and refinement steps tacked on to the V-cycle of PaToH. Another interesting issue is that although the parallel refinement is direct k-way, i.e., a k-way partition is projected in the refinement; the parallel initial refinement itself makes use of recursive bisection to generate that partition.

3.4.2 Parallel algorithm design considerations

As mentioned in Section 3.2, using the current sequential algorithms directly is not possible without incurring large performance penalties. It is, however, possible to sacrifice some of those algorithms' accuracy and use coarse-grain algorithms that can work with partial data and generate few, large updates instead of many

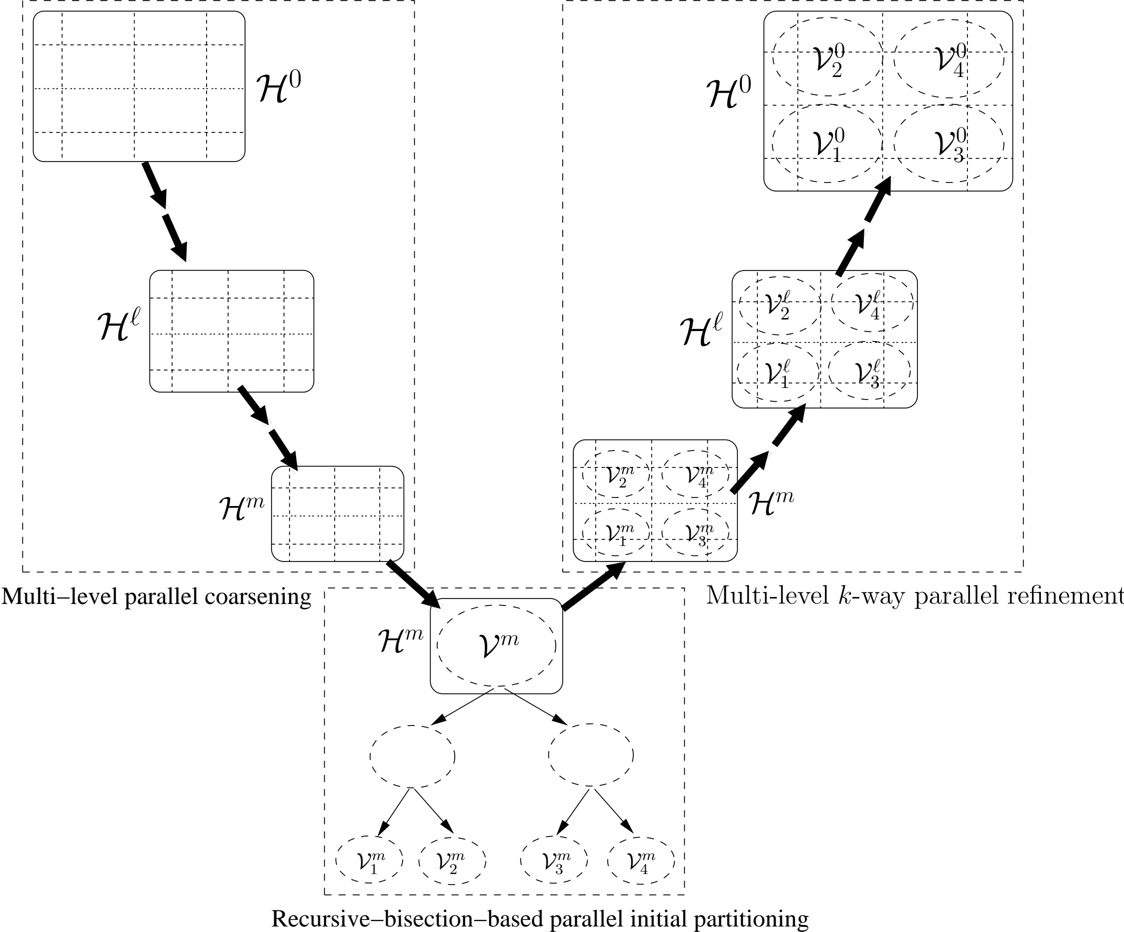

Figure 3.2: Parallel hypergraph partitioning

small ones. These properties are good for parallel computing; the first allows the processors to work without having to resort to expensive communication, whereas the second allows communication that can be performed efficiently. The details of the algorithms will be given in Chapter 4.

3.4.3 Restrictions of ParPaToH

Unfortunately, the ParPaToH implementation at the time of this writing is just an initial version and is far from perfect. While it performs the basic hypergraph partitioning operation, it lacks several features of sequential partitioners and has a few restrictions on the partitioning.

The first of these is a restriction on the number of parts a hypergraph can be divided into. The current implementation is restricted to a number that is an even numbered power of two $(4, 16, 64, 256 \ldots)$. This is usually not a problem as clusters generally are built in powers of two to allow hypercube-like communication, and it is not uncommon for them to have a square number of nodes to allow communication in a balanced grid formation.

Another one is a requirement on the number of processors. Currently, the number of processors performing the partitioning must be equal to the number of parts. Fortunately, this case is quite common in the parallel computing domain, where hypergraph partitioning is used to distribute data among processors and the parts represent the data elements a processor should be assigned.

Finally, it has to be said that none of the "advanced" features of sequential hypergraph partitioners are present. This means that there is no support for fixed vertices, uneven part sizes, multiple versions of algorithms that can be used during that partitioning or the ability to perform dynamic re-partitionings on already performed data. This list is by no means complete.

Chapter 4

Implementation

In this chapter, the implementation details of ParPaToH are described. Unless otherwise specified, the phase names (such as "refinement" or "matching") in this chapter refer to their parallel versions. The terms *row vertices* or *row set* refer to the set of hypergraph vertices that are assigned to a same row of processors. Similarly, the term *column nets* or *column set* refer to the set of nets that are assigned to a single processor column. The term *local hypergraph* refers to the sub-hypergraph formed by the row vertices and column nets of a processor. The term *row hypergraph* refers to the vertex-induced sub-hypergraph formed by the vertices in a row set. See Figure 4.1 for a graphical representation.

