University of London
Imperial College of Science, Technology and Medicine
Department of Computing

Parallel Algorithms for Hypergraph Partitioning

Aleksandar Trifunović

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of the University of London and
the Diploma of Imperial College, February 2006

Abstract

Near-optimal decomposition is central to the efficient solution of numerous problems in scientific computing and computer-aided design. In particular, intelligent *a priori* partitioning of input data can greatly improve the runtime and scalability of large-scale parallel computations. Discrete data structures such as graphs and hypergraphs are used to formalise such partitioning problems, with hypergraphs typically preferred for their greater expressiveness. Optimal graph and hypergraph partitioning are NP-complete problems; however, serial heuristic algorithms that run in low-order polynomial time have been studied extensively and good tool support exists. Yet, to date, only graph partitioning algorithms have been parallelised.

This thesis presents the first parallel hypergraph partitioning algorithms, enabling both partitioning of much larger hypergraphs, and computation of partitions with significantly reduced runtimes. In the multilevel approach which we adopt, the coarsening and refinement phases are performed in parallel while the initial partitioning phase is computed serially. During coarsening and refinement, a two-phase approach overcomes concurrency issues involved in distributing the serial algorithms, while a hash-based data distribution scheme maintains load balance. A theoretical analysis demonstrates our algorithms' asymptotic scalability.

The algorithms are implemented in the Par$k$way tool. Experiments on hypergraphs from several application domains validate our algorithms and scalability model in two ways. Very large hypergraphs (10^8 vertices) are partitioned with consistent improvements in partition quality (of up to 60%) over an approximate approach that uses a state-of-the-art parallel graph partitioning tool. The algorithms also exhibit good empirical scalability and speedups are observed over serial hypergraph partitioning tools, while maintaining competitive partition quality. An application case study of parallel PageRank computation is presented. Applying hypergraph models for one- and two-dimensional sparse matrix decomposition on a number of publicly-available web graphs results in a significant reduction in communication overhead and a halving of per-iteration runtime.

Acknowledgements

I would like to thank the following people:

- Dr. William Knottenbelt, my supervisor, for his help, guidance and a never-ending supply of enthusiasm.
- Keith Sephton and the London e-Science Centre for the use of the Viking Beowulf Linux cluster.
- The Engineering and Physical Sciences Research Council (EPSRC) for providing me with the funding to do my PhD.
- Members of the AESOP research group, for the pub lunches and evenings in the Holland club.
- The many people at the Department of Computing who made my PhD such a great experience.
- My family, for their love and support throughout my PhD.
- Ruth Coles, whose love and support has meant so much.

"I think there is a world market for maybe five computers."
Thomas Watson (1874-1956), Chairman of IBM, 1943

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Objectives

1.1.1 Partitioning in Science and Engineering

Partitioning is a process of decoupling that is fundamental across science and engineering. There are numerous applications of partitioning in both academia and industry. These include domain decomposition (in areas such as fluid dynamics, computational chemistry and astrophysics), load balancing for parallel computation, computer-aided design of very large digital circuits and data mining. For example, in the physical design process of very large digital circuits, effective partitioning is critical for achieving peak chip performance and reducing the cost and time of the design and manufacturing process.

Partitioning decomposes a set of interrelated objects into a set of subsets or *parts* to optimize a specified objective. In general, it is required that any two objects within the same part should be strongly related in some sense, whereas the converse should hold for any two objects found in different parts.

Partitioning also requires that the relationship between the objects is specified and the corresponding degree of association is quantified. These requirements are usually expressed using data structures such as graphs and hypergraphs.

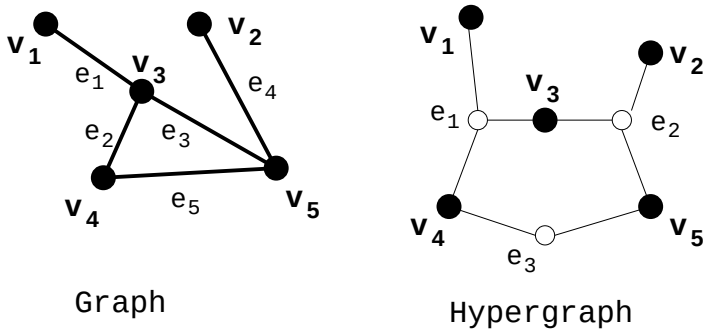

Figure 1.1. An example of a graph and a hypergraph.

1.1.2 Graphs and Hypergraphs

In a set-theoretic sense, graphs and hypergraphs are set systems. That is to say, they are collections of subsets of a given set of objects. The objects in this set are called vertices. The subsets within the collection are called edges when the collection is a graph, and hyperedges when it is a hypergraph. In a graph, the cardinality of any edge is two. On the other hand, the cardinality of a hyperedge may be any number greater than zero, up to and including the cardinality of the vertex set (cf. Figure 1.1).

Graphs and hypergraphs may also be considered as data structures that represent a set of related objects. The objects are represented by the vertices of the graph or the hypergraph. The existence of a relationship between objects is represented by an edge in the graph, or by a hyperedge in a hypergraph. A scalar weight is assigned to each edge or hyperedge to capture the degree of association between the vertices that the edge or the hyperedge connects.

Both graphs and hypergraphs may be partitioned to optimise some objective. Specifically, the set of vertices is partitioned, while the partitioning objective is usually defined over the set of edges or hyperedges. The next subsection discusses practical applications of graph and hypergraph partitioning.

Because there is no restriction on hyperedge cardinality, hypergraphs are more expressive than graphs in the context of modelling sets of related objects. That is to say, hypergraphs can capture a relationship between a group of objects,

whereas graphs can only capture binary relationships between objects.

1.1.3 Applications of Graph and Hypergraph Partitioning

Load Balancing of Parallel Computations

Load balancing (the assignment of work to processors) is a fundamental problem in parallel computing. The aim is to maximise parallel performance by ensuring that processor idle time and interprocessor communication time are as low as possible.

Load balancing is difficult in parallel applications that exhibit irregular computational structures. Irregularities are common in many applications, such as physical simulation and performance modelling. Since naïve problem decompositions are not satisfactory, load balancing is achieved through partitioning.

Load balancing may be static or dynamic. In applications with constant workloads, static load balancing is used as a pre-processing step to the parallel computation and partitioning is only done once. An example of static load balancing is sparse matrix partitioning prior to parallel sparse matrix–vector multiplication. Applications with changing workloads, for example due to evolution of the physical simulation, require dynamic load balancing. Here, partitioning may be done many times during the parallel computation. An example of such applications are adaptive finite element methods.

Traditionally, graphs have been used to model the computation and communication requirements of a parallel program. The vertices are used to model a subcomputation and the edges the communication requirement between the subcomputations.

However, graph models have a number of deficiencies; the most significant is that, in general, they can only *approximate* communication volume [Hen98]. As a result, hypergraph models for data partitioning (which quantify communication volume exactly) have been shown to be superior to graph models in many

cases of practical interest, including parallel sparse matrix–vector multiplication and computational modelling in areas such as electrical systems, biology, linear programming and nanotechnology [cA99, HBB02, UA04, VB05, DBH+05].

VLSI Computer-Aided Design

The task of designing integrated circuits is complex, as modern circuits have a very large number of components. Here, the hypergraph partitioning problem most commonly arises in the context of dividing a circuit specification into clusters of components (subcircuits), such that the cluster interconnect is minimised. Each subcircuit can then be assembled independently, speeding up the design and integration processes.

A circuit specification includes *cells*, which are pre-designed integrated circuit modules that implement a specific function and have input and output terminals. A *net* is a collection of input and output terminals connected together and is used to connect a group of cells together. Each connection between a cell and a net occurs at a *pin*. Cell connectivity information for the entire circuit is provided by the *netlist*, which specifies all the nets for a given circuit.

A hypergraph is used to represent the connectivity information from the circuit specification. Each vertex in the hypergraph represents a cell in the circuit and each hyperedge represents a net from the circuit's netlist. The weights on the vertices and hyperedges can, for example, convey information about the physical size of the cells and signal delay in the nets, respectively. The equivalence between a circuit specification and its hypergraph representation is exact if each net in the circuit has at most one pin on any cell [AHK95]. The application of hypergraph partitioning within VLSI design is longstanding and has been well-addressed in literature. For an extensive overview, see the surveys in [AHK95, Kah98, CRX03].

Data Mining

Hypergraph partitioning has been applied in the field of data mining. Clustering in data mining is a process that seeks to group data such that the intracluster similarity is maximised, while intercluster similarity is minimised. Hypergraph modelling of the clustering process is described in [HKKM97], where the set of vertices corresponds to the set of data items being clustered and each hyperedge corresponds to a set of related items. An application of this method to web document categorisation is presented in [BGG+99].

Parallel Databases

In high-performance database systems, where the amounts of data to be retrieved for individual queries is large, the I/O bottleneck is overcome through parallel I/O across multiple disks. The disks are accessed in parallel while a query is being processed; consequently, the response time can be reduced by balancing the amount of data that needs to be retrieved on each disk. The resulting declustering problem is to partition the data across multiple disks so that the data items that are more likely to be retrieved together are located in separate disks, while the disk capacity constraint is maintained. In [KA05], the declustering problem is modelled by a hypergraph where the set of vertices corresponds to the set of data items while the queries in the database system correspond to the hyperedges of the hypergraph.

1.1.4 Limitations of Serial Partitioning

The hypergraph partitioning problem has been extensively studied in literature and very good serial tool support exists. However, the performance of serial partitioning tools is limited because they can only run on a single processor. This imposes a limit on the size of the problem that can be tackled and also on the runtimes that can be achieved.

There are an increasing number of applications that demand large-scale partitioning which cannot be performed using serial partitioners. In [BDKW03, BDKW04], the hypergraph model for sparse matrix decomposition is key to scalability of the iterative Laplace Transform inversion algorithm that is, amongst other applications, used in the response time analysis of queueing systems. This is because parallel sparse matrix–vector multiplication is the kernel operation in such solvers. The authors note that in order to perform analysis of models that are of practical interest, parallel partitioning of hypergraphs that are too large for current serial hypergraph partitioners is necessary.

[DBH+05] describes forthcoming challenges in the field of dynamic load balancing. The authors emphasise that hypergraph models address many of the deficiencies of the hitherto-popular graph models. However, for large-scale dynamic load-balancing problems, parallel hypergraph partitioning is required.

Recently, direct implementation into reconfigurable hardware of programs written using high-level languages has attracted attention from researchers [BVCG04]. Currently, when compiling programs into hardware, circuit partitioning using serial hypergraph partitioners is performed [Mis04]. We anticipate that as the complexity of high-level programs for compilation into reconfigurable hardware increases, parallel hypergraph partitioning will be required[1].

1.1.5 Thesis Objectives

The aims of this thesis are:

- To develop scalable parallel algorithms for the hypergraph partitioning problem.
- To implement them in a parallel hypergraph partitioning tool.
- To conduct theoretical and empirical evaluations of the algorithms' scalability and partition quality comparisons with current state-of-the-art. In

[1] Personal communication with Mahim Mishra (`http://www.cs.cmu.edu/~phoenix/`).

particular, the objectives are:

  - To assess speedups achieved over serial hypergraph partitioning tools on a suite of test hypergraphs from a range of application domains.
  - To compare the quality of partitions produced by the parallel hypergraph partitioning algorithms with the quality of those produced by serial hypergraph, and parallel graph, partitioning tools.

- To investigate extensions to existing applications of hypergraph partitioning and develop novel applications.

1.2 Contributions

This thesis presents the first parallel multilevel algorithms for the hypergraph partitioning problem and describes their implementation within the parallel hypergraph partitioning tool `Par`*k*`way`. The work enables the solution of very large hypergraph partitioning problems that were hitherto intractable by serial computation. In addition, good empirical scalability enables faster solution of hypergraph partitioning problems. Solution quality is comparable with state-of-the-art serial multilevel hypergraph partitioning tools (typically within 5%).

Hypergraph partitioning algorithms based on the multilevel paradigm have been parallelised, using a one-dimensional distribution of the hypergraph across the processors. Serial partitioning algorithms based on the multilevel paradigm first construct a sequence of hypergraph approximations during the coarsening phase, and then, having computed a partition of the coarsest approximation during the initial partitioning phase, they project this partition onto the original hypergraph through the successive hypergraph approximations during the uncoarsening phase.

The approach taken parallelises the coarsening and uncoarsening phases of this paradigm, while the initial partitioning phase proceeds using existing serial hypergraph partitioning algorithms. The proposed parallel algorithms use a hash

function to balance computational load and a two-phase communication schedule to resolve interprocessor conflicts that potentially arise as a result of concurrent decision-making during the coarsening and uncoarsening phases. In addition, a serial multi-phase refinement technique (also known as V-cycling) is parallelised; this allows the quality of partition produced by the parallel multilevel algorithm to be further improved at an additional runtime cost.

An analytic average-case perfomance model of the parallel algorithms is presented. This shows that the proposed algorithms are asymptotically cost-optimal on a hypercube parallel architecture under the assumption that the input hypergraph is very sparse, i.e. has a very small average vertex degree; this is the case for hypergraphs across many areas of practical interest. The isoefficiency function of the parallel algorithms is also derived and is observed to be of the same order as Karypis and Kumar's parallel multilevel graph partitioning algorithm.

The proposed parallel algorithms are implemented in a parallel hypergraph partitioning tool `Par`*k*`way`. The tool is used as a basis for an experimental evaluation of the proposed parallel algorithms on a number of hypergraphs from various application domains. The experiments validate the parallel hypergraph partitioning algorithms against state-of-the-art serial multilevel hypergraph partitioning tools or (when the hypergraphs are too large to be partitioned serially) a state-of-the-art parallel graph partitioning tool. An empirical evaluation of the parallel hypergraph partitioning algorithm's runtime is performed and speedups over state-of-the-art serial multilevel tools are observed. We also perform experiments in order to investigate the scalability behaviour of the parallel multilevel hypergraph partitioning algorithms, as predicted by our analytic performance model. These seek to maintain a near-constant level of processor efficiency by increasing the problem size according to the isoefficiency function, as the number of processors is increased.

The `Par`*k*`way` parallel hypergraph partitioning tool is used to accelerate parallel PageRank computation. The substantial increase in partitioning capacity offered by our parallel algorithms is exploited by applying one- and two-dimensional

hypergraph partitioning-based schemes for sparse matrix decomposition. Such schemes previously could not be used because of the capacity limits of serial hypergraph partitioners. The use of the hypergraph partitioning-based schemes within parallel iterative PageRank computation yields a halving in per-iteration runtimes over existing purely load-balancing schemes on a gigabit Ethernet cluster.

1.3 Thesis Outline

The remainder of this thesis is set out as follows:

Chapter 2 formally introduces the definitions of a hypergraph and the hypergraph partitioning problem. A section on asymptotic notation and computational complexity is also included. Finally, we present background material relating to parallel algorithms. This consists of a brief overview of analytical modelling of parallel algorithm performance and scalability behaviour, as well as definitions of desirable properties that are sought in parallel algorithm design.

Chapter 3 presents related work on serial algorithms for graph and hypergraph partitioning and parallel algorithms for graph partitioning. Since the optimal graph and hypergraph partitioning problems are NP-complete, many suboptimal heuristic algorithms have been proposed. The most successful algorithms to date (such as Kernighan-Lin and Fidduccia-Mattheyses algorithms) are based on the iterative improvement strategy. More recently, these iterative improvement algorithms have been incorporated within the multilevel framework, which has significantly increased their ability to consistently compute good-quality solutions across a wide range of problem instances. We present a comprehensive survey of iterative improvement algorithms, and in the context of graph partitioning algorithms, we review spectral partitioning. A detailed summary of the multilevel paradigm is

also presented. A number of general combinatorial optimisation approaches have also been evaluated in partitioning literature. These include Simulated Annealing, Genetic Algorithms and Tabu search. We describe a summary of their application to graph and hypergraph partitioning. The chapter closes with a discussion of parallel graph partitioning. In particular, parallel multilevel graph partitioning algorithms are described in detail because they provide insight into possible parallel formulations for multilevel hypergraph partitioning algorithms.

Chapter 4 presents our work on parallel hypergraph partitioning. Drawing on experience from parallel graph partitioning, we note that only the coarsening and uncoarsening phases need to be parallelised. We first present an application-specific disk-based parallel multilevel hypergraph partitioning algorithm. In terms of partition quality, our algorithm consistently outperforms the approximate partitioning approach based on parallel graph partitioning. However, due to slow disk-access times and poor processor utilisation, its observed parallel runtimes are long. We then present the parallel multilevel hypergraph partitioning algorithms that are the main contribution of this thesis. In particular, parallel formulations of the serial coarsening and uncoarsening phases, based on a one-dimensional hypergraph distribution across the processors, are presented. The serial multiphase refinement algorithm is also parallelised. Finally, we describe an average-case analytical performance model of our parallel algorithms. Our model assumes that the input hypergraphs are sparse and have a low average vertex degree. Under these assumptions, asymptotic cost-optimality of the parallel algorithms is shown and the algorithms' isoefficiency function is derived. A description of an alternative approach to parallel hypergraph partitioning, developed at Sandia National Laboratories, concludes the chapter.

Chapter 5 describes the implementation of the parallel hypergraph partitioning algorithms proposed in Chapter 4 within the parallel hypergraph parti-

tioning tool Par*k*way2.0. We also describe the experimental evaluation of the algorithms using Par*k*way2.0. In order to provide a high-performance testbed for the parallel hypergraph partitioning algorithms, the London eScience's Beowulf cluster (comprised of 64 dual-processor nodes, with Intel 2.0GHz Xeon processors and 2GB RAM connected by Myrinet with peak throughput of 2Gbps) was used. The experiments were carried out on hypergraphs from several application domains. We compare the quality of partition produced by the parallel hypergraph partitioning algorithms with the quality of partitions produced by state-of-the-art serial multilevel partitioners and also with those produced by an approximate hypergraph partitioning approach using a state-of-the-art parallel graph partitioning tool. Speedup over a state-of-the-art serial multilevel partitioner achieved by Par*k*way2.0 across a number of different hypergraphs is observed. Finally, we perform an experiment which attempts to maintain a constant level of efficiency by increasing the problem size as the number of processors is increased according to the isoefficiency function derived in Chapter 4.

Chapter 6 describes a case study application of the parallel multilevel hypergraph partitioning algorithms in order to obtain a more efficient parallel PageRank computation. PageRank is typically computed using parallel iterative methods, whose kernel operation is parallel sparse matrix–vector multiplication. State-of-the-art hypergraph partitioning-based sparse matrix decomposition schemes could not be previously used in this context because of the capacity limits of serial hypergraph partitioners. We demonstrate that the use of hypergraph partitioning-based sparse matrix decomposition schemes leads to a halving of per-iteration runtime when compared to existing purely load-balancing sparse matrix decomposition schemes. In order to provide a realistic experimental setup for our PageRank case-study, a gigabit Ethernet-connected Beowulf Linux cluster was used. It comprised of 8 dual-processor nodes, each with two Intel Pentium 4 3.0Ghz processors and 2GB RAM.

Chapter 7 concludes the thesis by providing a summary and an evaluation of the work presented. This chapter also discusses opportunities for future work.

Appendix A describes in detail the hypergraphs used in the experiments in this thesis.

Appendix B presents in detail all the experimental results drawn upon in this thesis.

1.4 Statement of Originality and Publications

I declare that this thesis was composed by myself, and that the work that it presents is my own, except where otherwise stated.

The publications referred to below arose from work carried out during the course of this PhD. The chapters in this thesis where material from them appears are indicated below:

- **Workshop on Parallel and Distributed Scientific and Engineering Computing 2004** (PDSECA 2004) [TK04c] and **International Journal of Computational Science and Engineering 2006** (IJCSE 2006) [TK06b] present an application-specific disk-based parallel formulation of the multilevel k-way hypergraph partitioning algorithm. This was preliminary work that led to the development of the parallel hypergraph partitioning algorithms that form the main contribution of this thesis. Material from these papers appears in Chapter 4.

- **International Symposium on Parallel and Distributed Computing** (ISPDC 2004) [TK04b] presents the parallel multilevel algorithm for the k-way hypergraph partitioning problem, described in Chapter 4. It also compares the parallel hypergraph partitioning algorithm with the parallel graph partitioning approach to computing a partition of the hypergraph. Material from this paper appears in Chapter 4.

- **International Symposium on Computer and Information Sciences** (ISCIS 2004) [TK04a] describes an asymptotic analytical performance model for the proposed parallel multilevel hypergraph partitioning algorithms and derives its isoefficiency function. A set of experiments using the `Par`*k*`way2.0` tool in a comparison with state-of-the-art serial multilevel tools to evaluate scalability properties are performed. Material from this paper appears in Chapter 4 and Chapter 5.

- **Journal of Parallel and Distributed Computing** (submitted for publication) [TK06a] presents a detailed description of the parallel multilevel hypergraph partitioning algorithms and their analytical performance model from Chapter 4. It builds on earlier work in [TK04b] and [TK04a], by presenting a parallel formulation of multi-phase refinement. An extensive experimental evaluation of the algorithms on hypergraphs from a wide range of application domains is also included. Material from this paper appears in Chapter 4 and Chapter 5.

- **European Performance Evaluation Workshop** (EPEW 2005) [BKdT05] describes a case study application of parallel hypergraph partitioning to parallel iterative PageRank computation. Material from this paper appears in Chapter 6. This was joint work with Jeremy Bradley and Douglas de Jager.

Chapter 2

Preliminaries

2.1 Introduction

This chapter presents some preliminaries to the thesis. We define a hypergraph and the hypergraph partitioning problem. The remaining sections present background material related to computational complexity and parallel algorithms. Performance metrics used in the analysis of parallel algorithms are also introduced.

2.2 Problem Definition

2.2.1 Introductory Definitions

Given a finite set of n vertices, $V = \{v_1, \ldots, v_n\}$, a hypergraph on V is formally defined as follows.

Definition 2.1 (Hypergraph). *A hypergraph is a set system $(V, \mathcal{E})$ on a set V, here denoted $H(V, \mathcal{E})$, such that $\mathcal{E} \subset \mathcal{P}(V)$, where $\mathcal{P}(V)$ is the power set of V. The set $\mathcal{E} = \{e_1, \ldots, e_m\}$ is said to be the set of hyperedges of the hypergraph.*

Given the definition of hypergraph as above, we note that when $\mathcal{E} \subset V^{(2)}$, each hyperedge has cardinality two and the resulting set system is known as a *graph*.

Henceforth, in order to distinguish between graphs and hypergraphs, a graph shall be denoted by $G(V, \mathcal{E})$.

Definition 2.2 (Incidence). *A hyperedge $e \in \mathcal{E}$ is said to be incident on a vertex $v \in V$ in a hypergraph $H(V, \mathcal{E})$ if, and only if, $v \in e$ and this is denoted by $e \triangleright v$.*

Definition 2.3 (Incidence Matrix). *The incidence matrix of a hypergraph $H(V, \mathcal{E})$, $V = \{v_1, \ldots, v_n\}$ and $\mathcal{E} = \{e_1, \ldots, e_m\}$, is the $n \times m$ matrix $\mathbf{A} = (a_{ij})$, with entries*

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \in e_j \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Furthermore, let N_H denote the number of non-zeros in the hypergraph's incidence matrix $\mathbf{A}$.

Note that the transpose of the incidence matrix, $\mathbf{A}^T$, defines the *dual* hypergraph $H^*(V^*, \mathcal{E}^*)$ [Bol86].

Definition 2.4 (Vertex adjacency). *Vertices $u, v \in V$ are said to be* adjacent *in a hypergraph $H(V, \mathcal{E})$ if, and only if, there exists a hyperedge $e \in \mathcal{E}$ such that $u \in e$ and $v \in e$ (or equivalently, $e \triangleright u$ and $e \triangleright v$).*

Even though it is a hypergraph by definition, when talking about a graph $G(V, \mathcal{E})$, it is more usual to refer to its *adjacency matrix* instead of its incidence matrix.

Definition 2.5 (Adjacency Matrix). *The adjacency matrix of a graph $G(V, \mathcal{E})$, $V = \{v_1, \ldots, v_n\}$ and $\mathcal{E} = \{e_1, \ldots, e_m\}$, is the $n \times n$ matrix $\mathbf{A} = (a_{ij})$, with entries*

$$a_{ij} = \begin{cases} 1 & \text{if there exists } e \in \mathcal{E} \text{ such that } v_i \in e, v_j \in e \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

By general convention, the adjacency matrix of a graph $G(V, \mathcal{E})$ is also denoted by the symbol $\mathbf{A}$. Note that the adjacency matrix uniquely defines a graph, but does not, in general, uniquely define a hypergraph. Notational confusion with an

incidence matrix of a hypergraph is avoided, since the adjacency matrix will not be used in the context of hypergraphs in the remainder of this thesis. This is because it does not, in general, uniquely define a hypergraph. We also note that the adjacency matrix of a graph is necessarily symmetric for undirected graphs.

Definition 2.6 (Vertex degree and hyperedge length). *The degree of a vertex $v \in V$ is defined as the number of distinct hyperedges in $\mathcal{E}$ that are incident on v. The length of a hyperedge $e \in \mathcal{E}$ is defined to be its cardinality $|e|$.*

Henceforth, we let d_{max} denote the maximum vertex degree and e_{max} denote the maximum hyperedge length in a hypergraph $H(V, \mathcal{E})$.

2.2.2 Hypergraph Partitioning Problem

Definition 2.7 (Hypergraph Partition). *This is a partition $\Pi \subset \mathcal{P}(V)$ of the vertex set V (i.e. a finite collection of subsets of V, called parts), such that $P \cap P' = \emptyset$ is true for all $P, P' \in \Pi$ $(P \neq P')$ and $\bigcup_i P_i = \Pi$.*

When $|\Pi| = k$ and $k > 2$, the partition Π is called a k-way or a multi-way partition. Otherwise (when $k = 2$), we call Π a bipartition. The process of computing a k-way or multi-way partition is called k-way or multi-way partitioning. The process of computing a bipartition of a hypergraph is called hypergraph bipartitioning.

In order to make the hypergraph partitioning problem more meaningful (i.e. in a form where it could be used to model a practical problem), we define a *partitioning objective* and a set of partitioning constraints that identify a subset of feasible partitions from the set of all possible partitions. In order to precisely define the partitioning objective and the partitioning constraints, a scalar weighting on the vertices and hyperedges of the hypergraph is introduced.

In a hyperedge-weighted hypergraph $H(V, \mathcal{E})$, each hyperedge $e \in \mathcal{E}$ is assigned an integer weight (recall from Section 1.1 that the weight of a hyperedge may capture the degree of association between the vertices connected by that hyperedge).

Correspondingly, in a vertex-weighted hypergraph $H(V, \mathcal{E})$, each vertex $v \in V$ is assigned an integer weight (recall from Section 1.1 that vertex weight may model the size of the component or sub-computation being modelled by the vertex).

Note that the restriction to integer weights on the vertices and hyperedges is made with common applications of hypergraph partitioning in mind. However, the hypergraph partitioning problem definition is still meaningful when real-valued vertex and hyperedge weights are used.

We will assume that a hypergraph is always both vertex- and hyperedge-weighted. The weights of a vertex $v \in V$ and a hyperedge $e \in \mathcal{E}$ are denoted by $w(v)$ and $w(e)$ respectively.

As mentioned in Section 1.1.3, hypergraphs model systems that exhibit multi-way intercomponent relationships. The common objective of partitioning such hypergraphs is to divide the system into a number of sub-systems, such that components within each sub-system are strongly connected and components in different sub-systems are weakly connected. As a result, it is desirable to have fewer hyperedges that have vertices in more than one part of the partition and many hyperedges which have all their vertices in the same part. To model such behaviour, additional definitions concerning the state of a hyperedge in the context of a partition of the vertex set are required.

Definition 2.8 (Cut hyperedge). *A hyperedge $e \in \mathcal{E}$ is said to be cut by a partition Π if, and only if, there exist distinct parts $P, P' \in \Pi$ and distinct vertices $v, v' \in e$ such that $v \in P$ and $v' \in P'$. A hyperedge that is not cut by a partition Π is said to be uncut.*

The set of hyperedges that are cut by the partition is called the cutset. The cardinality of the cutset is called the partition cutsize. When the cardinality of each hyperedge in the hypergraph is two, so that Π is a partition of a graph, the cutset is also called an *edge separator* of the graph.

A hyperedge in the cutset of a partition Π has vertices in at least two distinct parts. It is said that a hyperedge $e \in \mathcal{E}$ spans (or connects) the part P of a

partition Π if, and only if, there exists a $v \in V$ such that $v \in e$ and $v \in P$. We write λ_e to denote the number of parts spanned by hyperedge $e \in \mathcal{E}$. We also write $\Pi(v)$ to denote the part that a vertex v has been assigned by partition Π and $P(e)$ to denote the number of vertices belonging to hyperedge e that are assigned to part P by a partition Π.

The objective of partitioning a hypergraph may be expressed as a function of the hypergraph $H(V, \mathcal{E})$ and its partition Π. The objective thus defines a measure of "quality" on the partition Π of $H(V, \mathcal{E})$, in terms of the relationships between the vertices (which are expressed by the set of hyperedges, $\mathcal{E}$).

Definition 2.9 (Partitioning Objective Function). *Let $f_o : (\mathcal{P}(V), \mathcal{P}(V)) \rightarrow \mathbb{Z}$ denote a function such that $f_o(\Pi, \mathcal{E})$ represents a cost (or another quantitative measure) of a partition Π when applied to a hypergraph $H(V, \mathcal{E})$. The function f_o is called the partitioning objective function.*

Once again, we note that the partitioning objective is expressed as an integer quantity to reflect hypergraph partitioning applications; a real-valued objective would also be meaningful. We also note that historically the word "cost" has been used in partitioning literature to represent a quantitative measure of partition quality. This is because hypergraph partitioning is commonly used to model decomposition problems where the requirement is that the interconnect within the decomposition (partition) is minimised and unneccesary interconnect is viewed as additional cost.

Having defined a suitable partitioning objective, a set of partitioning constraints is required. It is natural to define the partitioning constraints over the set of vertices V that is being partitioned. Since the hypergraph is vertex-weighted, it is possible to measure the weight of each part in the partition in terms of its constituent vertex weights.

Definition 2.10 (Part weight). *Let function $f_w : \mathcal{P}(V) \rightarrow \mathbb{Z}$ be such that for $A \subseteq V$, $f_w(A)$ defines the weight of a given subset of V. The function f_w is then used to define the weight of each part $P \in \Pi$.*

In almost all applications of hypergraph partitioning, the part weight function f_w is defined as the sum of the weights of the vertices within the part. Henceforth, we also assume this definition of part weight.

It is natural that the partitioning constraint should ensure a balance between the weights of different parts within the partition of the hypergraph. Formally, we express the balance constraint as follows: given the average part weight W_{avg} of a partition Π, the individual part weights should not exceed the average part weight by more than a prescribed factor of ϵ, where $0 < \epsilon < 1$.

In practice, the precise value of ϵ is usually chosen to reflect the requirements of the application domain. Note that in some applications [AHK95, Alp98, DBH+06], strictly enforcing the partitioning constraint defined by ϵ is not critical; that is, a slightly "unbalanced" partition (i.e. in which the weight of some part exceeds by a small factor the weight specified by the given value of ϵ) may be preferred if it results in a significant improvement in the objective function.

However, we always aim to compute partitions that satisfy the specified partitioning constraint. We expect that the algorithms presented in this thesis can be relatively easily modified to suit different cases where strict enforcement of the partitioning constraints is not critical.

We are now in a position to formally define the hypergraph partitioning problem.

Definition 2.11 (Hypergraph partitioning problem). *Consider a hypergraph $H(V, \mathcal{E})$. Given that functions f_o and f_w (as defined in Definition 2.9 and Definition 2.10, respectively) exist, and given an integer $k > 1$ and a real-valued balance criterion $0 < \epsilon < 1$, the goal is to find a partition $\Pi = \{P_1, \ldots, P_k\}$, with corresponding part weights $W_i = f_w(P_i)$, $1 \leq i \leq k$, such that:*

$$W_i < (1 + \epsilon)W_{avg} \tag{2.3}$$

holds for all $1 \leq i \leq k$ (where $W_{avg} = \sum_{i=1}^{k} W_i/k$) and $f_o(\Pi, \mathcal{E})$ is optimised.

In this thesis, the terms "computing partitions", "finding partitions" and "finding solutions" are all interchangeable and refer to the process of solving the hypergraph partitioning problem.

We note that hypergraphs arising in partitioning problems are typically sparse [Alp98, DBH+06]. Formally, we will say that a hypergraph $H(V, \mathcal{E})$ (with $n = |V|$ and $m = |\mathcal{E}|$) is sparse if, and only if, $N_H \ll m \times n$. That is, a hypergraph is sparse if, and only if, its incidence matrix is sparse.

2.3 Partitioning Objectives

This section describes the most important objective functions within hypergraph partitioning literature. The computational complexity of hypergraph partitioning with different objective functions is discussed in Section 2.4.

Recall that hypergraph partitioning has been most widely studied in the context of VLSI CAD and sparse matrix decomposition for parallel computation. Within these (and most other) application domains, the goal of the hypergraph partitioning is to find the partition that minimizes the objective function (subject to partitioning constraints). We assume this problem formulation in the remainder of the thesis.

A k-way hypergraph partition may be computed directly or by recursively bipartitioning the hypergraph, until k parts have been constructed, as shown in Figure 2.1. Recursive bisection is considered a divide-and-conquer approach. However, it is not able to directly optimise certain objective functions which depend on the knowledge of the parts spanned by a cut hyperedge. Examples of these are the sum of external degrees objective and the $k-1$ objective [Kar02].

In this thesis, parallel solution methods are primarily sought for the k-way hypergraph partitioning problem with the $k-1$ partitioning objective. However, in hypergraph partitioning literature a distinction is often made between objective functions arising in bipartitioning problems (bipartitioning objectives) and objective functions arising in multi-way partitioning problems (multi-way partitioning objectives) [Alp96, AHK95]. Consequently, in summarising the various partitioning objectives studied in literature, we will make a distinction between the bipartitioning and the k-way partitioning objectives (as was done in [Alp96, AHK95]).

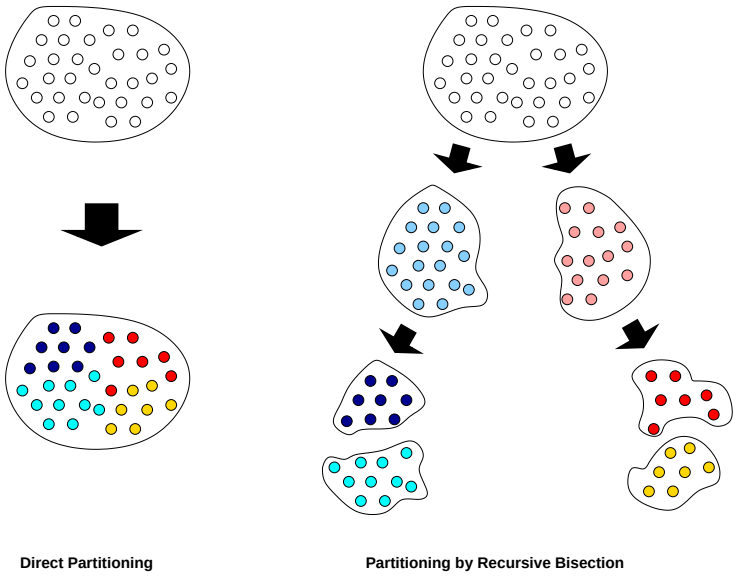

Figure 2.1. Approaches to computing a k-way partition.

Hypergraph (and graph) bipartitioning are also often called hypergraph (and graph) bisection in literature, and we will use these terms interchangeably.

2.3.1 Bipartitioning Objectives

A relatively simple partitioning objective is to minimise the cardinality of the cutset (i.e. the number of cut hyperedges). This defines the *hyperedge cut* or the *min-cut* objective. In graph partitioning, the equivalent objective function is called the *edge-cut* partitioning objective. The definition presented here sums the weights of the hyperedges in the cutset; setting the weight of each hyperedge to unity gives the problem formulation that minimises the cardinality of the cutset (the cutsize).

Definition 2.12 (Hyperedge cut objective). *The hyperedge cut objective function is defined as:*

$$f_o(\Pi, \mathcal{E}) = \sum_{e \in \mathcal{E}, \lambda_e > 1} w(e) \tag{2.4}$$

The hyperedge cut objective is the most widely studied partitioning objective in VLSI CAD literature [Alp96]. Since the cardinality of the cutset of a partition

is called its cutsize, the value of the hyperedge cut objective for a particular partition is often also called the partition cutsize in literature.

Different constraint formulations for min-cut bipartitioning are also addressed in VLSI CAD literature, in addition to the partitioning constraint described in Definition 2.11. For example, in *size-constrained* min-cut bipartitioning, explicit lower and upper bounds on part weight are prescribed [AHK95].

It is possible to combine the objective function on the hyperedges with the balance constraint on the part weights into a single objective function. In this problem formulation, the balance constraint on the part weights (cf. Equation 2.3) is omitted. In such an approach to partitioning, a balanced partition is implicitly required in order to optimise the objective function. However, bounds on part weight are not explicitly stated during the optimisation.

In [WC89], the *ratio-cut* objective, first proposed in [LR88], was studied in the context of VLSI CAD. Here, the objective function is

$$f_o(\Pi, \mathcal{E}) = \frac{1}{f_w(P_1) f_w(P_2)} \sum_{e \in \mathcal{E}, \lambda_e > 1} w(e) \tag{2.5}$$

where $\Pi = \{P_1, P_2\}$.

2.3.2 Multi-way Partitioning Objectives

We now consider partitioning objectives when $k > 2$ in Definition 2.11. Note that the hyperedge cut objective can be computed for a general $k \geq 2$ from Equation 2.4. However, this may not always provide a good model of a physical system because the hyperedge cut objective does not distinguish between hyperedges spanning two parts and hyperedges spanning many parts. For example, it has been noted that in the context of VLSI CAD, nets that span multiple parts can consume more I/O and timing resources than nets spanning only two parts [AHK95]. Instead, objective functions that take into account the number of parts spanned by each cut hyperedge are usually preferred.

Definition 2.13 (Min-cut k-way partitioning objective). *The min-cut k-way objective function is defined as:*

$$f_o(\Pi, \mathcal{E}) = \sum_{e \in \mathcal{E}, \lambda_e > 1} \lambda_e w(e) \tag{2.6}$$

The min-cut k-way partitioning objective is also known as the *sum of external degrees* (SOED) objective [KK98b]. In a related objective, called the $k-1$ objective function (or the $k-1$ metric), each cut hyperedge $e \in \mathcal{E}$ is counted $\lambda_e - 1$ times [San89, CL98, KK98b].

Definition 2.14 ($k-1$ partitioning objective). *The $k-1$ objective function is defined as:*

$$f_o(\Pi, \mathcal{E}) = \sum_{e \in \mathcal{E}, \lambda_e > 1} (\lambda_e - 1) w(e) \tag{2.7}$$

In [cA99], the $k-1$ objective was shown to correctly model the exact communication volume of parallel matrix–vector multiplication. We consider this in more detail in Section 6.3.

We note that for a bipartition, the $k-1$ objective reduces to the hyperedge cut objective (given by Equation 2.4). When the recursive bisection approach is used to solve the k-way hypergraph partitioning problem with the $k-1$ objective, during each bisection step, the objective function becomes the hyperedge cut objective.

Another objective related to the $k-1$ and SOED objectives is the $k(k-1)/2$ objective. Here, each cut hyperedge e that spans λ_e parts is counted $\lambda_e(\lambda_e - 1)/2$ times [San93].

As is the case with bipartitioning objectives, it is also possible to combine the objective function on the hyperedges with the balance constraint on the part weights into a single objective function.

In [CSZ94], the *minimum scaled cost* objective was proposed:

$$f_o(\Pi, \mathcal{E}) = \frac{1}{n(k-1)} \sum_{i=1}^{k} \frac{|S(P_i)|}{f_w(P_i)} \tag{2.8}$$

where $S(P_i)$ is the set of all hyperedges that span the part P_i, without having all their vertices in P_i. Alternatively, in [YCL92], the *minimum cluster ratio* objective was proposed:

$$f_o(\Pi, \mathcal{E}) = \frac{|C(\Pi, \mathcal{E})|}{\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} f_w(P_i) f_w(P_j)} \tag{2.9}$$

where $C(\Pi, \mathcal{E})$ is the set of all hyperedges in $\mathcal{E}$ that are cut by the partition Π.

2.4 Asymptotic Notation and Computational Complexity

2.4.1 Asymptotic Notation

Asymptotic notation is used to express bounds on functions and sequences of real numbers in the limit as the function or sequence parameter becomes very large; the bound is expressed as a function of the parameter. We consider a real-valued function $f(x)$:

The O notation: a function $f(x) = O(g(x))$ for some function $g(x)$ if, and only if, for a given constant $c > 0$ there exists an $x_0 \geq 0$ such that $f(x) \leq cg(x)$ for all $x \geq x_0$.

The Ω notation: a function $f(x) = \Omega(g(x))$ for some function $g(x)$ if, and only if, for a given constant $c > 0$ there exists an $x_0 \geq 0$ such that $cg(x) \leq f(x)$ for all $x \geq x_0$.

The Θ notation: a function $f(x) = \Theta(g(x))$ for some function $g(x)$ if, and only if, $f(x) = O(g(x))$ and $f(x) = \Omega(g(x))$.

2.4.2 Computational Complexity

We briefly consider the computational complexity of algorithms from a theoretical point of view and relate this to the hypergraph partitioning problem.

Computational complexity theory is concerned with characterizing the computational requirements (in terms of memory and time) for solving problems, expressed in terms of input size. The Turing model for sequential computation is used. It specifies a finite set of program states K and a finite set of symbols Σ (the alphabet). An algorithm begins in some initial state $s \in K$ and evolves according to a transition function $\triangle : K \times \Sigma \rightarrow K \times \Sigma$. The algorithm terminates if it reaches one of the prescribed halting states in K. This describes a deterministic Turing machine [Pap94]; in a nondeterministic Turing machine, $\triangle$ is instead a relation. Consequently, in a nondeterministic Turing machine, for each state-symbol combination, there may be more than one (or no) appropriate successor in $K \times \Sigma$ [Pap94].

Of particular interest are decision problems; i.e. problems that admit only a yes or no as output of the computation. An example of a decision problem is "given an integer, is it a prime number or not?". Importantly, an arbitrary problem can always be reduced to a decision problem. The hypergraph partitioning problem can be reformulated as a decision problem by asking the question "does there exist a partition satisfying Equation 2.3 such that the value of the objective function is less than or equal to x?". The optimal partition can be found by recomputing the decision problem, choosing values of x according to a binary search.

The time complexity of a problem is the number of computational steps that are required to solve an instance of the problem on a Turing machine using the most efficient algorithm, expressed as a function of the size of the input. The number of computational steps is quantified using asymptotic notation, which is described in Section 2.4.1. Space complexity is defined in a similar manner in terms of the memory (space) requirement [Pap94].

The set of problems whose time (or space) complexities are similar (for example, polynomial in the size of input) are said to belong to the same *complexity class* [Pap94]. The most important of these are the classes P and NP. A problem is said to be in the class P if it can be solved in time polynomial in the size of the input on a deterministic Turing machine. A problem is in NP if it can be

solved in time polynomial in the size of the input on a non-deterministic Turing machine. Clearly P $\subseteq$ NP, i.e. a problem that is in P is also in NP.

A problem A can be shown to belong to a particular complexity class if another problem B, which is already known to be in the complexity class, can be reduced to A. The reduction is done by an algorithm that transforms each instance of B to an equivalent instance of A. Of course, the complexity of the reduction algorithm must be at most that which defines the complexity class. A problem A is said to be complete with respect to a complexity class C if all other problems within C can be reduced to A; note that the required reductions may be transitive applications of simpler reduction algorithms (for example, it is enough to provide a reduction from an existing C-complete problem). Completeness is central to complexity theory because it categorizes the relative difficulty and enables a comparison of seemingly unrelated problems.

A long-standing conjecture in theoretical computer science is that P $\neq$ NP holds true [GJ79, Pap94]. Evidence for this is the difficulty in finding polynomial-time algorithms (polynomial in size of input) for problems in NP. In fact, if the conjecture holds, there can be no polynomial-time algorithms for problems in NP\P.

Optimal graph bipartitioning with the edge-cut objective is an NP-complete problem, by reduction from number partitioning [GJ79]. Similarly, optimal hypergraph bipartitioning and multi-way partitioning (with objective functions described in Section 2.3) are NP-complete problems [GJ79], which suggests that polynomial-time algorithms for these are also unlikely to exist.

2.5 Background on Parallel Algorithms

2.5.1 Introduction

Unlike a serial algorithm (that necessarily runs on a single processor), a parallel algorithm is able to execute multiple instructions at the same time because it

uses a computing environment with multiple processing units. The most general control structure for a parallel computing environment is multiple instruction stream, multiple data stream (MIMD).

The physical computing environment that a parallel algorithm runs on is called the *parallel architecture*. It consists of p processors and a physical processor interconnection network. In order to execute parallel programs, a parallel architecture needs to incorporate a communication model. This specifies how the parallel architecture resolves memory access.

A popular theoretical model of parallel computation is the Parallel Random Access Machine (PRAM) [Vis93, GGKK03]. The PRAM consists of p processors and a global memory of unbounded size that is uniformly accessible by all p processors. All the processors share the same address space and common clock, but are allowed to execute different instructions in each cycle. Types of PRAM are classed by how they handle simultaneous access to a particular memory location.

In practice, the communication model most similar to the PRAM is the *shared memory* parallel architecture. It supports a common data space that is accessible to each of the p processors. Memory access time may be uniform (same for each processor) or non-uniform [GGKK03]. An alternative communication model is the distributed memory or message-passing model, where each of the p processors has its own exclusive memory space. The exchange of messages between processors is used to transfer data, instructions and to synchronize actions among the processors.

The way in which the processors are connected to each other, and in the case of the shared memory model, to the memory itself, is called the interconnection topology. For a large number of processors, it is not practical to directly connect each processor to every other (requiring $O(p^2)$ connections), so a number of topologies have been proposed that trade off cost and scalability with performance [GGKK03].

Currently, the most prevalent parallel architecture is the distributed memory (message-passing) architecture, due to the relatively low cost and easy availability

of processing units. Examples of this architecture include clustered workstations and non-shared-address-space multicomputers [GGKK03].

When attempting a parallel formulation of a particular serial algorithm, the parallel algorithm designer decomposes the serial algorithm into sub-tasks and identifies opportunities for concurrency in the serial algorithm (sub-tasks that may be carried out in parallel). A decomposition into a large number of small tasks is called *fine-grained*. On the other hand, when the decomposition consists of a small number of larger tasks, it is called *coarse-grained*.

2.5.2 Performance Metrics for Parallel Algorithms

The performance of a serial algorithm is usually evaluated in terms of its execution time, or runtime, expressed as a function of the size of the input. The runtime of a parallel algorithm is, in addition, a function of the parallel architecture (the number of processing units used, the topology and the message-transfer performance of the interconnect, and potentially, the relative computational speed of the processors, if this differs between processors). It is therefore more pertinent to talk about performance of the *parallel system*; this necessarily takes into account both the parallel algorithm and the parallel architecture [GGKK03].

Let T_s denote the runtime of a serial algorithm, defined as the time elapsed between the beginning and the end of its execution on a single processing unit. Let T_p denote the runtime of a parallel algorithm on p processors, defined as the time elapsed between the beginning and the moment that the last of the processors finishes execution [GGKK03].

When comparisons are made between serial and parallel algorithms for the solution of a given problem, T_s refers to the runtime of the fastest serial algorithm. We seek parallelism because it may be possible to solve the given problem significantly faster if many processors are used. The natural metric to evaluate a parallel algorithm then becomes the *speedup* it achieves over the fastest serial algorithm, as a function of the number of processors used, p.

Definition 2.15 (Speedup). *Speedup, here denoted by S_p, is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the problem on a given parallel computer with p processors:*

$$S_p = \frac{T_s}{T_p} \tag{2.10}$$

Note that p represents a theoretical limit to the speedup that can be achieved using p processors (i.e. $S_p \leq p$). This is because achieving a speedup value that exceeds p when using p processors would contradict the fact that T_s is the runtime of the fastest serial algorithm (a faster serial algorithm may be constructed by combining the p sets of instructions executed across the p processors into a single set of instructions to be performed on one processor).

However, speedup values greater than p (using p processors) are sometimes observed in practice. So-called *superlinear* speedup is usually explained by hardware characteristics that disadvantage the serial algorithm, such as hierarchical memory that may enable the input data to fit inside the cache of each of the p processors but not the single processor that the serial algorithm runs on. It may also be caused by algorithmic differences between the serial and parallel algorithms; for example, the differences in node search order when comparing serial and parallel depth-first search in a graph [GGKK03].

Definition 2.16 (Cost). *The cost of a parallel algorithm on a parallel system with p processors, here denoted by C_p, is defined as the product of the parallel runtime and the number of processors used. Intuitively, this is the amount of processor-time "tied-up" in solving the problem in parallel.*

$$C_p = pT_p \tag{2.11}$$

Definition 2.17 (Efficiency). *Efficiency, here denoted by E, is the fraction of the parallel runtime that the parallel system is doing "useful" work. It is given*

by:

$$E = \frac{T_s}{C_p} \tag{2.12}$$

$$= \frac{T_s}{pT_p} \tag{2.13}$$

$$= \frac{S_p}{p} \tag{2.14}$$

Unless the cost of a parallel algorithm is equal to the best serial runtime, as in an idealised scenario involving a trivially parallel problem, the parallel algorithm will consume processor-time that could have been spent doing other work. This defines the parallel overhead.

Definition 2.18 (Parallel Overhead). *The parallel overhead, T_o, is given by:*

$$T_o = pT_p - T_s \tag{2.15}$$

$$= C_p - T_s \tag{2.16}$$

We expect that $T_o > 0$ will be observed in practice. This is because the processors will also be involved in communication and synchronisation with other processors while executing the parallel algorithm, none of which are done by the serial algorithm. During the parallel computation, the processors may also be doing additional computation that is not done by the serial algorithm. These sources of overhead can be summarised as follows [GGKK03]:

- Interprocessor communication – required when communicating data between processors and also required when processors synchronize.
- Idling – usually occurs at a synchronization point when one or more processors are idle, waiting for one or more processors to complete a step of the algorithm. This may also occur because part of the algorithm can only be done *sequentially* (i.e. only a single processor may work on that part).
- Excess computation – the fastest serial algorithm for a given problem may be difficult to parallelise, but a slower serial algorithm may offer better concurrency and so the parallel algorithm is based on it instead.

2.5.3 Scalability of Parallel Algorithms

We would like to design parallel algorithms whose runtimes decrease as the number of processors increases. It is possible to formalise this intuition.

Let W denote the problem size, expressed as the number of instructions required by the fastest serial algorithm. We (trivially) have that $T_s = \Theta(W)$.

Definition 2.19 (Cost-Optimality). *A parallel system can be said to be cost-optimal if the cost of solving the problem using the proposed parallel algorithm and parallel architecture has the same asymptotic growth when expressed as the function of the input size as the fastest known serial algorithm (running on a single processing element).*

Thus, a parallel system is cost-optimal if

$$pT_p = \Theta(W) \tag{2.17}$$

Note that it is not possible to maintain the increase in speedup by continuously increasing p, even for a cost-optimal parallel system. To see this, consider a serial component (defined as the set of instructions that must be computed sequentially) of size W_s within the problem of size W (W_s may perversely even be a single instruction). Then W/W_s is the upper bound on the speedup that can be achieved for that problem on any parallel system. This result is known as *Amdahl's Law* [Amd67].

Instead, the aim of parallel algorithm design is to maximise the efficiency of the parallel system. From Equation 2.14 and Equation 2.16, we have:

$$E = \frac{T_s}{pT_p} \tag{2.18}$$

$$= \frac{T_s}{T_o + T_s} \tag{2.19}$$

$$= \frac{1}{1 + \frac{T_o}{T_s}} \tag{2.20}$$

The aim of "maximising" efficiency can be interpreted as maintaining the term T_o/T_s as low as possible. However, for a fixed problem size W (and thus a fixed

value of T_s), as p increases, the efficiency of the parallel system is expected to decrease (because S_p is bounded by Amdahl's law, $T_o \to \infty$ as $p \to \infty$).

On the other hand, it may be possible to maintain a constant level of efficiency with increasing p by modifying the problem size W (and thus T_s). Parallel systems for which this is possible are called *scalable* parallel systems.

Consider the total parallel overhead T_o as a function of the problem size W and the number of processors p. From Equation 2.20, we then have:

$$E = \frac{1}{1 + \frac{T_o(W,p)}{W}} \tag{2.21}$$

Rearranging gives the following expression for W, in terms of efficiency and overhead:

$$W = \frac{E}{1 - E} T_o(W, p) \tag{2.22}$$

Treating the term $E/(1 - E)$ as a constant K (with E representing the desired efficiency), Equation 2.22 yields

$$W = KT_o(W, p) \tag{2.23}$$

The relationship in Equation 2.23, rearranged to give W in terms of p, is used to determine the required modification in the problem size W that is sufficient for maintaining the efficiency at a constant level for a scalable parallel system, as p is increased. The function $W = W(p)$, describing the required modification to the problem size, is called the *isoefficiency function*.

In fact, $\Omega(p)$ is an asymptotic lower bound on the isoefficiency function [GGKK03]. To see this, consider an isoefficiency function of lower order. Then, because the number of processing units increases at rate $\Theta(p)$, the number of processing units will eventually exceed W (which increases at rate $\Omega(p)$ by the isoefficiency function) and the remaining processors would be idle.

Naturally, it is desirable for scalable parallel algorithms to have isoefficiency functions of the lowest order possible. Poorly-scalable parallel systems have high-order isoefficiency functions, meaning that a very large increase in problem size is required to maintain constant efficiency as the number of processors is increased.

Chapter 3

Related Work

3.1 Introduction

This chapter presents an overview of existing approaches to the graph and hypergraph partitioning problems. As noted in Section 2.4, graph and hypergraph partitioning are NP-complete problems; as a result, research effort has focused on developing polynomial-time heuristic algorithms that yield good suboptimal solutions.