4.1 Coarsening

During the coarsening phase, the input hypergraph is gradually made smaller, using a HCM-like matching algorithm to determine the merging candidates. After the merge is performed on suitable vertices, a "crossing-over" operation takes place. This operation exchanges a half of a processor row's row vertices with the half of another processor row's row vertices. This exchange prevents the vertex space in which he matching is performed from being restricted to a $\sqrt{k}$-th of the entire vertex set, and can be seen as bringing in fresh candidates for merging.

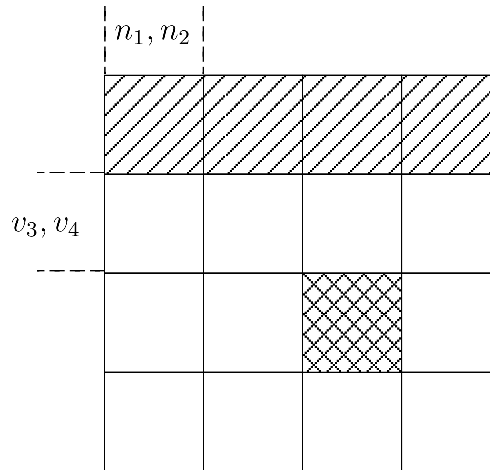

Figure 4.1: Explanation of terms: The doubly-hatched region forms a local hypergraph, the singly-hatched region a row hypergraph. The nets of the first processor column, n_1 and n_2 form that column's net set.The vertices of the second processor column, v_3 and v_4 form that row's vertex set.

Coarsening is completed in several levels, each containing the matching, merging and crossing-over steps.

4.1.1 Matching

The matching operation is restricted when compared to a sequential matching. Vertices are only allowed to merge within their row set. Furthermore, the matching decisions are based on the lossy agglomeration of $\sqrt{k}$ local decisions.

4.1.1.1 Local Affinity

The only matching information that can be computed in our setup, without having to resort to communication, is the similarity of two vertices with respect to the processor's column set. We name the numeric representation of similarity, *affinity*. The affinity is computed by running the HCM algorithm on the local hypergraph.

The affinity itself is a poor measurement of the overall similarity of two vertices. However, if we gather all of the affinity information from a processor row and add the affinities together we get the sequential HCM values for the row hypergraph. Unfortunately, gathering the information in a row is an expensive operation: Each pair of neighbor vertices contribute to the communication. Furthermore, there is no pattern in the communication, as a processor cannot, on its own, determine the connectivity of two vertices on the rows it does not know about.

4.1.1.2 Row affinity

The solution to this issue lies in the observation that only the highest value for a vertex is used when determining the matching. The lower values are only needed if a vertex has highest affinity to a vertex that is already matched, in which case it should be matched with the second highest, and if that fails with the third highest etc. If there was a way to determine and send the only the affinity values that are processed, the amount of communication could be dramatically reduced. While this is a chicken-and-egg problem (the highest row affinity values cannot be known without gathering all of the local values and the local values cannot be gathered efficiently without knowing the highest row affinity values) a heuristic approach could be used in determining the row affinity.

The proposeda approach is as follows: for each vertex in the local hypergraph, the neighbors and their affinities are calculated. The processors in the last column create an affinity array and insert the q most affectionate neighbors and their affinities (q is an external, user-customizable parameter). If a vertex has less than q neighbors, then it stores all of its neighbors' affinities. Then, these arrays are passed west on the processor rows, toward the first columns, concurrently. Once a processor receives the array from the east, it may make changes on it before passing it on. There are two types of changes that can be made. If, for a given vertex, there is a more affectionate neighbor not appearing in the array, the least affectionate of those in the array is replaced with the local one. If a neighbor in the local hypergraph also appears in the array, the affinity value in the

array belonging to that neighbor is incremented by the local amount. When the affinity arrays arrive at the first processor column of the processor mesh and those processors make their modifications, the final form of the arrays are redistributed to the rows; merging starts.

This approach is not guaranteed to find the q most affectionate neighbors for each vertex in the row set. However, the resulting affinity array gives a heuristic overview of the information contained in the row.

4.1.2 Merging

Merging is performed using the information in the affinity array exclusively. All of the vertex pairs in the array are first sorted on the affinity. Then, starting from the most affectionate pair, each vertex is matched to another. A vertex is only allowed to match an unmatched vertex. If a vertex cannot be matched, because it has no (remaining) neighbors, it stays unmatched. A new hypergraph is created with the reduced vertex size. Note that the old hypergraph is not discarded as it will be used in the corresponding level of the refinement phase.

4.1.3 Crossing-over

If the coarsening did not include a crossing-over step, the matching would be severely constrained. The only matches would be within the row set, and global information would be lost. To prevent this, after each merging step, one-half of the row vertices are exchanged. This exchange also includes information about the vertices weight and net connectivity information. The processors to which half of the vertices are sent and from which the new vertices are received change with each coarsening level. Figure 4.2 demonstrates how they are chosen so that after a few coarsening levels, the vertices are well distributed.

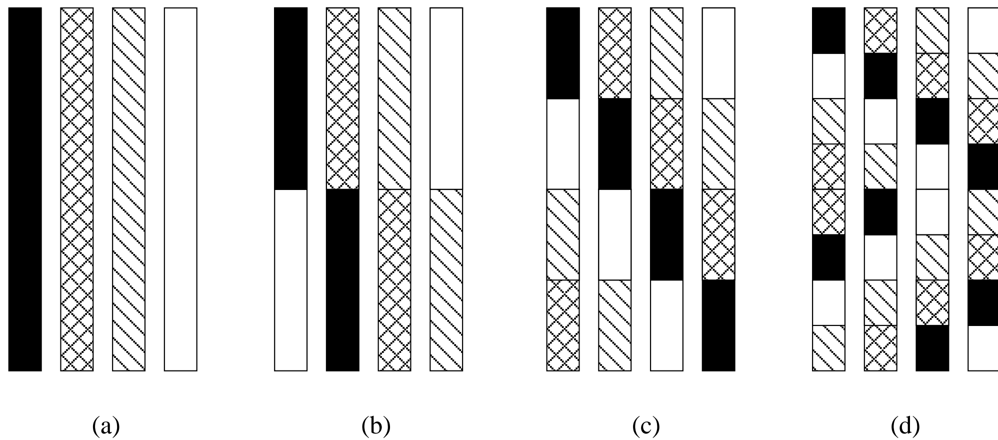

Figure 4.2: Crossing-over in action: At each crossing over half of the vertices are exchanged with a different processor. After $log_2\sqrt{k}$ cross-overs the initial vertex distribution is dispersed over all processors