Where graph partitioning is discussed, a graph $G(V, \mathcal{E})$, as opposed to the hypergraph $H(V, \mathcal{E})$, is the subject of the partitioning problem. Although it is a particular instance of the hypergraph partitioning problem, the graph partitioning problem has been the subject of considerable research and successful heuristics that are specific to graph partitioning have been developed.

Section 3.2 discusses empirical evaluation and benchmarking of heuristic partitioning algorithms in literature. Section 3.3 presents an overview of hypergraph partitioning approaches that first approximate the hypergraph by a graph, and then apply graph partitioning algorithms to compute a partition.

We then proceed to discuss move-based and multilevel approaches to the hypergraph and graph partitioning problems. The classification of algorithms used in [AHK95] is followed. It was noted in Section 2.3 that the k-way partition

may be computed directly or by recursive bisection. For both approaches, we make a distinction between *flat* algorithms (those that operate directly on the given hypergraph) and *multilevel* algorithms (those that construct a set of approximations to the given hypergraph). Section 3.4 presents an overview of flat move-based partitioning algorithms and Section 3.5 presents algorithms based on the multilevel paradigm.

Finally, we move on to describe existing parallel approaches to the graph partitioning problem in Section 3.6.

3.2 Experimental Evaluation of Partitioning Algorithms in Literature

In general, heuristic partitioning algorithms yield suboptimal partitions and usually no sufficiently tight bounds on the quality of partition with respect to the optimum are available. This means that an analytical comparison of heuristic partitioning algorithms is difficult. In order to compare the performance of heuristic algorithms, researchers have instead benchmarked their performance on suites of partitioning problem instances taken from applications. Because the algorithms tend to be randomized (rather than deterministic), it is common to report the average of objective function values attained and the average runtime over a number of runs of the algorithm.

Within the VLSI CAD research community, the performance of heuristic partitioning algorithms is usually reported on suites of real circuits. A number of benchmark circuits were released by the Microelectronics Centre of North Carolina (MCNC) and sponsored by ACM SIGDA [Brg93]. In literature, MCNC and ACM SIGDA are used interchangeably to denote this set of benchmark circuits. However, as the complexity of new circuits grows, benchmarks have a tendency to become obsolete over time. In addition, [Alp96] notes that the performance of partitioning algorithms on older benchmark circuits has tended to converge.

As a result, circuit benchmarks have been periodically updated in VLSI CAD literature, with the most recent partitioning benchmark being the ISPD98 suite [Alp98]. The largest circuit within the ISPD98 benchmark suite has 184 752 cells, 189 581 nets and 860 036 pins (so that the equivalent hypergraph representation has 184 752 vertices, 189 581 hyperedges and 860 036 non-zeros in its incidence matrix).

Because most industrial circuits are proprietary, they are not included within the standard circuit benchmarks and are thus not usually made available to researchers. To overcome this drawback, attempts have been made to synthetically generate circuits whose structural properties are similar to those of industrial circuits. In [VCS00], a survey of different synthetic generation methods is presented. Nevertheless, it remains hard to prove that such synthetically generated circuits are truly representative of actual physical circuits [VCS00, Alp98].

The scientific computing research community maintains an extensive set of publicly available sparse matrices from a variety of application domains, which are used for empirical evaluation of sparse matrix algorithms. Examples of sources include the University of Florida Sparse Matrix Collection [Dav05] and the Matrix Market repository [BPR+97].

Sparse matrices from these sources can be used to derive hypergraph test cases (for example, by considering the sparse matrix as the incidence matrix of a hypergraph). In general, the sparse matrix instances yield larger hypergraphs than the VLSI benchmark circuits – the largest matrix currently in the University of Florida Sparse Matrix Collection yields a hypergraph with 5.1 million vertices, 5.1 million hyperedges and 99.2 million non-zeros in its incidence matrix.

In [CKM99a], Caldwell et al. describe a detailed methodology for experimental evaluation and reporting of heuristic algorithms for hypergraph partitioning. They are motivated by the inability to reproduce some of the experimental results from previously published work and the hitherto underspecification of heuristic algorithms and experiments in partitioning literature. The authors propose a standardised approach to the reporting of algorithms and experimental results,

observing that:

1. There is a need for a robust experimental testbed when comparing heuristic algorithms. Best-available implementations are not necessarily competent and where possible, comparisons with proposed heuristic improvements should be made using fast and robust implementations of existing algorithms.

2. Implementation details should be reported, as underspecified features and ambiguities in the algorithm may allow a number of distinct implementations, resulting in a significant variance in runtime and solution quality across the possible implementations.

3. Experimental design should be relevant to the proposed heuristic and leading-edge alternatives. There should be a wide range of problem test instances that accurately reflect the application domain.

In this thesis, we aim to adopt the above suggestions within our design and reporting of experiments to evaluate the proposed parallel hypergraph partitioning algorithms.

3.3 Graph Partitioning-Based Approaches To Hypergraph Partitioning

The graph partitioning problem is discussed here as an alternative approach to computing a partition of a hypergraph. It first involves the transformation of the hypergraph into a graph and then the application of an existing graph partitioning algorithm to yield a partition of the hypergraph. This approach has been popular in VLSI CAD literature (applied directly to circuit netlists) and is discussed in detail in [Alp96].

3.3.1 Hypergraph-to-Graph Transformations

When *approximating* a hypergraph by a graph, the set of vertices in the graph is just the set of vertices in the hypergraph. Each hyperedge is typically modelled by a vertex *clique*. That is, for each hyperedge $e \in \mathcal{E}$, an edge is inserted into the graph for every pair of vertices in e, so that the clique connects all the vertices in e.

In [IWW93], the authors show that it is not possible, in general, to represent a hypergraph by a graph clique model whose cut properties correctly model the cut properties of the hypergraph (such as the number of hyperedges cut by a partition). Nevertheless, a number of schemes have been proposed for the weighting of the edges in the approximation graph; a detailed discussion of these is presented in [Alp96]. The standard scheme assigns a weight of $\frac{1}{|e|-1}$ to each clique edge, which ensures that the total vertex degrees remain unchanged [Len90].

As an alternative to a clique-based model, in [HK00], Hendrickson and Kolda propose a bipartitite graph model and modify a number of standard graph partitioning heuristics to run on bipartite graphs. They actually use the bipartite graph to model the non-zero structure within a sparse matrix, but since there is an equivalence between the non-zero structure in sparse matrices and hypergraphs, their approach can equally be applied to hypergraphs.

Having computed the hypergraph-to-graph transformation, we are interested in partitioning a graph $G(V, \mathcal{E})$ so that the edge-cut objective is minimised. Recall that the weights on the edges in $\mathcal{E}$ have been chosen so that the edge-cut provides an approximation to the hypergraph partitioning objective.

3.3.2 Related Graph Partitioning Algorithms

We note that move-based and multilevel approaches to graph partitioning are directly applicable to hypergraph partitioning and are thus reviewed in the context of hypergraph partitioning in Sections 3.4 and 3.5. The main alternatives to

these in the context of graph partitioning are the *spectral partitioning* approaches, which are based on the spectra of the graph's Laplacian matrix.

Spectral partitioning is related to the more general geometric partitioning approach. Geometric partitioning is so called because given a d-dimensional space (e.g. $\mathbb{R}^d$, for some $d > 0$), it tries to partition a set of n geometric objects (such as points), that are embedded in the given space, into k clusters such that some relation (e.g. the Euclidean distance between points in a cluster) is minimised [AHK95].

In the context of graph partitioning, geometric partitioning is usually applied as follows. First, a suitable transfomation of the vertex set V into the n points of a d-dimensional space is sought, such that minimising the intra-cluster relation by the geometric partitioning algorithm corresponds to minimising edge-cut in a partition of the graph. The geometric partitioning algorithm then partitions the set of n points and this partition is used to construct a partition of the graph by allocating vertices to parts according to the cluster that the corresponding point was allocated to.

Before providing an overview of the spectral approach, we note that the surveys [AHK95, DBH+05] provide a more thorough discussion of geometric techniques applied in graph partitioning.

Definition 3.1 (Degree Matrix). *The degree matrix of a graph $G(V, \mathcal{E})$ is the (diagonal) $n \times n$ matrix $\mathbf{D} = (d_{ij})$ with entries*

$$d_{ij} = \begin{cases} deg(v_i) & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Definition 3.2 (Laplacian Matrix). *The Laplacian matrix $\mathbf{Q}$ of the graph $G(V, \mathcal{E})$ is defined as $\mathbf{Q} = \mathbf{D} - \mathbf{A}$, where $\mathbf{A}$ is the adjacency matrix of $G(V, \mathcal{E})$.*

It is easily shown that $\mathbf{Q}$ is positive semidefinite [PSL90]. It thus has n non-negative eigenvalues (not necessarily distinct) and at least one zero eigenvalue. The multiplicity of the zero eigenvalue of $\mathbf{Q}$ is given by the number of connected

components in the graph and if G is connected, the second smallest eigenvalue is positive [PSL90].

In order to motivate the spectral approach to graph partitioning, let $\mathbf{A}$ be the adjacency matrix of a graph $G(V, \mathcal{E})$ and consider the following one-dimensional placement problem: given n points such that points i and j are connected if $a_{ij} = 1$ and not connected otherwise ($a_{ij} = 0$), find locations for the n points on a real line such that the sum of the squared distances between connected points is minimised. Stated formally, we would like to construct a vector $\mathbf{x}$, such that

$$d(\mathbf{x}) = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} (x_i - x_j)^2 a_{ij} \tag{3.2}$$

is minimised, subject to $\mathbf{x}^T\mathbf{x} = 1$. The latter constraint is imposed to avoid the trivial solution $x_i = 0$ for all i.

If the graph defined by $\mathbf{A}$ is connected, then Hall showed that the second eigenvector $\mathbf{x}_2$ of the Laplacian matrix $\mathbf{Q}$ minimises Equation 3.2 and the "cost" of the optimal solution to the one-dimensional placement problem is given by the second smallest eigenvalue [Hal70]. The spectral properties of $\mathbf{x}_2$ have been extensively studied by Fiedler [Fie73, Fie75a, Fie75b] and consequently this eigenvector is also known as the *Fiedler vector* in literature.

For a connected graph $G(V, \mathcal{E})$, the vertices can be ordered along the real line by assigning $v_i \in V$ to a position on the real line given by the corresponding entry in the Fiedler vector $\mathbf{x}_2$. Let x_i denote the position on the real line allocated to v_i (the i^{th} entry of $\mathbf{x}_2$). Hall's result implies that if v_i and v_j are connected by an edge in $\mathcal{E}$, it is likely that the distance $|x_i - x_j|$ is small. Thus, strongly connected vertices in the graph G are likely to also be close to each other in the ordering. In the language of geometric partitioning, the spectral approach effectively constructs an embedding of the vertices of the graph onto a one-dimensional space (the real line)[1].

[1]Hall's result generalises to an embedding of the n vertices of the graph $G(V, \mathcal{E})$ onto a d-dimensional space using d eigenvectors of $\mathbf{Q}$ [Hal70]. Multiple eigenvector approaches are discussed in more detail in [AHK95].

It is possible to induce a bipartition $\Pi = \{P_0, P_1\}$ of the graph $G(V, \mathcal{E})$ by choosing a real-value ρ, so that $P_0 = \{v_i \in V : x_i < \rho\}$ and $P_1 = \{v_i \in V : x_i \geq \rho\}$. With appropriate choice of ρ, Π can potentially be a good suboptimal partition of $G(V, \mathcal{E})$ (with respect to the minimum edge-cut) because the vertices that are close to each other in the ordering (and thus strongly connected in the graph) are likely to be assigned to the same part. This intuition forms the basis of *spectral partitioning*. In [AHK95], Alpert et al. note that spectral bipartitioning may, however, find solutions arbitrarily worse than the optimum.

In [PSL90], Pothen et al. present an algorithm for computing a vertex separator of a graph, that first uses spectral bipartitioning to compute an edge separator. Since computing the edge separator is equivalent to our stated graph partitioning problem, we only report details of this part of the algorithm. The authors compute the Fiedler vector $\mathbf{x}_2$ of the graph's Laplacian matrix using the Lanczos algorithm and determine the median value of its entries x_m. A bipartition $\Pi = \{P_0, P_1\}$ is constructed, such that $P_0 = \{v_i \in V : x_i \leq x_m\}$ and $P_1 = \{v_i \in V : x_i > x_m\}$.

Simon [Sim91] applies the spectral bipartitioning algorithm for computing the edge separator from [PSL90] within a recursive bisection framework and shows that recursive spectral bisection outperforms the recursive coordinate bisection algorithm. However, in [BS94], it is noted that recursive spectral bisection can be computationally expensive. When the graph has non-unit weights on the edges, in order to apply spectral bisection, the non-zero entries in the adjacency matrix $\mathbf{A}$ representing edges are equal to the corresponding edge-weights [HL95].

Hendrickson and Leland [HL95] study spectral graph partitioning using multiple eigenvectors of $\mathbf{Q}$. They derive a non-standard objective function from a communication-cost model of a parallel computation. A survey of spectral techniques applied within VLSI CAD is presented in [AHK95].

3.4 Move-Based Algorithms

A partitioning approach is defined to be *move-based* if it constructs a new candidate solution based on two considerations:

1. A *neighbourhood structure* that is defined over the set of feasible solutions
2. The previous history of the optimisation

The neighbourhood structure defines a topology over the feasible solution space and incorporates the means of moving from the current feasible solution to another (neighbouring) feasible solution by "perturbing" the current feasible solution. This perturbation may involve the movement of a single vertex or a number of vertices between parts. For example, given a neighbourhood structure defined by a single vertex move (necessarily across a partition boundary), the neighbours of a feasible solution are those feasible solutions that can be reached by making a single vertex move from one part of the partition to another. The solution space is explored by repeatedly moving from the current solution to a neighbouring solution, until some prescribed termination condition is satisfied.

The history of the optimisation may be used to guide the exploration of the solution space, or the approach can be memoryless. For example, greedy methods generate the next solution based only on the best possible move from the current solution.

An advantage of move-based approaches is that the optimisation framework is independent of the objective function being optimised. A number of the move-based algorithms described in this section have been successfully applied to other combinatorial optimisation problems.

3.4.1 Iterative Improvement Algorithms

An iterative improvement algorithm begins with a feasible solution and iteratively moves to the best improving neighbourhood feasible solution. The algorithm terminates when it reaches a solution for which all neighbours are worse

feasible solutions. Thus, iterative improvement algorithms converge to local minima with respect to the initial feasible solution and a neighbourhood structure of the algorithm.

Because the space of all feasible solutions to the hypergraph partitioning problem may be very large, the initial feasible solution is constructed using methods with a randomized component. In practice, multiple starts (or runs) of the algorithm are usually computed, and the partition that best optimises the objective function (out of all the runs of the algorithm) is chosen. This has the effect of randomly sampling the space of local minima with respect to the heuristic.

Iterative improvement algorithms typically use extended neighbourhood structures, since the neighbourhood structure defined by individual vertex moves is seen to be too restrictive.

A typical example of an extended neighbourhood structure is a *pass*. At the beginning of a pass, all vertices are free to move. Vertices are then moved in a greedy manner, so that at each step, the feasible vertex move that best improves the objective function is selected. The possible moves within a pass are defined by a simple neighbourhood structure, with the restriction that each vertex may be moved at most once during a pass. Note that a vertex may also not be moved at all during the pass, for example if all of its possible moves do not result in feasible partitions. An iterative improvement algorithm will proceed in passes from a given initial feasible solution. It will terminate when the most recent pass does not yield an improvement in the objective function.

The Kernighan-Lin (KL) Algorithm

In [KL70], Kernighan and Lin introduced an iterative improvement algorithm for the graph bipartitioning problem with the edge-cut objective (cf. Definition 2.12) that uses a pair-swap neighbourhood structure to generate vertex moves during a pass. A pair-swap neighbourhood structure implies that any pair of vertices $u, v \in V$, such that $\Pi(u) \neq \Pi(v)$, may be involved in a swap across a partition boundary, provided that the resulting partition is also feasible.

Each pass of the KL algorithm proceeds as follows. The vertices of the graph are swapped in iterative fashion, such that a swap leading to the feasible partition with the largest gain in the objective function is performed at each step of the pass. For each swap made, the gain in the objective function is recorded and the vertices moved are *locked* with respect to their new parts (so that they may not be moved again during the pass). The pass continues until no further swaps that lead to a feasible partition can be made. At this point, the sequence of swaps is traversed and the largest partial sum of the swaps' respective gains is noted. This corresponds to the best partition reached during the pass. If the largest partial sum of the gains is positive, the subsequent swaps made during the pass are taken back and this partition is the starting partition for the next pass. Otherwise, the pass has not yielded a gain in the objective function and the KL algorithm terminates. Note that the pass allows the algorithm to climb out of local minima with respect to the pair-swap neighbourhood structure. This is because during a pass, a pair of vertices that result in the highest gain are swapped, regardless of whether this gain is positive or not. A high-level description of the KL algorithm is given in Algorithm 1.

A naïve implementation of the KL algorithm needs $O(n^3)$ time per pass, since computing the highest gain swap involves $O(n^2)$ comparisons. This can be reduced to $O(n^2 \log n)$ per pass by maintaining a sorted list of gains. The number of passes is bounded by m for an unweighted graph, although in practice the number of passes before convergence has been observed to be a small constant.

The graph bisection algorithm was extended to hypergraph bisection in [SK72]. In [Dut93], the complexity per pass of the KL algorithm for graph partitioning was improved to $O(\max\{m \log n, d_{max} m\})$. Alpert notes in [Alp96] that a similar extension appears possible for the KL algorithm, when applied to hypergraph bisection.

Algorithm 1 High-level description of the Kernighan-Lin algorithm

Require: an initial feasible bipartition Π of $G(V, E)$

```
1: repeat
2:     i := 0
3:     S_i := 0
4:     unlock all vertices v ∈ V
5:     while there exist feasible vertex swaps involving unlocked vertices do
6:         make best feasible vertex swap
7:         lock vertices moved
8:         record gain g_i of vertex swap made
9:         S_{i+1} := S_i + g_i
10:        i := i + 1
11:    end while
12:    compute best partial sum S_x of gains
13:    if S_x < 0 then
14:        undo all vertex swaps
15:    else
16:        undo all vertex swaps from x^th to i^th iteration
17:    end if
18: until S_x < 0
```

The Fidduccia-Mattheyses (FM) Algorithm

In [FM82], Fidducia and Mattheyses presented an iterative improvement algorithm that reduced the run time for a pass to $O(N_H)$ for hypergraph bipartitioning. Unlike the KL algorithm, the FM algorithm uses a single vertex move neighbourhood structure to generate vertex moves during a pass, but otherwise proceeds in the same manner as KL.

As is the case with the KL algorithm, the number of passes to convergence has been observed to be a small constant. The reason for the relatively low complexity of the FM algorithm is the gain bucket data structure, shown in Figure 3.1. For computing a hypergraph bisection, there are two such structures; one for each direction of possible moves. This data structure enables constant-time selection of the vertex whose move results in the highest gain and it also enables fast gain updates after each move.

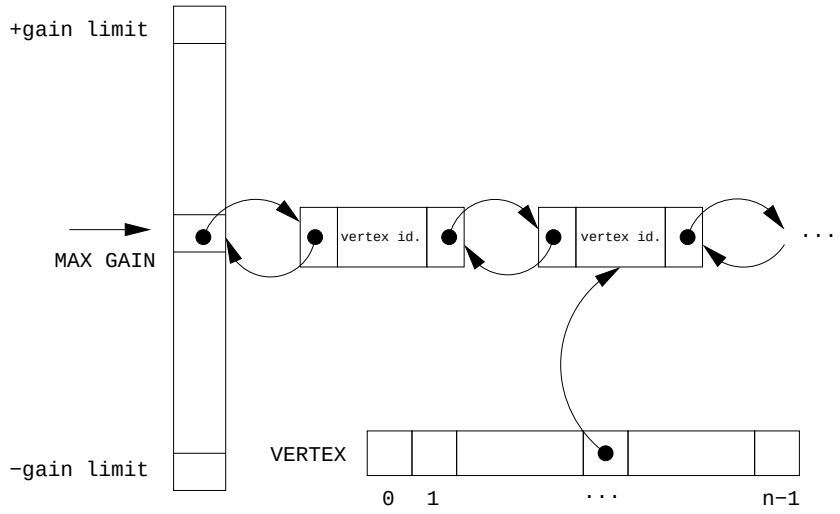

Figure 3.1. The gain bucket data structure used in the Fidduccia Mattheyses algorithm.

The gain buckets can be efficiently maintained because all possible vertex gains are integer-valued and bounded. For unit-weighted hyperedges, possible vertex gain is bounded above by d_{max}, and below by $-d_{max}$. The sparsity of $H(V, \mathcal{E})$ ensures that there is a relatively tight bound on the range of gain values, even when hyperedges are not unit-weighted.

A pass of the FM algorithm for hypergraph bipartitioning proceeds as follows. At the beginning of the pass, the gains of the (at most) n possible vertex moves are computed in $O(N_H)$ time and the gain bucket data structures are initialised by inserting the buckets for each vertex into a slot corresponding to the gain of the resulting vertex move. The vertices with the highest gain are stored in the slot with gain value `MAXGAIN`. This `MAXGAIN` pointer is used to extract the vertex move that results in the highest gain (given that there are unmoved vertices in both parts, moves in at least one direction will be feasible). The algorithm for computing the vertex gains is shown in Algorithm 2 (here $\bar{P}$ denotes the complement part to P in the bipartition Π). Once the move leading to `MAXGAIN` is made, the sparsity of $H(V, \mathcal{E})$ ensures that all subsequent gain updates for the adjacent vertices can be done in $O(1)$ time, with gain nodes adjusted using the pointers from the `VERTEX` array. This vertex gain update computation is shown in Algorithm 3. The `MAXGAIN` pointer is updated accordingly.

Algorithm 2 Computing vertex move gain in FM algorithm

Require: a feasible partition Π of $H(V, \mathcal{E})$
Require: each vertex $v \in V$ is free to move (unlocked)

```
1: for all v ∈ V do
2:     gain(v) := 0
3:     P := Π(v)
4:     for all e ▷ v do
5:         if P(e) = 1 then
6:             gain(v) := gain(v) + w(e)
7:         end if
8:         if P̄(e) = 0 then
9:             gain(v) := gain(v) − w(e)
10:        end if
11:    end for
12: end for
13: compute(MAXGAIN)
```

Enhancements to the KL and FM Algorithms

The seminal version of the FM algorithm leaves a number of implementation details underspecified. Given two or more vertices whose moves result in the same gain in the objective function, ties are broken in favour of the move resulting in the most balanced partition; failing this, no further tie-breaking strategy is specified.

In [Kri84], Krishnamurthy noted that, when breaking ties during a pass of the FM algorithm, one of the choices may often lead to a significantly better solution than the others. This is, however, not captured by the gain in the objective function as a result of the moves. The importance of a good tie-breaking strategy was also highlighted in [HHK97]. The authors note that 15 to 30 vertices will on average share the `MAXGAIN` value at any time during a pass of the FM algorithm on the Primary1 MCNC Benchmark circuit with 833 vertices.

In an attempt to rectify the deficiencies associated with underspecified tie-breaking, a *look-ahead* strategy was introduced by Krishnamurthy in [Kri84]. The strategy relies on computing different levels (or classes) of gain associated with each vertex move. The first level of gain of moving vertex $v \in V$ from part P to part $\bar{P}$

Algorithm 3 Updating vertex gain in the Fidduccia-Mattheyses algorithm

Require: $v \in V$ and *isfree*(v)
Require: $P = \Pi(v)$
1: $\Pi(v) := \bar{P}$
2: *lock*(v)
3: **for all** $e \triangleright v$ **do**
4: **if** $\bar{P}(e) = 0$ **then**
5: **for all** $v' \in e$ **do**
6: **if** *isfree*(v') /* $\Pi(v') = P$ */ **then**
7: $gain(v') := gain(v') + w(e)$
8: **end if**
9: **end for**
10: **else if** $\bar{P}(e) = 1$ **then**
11: **for all** $v' \in e$ **do**
12: **if** *isfree*(v') and $\Pi(v') = \bar{P}$ **then**
13: $gain(v') := gain(v') - w(e)$
14: **end if**
15: **end for**
16: **end if**
17: $P(e) := P(e) - 1$
18: $\bar{P}(e) := \bar{P}(e) + 1$
19: **if** $P(e) = 0$ **then**
20: **for all** $v' \in e$ **do**
21: **if** *isfree*(v') /* $\Pi(v') = \bar{P}$ */ **then**
22: $gain(v') := gain(v') - w(e)$
23: **end if**
24: **end for**
25: **else if** $P(e) = 1$ **then**
26: **for all** $v' \in e$ **do**
27: **if** $\Pi(v) = P$ and *isfree*(v') **then**
28: $gain(v') := gain(v') + w(e)$
29: **end if**
30: **end for**
31: **end if**
32: **end for**
33: *recompute*(`MAXGAIN`)

corresponds to the actual gain in the objective function, if that vertex move is made.

The i^{th} $(i > 1)$ level gain is defined in terms of the *binding number* of the hyperedges incident on v. The binding number $\beta_P(e)$ with respect to part P, of a hyperedge $e \in \mathcal{E}$, is defined as the number of unlocked vertices in $P \cap e$, if e does not contain any locked vertices in P and ∞ otherwise. It may be interpreted as the number of vertices that need to be moved from part P in order to move all the vertices of e out of P. As this cannot be done if e has a locked vertex in P, $\beta_P(e)$ in that case has value ∞.

For the hyperedge cut objective, the i^{th} level gain of v, $\gamma_i(v)$, is formally defined in Equation 3.3.

$$\gamma_i(v) = |\{e \triangleright v : \beta_P(e) = i, \beta_{\bar{P}}(e) > 0\}| - |\{e \triangleright v : \beta_P(e) > 0, \beta_{\bar{P}}(e) = i-1\}| \quad (3.3)$$

Note that the gain calculation in Equation 3.3 uses the cardinality of the cutset for the value of the hyperedge cut objective (cf. Definition 2.12). This can easily be modified to incorporate weights on hyperedges.

Instead of maintaining a single integer gain quantity for each vertex move, as in the original FM algorithm, Krishnamurthy's algorithm maintains a *gain vector*, whose length is some constant l, determined at the beginning of the algorithm. When computing the `MAXGAIN` pointer in the gain bucket structures, the gain vectors are compared lexicographically. Nevertheless, ties may still occur in the 1^{st} to l^{th} level gains, in which case they may be broken on balance considerations. Failing that, the implementation of the gain bucket structure will determine the vertex to be moved (since one of the buckets in the linked-list pointed to by the `MAXGAIN` pointer will need to be taken out from the linked-list). The Krishnamurthy heuristic increases the per-pass complexity of the FM algorithm to $O(lN_H)$. Note that l is bounded above by $e_{max} - 1$, since Equation 3.3 would not be valid for larger values of l.

As noted above, the way in which a bucket from the `MAXGAIN` slot is chosen within the gain bucket data structure implicitly determines how ties are broken. The

exact implementation of this operation is underspecified in most descriptions of FM-type algorithms. In [Alp96], Alpert infers that a Last-In-First-Out (LIFO) scheme was used in the original FM implementation ([FM82]).

In [HHK97], the authors also investigated the use of First-In-First-Out (FIFO) and random bucket selection schemes. In experiments on benchmark circuits using both the original FM and Krishnamurthy's algorithms, the authors found that the LIFO selection scheme consistently outperformed the FIFO and random gain bucket selection schemes. Furthermore, they proposed an alternative formula for the computation of higher-level gains. For each cut hyperedge that has at least one locked vertex (in a part $P \in \Pi$, say) and does not have locked vertices in $\bar{P}$, the contribution of the hyperedge to the gain computation is increased for vertex moves from P to $\bar{P}$. This is done to encourage the movement of vertices of a cut hyperedge into a single part.

In [San89, San93], Sanchis extends Krishnamurthy's algorithm to directly compute a k-way partition (for any $k > 1$). Here, the algorithm maintains $k(k-1)$ gain bucket structures (one for each possible direction of vertex move). We henceforth refer to this algorithm as the k-way FM algorithm. Prior to this, the FM bipartitioning algorithm was used within a recursive bisection framework to compute k-way partitions. Differences between direct and recursive bisection approaches were described in Section 2.3.

Sanchis presents the extended k-way FM algorithm for the hyperedge cut, $k-1$ and $k(k-1)/2$ objective functions. The respective time complexities per pass, using l levels of gain, are $O(N_H lk(\log d_{max} + d_{max} l))$, $O(N_H(e_{max} + lk)(\log d_{max} + d_{max} l))$ and $O(m(e_{max} + lk + k^2)(\log d_{max} + d_{max} l))$.

We now consider a number of other iterative improvement algorithms in literature that are based on the framework of the KL/FM algorithms.

In [DA97], Dasdan and Aykanat present two algorithms which relax the restriction that a vertex is moved at most once during a pass. Like Sanchis' algorithm, their algorithms compute the k-way partition directly.

The first algorithm, Partitioning by Locked Moves (PLM), proceeds in passes and

each vertex is allowed to move to any of the $k-1$ neighbouring parts during a pass, subject to the partition balance constraint. Each pass consists of a number of phases. A phase attempts to find a better part for each vertex by making tentative vertex moves; the actual vertex destination is not finalised until after the last phase. Because it can be moved in more than one phase, a vertex may be moved more than once during a pass.

The second algorithm, Partitioning by Free Moves (PFM), does not employ a locking mechanism. Instead, the algorithm uses the vertices' *mobility* to decide whether a particular vertex should be moved. The mobility may be interpreted as a measure of the probability that the vertex ought to be moved. The authors conjecture that this probability should be proportional to the gain and inversely proportional to the number of moves that a vertex has made.

The computational time-complexity analysis yields a runtime of $O(NN_Hk(k+G))$ per pass for the PLM algorithm, where N is the (predetermined) number of moves in a pass and G the maximum possible gain of a vertex move. For the PFM algorithm, the runtime per pass is shown to be $O(N_Hk + k^2S + N(k^2 + kd_{max}e_{max}S))$, where S the size of the bucket array. In their experiments, the authors observed an improvement in solution quality over their implementation of Sanchis' k-way FM algorithm, while the improvement in runtime over Sanchis' algorithm increased with partition size.

Dutt and Deng proposed two types of iterative improvement algorithms for hypergraph bipartitioning. They are based on vertex move probabilities and cluster detection, respectively.

In [DD96a, DD99, DD00], the iterative improvement algorithms based on vertex move probabilities are described. The vertex move probability corresponds to the probability that a particular vertex will be *actually moved* during a pass. Note that in a pass of KL/FM-type algorithms, vertex moves that were made after the best partial sum of the gains are taken back. These vertices are not actually moved during the pass.

The algorithm PROP bases the computation of vertex move gain on the vertex

move probability. In [DD99], it is shown that the sample space of events given by the vertex moves recorded at the end of the pass is actually a probability space. The so-called *probabilistic gain*, derived from approximating the vertex move probability, is then used to guide the optimisation algorithm together with the deterministic (actual) gain.

Dutt and Deng describe an added enhancement to the probability-based algorithm, called SHRINK_PROP, in [DD99, DD00]. During gain updates after a vertex move, this algorithm increases the contribution to the objective function of those hyperedges that have already had vertices moved during the pass. This is done to encourage the movement of vertices belonging to a cut hyperedge into a single part. The time-complexity of PROP and SHRINK_PROP is $O(N_H \log n)$ (where n is the number of vertices in the hypergraph), when an AVL tree data structure is used, or $O(N_H)$, when a gain bucket data structure is used.

The iterative improvement algorithms based on cluster detection are motivated by a well-known weakness of FM-style algorithms. Namely, they only find solutions corresponding to local minima and can only evolve from an initial partition through relatively short-sighted moves. Dutt and Deng note in [DD96b, DD02] that hypergraphs from the domain of VLSI CAD are typically an aggregation of a number of highly connected clusters of vertices. They also show that when a random starting partition is constructed, there is a high probability that many of the strongly connected clusters will have vertices in both parts of the starting partition. Because the FM algorithm tends to make short-sighted moves (that may initially result in objective function gains), it is not able to move the vertices of the strongly connected clusters into a single part and achieve the global minimum.

The proposed algorithms use a modified vertex gain-update calculation. The gain of a vertex move can be expressed as the sum of the initial gain (gain of the move computed at the beginning of the pass) and updated gain (the cumulative total of gain updates after each vertex move).

In the algorithm CLIP (CLuster-oriented Iterative-improvement Partitioner), the

initial gain of each vertex move is only used to provide an ordering of the vertices for the first vertex move; subsequent moves are then made based on the updated gain only. The algorithm is designed to move strongly-connected clusters of vertices into one part because the updated gain values for vertices not adjacent to those already moved is zero, while the updated gain values for vertices adjacent to those already moved may be positive.

The algorithm CDIP (Cluster-Detecting Iterative-improvement Partitioner) further incorporates a cluster-detection mechanism. After the vertex move process reaches a positive maximum improvement point and there is no further improvement in the objective function during the following δ vertex moves, a cluster is said to have been "pulled" into a part at the maximum improvement point. The algorithm then takes back the δ vertex moves and unlocks the vertices involved. All of the unlocked vertices are then ordered by their total gains (as at the beginning of the pass) and the updated gains are used to select subsequent vertex moves.

The CLIP algorithm has a runtime of $O(N_H)$ per pass, while the runtime of CDIP is $O(\max\{cn, N_H\})$ per pass, where c is the number of clusters detected in a pass of the algorithm.

Experimental evidence presented in [DD02] suggests that CDIP outperforms CLIP in terms of partition quality, while having slightly longer runtimes. Both algorithms result in a significant improvement in partition quality over the original FM algorithm. Best results were obtained when CLIP was overlaid onto the PROP algorithm. In experiments comparing the algorithms with the state-of-the-art multilevel tool `hMeTiS` [KK98a], algorithms based on CLIP/CDIP produced partitions of similar quality to those produced by `hMeTiS`, while exhibiting faster runtimes.

In [EC99], Eem and Chong propose the use of multiple gain bucket data structures for a single direction of vertex move. This is motivated by the CLIP algorithm (described above) that distinguishes between updated gain and actual gain values [DD96b, DD02]. Multiple gain bucket data structures are required because the

updated gain values are always given a higher priority than the actual gain values when computing vertex moves. Actual gain values are used when updated gain values cannot determine the best move or when tie-breaking is required.

Dutt and Theny [DT97] consider relaxing the balance constraint in the hypergraph partitioning problem at intermediate stages of the iterative improvement algorithm in order to improve its hill-climbing capability. Two methods are proposed. In the first, for every vertex move that results in constraint violation, the benefit and cost of such a move is estimated (using look-ahead formulations of the PROP algorithm [DD96a, DD00]). In the second method, a "benefit factor" and an "acceptance threshold" are computed for a constraint-violating vertex move. A move that violates the partition constraint is accepted if the benefit factor is greater than the acceptance threshold. The benefit factor and the acceptance threshold depend on the stage of the partitioning process, the location of the vertex and those of its neighbours.

The algorithm introduced in [CLL$^+$97] also attempts to move highly-connected clusters of vertices into a single part. The approach is motivated by considering the different states of a particular hyperedge during a pass. During a pass, a given hyperedge is *free*, when it contains only free vertices. It may be *loose*, when it has locked vertices in exactly one part. Finally, it may also be *locked*, when it has locked vertices in two (or more, in a case of a multi-way partition) parts. The *anchor* part of a loose hyperedge is defined as the part that contains its locked vertices and the remaining part(s) that have one or more of its free cells are called *tail* part(s).

The LR (Loose hyperedge Removal) algorithm increases the gains associated with moving the vertices from the tail part of a loose hyperedge to its anchor part, subject to a maximum gain threshold. If a hyperedge remains cut during an entire run of the iterative improvement algorithm, the authors refer to it as *stable*. They note from experiments performed in [SNK95] that more than 80% of the hyperedges in the final cutset are stable, trapping the FM algorithm into a local minimum and limiting the solution quality.

However, stable hyperedges may be removed following a run of the LR algorithm. A stable hyperedge is randomly picked and all of its vertices are moved into the part with the least weight. Vertices moved during this process cannot be moved to other parts during the handling of another stable hyperedge. This process is repeated until a pre-determined percentage of stable hyperedges are processed or no more moves are possible. Then, another run of the FM algorithm is performed using the outcome of the stable hyperedge removal as its initial partition. The integration of the two approaches is termed LSR (Loose and Stable hyperedge Removal).

Cong and Lim note in [CL98] that Sanchis' extension of the FM algorithm to multi-way partitioning is outperformed by the recursively-formulated FM bisection algorithm in terms of both the solution quality and run time on the MCNC and ISPD98 benchmarks. They note two main drawbacks to Sanchis' algorithm. Firstly, due to many candidate moves at any one time (as there are $k(k-1)$ possible directions of vertex move), the algorithm is prone to making the wrong choice. Secondly, the algorithm requires $O(k^2n)$ memory to store the gain buckets, which in the case of large k may be prohibitive. The authors' experiments empirically confirmed the informally accepted notion that Sanchis' algorithm was not suitable for partitioning large hypergraphs from the domain of VLSI CAD.

Cong and Lim instead propose a different formulation for the k-way extension of the FM algorithm. Their algorithm applies the FM bipartitioning algorithm in pair-wise fashion across the k parts. Note that there are $O(k^2)$ possible part pairs. During the algorithm the pairs may either be chosen at random, across all possible pair-wise configurations, be based on the cutset of hyperedges across pairs of parts or, finally, be based on the previously achieved gain across pairs of parts. In their experiments, the authors note a small improvement in terms of solution quality over the recursively-formulated FM on the MCNC and ISPD98 benchmarks, with comparable run times.

[CKM99b] considers the implication of *fixed* vertices in the context of hypergraph partitioning. Fixed vertices effectively provide additional constraints to the par-

titioning problem. A vertex is said to be fixed to a part because it must be allocated to that part in the final partition. This partitioning problem formulation arises in VLSI CAD applications. For example, during the circuit design process, cells in the sub-circuit currently being partitioned may be fixed because they are connected to chip I/O.

The authors note that in the presence of fixed vertices, the hypergraph partitioning problem becomes *easier* because the feasible solution space is reduced when compared to the problem formulation where no vertices are fixed prior to partitioning. As expected, the partitioning runtimes are observed to decrease substantially when a large percentage of vertices in the hypergraph are fixed. However, in some of the experiments where the fixing of vertices to parts is taken from a previously computed partition (a "good" partition), the solution quality did not improve over the unconstrained case (where no vertices were fixed). The authors conjecture that it is possible for a partitioning problem to be "overconstrained", so that the reduction of the solution space due to the presence of fixed vertices does not compensate for the loss in ability to explore more of the actual solution space.

The FM bipartitioning algorithm was more recently considered in [CKM00a], mainly in the context of multilevel partitioning (cf. Section 3.5). However, the following proposed modifications also apply to the flat FM bipartitioning algorithm:

1. The use of an "illegal" initial solution that puts all moveable vertices into a single part.

2. The relaxation of the acceptance criterion for legal moves – motivated by [DT97].

3. Randomisation of the gain computation at the beginning of the pass to compensate for deterministic inital solution generation. This is done by computing gains of legal moves in a random order.

4. The use of a LIFO gain bucket structure implementation [HHK97] and giving preference at the head of the gain buckets to vertices adjacent to fixed vertices (when fixed vertices are present).

The above proposed enhancements are combined into an FM implementation called VRW which was empirically compared to the simple FM [FM82] and CLIP algorithms [DD96b]. The authors recommend it for the hypergraph partitioning problem without fixed vertices in applications where only a small number of runs of the partitioning algorithm can be carried out; for example, due to restrictions on the runtime. This is because the VRW approach lacks the randomisation inherent in multiple runs of the original FM, since it generates the same initial solution for each run.

3.4.2 Simulated Annealing

Simulated Annealing was introduced in [KJV83, Cer85] as a general approach to optimisation and is based on the Metropolis procedure from statistical mechanics. Kirkpatrick et al. compare computing solutions in optimisation problems by iterative improvement with the microscopic rearrangement processes modelled by statistical mechanics [KJV83].

The Metropolis procedure is applied to move-based optimisation as follows. Given a starting solution and a prescribed neighbourhood structure, a random neighbouring feasible solution is selected (analogous to a collection of atoms in equilibrium at a given temperature being given a small random displacement). The resulting change in the objective function δf_o (or in the physical analogy, the energy δE) is computed. If $\delta f_o \leq 0$, then the resulting partition is taken to be the starting point of the next iteration. Otherwise ($\delta f_o > 0$), the acceptance of the resulting partition is treated probabilistically.

The probability of the resulting partition being accepted is given by $exp(-\delta f_o/T)$, where T is the so-called *temperature* parameter (in the physical process, T is the temperature of the system and the probability of the displacement occurring is

given by $exp(-\delta f_o/k_B T)$, where k_B is the Boltzmann constant). The temperature is taken to be a control parameter in the annealing process and a deterministic annealing schedule by which T varies is specified. Typically, at each temperature, the above iterative procedure is allowed to proceed long enough for the system to reach a steady state. Note that, as T approaches zero, the annealing process tends towards being entirely deterministic, allowing only greedy moves.

In [Haj88], it was shown that the simulated annealing algorithm will converge in probability to the globally optimum solution if, and only if, the annealing schedule allows T to tend to zero sufficiently slowly. More specifically, given an annealing schedule $\{T_i : i \geq 0\}$ such that

$$T_1 \geq T_2 \geq \quad \ldots \quad \geq T_i > 0 \tag{3.4}$$

$$\lim_{i \to \infty} T_i \quad = \quad 0 \tag{3.5}$$

the necessary and sufficient condition for convergence in probability to some global optimum solution is

$$\sum_{i=1}^{\infty} exp(-\delta^*/T_i) \quad = \quad \infty \tag{3.6}$$

where δ^* is a constant, defined in terms of local minima of the optimisation problem.

Although it has been successfully applied across many other optimisation problems, it is noted in [AHK95] that Simulated Annealing is not yet viewed as "practical" for hypergraph partitioning in the domain of VLSI CAD, due to its long runtimes. Simulated Annealing is, however, widely used in other optimisation problems within the VLSI CAD domain, such as cell placement [CKR+97, AYM+04].

3.4.3 Genetic Algorithms

The motivation behind genetic algorithms is Darwin's theory of natural selection [Hol75]. A genetic algorithm (GA) starts with an initial population of feasible

solutions that evolves over *generations*, so that solutions in the current generation are replaced with a set of *offspring* feasible solutions in the next generation. A GA usually determines the candidate offspring using a *crossover* operator on solutions in the current generation, while a *mutation* operator allows small random pertubations to the current solution. The crossover operator is analogous to mating, whereby two solutions are selected from the current population and their descriptors are partially mixed to produce offspring. Finally, a GA requires a *replacement* operator that determines the offspring that will replace members of the current population.

In [BM96], Bui and Moon describe a genetic algorithm for the graph bipartitioning problem. They achieve results comparable with the KL algorithm on different types of randomly generated graphs (rather than benchmark graphs or graphs taken directly from applications), so a comparison with alternative approaches is difficult. Alpert et al. propose a hybrid approach in [AHK96], whereby they use an existing state-of-the-art graph partitioner `MeTiS` [KK99] as a solution generator. The generated solutions are then combined within a genetic algorithm framework. Note that this approach involves a hypergraph-to-graph transformation (described in Section 3.3), and so the partitioner cannot directly optimise a desired objective function on the hypergraph. In experiments on the ACM/SIGDA (MCNC) benchmark suite, results comparable with the FM algorithm are reported.

In [Are00, AY04, SO99], genetic algorithms are applied directly to the hypergraph partitioning problem. The experimental study of the algorithm proposed in [SO99] lacks comparisons with state-of-the-art approaches and uses an older partitioning benchmark suite. The experimental evidence in [Are00, AY04] suggests that the genetic algorithm approach is slower than iterative improvement approaches. Kim et al. propose a combination of a genetic algorithm with an FM-based heuristic for hypergraph bipartitioning in [KKM04]. The algorithm is experimentally compared to a simple FM implementation and the `hMeTiS` multilevel tool [KK98a] on the (older) ACM/SIGDA benchmark circuits. The

authors observe that their FM-based heuristic without the genetic algorithm is an improvement over the simple FM algorithm. In combination with the genetic algorithm, the resulting partitioning algorithm computes partitions that are competitive with those produced by `hMeTiS`. The reported runtimes are also very competitive with `hMeTiS`. We note, however, that the test hypergraphs used are considerably smaller than those in the most recent circuit benchmark [Alp98].

3.4.4 Tabu Search

Tabu search is another well-known heuristic approach that has been applied to a number of optimisation problems. It was introduced by Glover in [Glo89].

The Tabu heuristic combines the recent history of the optimisation with iterative improvement to seek feasible solutions. A list of the r most recent moves is kept (with r a constant that is determined at the start of the optimisation), where a move is defined in terms of the neighbourhood operator. The list is called the Tabu list (from "taboo") and indicates the moves that cannot be made (these moves have been *prohibited*). An aspiration criterion can be used to temporarily release a move from its Tabu status [AV03]. Stopping criteria for the algorithm are also required and this may be a (fixed) maximum number of moves after which the search routine will terminate [AV00]. The Tabu heuristic requires careful implementation as cycling behaviour (where the algorithm repeats the same set of vertex moves over and over) must be avoided [Saa04].

In [BB99], Battiti and Bertossi apply Tabu search to the graph partitioning problem. They draw an analogy with the KL and FM algorithms; once the KL and FM algorithms move a vertex during a pass, they fix it so that it does not move again during that pass. A similar restriction is imposed by the Tabu list.

Battiti and Bertossi observe that the value of r, when fixed at the beginning of the algorithm, strongly influences its performance. Moreover, the best choice of r also depends on the graph being partitioned so that a value of r performing well on one graph does not necessarily lead to good performance on another.

Instead of keeping the value of r fixed, they propose reactive and randomised prohibition. In randomised prohibition, the value of r is varied according to a randomised function. Reactive prohibition implies a self-tuning of r based on the history of the optimisation. A previously successful value of r is chosen with higher probability and retained for a greater number of iterations.

The solution quality produced by the proposed algorithms is promising when compared to other state-of-the-art graph partitioning tools such as `MeTiS` [KK99] and `Chaco` [HL94]. However, the current implementation suffers from very long runtimes.

More recently, in [AV03], Areibi and Vanelli consider Tabu search in the context of hypergraph partitioning for VLSI CAD. Their Tabu implementation produces partitions of better quality than Sanchis' k-way FM algorithm, Simulated Annealing and a Genetic algorithm and also exhibits faster runtimes than the alternative approaches tested. However, given lack of experimental comparison with current state-of-the-art, it is difficult to draw meaningful conclusions about the relative quality of the proposed algorithm.

In [Saa04], Tabu search was combined within a multilevel framework for hypergraph partitioning. The reported experimental results, like those for graph partitioning in [BB99], suggest that the approach yields very competitive partitions. However, as was the case for the implementation in [BB99], the observed runtimes appear prohibitively long.

We note that unlike the KL/FM-based iterative improvement algorithms, which have matured and possess relatively simple and clearly-defined implementations, the Tabu heuristic presents the algorithm designer with a number of difficult and non-obvious implementation choices. These include the value of Tabu parameter r, aspiration and termination criteria. To that end, more research is necessary to establish whether Tabu-based algorithms can be competitive with current state-of-the-art hypergraph partitioning algorithms.

3.4.5 Other Move-Based Partitioning Approaches

In [RM03], Ramani and Markov develop a bipartitioning heuristic based on a combination of two different flat partitioning algorithms. These are a stochastic local search heuristic for the Boolean Satisfiability problem, WalkSAT [SKC94], and the FM algorithm, respectively.

WalkSAT is used to develop a partitioning algorithm called WalkPart. It performs a sequence of feasible vertex moves, until a prescribed number of moves have been made. Moves are chosen according to criteria motivated by WalkSAT; with probability x, a vertex is moved at random, and with probability $1 - x$, a move is chosen so that it minimises a cost function. In the authors' implementation, x has a fixed value of 0.1.

The hybrid approach alternates runs of the FM algorithm, via the FMPart implementation [CKM00b], with WalkPart. The authors observe better results with the hybrid algorithm than with either of the two approaches and conjecture that the algorithm would perform well within a multilevel context.

3.5 Multilevel Algorithms

3.5.1 Introduction

In literature, the multilevel approach is also known as the hierarchical or clustering-based approach [AHK95].

In [AHK95], Alpert et al. observe that most local minima attained by flat heuristic algorithms are of only "average" quality. That is, when a number of runs of a partitioning algorithm are performed, objective function values for most of the local minima attained will be clustered around the value of the sample mean. Kauffman and Levin [KL87] call this phenomenon the "Central Limit Catastrophe".

In general, the number of local minima (with respect to a heuristic) increases

as the size of the partitioning problem increases. Thus, the probability of a flat heuristic algorithm converging to a local minimum that is very close to the global minimum decreases, as the size of the partitioning problem increases. Consequently, there is a need to ensure that flat partitioning algorithms scale better with problem size, in terms of the partition quality produced.

In order to attain local minima values that are relatively close to the global minimum, the number of runs of the flat partitioning algorithm may be increased. However, the runtimes of such an approach are likely to become prohibitive for increasingly larger partitioning problem sizes.

In the multilevel approach, however, the original hypergraph is approximated by successively smaller hypergraphs. The space of feasible solutions, and hence the number of local minima corresponding to the partitioning heuristic used, should be significantly smaller for the approximate hypergraphs than for the original hypergraph.

Provided that a good approximation algorithm exists, a partition that is close to the optimum partition of the approximate hypergraph should correspond to a partition that is close to the optimum partition of the original hypergraph. The motivation for the multilevel approach is that it should be significantly easier to compute a good partition of the approximate hypergraph than to compute the corresponding partition of the original hypergraph.

An approximation to the original hypergraph can be constructed by clustering the vertices of the original hypergraph. In literature, the process of clustering vertices is also called vertex matching or coarsening. The resulting clusters form the vertices of the approximate (coarse) hypergraph. This method of approximation also maintains the cut properties of the hyperedges. This means that when a partition is projected from a coarse hypergraph onto the original hypergraph, hyperedges in the original hypergraph that are cut by the projected partition correspond to those that were cut in the coarse hypergraph. Thus, it should be possible to preserve the value of the partitioning objective function when projecting partitions from the coarse hypergraphs.

The multilevel approach naturally falls into three phases. The *coarsening* phase computes approximation(s) of the original hypergraph. Typically, many levels of approximation are used (hence *multilevel*). Next, the *initial partitioning* phase applies a flat heuristic partitioning algorithm to the coarsest approximation. Finally, the *uncoarsening* phase projects the partition of the coarsest hypergraph approximation through successive levels of approximation back onto the original hypergraph.

The idea of approximating the problem in order to make it "easier" for heuristic partitioning algorithms was first investigated in the context of graph partitioning. In [BCLS87], Bui et al. conjecture that constructing an approximation to the graph by contracting edges may improve the quality of partitions produced by the KL graph partitioning algorithm. The approximation scheme, using a single approximation graph, was formally investigated in [BHJL89] and applied to KL and Simulated Annealing bipartitioning algorithms.

Multilevel partitioning was (independently) investigated by researchers from the scientific computing and VLSI CAD communities. Multiple levels of approximation for graph partitioning were proposed in [BS94, HL93]. Both use a spectral partitioning algorithm to partition the coarsest graph. In [BS94], Barnard and Simon project the eigenvectors between the graphs at consecutive levels of approximation during the uncoarsening phase. It appears more natural to project partitions and this is what Hendrickson and Leland do in [HL93]. Moreover, they also note that further heuristic refinement to the projected partition is possible because the uncoarsened graph has more degrees of freedom than the coarse graph. Cong and Smith model the circuit hypergraph by a graph in [CS93]. They construct a single approximation G' to the original graph G, but perform refinement at multiple levels by unclustering only selected vertices of the approximation G' at each level. This continues until all the vertices in G' have been unclustered and the refinement is then performed on G. Hauck and Borriello consider a number of algorithms for constructing hypergraph approximations during the coarsening phase in [HB97].

3.5.2 The Coarsening Phase

In the first approximation-based algorithms, the coarsening phase involved the computation of a single approximation to the original hypergraph. In general, a sequence of successive approximations is preferred.

Formally, the original hypergraph $H(V, \mathcal{E})$ is approximated by a sequence of successively smaller hypergraphs $H_i(V_i, \mathcal{E}_i)$, $1 \leq i \leq c$. Whenever $i > j$, it is the case that $|V_i| < |V_j|$ and the number of vertices in the coarsest approximation H_c has some fixed upper bound that depends on the number of parts in the partition sought, k. In [Kar02], the requirements of a coarsening algorithm are formalised as follows:

1. Any partition of a coarse hypergraph should be projected easily to a partition of the successive finer hypergraph.

2. The value of the objective function for a projected partition should be less than or equal to the value of the objective function for the partition of the successive coarser hypergraph.

For hypergraph partitioning objective functions that are of practical interest (such as the $k-1$ objective), the above requirements are satisfied by grouping the vertices of a given hypergraph into clusters and allowing the resulting clusters to be the vertices of the coarse hypergraph. The clustering of the vertices of the hypergraph H_i, to form vertices of the coarse hypergraph H_{i+1}, is denoted by the map $g_i : V_i \rightarrow V_{i+1}$, where

$$\frac{|V_i|}{|V_{i+1}|} = r, r > 1 \tag{3.7}$$

and r is the prescribed reduction ratio between two successive levels of approximation.

The sum of the vertex weights in V_i is preserved by setting the weight of a coarse vertex formed by a cluster $C \subset V_i$ to be the sum of the weights of the vertices in C. The map g_i is used to construct $\mathcal{E}_{i+1}$ from $\mathcal{E}_i$ by applying it to every

vertex in each hyperedge $e \in \mathcal{E}_i$. Single vertex hyperedges in $\mathcal{E}_{i+1}$ are discarded because they will span at most one part (and thus cannot contribute to the objective function). Duplicate hyperedges are replaced by a single instance of that hyperedge, and the weight of this hyperedge is set to the sum of the weights of the duplicate hyperedges.