4.1.4 The termination of coarsening phase

The coarsening phase stops on two conditions: either the hypergraph is small enough or the size of the hypergraph has not shrunk significantly in the current level. Both conditions require the size of the hypergraph to be known after each coarsening level. This is accomplished by communicating and summing up the sizes of the hypergraph in a processor column.

4.2 Initial Partitioning

The initial partitioning is performed on the coarsest hypergraph. As the hypergraph is small enough, it can be gathered and replicated onto all processors without causing much difficulty. At this point, a sequential k-way hypergraph partitioner could be run on the small hypergraph on all nodes and the partitioning with the best cut among all the ones produced could be declared as the initial partitioning. While this is a good method, which was actually used during earlier versions of ParPaToH, a modest increase in performance can be obtained by a rather coarse-grain parallelization of the initial partitioning phase.

4.2.1 Parallel initial partitioning

The parallel version of the hypergraph partitioning can be seen as a naïve parallelization of the recursive bisection schema.

The small hypergraph is first gathered and replicated on all processors in the system. This can be simply and efficiently accomplished by letting each processor gather its respective row hypergraph; communicating along the rows of the processor mesh to gather parts form all of the processors in the row. Then $\sqrt{k}$ columnwise all-reduce type communications are performed in parallel to gather the individual row hypergraphs. These are merged together to form the global, coarsest hypergraph.

Once the coarsest hypergraph is replicated, all of the processors perform a bisection on the hypergraph. The result is a partition array of size equal to the number of vertices in the coarsest hypergraph. It will be referred to here as the *part array*. It denotes which part each vertex belongs to. Note that we are performing a bisection on the part array. The part imbalance values of the partition are also calculated based on that bisection. These cuts and imbalance values are distributed among all processors and the processor with the best cut, whose imbalance is below a threshold, is chosen to be the source for the partitioning. That processor broadcasts its part array to the others. Until this point, all of the processors are doing the same work.

After the initial bipartitioning is complete, half of the processors take the responsibility for further partitioning one of the two parts and the other half of the processors take the other part. Each processor group generates a sub hypergraph induced by one of the two vertex parts of the coarsest hypergraph; each of the two processor groups creates an even smaller sub-hypergraph that contains only vertices belonging to one of the parts. Each of the halves then bisect their sub-hypergraph once more, obtaining 2 more parts.

Again the best partitioning is chosen, the best of each of the processor groups are again divided into two, which further perform a bisection on the new, even

smaller vertex-part-induced sub-sub-hypergraph. This cycle continues until the bisectionings generate k parts in total. As k is a power of 2, the number of cycles is exactly log_2k. Once the final partitioning is made, and the best cut found, the many small binary part arrays are merged into a single, final, k-ary part array, which is the end result of the initial partitioning phase.

It may not be obvious at first why the parallel version is faster than that sequential version. The parallel implementation not only calls the sequential partitioner multiple times, but also has communication between the processors. The secret in the acceleration lies in the fact that bisections can be performed significantly faster than k-way partitionings. The decrease in number of target parts, coupled with the fact that the sub-hypergraphs get geometrically smaller makes the parallel version more efficient.

4.3 Refinement

Refinement, like its sequential version, is a phase composed of multiple levels and each level in the coarsening phase has its refinement counterpart. Each refinement level is composed of two stages: the uncoarsening stage, where the effects of the corresponding coarsening are undone, and the pairwise-refinement stage, where the cut is tried to be reduced even further.

4.3.1 Uncoarsening

The act of coarsening, while making the hypergraph smaller, causes a loss of information. This information needs to be recovered before the refinement can improve the cut. Additionally, as the ownership of the vertices change after the uncoarsening, the part array belonging to the current vertex set has to be constructed.

4.3.1.1 Decross-over

While the initial crossing-over required a large amount of communication, the only information needed to perform a decross-over at a processor is the part array of vertices that were "sent away" during the corresponding crossing-over. This information can be obtained by communicating with the processor in the row that currently owns the vertex. The current owner is known as the exchange was performed in a structured manner and it is trivial to reverse the order.

4.3.1.2 Unmerging vertices

Once the partial remote part array has been obtained, this can be combined with half of the local part array to derive the part array for the next level. As the one level coarser hypergraph was not discarded during the coarsening, a hypergraph that contains the unmerged form of the vertices already exists. The only act that needs to be performed is the propagation of the part information to the larger hypergraph, by assigning the supervertices' part numbers to its constituent vertices.

4.3.2 Improving the cut

Once the hypergraph is uncoarsened, the cut can be further improved with the new information. Our algorithms used are quite unlike their contemporary serial counterparts, which either only perform 2-way (the recursive bisection approach), or perform a simultaneous k-way refinement (the direct k-way approach). The recursive-bisection scheme is faster, but needs the entire V-cycle to be repeated $k-1$ times, this would be expensive in our parallel case. k-way refinement is slower than its 2-way counterpart but has the advantage that the entire partitioning can be generated in one V-cycle. Both have the disadvantage of performing multiple small updates.