In [Kar02], Karypis outlines some desirable characteristics that a coarsening algorithm should possess:

1. A near-optimal partition of the coarsest hypergraph H_c should project to a near-optimal partition of H.
2. The successive coarser hypergraphs should have significantly fewer large hyperedges than the original hypergraph.
3. The sum of the hyperedge weights in the successive coarser hypergraphs should decrease as quickly as possible.

Reducing the number of large hyperedges improves the performance of iterative improvement algorithms (such as FM and KL) because it is difficult for such algorithms to "pull" vertex clusters connected by large hyperedges into a single part when considering only the vertex move gain.

Coarsening Approaches

Coarsening algorithms attempt to cluster strongly connected vertices together. In a graph, relative connectivity between any two connected vertices is captured by the weight of the edge that connects them. In a hypergraph, it is necessary to consider the *vertex connectivity graph*. A connectivity graph $G_i(V_i, \mathcal{E}'_i)$ of a hypergraph $H_i(V_i, \mathcal{E}_i)$ is such that an edge $e \in \mathcal{E}'_i$ exists between vertices $v, v' \in V_i$ if, and only if, there exists a hyperedge $h \in \mathcal{E}_i$ such that $v \in h$ and $v' \in h$. The edge weights in $G_i(V_i, \mathcal{E}'_i)$ need to be chosen so that they reflect the relative connectivity of the vertices they connect. A function that quantifies this connectivity is called a vertex connectivity metric.

Unlike the graph-to-hypergraph transformations mentioned in Section 3.3, it is not necessary that the weights of the edges in $G_i(V_i, \mathcal{E}'_i)$ accurately model the cut properties of the objective function. We note that most coarsening algorithm implementations proceed in a greedy manner; the set of vertices is traversed and for each unmatched vertex $v \in V_i$, a neighbouring vertex $v' \in V_i$ that maximises some prescribed vertex connectivity metric is selected for matching with v. The graph $G_i(V_i, \mathcal{E}'_i)$ is implicitly traversed, rather than explicitly constructed.

The early algorithms for coarsening *graphs* [BHJL89, HL93] use a purely random edge matching scheme. Unmatched vertices are only allowed to match with other unmatched vertices, resulting in vertex clusters of size two. In [BS94], Simon and Barnard coarsen the graph by first computing a maximal independent set of vertices $V' \subseteq V_i$ and then constructing the coarse vertices by matching vertices in the maximal independent set with their neighbours[2].

In [Kar02], Karypis calls the random edge matching scheme Edge Coarsening (EC), when it is applied to hypergraphs. A weighting of the edges in the connectivity graph $G_i(V_i, \mathcal{E}'_i)$ is not computed. The vertices of $H_i(V_i, \mathcal{E}_i)$ are visited in a random order. For each vertex $v \in V_i$, all unmatched vertices that belong to hyperedges incident on v are considered and one is selected at random (with a uniform probability) to match with v. The ratio $|V_i|/|V_{i+1}|$ is controlled by terminating the algorithm early and copying the remaining unmatched vertices into V_{i+1}. Note that the EC coarsening algorithm can reduce the number of vertices in successive hypergraphs by a factor of at most two.

One may incorporate the length of the hyperedges, the weighting on the vertices and the weighting on the hyperedges of the hypergraph into the vertex connectivity metric. Roy and Sechen propose in [RS93] that each hyperedge $e \in \mathcal{E}_i$ should contribute $1/(|e| - 1)$ to the weight of each edge in the connectivity graph G_i. This is the heavy-edge variation of the edge coarsening algorithm in [Kar02].

[2] An independent set of vertices of a graph $G(V, \mathcal{E})$ is a subset $V' \subseteq V$, such that for any two vertices $u, v \in V'$, there is no edge in $\mathcal{E}$ that connects u and v. A maximal independent set of vertices is an independent set $V' \subseteq V$ such that for all $v \in V \setminus V'$, the set $V' \cup \{v\}$ is no longer an independent set.

Alpert et al. propose the following more sophisticated vertex connectivity metric in [AHK97] for their Match coarsening algorithm:

$$conn(v_i, v_j) = \frac{1}{w(v_i)w(v_j)} \sum_{\{e \in \mathcal{E} : e \rhd v_i, e \rhd v_j\}} \frac{1}{|e|} \tag{3.8}$$

The connectivity value between any two vertices is inversely proportional to the product of their weights in order to discourage formation of large clusters in successive coarser hypergraphs.

Many authors have noted that it may be beneficial to allow unmatched vertices to match with vertices that have been already matched, if this is suggested by the vertex connectivity metric [KK98b, HB97]. Hauck and Borriello propose a vertex connectivity metric that uses hyperedge bandwidth in [HB97]. They define the bandwidth $b(e)$, of a hyperedge $e \in \mathcal{E}$, to be $1/(|e| - 1)$. Their metric is shown in Equation 3.9 below:

$$conn(v_i, v_j) = \frac{1}{w(v_i)w(v_j)} \sum_{\{e \in \mathcal{E} : e \rhd v_i, e \rhd v_j\}} \frac{b(e)}{(deg(v_i) - b(e))(deg(v_j) - b(e))} \tag{3.9}$$

Karypis and Kumar propose the First Choice (FC) coarsening algorithm [KK98b]. In the FC algorithm, each hyperedge contributes $1/(|e| - 1)$ to an edge in the vertex connectivity graph G_i. The vertex set V_i is traversed in random order. For each vertex $v \in V_i$, all vertices (both matched and unmatched) adjacent to v are considered. The vertex connected via the edge (in the vertex connectivity graph) with the largest weight is matched with v, subject to a maximum cluster weight threshold. Ties are broken in favour of unmatched vertices.

In [CKM00a], an enhancement to the heavy-edge coarsening algorithm was proposed. The algorithm, called PinEC, computes the following vertex connectivity metric:

$$conn(v_i, v_j) = \frac{1}{w(v_i) + w(v_j)} \sum_{\{e \in \mathcal{E} : e \rhd v_i, e \rhd v_j\}} \frac{1}{b(e)} \tag{3.10}$$

where the bandwidth $b(e)$ is two for hyperedges of size two and one for larger hyperedges (i.e. different from the bandwidth definition in Equation 3.9).

Çatalyürek and Aykanat propose the Heavy Connectivity Clustering (HCC) algorithm in [cA99]. Here the connectivity measure between a vertex v and a cluster

of vertices C (possibly consisting of a single vertex) is given by:

$$conn(v, C) = \frac{\mathcal{E}_{v+C}}{W_{v+C}} \tag{3.11}$$

where $\mathcal{E}_{v+C}$ is the cardinality of the set of hyperedges that contain the vertex v and at least one vertex from C, and W_{v+C} is the total weight of the cluster consisting of v and the vertices from C.

A number of algorithms try to identify more intuitive cluster properties through global considerations. As this requires traversal of the entire vertex connectivity graph G_i, such algorithms have larger runtime complexities than the greedy schemes described above.

Hagen and Kahng [HK92] suggest cycles in random walks of the connectivity graph as a clustering measure. They begin by asking the question: "how hard is it to separate two vertices x and y in a graph?". If there are more distinct paths from x to y, then x and y likely belong to the same natural cluster. On the other hand, if there are fewer distinct paths from x to y, x and y do not belong to the same natural cluster.

The proposed coarsening algorithm computes a random walk that covers the entire graph and extracts cycles from it. The cycles are then used to compute a connectivity value between x and y called *sameness*. This considers how often y occurs in cycles originating at x and vice versa. This random walk-based coarsening algorithm is prohibited by the $O(n^3)$ runtime (where $n = |V_i|$).

Cong and Lim [CL00] use the *edge separability* between two vertices connected by an edge in the vertex connectivity graph as a clustering measure. Recall from Section 2.2.2 that an edge separator of a graph $G(V, \mathcal{E})$ is a subset of edges $E \subseteq \mathcal{E}$ that are cut by a bipartition of G. Edge separability of vertices x and y connected by the edge $e = (x, y)$ is the cardinality of the smallest edge separator (i.e. the edge-cut of the minimal bipartition), such that e is cut by the bipartition. If x and y have a high edge separability, then it is likely that x and y belong to the same cluster (they are hard to separate). If, on the other hand, x and y have low edge separability, then they probably do not belong to the same natural cluster.

When constructing $G_i(V_i, \mathcal{E}_i')$, Cong and Lim let each hyperedge $e \in \mathcal{E}_i$ contribute $1/(|e|-1)$ to the edge weights in $\mathcal{E}_i'$. They use the algorithm proposed in [NI92] to compute a tight lower bound on edge separability of two vertices. A set of contractable edges $Z(G_i)$ (with edge-separabilities greater than a specified threshold) is then computed. The contractable edges in $Z(G_i)$ are first sorted then contracted in decreasing order of *rank*, subject to a maximum allowed cluster weight. The rank of an edge $e = (x, y)$ is given by Equation 3.12 below:

$$rank(e) = \frac{q(e)}{\min\{deg(x), deg(y)\}} \tag{3.12}$$

where $q(e)$ is the computed lower-bound on edge separability. The rank values are chosen in such a way that they give higher priority to edges with larger edge separability values and edges whose contraction results in a smaller increase of degree. The overall complexity of the edge separability-based coarsening algorithm is $O(n \log n)$ (where $n = |V_i|$).

Absorption has also been proposed for identifying clusters in a hypergraph [SS93]. The following definition from [CL00] (in terms of the vertex connectivity graph) is equivalent to the definition given in [SS93] for absorption in hypergraphs. Each hyperedge $e \in \mathcal{E}_i$ contributes $1/(|e|-1)$ to the edge weight in the vertex connectivity graph G_i. The absorption of a cluster of vertices C is given by:

$$absorption(C) = \sum_{\{e'=(x,y):x \in C, y \in C\}} w(e') \tag{3.13}$$

A coarsening algorithm may also extract clusters from a pre-computed vertex ordering, as proposed in [AK94a]. A vertex ordering S is initialised with a randomly chosen vertex and unordered vertices are then iteratively added in a greedy manner, so that the chosen vertex maximises a general attractor function with respect to S. This provides a vertex ordering which can be computed in $O(n \log n)$ time using a Fibonacci heap, if the attractor function is monotonic (decreasing or non-decreasing) in the size of the ordered set S.

When computing the attraction between an unordered vertex v and S, a window of size W is defined, so that at most the last W of the ordered vertices exert

full attraction on v. The coarsening map g_i is then computed by forming vertex clusters from consecutive vertices in the ordering [AK94a, AK94b].

A clique is an intuitive cluster property that has been proposed as a coarsening metric. In [CS93], Cong and Smith suggest forming clusters by contracting cliques of the vertex connectivity graph. A clique is contracted if it does not exceed prescribed size (percentage of total number of vertices contracting into a single coarse vertex), area (percentage of the total weight of the graph) and density thresholds. Here, the *density* of a vertex cluster C is given by $W_C|C|(|C|-1)/2$, where W_C is the total weight of the edges in the cluster.

Similar in spirit to the clique clustering approach described above is hyperedge coarsening [KAKS97, KK98b, Kar02]. Here, all the vertices in a hyperedge will form a cluster. Hyperedge coarsening potentially removes a large number of hyperedges in the successively coarser hypergraphs.

Toyonaga et al. [TYAS94] sort the hyperedges in non-decreasing order by cardinality and then traverse the sorted list of hyperedges, with vertices of a hyperedge forming a cluster if they have not been matched previously as part of another hyperedge. Karypis et al. [KAKS97] first sort the hyperedges in a non-increasing order of weight and then hyperedges of the same weight are sorted in a non-decreasing order of cardinality. The sorted set of hyperedges is then traversed and vertices of a hyperedge form a cluster if they have not been matched previously as part of another hyperedge.

Karypis et al. [KAKS97] also note that hyperedge coarsening leads to a large variance in vertex weight and does not sufficiently reduce the size of large hyperedges at coarser levels. They instead propose a modified hyperedge coarsening algorithm, which initially proceeds like the hyperedge coarsening algorithm. However, having contracted a maximal set of disjoint hyperedges, modified hyperedge coarsening traverses the list of hyperedges again. Then, for each hyperedge that has not been contracted, the vertices that do not belong to any other contracted hyperedge are contracted to form a vertex in the coarse hypergraph.

An interesting approach to coarsening is presented in [Saa04]. Here the coarsening

is a by-product of a pass of an iterative improvement bipartitioning algorithm. As vertices are moved and locked during the pass, they are deemed *contractable* and available for matching with other vertices.

The vertex matching procedure proceeds as follows. After a vertex $v \in V_i$ is moved, it is made contractable and a set X, consisting of v and all other contractable vertices that are connected to v via *critical* hyperedges, is computed (a hyperedge $e \in \mathcal{E}_i$ is defined as critical with respect to a vertex $v \in V_i$ if it is removed from the cutset after v is moved). If X has more than one vertex in it, then all vertices in X are clustered together to make a new coarse vertex and they are no longer deemed contractable. Otherwise, v remains contractable.

Remarks on Coarsening Algorithm Implementation

The performance of a coarsening algorithm (that is, its ability to identify natural vertex clusters) across different hypergraph problem instances strongly depends on a number of implementation choices. A detailed discussion of these is presented in [Kar02], based on experiences gained through the development of the `hMeTiS` hypergraph partitioning tool [KK98a].

In a "natural" implementation of a coarsening algorithm, an increase in the variance of vertex weight at coarser levels is expected. This is because hypergraphs occurring in applications such as VLSI CAD usually have a number of strongly connected vertex clusters, which will form increasingly large vertices at successive coarser levels. Potentially, this behaviour reduces the number of feasible partitions of the coarsest hypergraph. The choice of the maximum allowed coarse vertex weight (i.e. imposing an upper bound on vertex weight in the coarser hypergraphs) is thus a parameter that strongly influences coarsening algorithm performance.

The rate at which successive coarser hypergraphs are reduced (r in Equation 3.7) will also affect the performance of the multilevel partitioning algorithm. A greater number of levels (given by a lower reduction ratio) potentially improves the partition quality produced by the multilevel algorithm because refinement is per-

formed at more levels. However, this comes at the cost of longer runtimes and larger memory utilisation. Thus, there is a trade-off, for which the optimal configuration should be dependent on the nature of the hypergraph problem instance. Karypis and Kumar report that they achieve an acceptable balance between runtime and partition quality with $1.5 \leq r \leq 1.8$ [Kar02].

The remaining significant implementation issue is determining when to stop the coarsening process. We note that there are two potentially conflicting factors to consider. If the coarsening process terminates too early and the coarsest hypergraph is relatively large, it may not be possible to find a sufficiently good partition of the coarsest hypergraph. On the other hand, if the coarsest hypergraph is too small, the space of feasible solutions may be too small and the optimal partition of the coarsest hypergraph may be unacceptably worse than the optimal partition of the original hypergraph.

This presents another trade-off for which the optimal configuration depends on the given partitioning problem. In [Kar02], Karypis terminates the coarsening algorithm when the coarse hypergraph has around $30k$ vertices, where k is the number of parts in the partition sought. The coarsening process may also be terminated if the coarsening algorithm can only reduce the current hypergraph H_i by an insufficient amount, thus making further coarsening unproductive.

Finally, the runtime of the coarsening phase may be reduced, without greatly affecting coarsening quality in many cases, by not considering large hyperedges. In [cA99], Çatalyürek and Aykanat use a threshold of $4e_{avg}$, where e_{avg} denotes the average hyperedge length in the hypergraph H_i.

3.5.3 The Initial Partitioning Phase

The aim of the initial partitioning phase is to construct a partition of the coarsest hypergraph $H_c(V_c, \mathcal{E}_c)$ that satisfies the partition balance constraint and optimises the objective function. Since H_c should be significantly smaller than H, the time taken by the initial partitioning phase will usually be considerably less

than the time taken by the coarsening and uncoarsening phases of the multilevel framework.

Effectively, any heuristic optimisation algorithm that can be applied to hypergraph partitioning may be used to compute the *initial partition*. For example, in the context of graph partitioning, [HL93, BS94] use a spectral partitioning algorithm, while for hypergraph partitioning, [AY04] use a genetic algorithm. For hypergraph partitioning, however, iterative improvement algorithms (cf. Section 3.4.1) have typically been used to generate the initial partition. A number of partitions are computed from multiple runs of a flat partitioning algorithm.

Hauck and Borriello [HB97] evaluate a number of simple methods for generating a starting feasible partition, prior to using a heuristic partitioning algorithm. Random partition creation (by bisecting a random ordering of the vertices), seeded partition creation (by "growing" a partition around randomly chosen vertices using a greedy algorithm), breadth-first and depth-first partition creation (by breadth-first or depth-first partition initialisation around randomly chosen vertices) are compared on a number of benchmark circuits. The random partition creation (which bisects a random ordering of the vertices) is found to perform best, both in terms of partition quality and runtime. We note that the authors report the minimum objective function value attained over ten runs of the partitioning algorithm, rather than reporting the average. Similar initial partition generation methods are also used in [KAKS97, KK98b, Kar02].

In [CKM00a], the authors suggest that the starting feasible partition may be generated by applying the FM algorithm to a partition with all the vertices in a single part. Since this would necessarily be deterministic, the computation of gains at the beginning of the pass is randomised (as described in Section 3.4.1) and a relaxation of the balance constraint is introduced.

The authors also generate the starting feasible partition using a biased random selection method. The assignment probabilities (of vertices to parts) are proportional to the "slack" in the part weights after the given vertex is hypothetically allocated to each part (where the slack is defined to be the difference between

the maximum part weight and the current part weight plus the vertex weight).

An important implementation decision is the number of partitions of the coarse hypergraph that are projected to successive finer hypergraphs. We note that it is not necessarily the best partition of the coarsest hypergraph H_c which yields the best partition of a finer hypergraph H_i $(i < c)$ after heuristic refinement in the uncoarsening phase. As a result, Karypis et al. [KAKS97] suggest projecting some or all the partitions computed at the initial partitioning stage. In their experiments, they compute ten partitions of the coarsest hypergraph and project them onto the next-level finer hypergraph.

3.5.4 The Uncoarsening and Refinement Phase

During this phase, a partition of the coarsest hypergraph is projected through each successive finer hypergraph ($H_c \xrightarrow{\Pi_c} H_{c-1}, \ldots, H_{i+1} \xrightarrow{\Pi_{i+1}} H_i, \ldots, H_1 \xrightarrow{\Pi_1} H$). A simple algorithm for projecting a partition which runs in $O(n)$ time proceeds as follows. The vertices of a hypergraph $H_i(V_i, \mathcal{E}_i)$ are traversed, and for each vertex $v \in V_i$, the part corresponding to the coarse vertex $g_i(v) \in V_{i+1}$ is noted from Π_{i+1} and v is assigned to the same part in Π_i.

At each successive level (after a projection of partition Π_{i+1} to the finer hypergraph H_i), the partition Π_i can be further *refined* (the value of the objective function may be further optimised) because the finer hypergraph H_i has more degrees of freedom than H_{i+1}. As is the case with the initial partitioning phase, in principle any flat hypergraph partitioning algorithm may be used to refine Π_i. Typically, iterative improvement algorithms based on the KL/FM framework are used, although [Saa04] uses a Tabu search-based algorithm.

As previously noted in Section 3.5.1, the commonly noted weakness of the original FM algorithm within a flat setting is that it tends to find solutions corresponding to local minima of only "average" quality. This is because it greedily explores the space of feasible solutions, using only a local view. A number of sophisticated enhancements that incorporate a look-ahead capability have been proposed (cf.

Section 3.4.1). In general, they aim to identify and move strongly connected vertex clusters into a single part.

In contrast to this, the multilevel approach implicitly offers a cluster-detection capability through the coarsening process because vertices in coarser hypergraphs will usually correspond to strongly connected clusters of vertices in the original hypergraph. Multilevel approaches also have the ability to consider clusters of (strongly connected) vertices in the original hypergraph at different levels of granularity (through the different levels of approximation).

As a result, even though sophisticated refinement algorithms can be integrated into the multilevel scheme, we note that current multilevel implementations seem to favour simpler refinement algorithms that possess significantly shorter runtimes [Kar02]. Because the quality of partitions produced by multilevel algorithms strongly depends on the coarsening algorithm, simpler refinement algorithms tend to offer a better trade-off between runtime and partition quality than more sophisticated refinement algorithms [Kar02].

A k-way partition may also be computed directly via the multilevel framework. In this case, a k-way refinement algorithm is required. In the following discussion, we make a distinction between algorithms that are used to refine a bisection and algorithms that directly refine k-way partitions, with $k > 2$.

Bisection Refinement Algorithms

The partition Π_i, obtained by projecting Π_{i+1} onto H_i, is used as the starting partition for the refinement algorithm. This should be a significantly better partition of H_i when compared to a randomly generated feasible partition. In [KAKS97], Karypis et al. propose a number of optimisations to the FM algorithm that reduce the runtime with little observed impact on solution quality.

In particular, the authors propose to limit the number of passes of the FM algorithm to only two, since they observe that the most significant improvement in the objective function is achieved during the first couple of passes. Because

Π_i is considered to be a good partition, it is observed that hill-climbing after a long sequence of moves with negative gain is unlikely. Consequently, the authors also abort each pass after a prescribed number of consecutive moves have been made that do not improve the value of the objective function. This prescribed threshold x is expressed as a percentage of the total number of vertices and is set prior to the optimisation; in [KAKS97, Kar02], the recommended values were $1 \leq x \leq 5$. The resulting algorithm is called the Early Exit FM (FM-EE) bisection refinement algorithm.

The second refinement algorithm proposed in [KAKS97] is Hyperedge Refinement (HER). It is motivated by the observation that FM-type algorithms do not perform as well when the hypergraph has many large hyperedges. The HER algorithm is similar to the LSR algorithm for removing *stable* hyperedges [CLL$^+$97] (cf. Section 3.4.1).

A pass of the HER algorithm proceeds as follows. The set of hyperedges is traversed in a random order. For each hyperedge $e \in \mathcal{E}_i$ that has vertices in both parts of the bipartition, the gains of moving all the vertices into one of the two parts are computed. Subject to the balance constraint, the vertices are moved into the part that yields the largest positive gain in the objective function. It is also noted that, in principle, it would be possible to develop an FM-style refinement algorithm whose "moves" involve the movement of all the vertices of a hyperedge that straddles the partition boundary into one part. However, the gain update computation of such an algorithm would be considerably more expensive than the gain update computation for a single vertex move. As in [CLL$^+$97], a run of the FM algorithm is performed after an iterative run of the HER algorithm.

In [cA99], Çatalyürek and Aykanat propose the Boundary FM (BFM) algorithm. In this version of the FM algorithm, only *boundary* vertices are moved during a pass. A vertex is said to be a boundary vertex if it is connected to at least one cut hyperedge. Note that a vertex that is not a boundary vertex at the beginning of a pass may subsequently become a boundary vertex during the

pass. Such vertices, when they become boundary vertices, are inserted into the gain bucket structure based on their updated gain value (rather than the *actual* gain value of the vertex move), in the spirit of the CLIP and CDIP algorithms [DD96b, DD02] (cf. Section 3.4.1). The BFM algorithm also incorporates an early-exit criterion to terminate a pass if no more feasible moves are possible or if a sequence of the most recent moves does not yield an improvement in the objective function. In the `PaToH` multilevel hypergraph partitioning tool [cA01b], the BFM algorithm is implemented so that it performs at most two passes at each level of approximation.

Finally, Karypis notes that only a small number of feasible vertex moves may be possible at the coarser levels [Kar02]. He recommends that partitions are allowed to violate the balance constraint during a pass of the refinement algorithm, provided that a feasible solution can be ensured by the end of the refinement process at that level of approximation. The alternative is to relax the balance constraint at coarser levels and then incrementally tighten it as the partition is projected onto successive finer hypergraphs [Kar02]. These observations are also valid in the context of multi-way refinement.

Multi-way Refinement Algorithms

We noted in Section 3.4.1 that flat multi-way partitioning algorithms have not, in general, been competitive with the recursive bisection approach. Moreover, multi-way partitioning that applies the FM algorithm across pairs of parts [CL98] may lead to prohibitive runtimes for larger values of k, since there are $O(k^2)$ such pairs. Nevertheless, in [UA04], Sanchis' k-way algorithm was preferred to other direct multi-way refinement algorithms within the multilevel framework.

Karypis notes in [Kar02] that the hill-climbing capability of FM-type algorithms becomes less important in a multilevel setting, citing the fact that movement of vertices at coarser levels actually represents movement of strongly-connected clusters of vertices in the original hypergraph. He also observes that vertex moves yielding a large positive gain would be made during a pass whether or

not a priority order of move gain is maintained. Based on these observations, a (relatively simple) greedy multi-way refinement algorithm (henceforth denoted greedy k-way) was proposed in [KK98b].

As in [cA99], it is noted that only moves involving boundary vertices will yield a positive gain in the objective function (recall that a vertex is said to be a boundary vertex if it is incident on at least one cut hyperedge). A vertex is said to be *internal* to a part if it is not incident on any cut hyperedges.

A pass of the greedy k-way algorithm proceeds as follows. The vertices of the hypergraph $H_i(V_i, \mathcal{E}_i)$ are visited in random order. For each vertex $v \in V_i$, if v is internal to its current part, it is not moved. If it is a boundary vertex, then it may be moved to any of the $N(v)$ parts that vertices adjacent to v belong to; the set $N(v)$ is called the neighbourhood of v. Let $N'(v)$ denote the subset of $N(v)$ that contains the parts such that the move of v to one of those parts does not violate the balance constraint and results in a positive gain in the objective function. If $N'(v)$ is nonempty, v is moved to the part in $N'(v)$ that maximises the gain. In [KK98b], the authors note that this greedy algorithm converges within a small number of passes and leads to reasonably good partitions within the multilevel setting.

Refinement Algorithms Based on the Multilevel Framework

Suppose that a partition Π_i for the hypergraph $H_i(V_i, \mathcal{E}_i)$ has been computed within the multilevel sequence (following projection from H_{i+1} and subsequent refinement). When compared to the prior coarsening of $H_i(V_i, \mathcal{E}_i)$ at the i^{th} multilevel step, we note that now the partition Π_i also provides additional vertex connectivity information within V_i; namely, vertices within strongly connected clusters will tend to lie within the same part of the partition. This information can be used to improve the quality of the coarsening algorithm and thus also potentially improve the quality of partition computed for H_i.

Karypis et al. introduce a multi-phase refinement algorithm in [KAKS97]. It uses *restricted coarsening* to compute the map g_i by clustering together vertices

$u, v \in V_i$ if, and only if, $\Pi_i(u) = \Pi_i(v)$. This scheme does not otherwise restrict the type of coarsening algorithm used. The restricted coarsening procedure will construct a new multilevel sequence $\{H_i, H_{i+1}, \ldots, H_{c'}\}$. The partition Π_i is easily translated to coarser levels by projecting it onto H_{i+1} to yield Π_{i+1}. The restricted coarsening algorithm uses Π_{i+1} to compute g_{i+1}, and so on, until the coarsest hypergraph $H_{c'}$ has been constructed. This coarsest hypergraph in the new sequence, $H_{c'}$, is partitioned and its partition is projected and further refined at each level $i' \geq i$, until a new partition Π'_i of the hypergraph H_i is constructed.

A single iteration of this multi-phase approach is called a *V-cycle*. Successive calls to V-cycles (multi-phase refinement iterations) may be terminated when the most recently completed V-cycle has not yielded an improvement in the objective function of the partition or if the number of V-cycles performed at the current multilevel step thus far has reached a prescribed threshold. We note that V-cycles can be recursively applied at each stage of a multilevel algorithm.

Coarsening in the multi-phase refinement setting was considered in [WA98]. Here, the authors augment the vertex connectivity metric to include information about how frequently a hyperedge has been cut in computing partitions of H_i. This is quantified by the function $exp(-\gamma f(e))$, where $f(e)$ is the frequency of hyperedge e in the cutset of the computed bipartitions of H_i with objective function values below a prescribed threshold and γ is a damping factor. The exponential term encourages vertices to match together if they are connected by low-frequency hyperedges. Hyperedges with high frequency would most likely appear in the cut of high-quality solutions and vertices in these ought not be clustered together.

A similar scheme could be adapted to the hyperedge coarsening algorithm (cf. Section 3.5.2). The authors of [WA98] also adjust the reduction ratio r (cf. Equation 3.7) during the restricted coarsening phase, based on the number of passes performed by their FM-based refinement algorithm during the last x levels of the multilevel algorithm, with x some prescribed constant. Intuitively, a greater number of passes performed would indicate a greater difference in the degrees of freedom between successive levels of approximation and would thus imply that

smaller values of r should be chosen to achieve a better quality of coarsening.

3.5.5 Remarks on the Multilevel Paradigm

Algorithms based on the multilevel approach are currently the heuristic methods of choice for generating solutions to the graph and hypergraph partitioning problems. Intuitive explanations for the robustness of the multilevel approach are described by Karypis in [Kar02]. We outline the main advantages over alternative approaches below:

No restriction on partitioning algorithm used: The multilevel approach imposes no restrictions on the type of partitioning algorithm used to compute the partition during the initial partitioning phase.

Makes partitioning easier: The coarsening phase reduces the total hyperedge weight from the original hypergraph and also reduces the space of feasible solutions. Worst-case partitions and randomly-generated partitions of the coarsest hypergraph will be better than those of the original hypergraph.

Allows refinement at multiple levels of granularity: Sophisticated partitioning algorithms are motivated by the need to identify and move strongly connected clusters of vertices into the same part or achieve the same effect by "looking ahead" before making vertex moves. The multilevel paradigm implicitly achieves this during the uncoarsening phase with refinement at different levels of granularity. Refining a coarse hypergraph is equivalent to the movement of strongly connected vertices in the original hypergraph.

Makes refinement easier for FM/KL algorithms: The FM/KL-based iterative improvement algorithms are limited in their ability to escape local minima in the presence of large hyperedges. Multilevel approaches improve the performance of the FM/KL-based algorithms by reducing the number of large hyperedges at coarser levels.

Scales well with increasing problem size: Partition quality of multilevel algorithms scales better with problem size than partition quality of flat algorithms.

Runtimes of multilevel algorithms also scale better with problem size than runtimes of flat algorithms because fewer partitioning runs are necessary to achieve acceptable partition quality.

Offers runtime/partition quality trade-off: Multilevel algorithms offer better opportunities for controlling the runtime/partition quality tradeoff than flat partitioning algorithms, through the coarsening parameters such as the reduction ratio between successive coarser hypergraphs and the maximum size of the coarsest hypergraph. Also, during the uncoarsening phase, application of multiphase refinement should result in better quality partitions, albeit at the expense of longer runtimes.

Computational Time Complexity of Multilevel Algorithms

In the following runtime complexity analysis, we consider a sparse hypergraph $H(V, \mathcal{E})$, with $n = |V|$ and $m = |\mathcal{E}|$, such that n and m are of the same order of magnitude. We assume that $d_{max} \ll n$ and $e_{max} \ll m$. The computational time complexity analysis that follows should be viewed as an informal justification for the apparent linear time complexity of multilevel algorithms in practice rather than a rigorous derivation.

We let n_i be the number of vertices in hypergraph H_i at multilevel stage i; e^i_{max} and d^i_{max} denote the maximum hyperedge cardinality and maximum vertex degree, respectively, of H_i.

It is assumed that the coarsening algorithm is implemented in a greedy manner (cf. Section 3.5.2) and that it reduces the numbers of vertices and hyperedges at each coarsening step by constant factors $1 + \nu$ and $1 + \omega$ ($\nu, \omega > 0$), respectively. The runtime complexity of the initial partitioning phase is assumed to be dominated by the runtime complexities of the coarsening and uncoarsening phases, as the coarsest hypergraph is an order of magnitude smaller than the original hypergraph. Because the number of vertices is reduced by a factor greater than one at each coarsening step, there are $O(\log n)$ multilevel steps. We note that e^i_{max} is bounded above by e_{max} for all multilevel steps, while d^i_{max} is typically

observed to increase during the coarsening phase (as the number of hyperedges is usually reduced at a slower rate than the number of vertices by the coarsening algorithm). We assume here that this increase is modest and, in any case, d^i_{max} is bounded above by the number of hyperedges in hypergraph H_i, which in the latter stages of the coarsening phase will be an order of magnitude less than the number of hyperedges in the original hypergraph.

During a single coarsening step i, a coarsening algorithm performs $O(n_i e^i_{max} d^i_{max})$ operations in computing the map g_i, because for each vertex it is required to compute connectivity values with each adjacent vertex. Having computed the map g_i, the construction of the coarse hypergraph H_{i+1} has complexity $O(n_i e^i_{max})$.

During a single uncoarsening step, the projection of partition Π_{i+1} from hypergraph H_{i+1} onto H_i has complexity $O(n_i)$. Iterative improvement algorithms adapted for multilevel refinement (e.g. early-exit FM) have complexity $O(n_i e^i_{max})$ per pass (cf. Section 3.4.1).

Because e^i_{max} and d^i_{max} are small relative to the number of vertices (and hyperedges) in hypergraph H_i and the number of passes of the refinement algorithm at each uncoarsening step is also assumed small, each multilevel step has runtime complexity $O(n_i)$. Then, the runtime of the serial multilevel partitioning algorithm is given by:

$$\begin{aligned} T_s &= \sum_{i=0}^{O(\log n)} O(n_i) && (3.14)\\ &= \sum_{i=0}^{O(\log n)} O(n(1+\nu)^{-i}) && (3.15)\\ &\leq \sum_{i=0}^{\infty} O(n)(1+\nu)^{-i} && (3.16)\\ &= O(n) && (3.17) \end{aligned}$$

Outstanding Issues

Aside from the development of parallel multilevel partitioning algorithms, there are a number of other possible directions for further research.

In [CRX03], Cong et al. perform a study of leading-edge hypergraph partitioning algorithms using existing state-of-the-art tools on a number of purpose-engineered hypergraphs. These hypergraphs were constructed in such a way that an upper bound on the optimal value of the partitioning objective was known upon the construction of the hypergraph. This bound was also sufficiently close to the optimum, so that it corresponded to solutions which were generally better than those obtained by state-of-the-art algorithms.

They observe that the multilevel bipartitioning algorithms consistently matched the authors' upper bound for the bipartitioning problem. On the other hand, when computing multi-way partitions (using multi-way partitioning or recursive bisection algorithms), state-of-the-art tools performed less well (up to 18% from the upper bound on the optimal solution). This suggests that while bipartitioning algorithms appear to be fairly mature, there is still scope for potentially significant improvement in the case of the multi-way partitioning problem.

In [Kar02], Karypis identifies the coarsening phase as offering the best scope for further research, since there is still no coarsening method that outperforms others across a wide range of hypergraphs.

We also note a number of alternative directions for further research. Relatively little research has been done into the feasibility of hybrid approaches that combine different optimisation techiques into the multilevel framework. For example, a very sophisticated refinement algorithm may be used during the coarser levels, while a refinement algorithm optimised for fast runtimes may be used at the finer levels of the multilevel algorithm.

Research may also be application driven. In certain applications, such as load balancing in scientific computing, the runtime/solution quality trade-off is usually skewed towards faster runtimes. Runtime optimisations of the multilevel framework that do not significantly degrade solution quality are sought. Different application domains also give rise to partitioning problems that exhibit particular common structures. Thus, coarsening and refinement schemes may be developed that exploit characteristics of the hypergraphs within a particular

application domain.

3.6 Parallel Graph Partitioning Algorithms

3.6.1 Data Distribution Strategies

Before presenting an overview of parallel graph partitioning literature, we describe the two most common ways in which a graph $G(V, \mathcal{E})$ may be allocated to p processors on a distributed-memory parallel machine. This involves the distribution of the set of vertices and the set of edges of the graph, as well as the associated vertex and edge weights. The graph decomposition is particularly important for message-passing architectures, because interprocessor communication is necessary when a processor needs to access a part of the graph that is stored on another processor. We assume here that each processor is assigned a roughly equal portion of the graph.

The natural way to store a graph across p processors is for each processor to store $|V|/p$ vertices and the adjacency list associated with those vertices (i.e. all the edges incident on the $|V|/p$ vertices). We call this the one-dimensional decomposition of the graph to the p processors. The name is intuitive; if we imagine the set of vertices to be stored in a one-dimensional array, the p processors are organised into a line so that each processor is allocated a contiguous portion of the vertex array.

Alternatively, the vertex set V can be partitioned into $\sqrt{p}$ roughly equal subsets (i.e. $V = \{V_0, \ldots, V_{\sqrt{p}-1}\}$) with the p processors organized into a $\sqrt{p} \times \sqrt{p}$ grid. Then, processor $p_{i,j}$ is assigned the edges from $\mathcal{E}$ that connect vertices from V_i with vertices from V_j. Vertices in V_i are assigned to the diagonal processor $p_{i,i}$. We call this the two-dimensional decomposition of the graph, since the processors are organized into a two-dimensional grid. It is analogous to a two-dimensional checkerboarding decomposition of the graph's adjacency matrix onto p processors [GGKK03].

Henceforth, when necessary, we will classify the parallel graph partitioning according the graph-to-processor allocation. We note that most parallel graph partitioning algorithms in literature use the one-dimensional graph-to-processor allocation.

3.6.2 Early Work

In [GZ87], Gilbert and Zmijewski develop a parallel graph partitioning algorithm (based on the KL algorithm) for a message-passing parallel architecture. They investigate parallel computation of a p-way partition using p processors and attempt a fine-grained parallel formulation of KL. The algorithm is said to be fine-grained because communication is induced after individual vertex moves. The scalability of this algorithm appears to be limited by the potentially large amount of communication required.

Parallelism in graph partitioning was formally investigated by Savage and Wloka, in [SW91]. The authors show that local search under the KL (and thus also the FM) neighbourhood structure with unit-weighted edges is P-complete. This suggests that highly-parallel formulations of the KL/FM algorithms are unlikely[3]. The authors also present a parallel algorithm in the spirit of the KL heuristic that swaps multiple vertices across the partition boundary at a time. The algorithm was implemented on the Connection Machine 2 massively parallel computer that can use up to 64K one-bit processors. Results that are competitive with serial KL in terms of partition quality are reported on randomly generated graphs.

In [DPHL95], Diniz et al. parallelise the Inertial geometric partitioning algorithm [NORL87] and the FM algorithm. The Inertial algorithm is used to construct a starting partition for the FM algorithm. In the first parallel algorithm, a bi-

[3]It is thought that decision problems in the class NC (decidable in poly-logarithmic time on a PRAM with a polynomial number of processors) offer the best potential for parallelism. On the other hand, P-complete problems have been traditionally difficult to parallelise. A long-standing conjecture is that P $\neq$ NC, which, if true, makes P-completeness a useful measure of whether a problem is inherently sequential [GHR95].

partition produced by the parallel Inertial algorithm is improved using a parallel FM algorithm. In the second, a p-way partition is produced by recursive application of the parallel Inertial algorithm. Then, the p-way partition is further improved by a pairwise application of the parallel FM algorithm (similar to the serial pairwise FM from [CL98]).

We focus here on the parallel FM algorithm. When applied to a bipartition using p processors following the parallel Inertial algorithm, the parallel FM algorithm stores vertices belonging to part P_0 on $p/2$ processors and vertices belonging to part P_1 on the remaining $p/2$ processors. The processors then pair up, such that in each processor pair, one processor stores vertices from P_0 and the other from P_1. The paired processors interact in performing passes of the FM algorithm.

In the p-way parallel FM algorithm, parallelism associated with the pairwise application of FM is exploited. Vertices from part i are stored on processor p_i. Processors pair up with other processors to perform parallel FM, as in the parallel bipartitioning algorithm. Note that processors only pair-up with other processors with whom they share common edges. Pairs of processors that do not share common edges may perform parallel FM concurrently.

In order to identify such processor-pairs, the authors define a quotient graph of the p-way partition, $\mathcal{G}$. Each vertex in $\mathcal{G}$ is a part and an edge in $\mathcal{G}$ is induced between parts if the two parts have vertices that are connected by an edge in $G(V, \mathcal{E})$. The quotient graph $\mathcal{G}$ is edge-coloured using a heuristic algorithm[4], so that the pairwise FM algorithm may be applied concurrently to pairs of parts connected by edges of the same colour.

Parallel algorithms were run on nCUBE 2 and Paragon multiprocessors. The partition quality produced by the parallel algorithms is dominated by the serial multilevel algorithm used in the `Chaco` tool [HL94]. Speedups in the range of 9-18 are observed using 64 processors and 19-50 using 1024 processors on graphs

[4]An edge colouring of a graph is the assignment of colours to edges, such that any two edges that share a vertex must be assigned a different colour. Computing the minimum number of colours required for a general graph is NP-complete [GJ79].

with around 150 thousand vertices and 1 to 1.9 million edges. We note that these parallel algorithms do not explicitly enforce the partitioning balance constraint.

In [HR95], Heath and Raghavan present a parallel formulation of a geometric graph partitioning algorithm that uses a one-dimensional allocation of the graph to p processors. The parallel asymptotic time complexity is derived to be $O((m/p) \log n)$, meaning that the parallel algorithm is not cost optimal (since the time complexity of the serial multilevel algorithm is $O(n)$).

As noted in Section 3.5, multilevel algorithms are currently the heuristic method of choice for graph and hypergraph partitioning. Recent work on parallel graph partitioning has yielded a number of parallel formulations of the serial multilevel approach. Some of these algorithms have also been implemented within the parallel graph partitioning tools `ParMeTiS` [KSK02] and `pJOSTLE` [Wal02].

3.6.3 Parallel Multilevel Recursive Spectral Bisection

In [Bar95], Barnard presents a parallel formulation of the multilevel recursive spectral bipartitioning algorithm. It first computes a bipartition of the graph using p processors, following which successively smaller teams of processors are used for each recursive application of the bipartitioning algorithm.

The multilevel spectral bipartitioning algorithm from [BS94] is parallelised, so that the Fiedler vector is transferred between the levels in the uncoarsening phase (rather than the partition of the coarser hypergraph). Barnard identifies the coarsening phase as the hardest operation to parallelise, noting two main difficulties. The first is computing a maximal independent set of vertices in parallel (which is required by the serial coarsening algorithm from [BS94], as described in Section 3.5.2); the second is that the computation of the Fiedler vector (during the initial partitioning phase) requires a connected graph (cf. Section 3.3.2). Recall that in the serial multilevel algorithm from [BS94], heuristic refinement is not used during the uncoarsening phase.

Barnard uses Luby's CRCW (Concurrent-read, Concurrent-write) PRAM algo-

rithm for computing a maximal independent set of vertices in a graph [Lub86]. In the initial partitioning phase, if the coarsest graph is not connected, the Fiedler routine is separately applied to each of the connected components. The disconnected components are identified using a PRAM-based parallel non-deterministic random-mating algorithm [Gaz93].

Barnard implemented the parallel algorithm on a Cray T3D parallel computer with 256 processors. Each processor is a 150Mhz DEC Alpha chip with 64MB of memory. The parallel implementation used Cray's `SHMEM` (SHared MEMory) library. In this communication model, the T3D has a segmented global shared memory, with each processor having direct access to its local memory and access to the memories of other processors through calls to the `shmem_get` and `shmem_put` routines. Although these are similar to send and receive operations on distributed-memory message-passing architectures, the main difference is that the `SHMEM` routines do not require cooperation from the remote processor. Thus, the T3D is a good approximation to the (theoretical) shared memory model.

In the experiments, a p-way partition is computed using p processors. Partition quality is comparable with serial multilevel spectral bipartitioning. On a graph with 50 000 edges, a speedup of almost 20 over the single-processor implementation [BS94] is observed using 256 processors. On a larger problem (with 765 000 edges), the observed speedup is 140 using 256 processors. These results suggest good empirical scalability, albeit observed on an approximation to the shared-memory platform. To that end, Barnard notes that the algorithm would be inefficient on a general distributed memory message-passing architecture because of the high interprocessor communication requirements of the PRAM-based parallel algorithms.

3.6.4 Karypis and Kumar's Two-Dimensional Parallel Algorithm

In [KK98c], Karypis and Kumar present a parallel multilevel graph partitioning algorithm that is based on the serial algorithm proposed in [KK99]. With a distributed-memory message passing architecture in mind, the authors try to limit interprocessor communication (as compared to other parallel approaches).

The algorithm uses a two-dimensional distribution of the graph $G(V, \mathcal{E})$ across the processors, with the adjacency matrix of the graph stored in Cartesian fashion (cf. Section 3.6.1). A p-way partition of the graph using p processors in parallel is sought, and is computed by recursive bisection. Karypis and Kumar note that only the coarsening and uncoarsening phases need to be parallelised; the coarsest graph should be sufficiently small to be partitioned serially without significantly affecting the parallel runtime. We describe the three phases of the parallel multilevel algorithm in more detail below.

Parallel Coarsening Phase

Karypis and Kumar note that Luby's PRAM-based algorithm [Lub86] (used by Barnard [Bar95] to find a maximal independent set of vertices) may be modified to compute a set of independent edges that could be contracted to form vertices of the successive coarser graph; however, this approach is rejected because of its high communication requirement on a distributed-memory parallel computer. They instead seek an algorithm that computes the coarsening in parallel without interprocessor communication. This is achieved if vertices on each processor are only allowed to form clusters with other vertices on that processor.

It is noted that if a one-dimensional decomposition of the graph to processors is used, the quality of the coarsening is expected to deteriorate as p increases (because each processor will have fewer vertices to cluster together). However, when the two-dimensional graph decomposition is used, only the $\sqrt{p}$ diagonal processors are involved in the computation of the vertex matches; this should

ensure that there are sufficient local candidate vertices on each processor.

Having computed the vertex matching, the construction of the coarse graph G_{i+1} proceeds in parallel, as follows. The diagonal processors broadcast the required vertex matches to processors along the rows and columns of the processor grid. Once each processor has the required vertex matches, it can proceed to contract the locally stored edges without further communication.

The parallel coarsening algorithm uses p processors, until the reduction in the number of vertices between successive coarser graphs is smaller than a prescribed threshold. Then, the graph is transferred onto a (smaller) $(\sqrt{p}-1) \times (\sqrt{p}-1)$ processor subgrid and the parallel coarsening algorithm continues using $(\sqrt{p}-1)^2$ processors. The use of fewer processors as the graph shrinks is done to maintain the quality of coarsening. The parallel coarsening algorithm terminates when the entire coarse graph has been "folded" onto a single processor.

Initial Partitioning Phase

This is computed serially, using a number of independent runs of the Greedy Graph Growing Algorithm from [KK99]. The authors note that it is possible to utilise each or a subset of the p processors in computing the initial partitions, although this would require a broadcast of the coarsest graph. Because the time taken by the initial partitioning phase is small, the authors did not implement this extension.

Parallel Uncoarsening Phase

The partition of the coarsest graph G_c is projected back onto the original graph G_0 via the sequence of intermediate graphs $G_{c-1}, \ldots, G_1$. The processor sub-grid used for each intermediate graph G_i is the same as that used for G_i during the coarsening phase.

For refinement, the boundary KL algorithm [KK99] is parallelised. The serial version of this algorithm has a hill-climbing capability; the authors note that

this is less significant within a multilevel context. Thus, in order to achieve better concurrency, they remove the hill-climbing capability from the parallel formulation.

The parallel formulation of the KL bipartitioning algorithm proceeds in a number of steps, during each of which vertex moves are only made in one direction. Since the vertex moves are being made concurrently across the processors, this restriction guarantees that each vertex move yields a reduction in edge-cut. The algorithm starts the vertex moves from the heaviest part; in each subsequent step, the part from which vertices are moved alternates. This continues until no improvement in edge-cut is made by the most-recently completed step or until the maximum number of steps have been made. There is an explicit rebalancing step at the end if the final partition violates the partitioning constraint.

At the beginning of each step, the gain of moving each vertex is computed across processor columns using only locally stored edges; each processor $p_{i,j}$ computes the local gain in edge-cut lg_v obtained from moving a vertex $v \in V_j$. The total gain of moving vertex $v \in V_j$, denoted by g_v, is computed by a reduction across the processor column j and is stored on the diagonal processor $p_{j,j}$. The diagonal processor $p_{j,j}$ is responsible for actually making the moves of the set of vertices $U_j \subset V_j$ that yield a positive gain.

The set U_j is then broadcast by $p_{j,j}$ along column j and row j of the processor grid. The updated local gain values for vertex moves are computed by each processor $p_{i,j}$ and the *actual* updated gain values are obtained at each diagonal processor via a reduction across the processor column.

Analytical and Empirical Evaluation

The analytical performance model of the algorithm presented in [KK98c] yields an asymptotic runtime complexity of $O(\frac{n}{\sqrt{p}} \log p)$, where $n = |V|$. Since the complexity of the serial algorithm is $O(n)$, this runtime complexity implies that the parallel graph partitioning algorithm is not cost-optimal. However, the authors note that experimental evidence suggests a larger constant is associated with the

term $O(|\mathcal{E}|/p)$ for graphs with a high average degree (this term is hidden, as it is asymptotically dominated by the term yielding the parallel runtime).

The experiments were carried out on a 128-processor Cray T3D parallel computer and implemented using the `SHMEM` library (cf. Section 3.6.3). Results are reported for p-way partitions using p processors, on a suite of matrices with up to 181 200 vertices and 2 313 765 edges. The quality of partitions produced by the parallel algorithm is within 10% of the serial algorithm and the authors note that this difference decreases as p increases. A sub-linear decrease in runtime as p increases is reported, as expected from the asymptotic scalability results. However, for graphs with high average degree, speedup in the range of 14 to 24 was achieved on 32 processors and in the range 22 to 56 on 128 processors.

We note that the algorithm is not restricted to square values of p; p can be any non-prime number with factors i and j such that the processors are arranged into an $i \times j$ grid. However, for a reasonably regular graph, better performance is expected when the difference between i and j is small. This is because in this case, more efficient collective communication operations along the rows and columns of the processor grid are performed and a better computational load balance across the processors is achieved.

3.6.5 Karypis and Kumar's One-Dimensional Parallel Algorithm

In [KK96, KK97], Karypis and Kumar present a parallel formulation of their multilevel k-way graph partitioning algorithm from [KK95], using a one-dimensional graph-to-processor distribution. The authors summarise the difficulties encountered in developing a parallel formulation of the multilevel approach, specifically focusing on the coarsening and uncoarsening phases.

In particular, they note that the two-dimensional algorithm [KK98c] can only form clusters of local vertices during the coarsening phase and that the quality of its coarsening degrades when the average vertex degree in the graph is small.

Likewise, they note that the parallel coarsening algorithm from [Rag95] uses a one-dimensional distribution of the graph across processors and forms processor pairs, so that a vertex may match with any other vertex stored within its processor pair, but not with vertices on other processors.

The above algorithms sacrifice the quality of the coarsening in order to reduce the interprocessor communication during the coarsening phase; Karypis and Kumar observe that allowing vertices to match with any other vertex in the graph may result in an excessive communication requirement within the parallel setting.

During the uncoarsening phase, a parallel refinement algorithm needs to move vertices concurrently on all or some of the processors. As processors make independent choices about vertex moves, it is possible that the concurrent movement of adjacent vertices by different processors yields an overall increase in edge-cut, even though the individual vertex moves yield a decrease in edge-cut. Karypis and Kumar also note that the partition quality produced by the parallel FM-formulation from [DPHL95] is potentially much worse than that produced by the serial algorithm.

The above two obstacles to successful parallelism are tackled by first colouring the vertices of the graph at each multilevel step[5]. The potential conflicts during the coarsening and refinement computations are avoided by only allowing concurrent operations involving vertices of a single colour.

The colouring computation proceeds iteratively; during each iteration, a maximal independent set of vertices is computed and assigned a colour. The vertices of this colour are removed from the vertex set prior to the next iteration. The authors use a variant of Luby's parallel algorithm [Lub86], which computes a maximal set of vertices that is not necessarily independent, but significantly reduces the parallel runtime of Luby's original algorithm. In experiments, the number of

[5]A vertex colouring of a graph $G(V, \mathcal{E})$ is the assignment of colours to vertices, such that any two vertices that are connected by an edge must be assigned a different colour. Computing the minimum number of colours required for a general graph (i.e. the graph's *chromatic number* $\chi(G)$) is NP-complete [Pap94].

colours produced by the algorithm is small compared to the number of vertices in the graph, ranging from five to twenty.

Parallel Coarsening Phase

Karypis and Kumar present a parallel formulation of the serial heavy-edge coarsening algorithm [KK99].

The parallel coarsening algorithm proceeds in iterations, one for each colour. During the c^{th} iteration, vertices of colour c that have not been matched already choose one of their unmatched neighbours using the heavy-edge heuristic. Formally, let u be a vertex of colour c and suppose that the edge (u, v) is chosen for contraction. As the colour of v is not c during the current iteration, v will not be seeking a match. However, it is possible that another vertex stored on a different processor, call it w, also selects v for matching during the same iteration.

Such conflicts are dealt with as follows. When all vertices of colour c have selected an unmatched neighbour, a synchronisation step is performed across all the processors. The processors storing u and w send a message to the processor storing v, requesting a match with v. Since only one vertex is allowed to match with v, the one connected via the heaviest edge is chosen by the processor storing v and the processors storing u and w are informed as to whether or not their request for a match was successful.

Suppose that, without loss of generality, u is chosen for matching with v. The processor storing v also determines whether or not it will store the coarse vertex resulting from contracting (u, v) (making a random choice between itself and processor storing u) and informs the processor storing u of the outcome. Note that if u's request for a match is not successful in the current iteration, it may still match with another vertex during subsequent iterations, if selected for matching by a vertex of another colour.

When constructing G_{i+1}, coarse vertices resulting from local matches are retained on the same processor, otherwise they are retained on one of the two processors

that store the matched vertices, as described above. Unlike the algorithm in [KK98c], all p processors are involved during every multilevel step. The coarsening process terminates when the coarse graph has $\Theta(p)$ vertices.

Initial Partitioning Phase

Because the coarsest graph is small, the p-way partition is computed serially, using a recursive bisection algorithm. Parallelism during the recursive step is exploited; when a graph bipartition is computed on a processor, the subsequent bipartitioning of the two "halves" of the bipartition is done on different processors concurrently [KK97].