We believe that the solution to the parallel refinement problem does not lie in

these algorithms optimized for sequential execution. Instead, we propose that the "old way" of refinement, the pairwise refinement approach as described in [19], allows a coarse-grain parallel solution. The main advantage of this method is that it allows for independent operations on multiple disjoint vertex sets. Our basic idea is the same as the original approach: to perform multiple pairwise refinements, with different pairs, until all (or most) of the possible pairs are processed. Since we can perform multiple pairwise refinements in parallel, we believe that the original reason of this method's dismissal, its slowness, can be overcome.

Unfortunately, the distributed nature of the hypergraph prevents us from using the old method directly. The main issue is that the local hypergraph is not aware of the state of the nets in the other processors in the same mesh column. Thus freeing a net from the cut in the local view may not translate into a gain in the global view. However, the additional knowledge of the nets' connectivities to different parts on the other processors allows us to reduce this partial pairwise refinement problem to a hypergraph refinement with fixed vertices problem.

To to enable this reduction, two additional, fixed vertices whose part numbers are the parts to be refined are added to the hypergraph. These serve as anchors to the nets whose gains do not improve in the global view. If a new hypergraph is generated, where nets that are remotely attached to the two parts to be refined are made to have an additional pin to the relevant anchors, this hypergraph, once partitioned, will generate partitions that incorporate the global information in the refinement.

We implement a reduction to this model, but apply it on row hypergraphs to make the operation coarser grained. We make use of a net-to-part connectivity matrix to extract the global net-part connectivity information.

4.3.2.1 The refinement cycle

Our refinement algorithm operates on a local copy of the row hypergraph. While the collection and replication of the row hypergraph is admittedly an expensive

procedure, it only needs to be collected once per refinement level and can be used for all of the refinement cycles in the current level.

In each refinement cycle, $k/2$ non-overlapping pairwise refinement operations are performed on the hypergraph simultaneously (i.e., all vertices are processed exactly once during each cycle). The refinement cycle is further divided into $\sqrt{k}$ sub-cycles. In each sub-cycle all of the row hypergraphs undergo $\sqrt{k}/2$ simultaneous pairwise refinements, so that after the $\sqrt{k}$ sub-cycles all of the row hypergraphs are refined with $k/2$ pairwise refinements in total. Since the union of all of the row hypergraphs form the entire hypergraph, after the completion of the sub-cycles the entire hypergraph is refined with respect to the $k/2$ parts.

4.3.2.2 Scheduling

Each refinement cycle starts with the determination of a pairwise refinement schedule. The schedule first determines the $k/2$ pairs. Then it assigns the pairs in a round-robin manner to the $\sqrt{k}$ processor rows. Finally the $\sqrt{k}/2$ pairs of each row are assigned, again in a round-robin manner, to the processors of each row. The resulting schedule is that the first half of each processor row is assigned one pair and the remaining are assigned none. Currently, the pairing of parts and their distribution are random, using a shared seed.

This part-to-processor assignment is only valid for the first refinement sub-cycle. After each sub-cycle the schedule is "rotated", so that each processor is assigned to the parts that were processed by the neighbor processor in the previous row (the northern neighbor). See Figure 4.3 for a visualization of this rotation.

4.3.2.3 The connectivity matrix and global part weights

The net-to-part connectivity matrix, δ, is a large, two-dimensional, matrix of size $k \times N$ that contains the connectivity of all nets to parts. If, in the global hypergraph, the net n_i contains m vertices that are in part j, then $\delta_{j,i} = m$. This

1/3	4/7		
2/11	15/16		
6/9	10/13		
8/14	5/12		

(a)

8/14	5/12		
1/3	4/7		
2/11	15/16		
6/9	10/13		

(b)

6/9	10/13		
8/14	5/12		
1/3	4/7		
2/11	15/16		

(c)

2/11	15/16		
6/9	10/13		
8/14	5/12		
1/3	4/7		

(d)

Figure 4.3: Rotating the refinement schedule: After each sub-cycle the parts to be pairwise refined are rotated within the column

matrix is used to provide the global view when performing pairwise refinement on the row hypergraphs.

Before the matrix is created, the refinement schedule needs to be determined. To create the matrix the following steps are taken: First the local δ matrix is computed. This generates a $k \times N_c$ sized matrix, where N_c is the numbers of nets in the processor column c. This matrix is sent row-by-row to the processor in the current column whose row, according to the refinement schedule, will process that part. Note that it is possible for a processor to send to itself. For each part its row is responsible for, the processor forms a merged δ matrix by superimposing and adding the received δ matrices. At the end of the operation, each processor contains the δ matrices of $\sqrt{k}$ parts, since there are $\sqrt{k}$ pairs to be refined. However, these δ matrices are only partial. They only contain information about the nets in the current processor column. A second communication is performed and each processor sends the partial data matrices to the processor who will process the part; again, using the refinement schedule. In the current implementation, each processor will either refine two parts or no parts at all. Hence, the number of rows gathered in the final step is at most two.

The generation of the global δ matrix (δ_g) is expensive, and has to be performed at the beginning of each cycle.

There is only one more action that needs to be performed before the actual pairwise refinement can begin. Each processor calculates the k part weights for the row hypergraph. Then, using an all-reduce operation, the part weights of the

entire hypergraph are obtained.

4.3.2.4 Pairwise refinement on the row hypergraph

Finally, the pairwise refinement can be run on the parts of the row hypergraph. Each processor only refines the row hypergraphs vertices that belong to the parts assigned to that processor by the refinement schedule. During this refinement the information contained in the δ_g matrix is used to obtain global information. The actual refinement is performed using PaToH's refinement routines. However, since those are unable to make use of the δ_g matrix, the problem has to be modified to a form that is understandable by them, the problem of fixed-vertex refinement.

The problem of performing pairwise refinements on the row hypergraphs (instead of the whole hypergraph) is that we loose information. The gain of moving one vertex to another part cannot be calculated accurately without knowing all neighbors of that vertex. By constructing a row δ matrix (δ_r) and comparing it to the δ_g matrix we can gain additional data. A δ_r matrix is a local version of the δ_g matrix; it only contains net-to-part connectivity values restricted to the row hypergraph's vertices and the parts to be refined by the current processor. As the row hypergraph is available in full, this can be calculated locally. As a pairwise refinement operation is performed, this δ_r matrix will have two rows, .