Parallel Uncoarsening Phase

The k-way greedy refinement algorithm from [KK95] is parallelised. A pass of the *serial* algorithm proceeds as follows. The vertices of the graph G_i are visited in random order. For each vertex v, if v is internal to its current part (i.e. all the vertices adjacent to v are also in that part), then it is not moved. Otherwise, if possible, it is moved to the part yielding the largest improvement in the edge-cut such that the balance constraint is not violated.

Each pass of the *parallel* refinement algorithm proceeds in c steps, where c is the number of colours computed for the given graph. During the i^{th} step, all vertices of colour i may be moved and a subset that leads to the greatest reduction in the edge-cut is actually moved. Having performed the movement of these vertices, processors storing adjacent vertices are informed of the moves. When vertex moves are made, vertices are not explicitly moved between the processors; only their part index variable is updated.

At the beginning of a refinement step, each processor knows the weight of each of the p parts. The local moves are all made subject to these weights. At the end of each step, the part weights are recomputed and each processor again knows the exact weights. We note that the graph partitioning problem considered in

[KK96, KK97] does not impose explicit partition balance constraints. Should the recomputed part weights violate a prescribed balance constraint, the algorithm described in [KK96, KK97] does not take back moves in order to return a balanced partition.

Analytical and Empirical Evaluation

In their analytical performance model, the authors assume that every vertex in each of the coarse graphs has a small, bounded degree. Consequently, the chromatic number of the graphs at all coarsening levels is small[6] (when compared to the number of vertices in the original graph) and it is assumed that graphs are reduced by a constant factor at each successive coarsening step. These assumptions yield an asymptotic parallel runtime complexity of $O(n/p) + O(p \log n)$. The algorithm is asymptotically cost-optimal if $p^2 = O(n/\log n)$ and has an isoefficiency function of $O(p^2 \log p)$.

The algorithm was implemented on the Cray T3D parallel computer with 128 processors, using the `SHMEM` library (cf. Section 3.6.3) The reported partition quality is comparable with that produced by the serial algorithm on a number of finite element mesh graphs. The parallel algorithm also exhibits good empirical scalability.

The performance of the algorithm can be improved by using an intelligent initial distribution of vertices to processors to reduce the number of edges with vertices on two processors. However, this is equivalent to solving the given graph partitioning problem (with $k = p$). When multiple runs of the partitioning algorithm are required, adaptive repartitioning of the vertices across the processors can be performed after each partitioning run.

[6] Given a graph $G(V, \mathcal{E})$, an upper bound on its chromatic number $\chi(G)$ is given by $d_{max} + 1$ [Bro41].

A Coarse-Grain Formulation

In [KK96], Karypis and Kumar modify the one-dimensional algorithm for a distributed-memory message-passing architecture. The resulting parallel algorithm no longer performs colouring of the graphs prior to each multilevel step.

During the parallel coarsening phase, the computation of the vertex matching proceeds in a number of steps. During each step i, every processor visits its unmatched vertices. For each such vertex u, an unmatched adjacent vertex v is chosen for matching, based on the heavy-edge heuristic. If v is locally stored, then the match proceeds; otherwise a request may be made to the processor that owns v. If i is odd, then a request is made only if $v < u$ (based on the vertices' respective indices) and if i is even then a request is made only if $v > u$. Remote requests to the processor storing v are granted in the same way as in [KK97].

During the uncoarsening phase, a pass of the parallel refinement algorithm now consists of only two steps. During the first step, vertices from part i are only allowed to move to part j if $i < j$. During the second step, this condition is reversed (i.e. vertices only move from part i to j if $i > j$), so that the pass may consider all possible vertex moves.

As in [KK98c], the coarse-grain parallel algorithm folds successive coarser graphs onto fewer processors, in order to reduce communication overheads that begin to dominate the parallel runtime as the size of the graph is reduced.

The parallel algorithm was implemented using the Cray MPI message passing library and run on the Cray T3D parallel computer. The algorithm produces partitions of comparable quality to the finer-grained parallel algorithm from [KK97]. However, the coarse-grained algorithm achieved slower runtimes and the difference in runtime between the two parallel algorithms increases with the number of processors.

3.6.6 Walshaw's Parallel Multilevel Graph Partitioning Algorithm

In [WCE97, WC99], Walshaw et al. describe a parallel multilevel graph partitioning algorithm that uses a one-dimensional graph distribution across the processors.

Parallel Coarsening Phase

The algorithm uses the heavy-edge heuristic [KK99] for computing vertex matches. For each locally stored vertex, the processors distinguish between adjacent local vertices and adjacent remote vertices. The coarsening algorithm first computes matches for vertices adjacent to local vertices. This may be sufficient to contract the graph by the prescribed amount; if not, it is necessary to seek remote matches for local vertices. A parallel iterative matching procedure is used, which terminates when no vertex is left unmatched.

During every iteration, each processor first informs neighbouring processors (processors that store vertices adjacent to locally held vertices) of local vertices that are unmatched. The list of locally unmatched vertices is then visited and vertices with no unmatched adjacent vertices are copied over to the coarse graph; otherwise they are tentatively matched to an unmatched adjacent vertex according to a heavy-edge heuristic. Neighbouring processors are again updated with the recent matches and matches spanning two processors are accepted whenever the vertices involved are mutually requested.

The algorithm avoids selection cycles (i.e. where vertex u selects vertex v, which in turn selects vertex w, which in turn selects u) as follows. For each edge (u, v) in the cycle, a pseudorandom number generator is seeded with the sum of the indices of u and v and a pseudorandom number is generated. The edge with the highest number is chosen.

Initial Partitioning Phase

The coarsening procedure continues until a coarse graph with p vertices has been constructed. This *initial* partition is then used as input to the uncoarsening phase.

Parallel Uncoarsening Phase

Walshaw's algorithm differs from other parallel graph partitioning algorithms in that it accepts unbalanced partitions at the coarse levels and rebalances the projected partition through successive finer graphs. The rate at which the rebalancing is done is called the *Multilevel Balancing Schedule*. The algorithm always seeks a p-way partition, with each processor storing all the vertices in a part. It explicitly moves vertices between the processors (rather than modifying the vertices' part index as in [KK96, KK97]), although the corresponding vertices in the finer graphs are not moved. Three distinct refinement algorithms are described.

Common to each of the algorithms, the authors identify the *subdomain* graph $G_\pi(S, \mathcal{L})$ (the so-called quotient graph in [DPHL95]). The vertices of G_π are the parts of the partition and there exists an edge in $\mathcal{L}$ between parts P_i and P_j if, and only if, there exists an edge $e = (x, y) \in \mathcal{E}$ such that $x \in P_i$ and $y \in P_j$. The edges in the subdomain graph identify the partition boundaries that contribute to the edge cut and these boundaries are refined (the sum of the edge weights on the boundary reduced). The subdomain graph is also used to compute the required balancing flow (expressed in terms of the weight of vertices in V) along its edges in order to rebalance the partition.

In *interface optimisation* (which appears very similar in nature to the pairwise application of FM in [DPHL95]), the processors pair-up to apply the KL algorithm on the partition boundary common to them. The boundaries are identified using $G_\pi(S, \mathcal{L})$. Vertex moves are accepted if a condition on the vertex weight and the appropriate flow (in favour of a more balanced partition) is satisfied. The flow-based mechanism is used to balance the partitions (rather than an explicit

balancing scheme).

In *alternating optimisation*, the vertices are only allowed to move in one direction during a particular pass, but otherwise the algorithm proceeds like the interface optimisation algorithm. The direction of allowed vertex moves alternates from pass to pass. Finally, the *gain optimisation* algorithm uses relative gain. The relative gain of a vertex move is defined to be its actual gain minus the average gain of vertex moves in the opposite direction.

Empirical Evaluation

The algorithms were implemented within the tool `pJOSTLE` on a Cray T3E parallel computer, using the Cray MPI message-passing library for interprocessor communication. Interface optimisation was observed to produce the partitions of highest quality, although running slower than alternating optimisation and relative gain optimisation. A hybrid of relative gain and interface optimisation computed partitions that were better balanced, but of slightly lesser quality, than those produced by interface optimisation. The hybrid and interface optimisation algorithms yielded comparable runtimes.

The parallel algorithms were also compared to the parallel graph partitioning tool `ParMeTiS` [KSK02], which implements the one-dimensional parallel algorithm described in Section 3.6.5.

Interface optimisation produced better quality partitions than `ParMeTiS` on average, albeit with considerably slower runtimes. However, reasonably good speedups relative to the `JOSTLE` [WCE95] serial graph partitioner were reported. The authors observe that the partitioning time (for all the parallel algorithms) is strongly dependent on the initial vertex distribution, since a good initial distribution of vertices to processors decreases the number of edges that span multiple processors, reducing interprocessor communication.

Chapter 4

Parallel Multilevel Hypergraph Partitioning

4.1 Introduction

As discussed in Section 1.1.4, many large-scale hypergraph partitioning applications demand parallel partitioners, because existing serial partitioners do not have sufficient solution capacity. Since parallel hypergraph partitioners are not currently available, but parallel graph partitioners (e.g. `ParMeTiS` [KSK02]) are, the best that can be done for large problem instances is to use an approximate graph model in combination with a parallel graph partitioner.

This chapter describes our work on parallel multilevel hypergraph partitioning. We first present our preliminary work, in the form of an application-specific disk-based parallel multilevel hypergraph partitioning algorithm. It was designed to exploit the structure inherent in hypergraphs derived from Markov and semi-Markov transition matrices that are constructed by a breadth-first traversal of the state-transition graphs.

We then present the main contribution of this thesis: a general parallel multilevel k-way hypergraph partitioning algorithm, developed upon experience from our initial work. We show conditions under which our parallel algorithm is cost-

optimal and derive its isoefficiency function. Finally, we describe a very recent approach to parallel hypergraph partitioning developed at Sandia National Laboratories.

The remainder of this chapter is organised as follows. Section 4.2 outlines a general framework for parallel multilevel hypergraph partitioning algorithms. Section 4.3 describes our disk-based parallel multilevel hypergraph partitioning algorithm. Section 4.4 summarises the insights gained from our preliminary work on the disk-based algorithm and discusses concurrency within serial multilevel hypergraph partitioning algorithms. Section 4.5 describes our general parallel multilevel k-way hypergraph partitioning algorithm. The algorithm's analytical performance model is described in Section 4.6. Finally, Section 4.7 describes another very recent approach to parallel hypergraph partitioning, proposed in [DBH+06].

4.2 Parallel Hypergraph Partitioning Considerations

There are two main objectives behind the development of a successful parallel hypergraph partitioning algorithm. Firstly, we seek a parallel hypergraph partitioning algorithm that produces partitions of comparable quality to those produced by the serial algorithm. Note that no parallel partitioning algorithm can guarantee better partition quality than a serial algorithm. This is because the performance of the parallel algorithm on p processors can be replicated serially by lumping together the p sets of computations (that are performed across p processors) into a single set S and performing the computations of S in a suitable order. Secondly, we would also like our parallel partitioning algorithm to be technically scalable, as described in Section 2.5.2. This means that we would like to be able to maintain a constant level of processor efficiency with an increasing number of processors, by appropriately increasing the input problem size.

Experience from parallel graph partitioning suggests that finding true concur-

rency in serial multilevel hypergraph partitioning algorithms is difficult[1]. However, the similarity in their respective serial partitioning algorithms suggests that the ideas behind parallel graph partitioning algorithms could be a useful starting point for development of parallel hypergraph partitioning algorithms. We seek a parallel formulation of the multilevel approach, since it is at the core of leading-edge heuristics for both serial hypergraph partitioning and serial and parallel graph partitioning.

4.2.1 Graphs vs. Hypergraphs

It is instructive to consider the primary difference between graphs and hypergraphs and the impact this has on a naïve adaptation of parallel graph partitioning algorithms to hypergraphs. Recall that, whereas the cardinality of every edge in a graph is two, the cardinalities of hyperedges in a hypergraph may vary from a lower bound of one to an upper bound given by the number of vertices in the hypergraph.

This is significant because the main obstacle to parallelism in graph and hypergraph partitioning is the presence of adjacent vertices on different processors. Firstly, consider a move-based partitioning algorithm. Concurrent movement of adjacent vertices on different processors potentially causes a conflict because the gain of each vertex move, as computed by the respective processor, is conditional upon its adjacent vertices remaining fixed in their respective parts.

Secondly, consider the edge coarsening algorithm; serially, it matches an unmatched vertex with the most strongly connected neighbouring unmatched vertex. In parallel, processors compute vertex matches concurrently. A processor may match an unmatched local vertex with (what it thinks) is another unmatched vertex on another processor; however, this remote vertex may have already been matched by its owner processor.

[1]Recall from Section 3.6.2 that [SW91] showed that KL and FM-based partitioning is P-complete, which suggests that finding true concurrency within these algorithms is difficult [GHR95].

In [KK97], in the context of parallel graph partitioning, Karypis and Kumar coloured the vertices of the graph to identify vertices that did not share any common edges. In any given step, the algorithm would operate concurrently only on vertices corresponding to the same colour, avoiding the conflicts outlined above. Identifying groups of vertices that do not share any common hyperedges in the hypergraph may be achieved by constructing a graph clique model and then colouring this vertex connectivity graph. However, because large hyperedges will induce large cliques in the vertex connectivity graph, its chromatic number is also likely to be large[2].

4.2.2 A Parallel Framework for the Multilevel Approach

We outline a framework for the development of parallel multilevel hypergraph partitioning algorithms. We note that only the coarsening and uncoarsening phases need to be parallelised. During the initial partitioning phase, the coarsest hypergraph should be small enough to be partitioned serially on a single processor and the time-complexity of this serial component should be dominated by the time-complexities of the parallel coarsening and parallel uncoarsening phases. The proposed parallel multilevel pipeline is illustrated in Figure 4.1.

The design of a parallel hypergraph partitioning algorithm may be application-specific. That is to say, the parallel algorithm may be designed to exploit an inherent structure within hypergraphs from a particular application domain. We also note, drawing from experiences in parallel graph partitioning literature, that formulating a parallel algorithm that exactly replicates the behaviour of the serial partitioning algorithm is not necessary. By approximating the serial algorithms, parallel algorithms may be able to achieve better concurrency.

[2]The number of colours used is the chromatic number of the graph. Suppose that in a graph $G(V, \mathcal{E})$ the largest vertex clique has d vertices. Then, d is (trivially) a lower bound on its chromatic number $\chi(G)$.

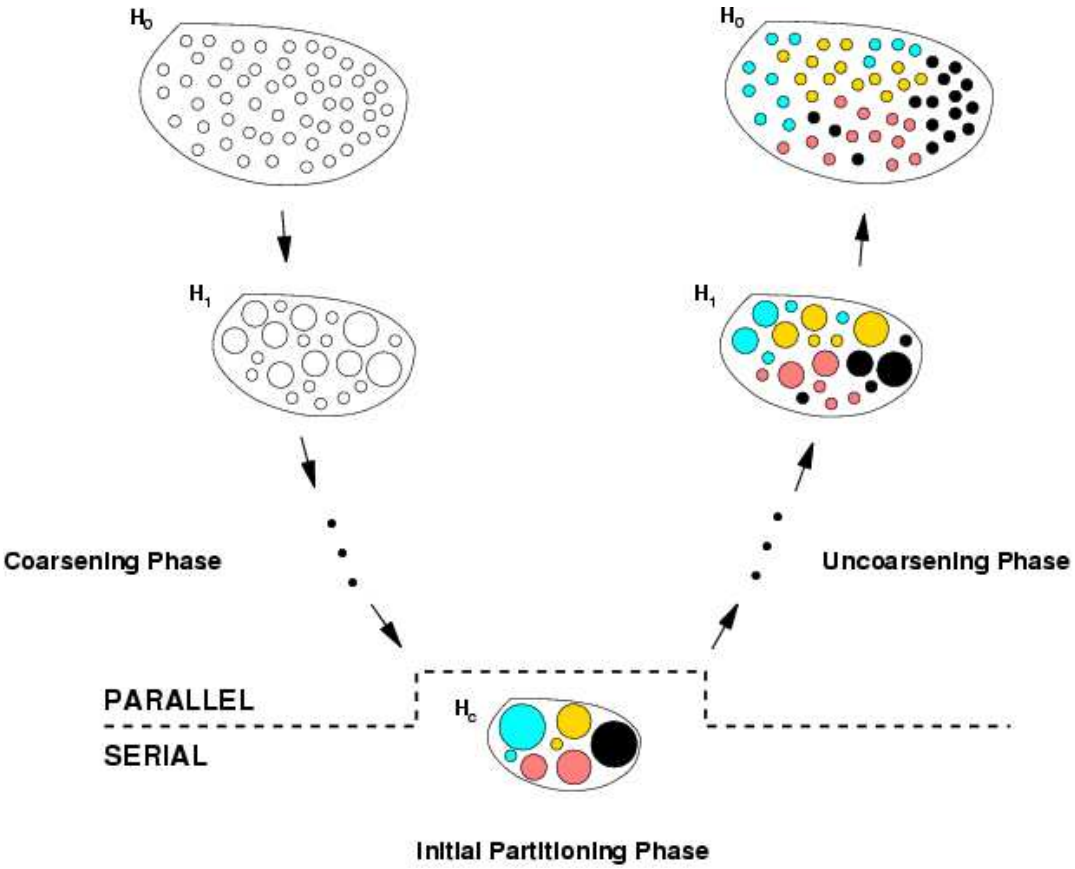

Figure 4.1. The parallel multilevel hypergraph partitioning pipeline.

4.3 An Application-Specific Disk-Based Parallel Algorithm

This section describes the application-specific disk-based parallel multilevel partitioning algorithm. The disk-based parallel hypergraph partitioning algorithm is the first contribution of this thesis and it was first presented in [TK04c]. We sought a high-capacity application-specific solver, motivated by the successful use of out-of-core techniques to solve large systems of linear equations [KH99].

The algorithm was designed to exploit an inherent characteristic of transition matrices of Markov and semi-Markov chains, when generated by a breadth-first search of the state-transition graph. An example of such a transition matrix, with the distinctive, almost lower-triangular structure, is shown in Figure 4.2

[Kno00].

Our target application is parallel sparse matrix–vector multiplication within an iterative Laplace Transform inversion algorithm used to compute probability distributions of response times in queueing systems [BDKW03, BDKW04]. The Laplace transform inverter solves a large number of sparse systems of complex linear equations; each sparse system of linear equations is solved using a parallel iterative method, for which the kernel operation is parallel sparse matrix–vector multiplication. In [BDKW03, BDKW04], serial hypergraph partitioning was used to accelerate the parallel sparse matrix–vector multiplication within the Laplace Transform inverter. However, in order to analyse systems of realistic size, high-capacity hypergraph partitioning is required.

We describe in more detail the hypergraph models used for sparse matrix decomposition in parallel sparse matrix–vector multiplication in Section 6.3.1. Here, the hypergraph is derived from the transition matrix by considering it as the incidence matrix of a hypergraph. The matrix rows define the vertices and the matrix columns the hyperedges of the hypergraph, so that a partition of the hypergraph corresponds to a row-wise decomposition of the matrix to processors; the partitioning objective exactly quantifies the communication volume of parallel sparse matrix–vector multiplication and is minimised. This is consistent with the one-dimensional hypergraph row-wise decomposition model for parallel sparse matrix–vector multiplication [cA99].

Following the remarks in Section 4.2.2, only the coarsening and the uncoarsening phases of the multilevel approach are parallelised. The target architecture for the algorithm is a cluster of commodity workstations connected by switched 100 Mbps Ethernet. We restrict our parallel algorithm formulation so that both the desired number of parts in the partition, k, and the number of processors, p, must be a power of two and $k \geq p$.

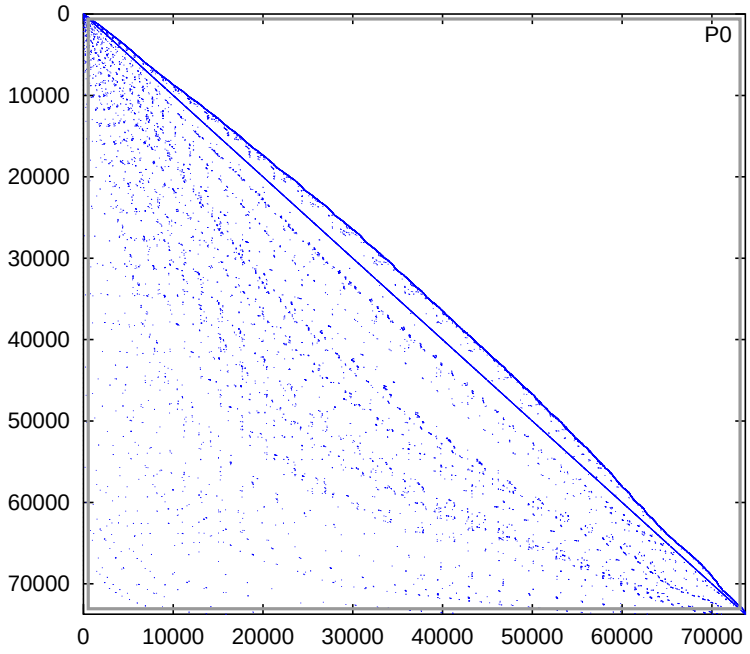

Figure 4.2. An example of a semi-Markov transition matrix generated by breadth-first state traversal.

4.3.1 Data Distribution

A natural way to store a hypergraph $H_i(V_i, \mathcal{E}_i)$ at stage i in a multilevel algorithm across p processors is to store $|V_i|/p$ vertices and $|\mathcal{E}_i|/p$ hyperedges on each processor. The processors are assigned non-overlapping sets of vertices; $V_i^{p_j}$ on processor p_j, such that $\bigcup_j V_i^{p_j} = V_i$.

In order to eliminate the communication overhead that would arise when locating remote vertices that are adjacent to a local vertex $v \in V_i^{p_j}$ on processor p_j, *all* hyperedges incident on vertices in $V_i^{p_j}$ are assigned to $\mathcal{E}_i^{p_j}$ (i.e. allocated to processor p_j).

Note that this hyperedge-to-processor allocation may result in some hyperedges being replicated across several processors, since vertices incident on such hyperedges will be assigned to different processors. We refer to these as *frontier* hyperedges. To increase capacity, the algorithm only stores in memory the hypergraph corresponding to the current stage in the multilevel pipeline; the remaining hypergraphs are stored on disk and are loaded into memory as required.

4.3.2 Parallel Coarsening Phase

The serial coarsening algorithm chosen for parallelisation is the First Choice (FC) coarsening algorithm from [KAKS97]. In a naïve parallel formulation, if the processors were to apply the serial FC algorithm to their local vertex sets concurrently during coarsening stage i, potentially excessive interprocessor communication may result (since adjacent vertices may be located across several processors).

In the belief that the approximate lower-triangular structure of the transition matrix can ensure a sufficient number of strongly connected local vertex clusters, our parallel coarsening algorithm only matches together vertices local to the processor. This avoids interprocessor communication during the vertex matching computation.

We first perform an experiment to indicate whether restricting the parallel coarsening algorithm in this manner would still yield a sufficient number of good vertex matches, when compared to the unrestricted coarsening algorithm. A simple serial vertex matching algorithm (edge coarsening [KAKS97]) was applied to hypergraphs derived from transition matrices of a semi-Markov model of a voting system [BDHK03]. The main characteristics of these voting hypergraphs can be found in Appendix A. It was conjectured that two vertices close in terms of their index (corresponding to two rows that are close in the transition matrix) would form good matches. This is because the upper triangular part of the matrix is mostly zero, while the diagonal region is the densest part of the matrix.

The serial edge coarsening algorithm traverses the vertex set V_i of the hypergraph in random order (cf. Section 3.5.2). Given an unmatched vertex $v \in V_i$, it looks for a maximal vertex match with another unmatched vertex, subject to the maximum prescribed cluster weight. If no suitable matches are found, v is copied over to V_{i+1}. The algorithm terminates when the coarse hypergraph has been reduced by a prescribed amount. We compute the connectivity metric between two given vertices as the sum of the weights of the hyperedges that connect the two vertices.

Hypergraph	partition size	% local	% remote	% singleton
voting100	2	90.1	0.9	9.0
voting100	4	88.1	2.9	9.0
voting100	8	84.4	6.7	8.9
voting100	16	77.2	13.9	8.9
voting100	32	62.8	28.2	9.0
voting125	2	90.4	0.7	8.9
voting125	4	88.8	2.3	8.9
voting125	8	85.9	5.2	8.9
voting125	16	79.9	11.2	8.9
voting125	32	68.6	22.5	8.9
voting150	2	91.5	0.7	7.8
voting150	4	90.2	2.0	7.8
voting150	8	87.5	4.7	7.8
voting150	16	82.4	9.8	7.8
voting150	32	72.4	19.8	7.8
voting175	2	91.6	0.6	7.8
voting175	4	90.5	1.7	7.8
voting175	8	88.2	4.0	7.8
voting175	16	83.8	8.4	7.8
voting175	32	75.1	17.1	7.8

Table 4.1. Percentages of match types in the edge coarsening algorithm during the vertex connectivity analysis.

We performed the experiment as follows. The vertex set V_i was partitioned into a number of subsets, each of which contained vertices with contiguous index. During the edge coarsening procedure, whenever unmatched vertices matched with unmatched vertices within the same subset, the match was called a *local* match. When vertices matched with others from a different subset, it was called a *remote* match. Finally, vertices that were simply copied over to the coarser hypergraph were denoted singleton matches. Table 4.1 shows the average percentages of the above match types over ten runs of the algorithm on the voting hypergraphs. The results of our experiment suggest that vertices seek matches with their immediate neighbours (which are defined in terms of their indices) in hypergraphs representing Markov and semi-Markov transition matrices. This follows from observing a high percentage of local matches and a low percentage of remote matches.

Our parallel coarsening algorithm exploits this property by allocating vertices of contiguous index to the set $V_i^{p_j}$ on processor p_j (this corresponds to allocating contiguous rows of the transition matrix to p_j).

Once the map g_i has been computed across the processors, we construct the hypergraph $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$ using $H_i(V_i, \mathcal{E}_i)$ and g_i. A b–bit hash key is associated with each hyperedge $e \in \mathcal{E}_i$ and used to assign "ownership" of hyperedges to processors. This hash key is computed using a hash-function $h : \mathbb{N}^a \rightarrow \mathbb{N}$ (described in more detail in Section 4.5.1), where a is the maximum hyperedge cardinality in $\mathcal{E}_i$. It posesses the desirable property that for an arbitrary set of hyperedges E, $h(e)$ `mod` p, $e \in E$, is near-uniformly distributed.

Each processor only contracts those hyperedges that have been assigned to it by the hash function. This ensures that only one copy of a given hyperedge is contracted (as potentially multiple copies of the given hyperedge exist across processors). Each processor first contracts the local hyperedges using its portion of the map g_i, then communicates this portion of the vector representation of g_i to the other $p - 1$ processors in a round-robin fashion. This enables every processor to fully contract its hyperedges. There is no need for explicit removal of duplicate hyperedges following the hyperedge contraction, because this is done when the hyperedges are read in from disk by the processors at the beginning of the subsequent coarsening step.

Motivated by [KK98c], we use a smaller number of processors as the hypergraph is reduced in successive coarsening steps. In our implementation, we only use processor numbers that are powers of two. When the hypergraph is considered to be small enough to fit on a single processor, a serial multilevel algorithm is used; in our implementation, we require that the hypergraph has been reduced by a factor of p before using a serial algorithm.