By knowing how many of a net's vertices belong to a specific part and comparing that value to its counterpart on the row hypergraph, we can determine if a move in the row hypergraph will result in a gain in the global hypergraph. If, for example, the net n_i is connected to part m with 3 vertices in the global hypergraph (i.e., the ith entry in the δ_g row belonging to part m is 3) but only 2 of them are in the row hypergraph (i.e., the ith entry in the δ_r matrix row belonging to part m is 2) any moves of those will not affect the cut. If all of the 3 vertices were available, however, moving all three away from part m would lead to a decrease in the cut.

Another problem that is caused by the local view is that the local part balances are not an indicator of the overall balance. This problem can be easily remedied

by taking the (previously calculated) global part weights into account during the execution of the refinement algorithms.

We solve both problems by a reduction of the problem to a hypergraph refinement with fixed vertices problem.

First, a part-induced sub-hypergraph of the row hypergraph is formed. This makes sure that no redundant work is performed by the algorithms and only the parts that are to be refined are processed. Two additional vertices are added to the new hypergraph, one for each part. These fixed to prevent their move and will act as anchors to nets whose vertices' moves will improve the cut on the row hypergraph, but will not make a difference in the global hypergraph. This anchoring will provide the routines with the global information by telling them that there is an unmovable vertex and that moving the remaining vertices away form the unmovable vertex will not make a difference. It has to be noted that, when nets are anchored to both parts there is no way of preventing that net from appearing in the global cut. Hence these nets can be removed, along with any vertices that are unique to them, from the new hypergraph.

Providing the global part weight information in the new hypergraph can be accomplished setting the two anchor vertices weights to the global part weights.

This new hypergraph can after these modifications be processed by the sequential hypergraph refinement routines. The routine implement the FM refinement algorithms, modified to make vertex fixing possible. The resulting part array is used to form a list of vertices that change parts. After each subsequent sub-cycle, the updates performed in that sub-cycle will be appended to this list. This list will be row broadcast at the end of the refinement cycle (so that the entire row is aware of the changes made to their vertices and can update their local and row hypergraphs).

4.3.2.5 Preparing for the next sub-cycle

Before the next sub-cycle can begin two pieces of information needs to be updated and passed to the processor that will process the currently processed parts.

The first of these is the updated δ_g matrix. It needs to be updated to reflect the changes that occurred in the refinement. More specifically, nets whose vertices are in different parts after the refinement need to be updated in the δ_g matrix. The second of the updates is the change in the global part weight due to the moving vertices. This is simply equal to the old part weights minus the weights of leaving vertices, plus the weights of the incoming vertices.

Once these two updates are computed they are transmitted to the processor that will next process the currently refined parts. Although "rotating" the δ_g matrix in this manner is expensive, the communication pattern is regular and point-to point.

4.3.2.6 Run-length encoding

Observation has shown that the δ_g matrix usually contains many zero elements. This is due to the fact that a net is usually only connected to a few of the k parts and hence the part connectivity for the remaining parts are zero. We have exploited this sparseness with a run-length-encoding algorithm to compress the data to be transmitted. This allows us to reduce the communication amount while constructing and rotating the δ_g matrix significantly. Size reductions up to 80% of the original size were observed during experiments.

4.3.2.7 Putting the parts together

At this point it would be wise to summarize a single refinement pass, in full, to prevent any confusion. Each refinement level has multiple refinement cycles, and each refinement cycle has $\sqrt{k}$ sub-cycles. Each refinement cycle refines all of the

k parts of the *global* hypergraph in a pairwise manner. In each refinement sub-cycle $k/2$ processors perform concurrently refinement operations on the disjoint part-induced *row* hypergraphs.

First, the row hypergraph is gathered and replicated. This is done only once in each level. A refinement cycle begins. A refinement schedule is formed. The δ_g matrix is formed and distributed to the processors that will use it in the first refinement sub-cycle. The global part weights are computed and replicated. A refinement sub-cycle begins. A δ_r matrix is calculated using the row hypergraph. A smaller sub-hypergraph is formed from the row hypergraph using the δ_g and δ_r matrices. That smaller sub-hypergraph's cut is improved using a refinement alogrithm that allows fixed vertices. The vertices that change part are noted and the part changes are updated in the δ_g matrix. The δ_g matrix, along with the part weights of the updated parts is rotated. The refinement sub-cycle ends. This sub-cycle is repeated k times. The previously noted vertex changes are broadcast to the processor row, the row and local hypergraph are updated. The refinement cycle ends, but is repeated a number of times to ensure the cut is thoroughly improved. A refinement level ends, the global hypergraph is uncoarsened, and this process repeats until the last level of uncoarsening has been improved upon.

The resulting final part array is gathered from the processors and returned.

Chapter 5

Experimental Results

Here, we present how our tool behaves in real life. This data was on a 16-processor Linux WareWulf Cluster using the LAM implementation of the MPI message passing interface. We examine the quality of the cuts generated by the tool and discuss its scalability to multiple processors.

5.1 Cut vs. time

Several batches of experiments were run on different hypergraphs, each illustrating the effects of modifying one of the parameters detailed below. The average of 10 runs each are reported in the tables below. The maximum allowable imbalance was given to be 10%; this was met in all cases by the partitioner. In order to examine the quality of the output generated by the tool in relation to the output generated by sequential partitioners, the cut values are given normalized to the cuts of a sequential partitioner, PaToH, running with default parameters.

12 different hypergraphs were partitioned, ranging from the quite small (big and ebp1) to quite large ones (cage 13 and stomach). All of the hypergraphs partitioned were derived from readily available symmetric-square matrices. Table 5.1 gives an overview of the processed hypergraphs. In the experiments, Cage

Table 5.1: Properties of the partitoned hypergraphs

Hypergraphs	Net & Vertex Count	Pin Size
Zhao	33861	166453
Big	13209	91465
Cage 11	39082	559722
Cage 12	130228	2032536
Cage 13	445315	7479343
Epb1	14734	95053
Epb2	25228	175027
Epb3	84617	7463625
G7jac050sc	14760	157990
K3plates	11107	378927
Mark3jac060	27449	170695
Olafu	16146	1015156
Stomach	213360	3021648

13 could not be partitioned 4-way due to its size, hence when results are given for 4-way partitionings it is left blank.

5.1.1 Tunable parameters

Our tool can be adjusted to generate a partition based on a cut/time trade-off. Better cuts are possible by spending more time on the refinement. We examine this parameter with different partition sizes. Another factor the affects the quality and the execution time is the maximum coarsening depth, how small the coarsest hypergraph should be made. We examine and discuss the depth. Finally, we look at the "crossing-over" and see how it affects the result.

5.1.1.1 Refinement count

The refinement count is the number of cycles at each refinement level. The results of partitioning hypergraph with differing refinement counts are given in Tables 5.2 and 5.3

Table 5.2: Effects of the refinement count on the cut size

	4-way partition			16-way partition		
Hypergraph	RC=1	RC=2	RC=4	RC=1	RC=4	RC=10
Zhao1	1.20	1.17	1.08	1.25	1.19	1.11
Big	1.18	1.16	1.12	1.07	1.06	1.05
Cage 11	1.10	1.02	0.99	1.36	1.22	1.10
Cage 12	1.18	1.07	0.99	1.59	1.27	1.09
Cage 13	-	-	-	1.82	1.36	1.11
Epb1	1.04	1.02	0.99	1.03	1.03	1.02
Epb2	1.41	1.18	1.14	1.20	1.13	1.10
Epb3	1.02	0.99	0.95	2.15	1.95	1.67
G7jac050sc	1.04	0.93	0.91	1.10	1.07	0.97
K3plates	0.99	0.99	0.98	1.10	1.08	1.08
Mark3jac060	1.70	1.34	1.17	1.47	1.33	1.19
Olafu	1.15	1.10	1.11	1.18	1.14	1.09
Stomach	1.28	1.12	1.09	1.62	1.32	1.16

Predictably, the more cycles are performed in this step, the smaller the resulting cut gets. However this gain comes with a significant increase in processing time. A ten-fold repetition of the refinement cycle improves the quality in the cut of Cage 13 to a value close to the sequential partitioner's cut, but this effect is only achieved through a tripling in the processing time.

5.1.1.2 Coarsening depth

The coarsening depth also affects the cut. Three different batches of 16-way partitionings were performed where the partitioner was told to coarsen until 100, 500 and 1000 vertices remained per part in the global hypergraph. The result of the runs are given in Tables 5.4 and 5.5.

It can be seen that the more information is given to the initial partitioner, the better the initial partitioning will be; this quality improvement propagates to the final cut. Furthermore, since the sequential partitioning routines are more efficient and effective than their parallel counterparts; using them in a highly coarse-grained parallelization still allows them to work effectively.

Table 5.3: The effects of the refinement count on the partitioning time in seconds

	4-way partition			16-way partition		
Hypergraph	RC=1	RC=2	RC=4	RC=1	RC=4	RC=10
Zhao1	2.66	3.44	4.51	1.57	1.62	2.83
Big	3.33	3.06	4.61	0.69	0.79	1.03
Cage 11	1.84	2.28	2.75	2.69	3.73	5.12
Cage 12	1.96	2.31	2.97	10.80	17.18	27.03
Cage 13	-	-	-	44.68	65.26	117.09
Epb1	4.58	5.17	6.13	0.82	0.83	1.69
Epb2	3.69	5.61	5.95	1.09	1.20	1.58
Epb3	4.61	5.34	6.43	2.51	3.33	4.84
G7jac050sc	3.02	3.31	4.95	0.89	1.21	1.30
K3plates	1.72	1.93	2.46	1.01	1.03	1.17
Mark3jac060	3.41	3.76	4.83	1.02	1.38	1.86
Olafu	1.26	1.45	1.79	1.66	1.79	2.13
Stomach	4.49	5.22	5.46	9.87	14.12	23.36

The down side of increasing the coarsening depth to higher values is the communication time that needs to be spent to distribute the coarsest hypergraph increases; this explains the increase in time for Zhao1 at 500 and 1000 vertices per part.

5.1.1.3 Effects of crossing-over

While crossing-over during the refinement "makes sense" quality-wise, does the gain in quality justify the large amount of communication performed? The answer to his turns out to be yes, the relatively large increase in quality justifies the extra time spent on data distribution and processing. This can be seen in Tables 5.6 and 5.7

In some cases, crossing-over actually reduces the total run time. Crossing-over allows a better refinement, increasing the number of formed supervetices per level. As the number of levels in the V-cycle depend on the size of the hypergraph, more effective merging decreases the total number of levels, which in turn tranlates into a reduction in execution times.

Table 5.4: Effects of coarsening depth (given in vertices/part) on the cut.

Hypergraph	100 v/p	500 v/p	1000 v/p
Zhao1	1.47	1.19	1.14
Big	1.36	1.06	0.97
Cage 11	1.34	1.22	1.17
Cage 12	1.40	1.27	1.27
Cage 13	1.75	1.36	1.33
Epb1	1.23	1.03	0.99
Epb2	1.33	1.13	1.05
Epb3	2.14	1.95	1.76
G7jac050sc	1.13	1.07	0.98
K3plates	1.30	1.08	0.97
Mark3jac060	2.07	1.33	1.25
Olafu	1.26	1.14	1.04
Stomach	1.64	1.32	1.26

5.1.2 Time dissection

Figure 5.1 shows the normalized times for the individual phases in a 16-way partitioning, a refinement count of 4 and a coarsening depths of 500 vertices par part. As can be seen, for larger hypergraphs (like Cage 13 and Stomach), refinement takes up a significant portion of the total partitioning time. In smaller hypergraphs (like big and epb1) the initial partitioning phase dominates the overall execution time.

5.2 Scaling

The smaller hypergraphs in the above list were partitioned 16-way on a single computer to examine the scalability of the tool. As the tool requires the presence of 4 instances for a 4-way partitioning, multiple processes were started on the computer. We believe that this is a valid test as the communication is simulated using fast memory transfers and thus can be neglected. Only small hypergraphs were used as we wanted to prevent paging; this would skew the results unfairly

Table 5.5: Effects of coarsening depth (given in vertices/part) on the execution times (in seconds)

Hypergraph	100 v/p	500 v/p	1000 v/p
Zhao1	2.64	1.62	1.73
Big	1.34	0.79	0.48
Cage 11	5.19	3.73	3.21
Cage 12	21.13	17.18	16.13
Cage 13	84.76	65.26	61.68
Epb1	1.06	0.83	0.55
Epb2	1.80	1.20	0.92
Epb3	4.37	3.33	2.85
G7jac050sc	1.57	1.21	0.67
K3plates	1.47	1.03	0.88
Mark3jac060	2.43	1.38	1.42
Olafu	1.56	1.79	1.56
Stomach	18.26	14.12	12.95

toward the parallel partitioner.

Table 5.8 shows that despite the network communication overhead the parallel partitioner performs better in its distributed form, yielding an execution time that is around a half of the single processor execution.

Table 5.6: Effects of crossing-over on the cut

	4-way partition		16-way partition	
Hypergraph	Crossing-Over	No Crossing-Over	Crossing-Over	No Crossing-Over
Zhao1	1.08	1.11	1.19	1.21
Big	1.12	1.09	1.06	1.07
Cage 11	0.99	1.04	1.22	1.22
Cage 12	0.99	0.99	1.27	1.34
Cage 13	-	-	1.36	1.52
Epb1	0.99	1.16	1.03	1.03
Epb2	1.14	1.21	1.13	1.17
Epb3	0.95	1.39	1.95	2.66
G7jac050sc	0.91	0.98	1.07	1.07
K3plates	0.98	0.99	1.08	1.08
Mark3jac060	1.17	1.34	1.33	1.61
Olafu	1.11	1.11	1.14	1.19
Stomach	1.09	1.27	1.32	1.64

Table 5.7: Effects of crossing-over on the execution times (in seconds)

	4-way partition		16-way partition	
Hypergraph	Crossing-Over	No Crossing-Over	Crossing-Over	No Crossing-Over
Zhao1	1.58	1.63	1.62	2.11
Big	0.53	0.61	0.79	0.78
Cage 11	3.34	3.05	3.73	3.33
Cage 12	18.95	18.55	17.18	16.10
Cage 13	-	-	65.26	62.61
Epb1	0.66	0.64	0.83	0.96
Epb2	1.15	1.44	1.20	1.28
Epb3	3.48	4.35	3.33	3.17
G7jac050sc	0.84	0.81	1.21	0.78
K3plates	0.65	0.67	1.03	1.07
Mark3jac060	1.38	1.57	1.38	1.66
Olafu	1.41	1.40	1.79	1.75
Stomach	20.58	20.86	14.12	16.05

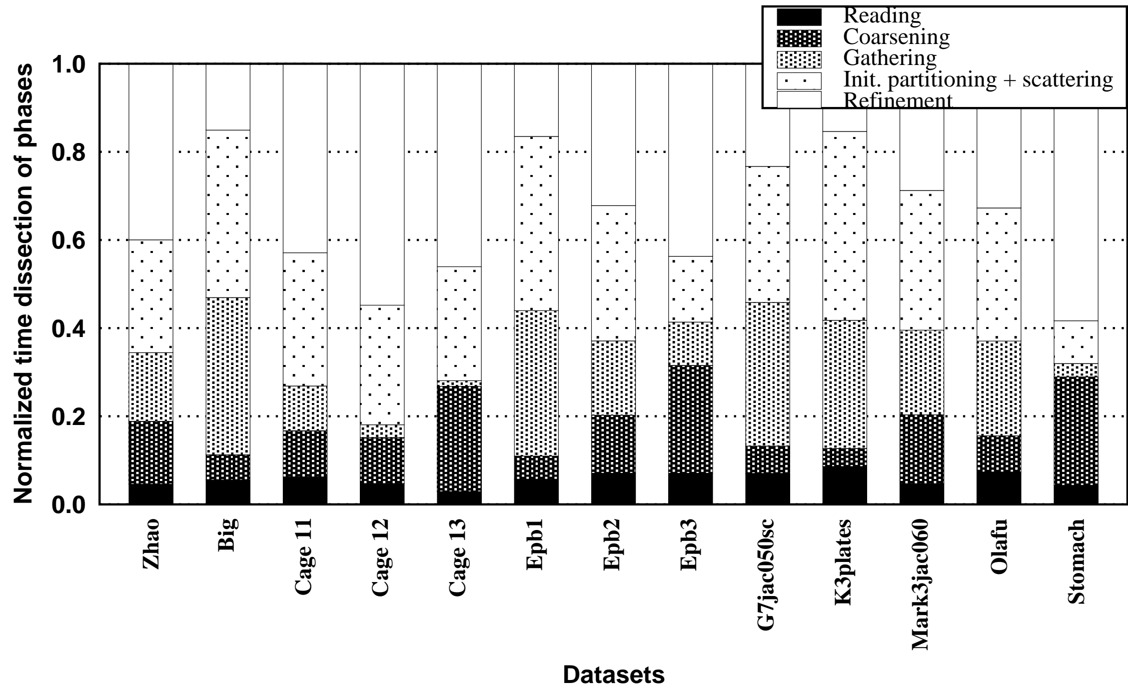

Figure 5.1: The time dissection of a V-cycle

Table 5.8: Scaling to multiple processors

Hypergraphs	1 Processor	4 Processors
Zhao1	3.36	1.58
Big	0.99	0.53
Cage 11	7.75	3.34
Epb1	1.13	0.66
Epb2	2.50	1.15
G7jac050sc	1.65	0.84
K3plates	1.34	0.65
Mark3jac060	2.92	1.38

Chapter 6

Discussions & Conclusions

6.1 Evaluation of the results

The inital results of our parallel hypergraph partitioner are promising. Even though the implementation is incomplete, given enough processing time, partitions of a quality similar to sequential partitioners can be achieved. We believe that this justifies the validity of our 2D data distribution and parallel pairwise refinement model.

6.2 Ideas for future work

There are many improvements that could be implemented in the future. Some of these are described below.

6.2.1 Generalization and improvement of the current code-base

Restrictions of the current code should be removed. This includes the restrictions on k, restrictions on the number of processors used to perform the cut and the processor arrangement. Furthermore, refinement should be modified to allow all of the processors to participate.

6.2.2 Merging identical nets

Merging identical nets is the act of finding multiple nets that share the same pins and merging them into a single net with a cost equal to the sum of the merged nets' costs. It can be used to decrease the number of nets during the coarsening, reducing the processing times for all of the subsequent operations.

6.2.3 Different refinement algorithm

We are unhappy with our refinement implementation. A custom made refinement could be written that does not incur the problem reduction overhead. Furthermore, such a custom algorithm could also be adjusted to allow a two-level gain scheme in which local information is taken into account along with global information, breaking ties in favor of a better local cut.

6.2.4 Other features

Features found in sequential partitioners could be added. These include vertex fixing, partitioning into uneven parts and dynamic repartitioning.

6.3 Conclusion

While the current version is far from being an ideal hypergraph partitioner, it provides a good starting point for the future. As its partitioning quality and speed improves it will provide an efficient and effective solution to the parallel hypergraph partitioning problem.

Bibliography

[1] C. J. Alpert and A. B. Kahng. Recent directions in netlist partitioning: A survey. *VLSI Journal*, 1995.

[2] C. Aykanat, A. Pinar, and Ü. V. Çatalyürek. Permuting sparse rectangular matrices into block-diagonal form. *SIAM J. Scientific Computing*, 2004.

[3] C. Berge. *Graphs and Hypergraphs.* North-Holland Publishing Company, 1973.

[4] R. H. Bisseling, J. Byrka, S. Cerav-Erbas, N. Gvozdenovic, and M. Lorenz. Partitioning a call graph. In *Second International Workshop on Combinatorial Scientific Computing (CSC05)*, Toulouse, France, June 2005.

[5] B. B. Cambazoglu and C. Aykanat. Hypergraph-partitioning-based remapping models for image-space-parallel direct volume rendering of unstructured grids. Revised manuscript submitted to *IEEE Transaction on Parallel and Distributed Systems* 2005.

[6] Ü. V. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel and Distributed Systems vol. 10, no. 7, pp. 673–693*, 1999.

[7] Ü. V. Çatalyürek and C. Aykanat. Patoh: A multilevel hypergraph partitioning tool, version 3.0. Bilkent University,Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/ umit/software.htm, 1999.

[8] U. V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 118, 2001.

[9] U. V. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, page 28, 2001.

[10] C. Chang, T. M. Kur, A. Sussman, U. V. Çatalyürek, and J. H. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[11] C. Clifton, R. Cooley, and J. Rennie. Topcat: Data mining for topic identification in a text corpus. *Transaction on Knowledge and Data Engineering*, 16(8):949–964, 2004.

[12] E. Demir, C. Aykanat, and B. B. Cambazoglu. Modeling and clustering of spatial networks for aggregate queries: A hypergraph approach. Submitted to *Information Systems* 2005.

[13] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and Ü. V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. To appear in *IEEE International Parallel & Distributed Processing Symposium*, 2006.

[14] N. J. Dingle, P. G. Harrison, and W. J. Knottenbelt. Uniformization and hypergraph partitioning for the distributed computation of response time densities in very large markov models. *J. Parallel Distrib. Comput.*, 2004.

[15] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 175–181, 1982.

[16] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in vlsi domain. *IEEE Trans. Very Large Scale Integration Systems*, 1999.

[17] G. Karypis and V. Kumar. hmetis: A hypergraph partitioning package. Technical report, Dept. of Computer Science, University of Minnesota, 1998.

[18] K. Kaya and C. Aykanat. Iterative-improvement-based heuristics for adaptive scheduling of tasks sharing files on heterogenous master-slace environments. Accepted for publication in IEEE Transaction on Parallel and Distributed Systems.

[19] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 1970.

[20] G. Khanna, N. Vydyanathan, T. Kurc, U. Catalyurek, P. Wyckoff, J. Saltz, and P. Sadayappan. A hypergraph partitioning based approach for scheduling of tasks with batch-shared io. In *Proceedings of Cluster Computing and Grid*, 2005.

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

[22] M. M. Ozdal and C. Aykanat. Hypergraph models and algorithms for data-pattern-based clustering. *Data Mining and Knowledge Discovery*, 9(1):29–57, 2004.

[23] D. G. Schweikert and B. W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proceedings of the 9th Workshop on Design Automation*, pages 57–62. ACM/IEEE, 1972.

[24] S. Shekhar, C.-T. Lu, S. Chawla, and S. Ravada. Efficient join-index-based spatial-join processing: A clustering approach. *IEEE Transactions on Knowledge and Data Engineering*, 14(6):1400–1421, 2002.

[25] A. Trifunovic and W. Knottenbelt. Par*k*way2.0: A Parallel Multilevel Hypergraph Partitioning Tool. In *Proc. 19th International Symposium on Computer and Information Sciences*, volume 3280 of *Lecture Notes in Computer Science*, pages 789–800. Springer, 2004.

[26] B. Ucar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies. *SIAM J. Scientific Computing*, 25(6):1837–1859, 2004.

[27] B. Ucar, C. Aykanat, M. Pinar, and T. Malas. Parallel image restoration using surrogate constraints methods. Submitted to *Journal of Parallel and Distributed Computing*.

[28] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.