4.3.3 Initial Partitioning Phase

During this phase, a partition of the coarse hypergraph is computed serially on a single processor. In principle, any serial algorithm may be used, as this is the least time-critical phase in the parallel multilevel algorithm. For our experiments, we used the `HMETIS_PartKway()` routine from the `hMeTiS` library [KK98a].

4.3.4 Parallel Uncoarsening Phase

During the i^{th} level in the uncoarsening phase, each processor is responsible for vertices from k/p parts and the hyperedges incident on these vertices. As in the coarsening phase, some hyperedges are replicated across multiple processors (frontier hyperedges). The parallel refinement algorithm proceeds in a number of steps. During each step, every processor performs local refinement on the parts it currently owns (using serial FM when $k = 2p$, and using the serial greedy k-way refinement [KK98b] if $k > 2p$). If $p = k$, the processors pair-up and only one processor performs serial FM refinement on the two parts (rather than the processors interacting to perform FM refinement, as in [DPHL95]).

A round-robin communication of vertices and incident hyperedges is then performed, in order for subsequent steps to consider different directions of vertex move. The partition balance constraint is enforced locally, since each processor refines an independent set of parts. As in the parallel coarsening phase, fewer processors are used at the coarser levels.

4.3.5 Implementation and Experimental Evaluation

The parallel algorithm was implemented in the C++ language using the Message Passing Interface (MPI) standard [SOHL$^+$98] for interprocessor communication, forming the `Par`*k*`way1.0` tool. This experimental implementation consisted of three phases:

1. Parallel coarsening using the parallel formulation of the First Choice coarsening algorithm from Section 4.3.2. A reduction ratio of 2.0 (cf. Equation 3.7) was enforced on each processor.

2. Serial initial partitioning performed by the `HMETIS_PartKway()` routine of the `hMeTiS` library. This is called on a single processor when the coarse hypergraph has $\Theta(n/p)$ vertices.

3. Parallel refinement on the partition output by `HMETIS_PartKway()` using the parallel refinement algorithm described in Section 4.3.4.

The architecture used in the experiments consisted of a cluster of commodity PC workstations, connected by a switched 100 Mbps ethernet network. Each PC was equipped with a 2.8GHz Pentium(4) CPU and 1GB RAM.

The experimental evaluation was carried out on hypergraph representations of transition matrices from the voting model with 250 and 300 voters [BDHK03], yielding the voting250 and voting300 hypergraphs respectively. The hypergraphs were constructed from the sparse matrices according to the one-dimensional row-wise hypergraph model for sparse matrix decomposition [cA99]; their main characteristics can be found in Appendix A. In order to quantify the communication volume of parallel sparse matrix–vector multiplication exactly, the partitioning objective used in the experiments was the $k-1$ metric (cf. Definition 2.14). A partitioning balance constraint of 5% was imposed, equivalent to setting $\epsilon = 0.05$ in Equation 2.3.

Both problem instances were too large to be partitioned on a single workstation, so a suitable comparison was provided by the parallel graph partitioning tool `ParMeTiS` [KSK02], which implements the coarse-grained parallel algorithm described in Section 3.6.5. The transition matrices were converted into appropriate input for `ParMeTiS` according to the transformations described in [cA99]. We used default parameter values when running `ParMeTiS`. Note that it is not possible to explicitly enforce the balance constraint on partitions produced by

	voting250 results using 4 processors			
	Par$k$way1.0		ParMeTiS	
Partition size	$k-1$ objective	time(s)	$k-1$ objective	time(s)
8	91 511	1 309	117 354	25
16	182 206	1 393	249 415	27
32	354 561	1 495	402 681	32
64	525 856	1 777	610 597	33
total:	1 154 134	5 974	1 380 047	117

Table 4.2. Par$k$way1.0 and ParMeTiS: variation in partition quality and runtime for the voting250 hypergraph.

	voting300 results using 8 processors			
	Par$k$way1.0		ParMeTiS	
Partition size	$k-1$ objective	time(s)	$k-1$ objective	time(s)
16	322 737	4 827	442 387	85
32	529 763	4 762	687 659	61
64	874 652	5 007	1 033 312	80
total:	1 727 152	14 596	2 163 358	246

Table 4.3. Par$k$way1.0 and ParMeTiS: variation in partition quality and runtime for the voting300 hypergraph.

ParMeTiS; however, the vast majority of partitions produced satisfied the 5% balance constraint.

Table 4.2 presents the experimental results for the voting250 hypergraph and Table 4.3 results for the voting300 hypergraph. For completeness, these two tables also appear in Appendix B. The results indicate that the proposed parallel hypergraph partitioning algorithm significantly dominates the approximation given by parallel graph partitioning in terms of partition quality. On average, the algorithm produces partitions with $k-1$ objective values 20% lower than those produced by ParMeTiS on the voting300 hypergraph and 16% lower on the smaller voting250 hypergraph. In turn, ParMeTiS significantly dominates our disk-based algorithm in terms of runtime.

There are a number of reasons for the large difference in the respective runtimes. Firstly, hypergraph partitioning is inherently more "difficult" than graph partitioning and should thus take more time to compute. Secondly, our implementation experienced slow disk access time due to high disk contention; this may partly be due to the use of disk storage on a shared departmental file server. Thirdly, the parallel refinement algorithm required explicit communication of vertices and corresponding incident hyperedges in order to consider the different directions of vertex move. The volume of this communication was observed to be very large and increased with the number of processors used. Finally, as the hypergraph was "folded" onto fewer processors relatively early in the multilevel process, the parallel partitioning algorithm delivered relatively poor overall processor utilisation.

Nevertheless, we note that the very long runtimes of the disk-based parallel partitioning algorithm are not prohibitive in the context of our target application (parallel Laplace Transform-based response time computations) [BDKW03, BDKW04]. Here, parallel sparse matrix–vector multiplication is typically performed several hundred thousand times (there are thousands of systems of complex linear equations to solve, each requiring hundreds of iterations) using the same sparse matrix decomposition; the reduction in communication volume associated with better quality of decomposition when using a hypergraph model is thus also magnified by a factor of several hundred thousand.

4.4 Developing a General Parallel Hypergraph Partitioning Algorithm

4.4.1 Insights Gained From Preliminary Work

In Section 4.3, we described an application-specific disk-based parallel formulation of the multilevel hypergraph partitioning approach. Although the disk-based parallel algorithm exhibited poor scalability and long runtimes, it was able to pro-

duce suboptimal partitions of good quality. We will briefly summarise the most important conclusions drawn from our preliminary experiments.

The parallel coarsening and parallel refinement algorithms implemented within the disk-based parallel partitioning algorithm were motivated by the desire to avoid interprocessor conflicts occuring during coarsening and uncoarsening. We consider these schemes to be too restrictive. In particular, during the coarsening phase, disallowing inter-processor vertex matches in order to reduce the communication cost of the algorithm may significantly degrade the quality of the coarsening, as the number of processors is increased. In order to maintain the quality of the parallel coarsening algorithm during coarser levels, fewer processors were used by the disk-based parallel algorithm; however, this resulted in poor processor utilisation. Also, although conflicts were avoided in the parallel refinement algorithm, it was necessary to explicitly move vertices and the corresponding adjacency lists between processors. In addition, the parallel refinement algorithm did not provide a global view because it limited vertex moves to parts associated with each processor.

4.4.2 Discussion of Parallelism in Multilevel Hypergraph Partitioning

In this section, we consider in more detail the obstacles to, and opportunities for, concurrency in multilevel hypergraph partitioning algorithms.

Consider a one-dimensional distribution of a hypergraph $H_i(V_i, \mathcal{E}_i)$ across p processors during the i^{th} multilevel step, as described in Section 4.3.1. We assume that the number of vertices and the number of hyperedges in the hypergraph are of the same order of magnitude. Without loss of generality, we also assume that each processor stores n/p of the vertices. For the moment, we make no assumptions about the hyperedge-to-processor allocation.

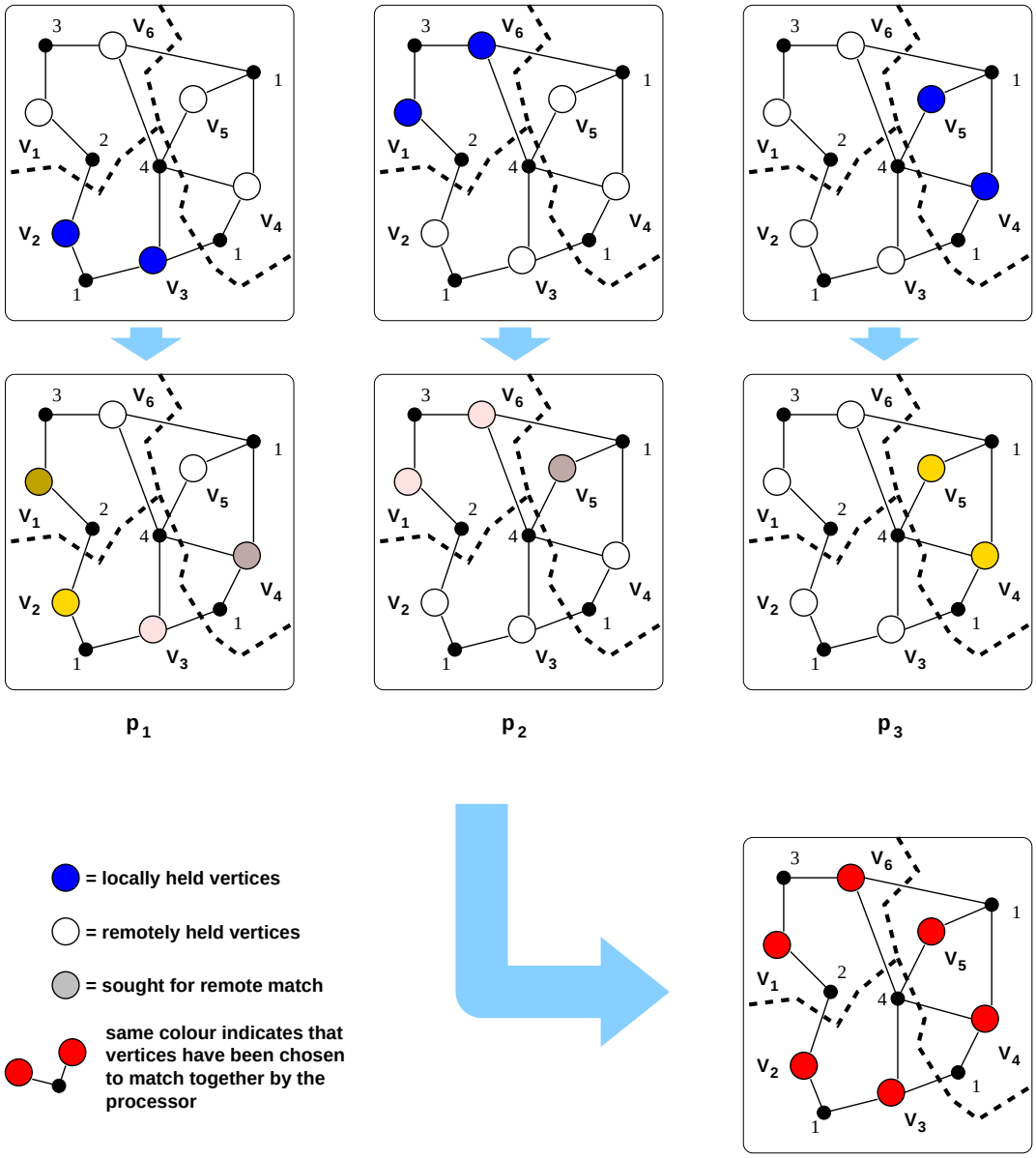

Figure 4.3. An example of conflict occurrence during parallel coarsening.

Parallelism during coarsening

Recall that step i in the coarsening phase first constructs the coarsening map g_i (cf. Section 3.5.2) by mapping strongly connected vertex clusters to a single coarse vertex. In parallel, the processors are required to make vertex matching decisions concurrently. We consider a naïve parallel formulation of the First Choice serial coarsening algorithm, in the scenario shown in Figure 4.3. Here, the numbers adjacent to the hyperedges denote their weight.

Processors p_1, p_2 and p_3 all proceed concurrently, with p_1 responsible for v_2 and v_3, p_2 responsible for v_1 and v_6, and p_3 responsible for v_4 and v_5. Processor p_1 considers v_2 to be most strongly connected to v_1 (which is held by p_2) and will send a message to p_2 requesting this match. Processor p_1 also considers v_3 to be most strongly connected to v_4 (held by p_3) and will also send a message to p_3 proposing to match v_3 with v_4. Processor p_2 has (locally) matched v_1 and v_6 together, and would like to match both with v_5 (held on processor p_3) because v_5 is most strongly connected to v_6. Processor p_2 will send a message to processor p_3 to request this match. Finally, processor p_3 locally matches v_4 and v_5.

Indiscriminately allowing all the requested interprocessor matches to proceed forms a cluster (consisting of vertices v_1 to v_6, inclusive) that will collapse to a single coarse vertex. Such an algorithm may produce coarse vertices that exceed the prescribed vertex weight threshold. It is possible to allow only a subset of the requested matches; however, this requires additional communication and processor synchronisation.

Having computed the map g_i, the serial coarsening algorithm uses it to construct $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$ from $H_i(V_i, \mathcal{E}_i)$. In parallel, each processor p_j stores a subset of hyperedges $\mathcal{E}_i^{p_j} \subset \mathcal{E}_i$ and a subset of vertices $V_i^{p_j} \subset V_i$. To contract hyperedges in $\mathcal{E}_i^{p_j}$, p_j may need values of g_i that are stored on other processors; this will induce interprocessor communication. Moreover, we note that a round-robin communication of g_i to all the processors (as in the disk-based parallel algorithm from Section 4.3) is not cost-optimal on most distributed-memory message-passing architectures.

Another important issue during the coarsening phase is maintaining the computational load balance during the coarser levels. Even though the initial distribution of the hypergraph $H(V, \mathcal{E})$ to processors ensures a computational load balance at the beginning of the coarsening process, a computational load imbalance may still occur at coarser levels because a hypergraph may coarsen at a faster rate on one or more processors than on the remaining processors. Finally, in order to ensure that the hypergraph shrinks by a sufficient amount during the coarsening phase,

the parallel coarsening algorithm may need to ensure that duplicate hyperedges in $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$ are eliminated. Again, we note that a round-robin communication of hyperedges in $\mathcal{E}_{i+1}$ followed by a local comparison of hyperedges on each processor is not cost-optimal on most distributed-memory message-passing architectures.

Parallelism during uncoarsening and refinement

At the beginning of step i in the uncoarsening phase, the serial multilevel algorithm projects the partition of the coarser hypergraph Π_{i+1} onto the hypergraph H_i. We note that in parallel this is similar to computing the hyperedge set $\mathcal{E}_{i+1}$ using the map g_i and $\mathcal{E}_i$, because processors may not store locally part values of vertices in H_{i+1} needed by local vertices in H_i.

After projecting Π_{i+1} onto H_i during step i in the uncoarsening phase, the serial multilevel algorithm performs heuristic refinement. KL and FM-based refinement algorithms require an update of the gain bucket data structures (or priority queues in KL) after every vertex move (cf. Section 3.4.1). Because vertices adjacent to the most recently moved vertex may be stored across many processors, maintaining and updating the correct part weights and the correct gain values after each vertex move requires repeated processor synchronisation, potentially involving all p processors.

In the context of graph partitioning, parallel refinement algorithms do not explicitly maintain priority queues or gain bucket data structures in order to reduce processor synchronisation and interprocessor communication (cf. Section 3.6). These parallel *approximations* to the serial refinement algorithm instead greedily move only those vertices whose moves lead to a gain in the objective function. In the context of serial multilevel hypergraph partitioning, Karypis and Kumar note that the serial greedy k-way refinement algorithm has good opportunities for concurrency [KK98b]. Recall from Section 3.5.4 that each pass of the serial greedy k-way refinement algorithm visits the vertices of the hypergraph $H_i(V_i, \mathcal{E}_i)$ in a random order. Each vertex $v \in V_i$ is moved to the part such that its move

does not violate the balance constraint and maximises the positive gain in the objective function. If no feasible move yielding a positive gain can be made, the vertex is not moved.

Now, consider the scenario in Figure 4.4, where again the numbers adjacent to the hyperedges denote their weight. Here, a naïve parallel formulation of the serial greedy k-way refinement algorithm from [KK98b] is running concurrently on processors p_1 and p_2. Processor p_1 computes a gain of two for moving vertex v_2 from part P_1 to part P_2 (a hyperedge of weight three is removed from the cutset, while a hyperedge of weight one is introduced into the cutset by the move). Meanwhile, processor p_2 computes a gain of two for moving vertex v_4 from part P_2 to part P_1 (a hyperedge of weight two is removed from the cutset by the move). However, when the two moves are executed concurrently, the overall gain is actually minus one, i.e. the moves have increased the cutsize of the partition from five to six. This phenomenon was called *vertex thrashing* in the context of parallel graph partitioning [KK97].

Finally, enforcing the balance constraint from Equation 2.3 while permitting concurrent vertex moves during refinement is non-trivial. Unless processors synchronise after every vertex move, it appears difficult to guarantee that concurrent vertex moves do not violate the balance constraint. It is possible to set tighter local balance constraints (on individual processors), such that the overall balance can be guaranteed if the local balance constraints are satisfied on each processor. However, this scheme may be too restrictive as it may not allow vertex moves with large gain that are not feasible in the context of the local balance constraint, but are nevertheless feasible within the (actual) global balance constraint (cf. Equation 2.3).

A remark on data distribution

There are also a number of data distribution issues that may potentially affect the scalability of the algorithm. Consider the case where the hyperedge sets stored across the processors are disjoint (i.e. there are no replicated frontier hy-

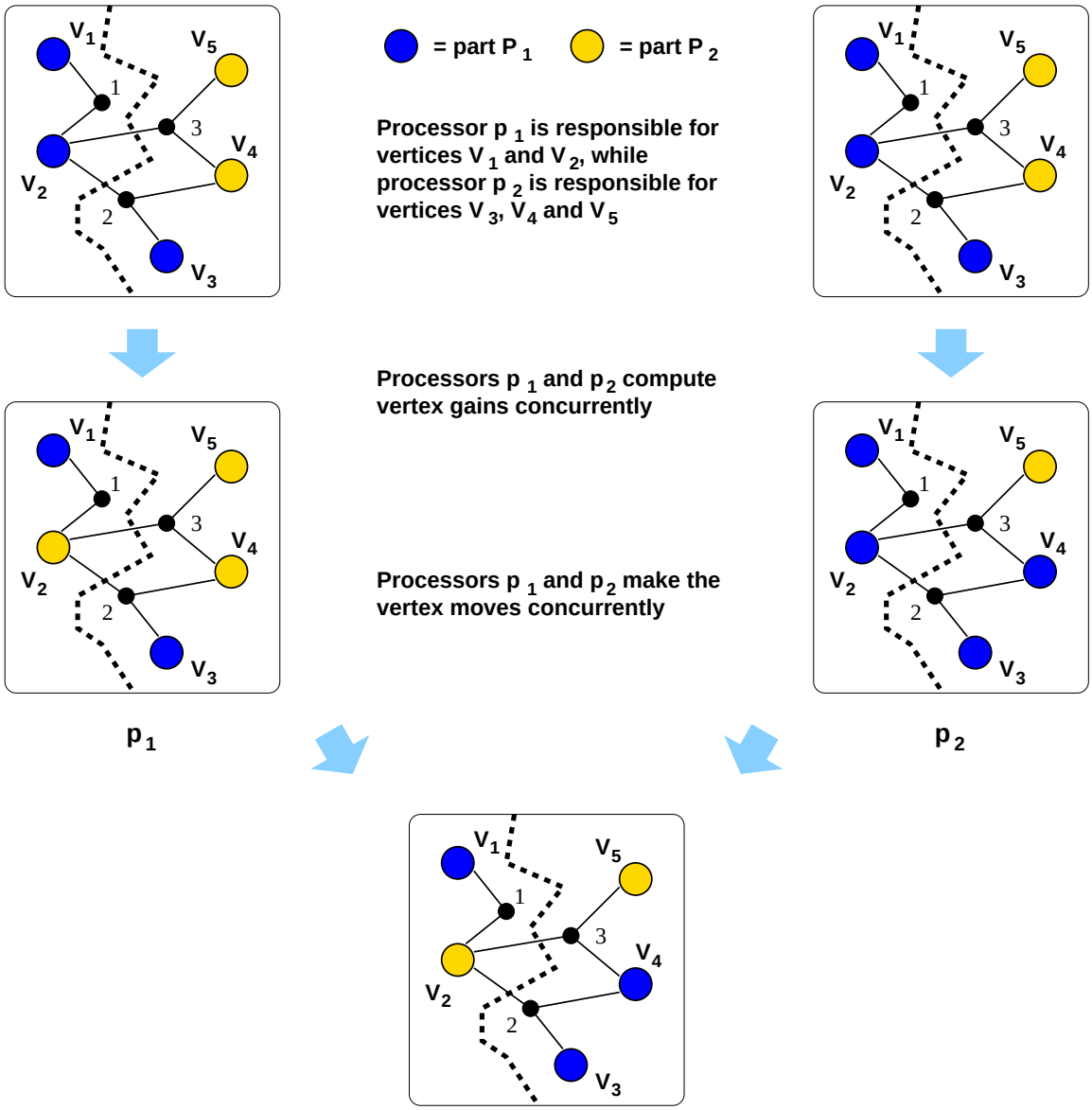

Figure 4.4. An example of conflict occurrence during parallel refinement.

peredges). It is difficult to guarantee that the locally-held hyperedge set $\mathcal{E}_i^{p_j}$ on processor p_j will contain all the hyperedges incident on the locally-held vertices (in $V_i^{p_j}$). In general, a large number of hyperedges incident on the locally-held vertices in $V_i^{p_j}$ may be stored on remote processors and in order for processor p_j to make meaningful coarsening and refinement decisions, they will need to be communicated to p_j.

The performance of a parallel algorithm is also affected by the computational load balance across the processors. At the beginning of the parallel multilevel algorithm, the locally-held hyperedge sets $\mathcal{E}^{p_j}$ may approximately be of the same size, but as the coarsening phase progresses, this may no longer hold (for example, hyperedges incident on strongly-connected groups of vertices are more likely to be removed in the coarser hypergraphs).

4.5 Parallel Multilevel Partitioning Algorithms

4.5.1 Data Distribution

The hypergraph $H(V, \mathcal{E})$ is distributed across the processors as follows. Each processor stores $|V|/p$ vertices and $|\mathcal{E}|/p$ hyperedges. The vertices are allocated to processors lexically, so that the first $|V|/p$ vertices (in terms of their index) are allocated to the first processor, the subsequent $|V|/p$ vertices to the second processor and so on. For each vertex $v \in V$, its weight and current part index in the partition are stored on the processor holding v and similarly, for each hyperedge $e \in \mathcal{E}$, its weight is stored on the processor holding e.

A randomized allocation of vertices to processors can be achieved by computing a pseudorandom permutation of the indices of the elements of the set V and then modifying the hyperedge set $\mathcal{E}$ by assigning to every vertex in each hyperedge a new index, as given by the permutation.

A b-bit hash key is associated with each hyperedge $e \in \mathcal{E}$. It is computed using the hash-function $h : \mathbb{N}^a \rightarrow \mathbb{N}$, where a is the maximum hyperedge cardinality. This hash function is a variant of the load balancing hash function from [Kno00]. An implementation of the function computing a 32-bit hash key is illustrated in Listing 4.1.

The hash-function has the desirable property that for an arbitrary set of hyperedges E, $h(e)$ `mod` p, $e \in E$, is near-uniformly distributed. Consequently, in order to ensure an even spread of hyperedges across the processors, each hyperedge

```
typedef unsigned int hashkey;

hashkey hashf(int *hedge, int length) {
  int slide1 = 0, slide2 = sizeof(hashkey) / 2, sum = 0;
  hashkey key = 0;
  for (int i = 0; i < MAX_HYPEREDGE_LENGTH; ++i) {
    if(i < length) {
      sum = sum + hedge[i];
      key = XOR(key, rotateLeft(hedge[i], slide1));
    } else {
      sum = sum + 1;
      key = XOR(key, rotateLeft(1, slide1));
    }
    key = XOR(key, rotateLeft(sum, slide2));
    slide1 = Mod(slide1 + 7, sizeof(hashkey));
    slide2 = Mod(slide2 + 13, sizeof(hashkey));
  }
  return key;
}

unsigned int rotateLeft(unsigned int num, int times) {

  int right = sizeof(hashkey) - times;
  return (OR(Shiftr(num, right), Shiftl(num, times)));
}
```

Listing 4.1. Load-balancing hash function with 32-bit hash keys.

e resides on the processor given by $h(e)$ `mod` p. To calculate the probability of collision, assume that h distributes the keys independently and uniformly across the key space (i.e. that all $M = 2^b$ key values are equally likely) and let $C(N)$ be the number of hash key collisions among N distinct hyperedges. We then have

$$\begin{aligned} \mathbb{P}(C(N) \geq 1) &= 1 - \mathbb{P}(C(N) = 0) && (4.1) \\ &= 1 - \frac{M!}{(M-N)!M^N} && (4.2) \\ &\leq 1 - e^{\frac{-N^2}{2M}} && (4.3) \end{aligned}$$

if $N^2 \ll M$, as shown in [Kno00]. Suppose that, for example, $|\mathcal{E}| = 10^8$, the

hyperedges are distinct and that $b = 64$. Then $\mathbb{P}(C(10^8) \geq 1) \leq 0.0003$ – ensuring that the probability of collisions is remote. This facilitates rapid hyperedge comparison, since given hyperedges e and e', $h(e) \neq h(e')$ implies that $e \neq e'$. The converse does not hold, but collisions do not affect the correctness of the algorithm; when a collision occurs between hyperedges $e, e' \in \mathcal{E}$, full sets e and e' can be compared.

At the beginning of every multilevel step, each processor assembles the set of hyperedges that are incident on each of its locally held vertices using an all-to-all personalized communication. A map from the local vertices to their adjacent hyperedges is then built. At the end of the multilevel step, the non-local assembled hyperedges are deleted together with the vertex-to-hyperedge map. Note that during the multilevel step, frontier hyperedges may be replicated on multiple processors, but only for the hypergraph used in the current multilevel step. Experience suggests that, given a sparse hypergraph with small average vertex degree, the memory overhead incurred by duplicating frontier hyperedges at the current multilevel step is modest.

4.5.2 Parallel Coarsening Phase

This section describes the parallel formulation of the First Choice (FC) serial coarsening algorithm [KAKS97, KK98b]. In principle, a number of the other algorithms described in Section 3.5.2 may also be parallelised in this fashion, with the data distribution as described in Section 4.5.1.

Recall from Section 3.5.2 that given a hypergraph $H_i(V_i, \mathcal{E}_i)$, the serial FC coarsening algorithm proceeds by visiting the vertices of the hypergraph in a random order. For each vertex $v \in V_i$, all vertices (both those already matched and those unmatched) that are connected via hyperedges incident on v are considered for matching with v. A connectedness metric is computed between pairs of vertices and the most strongly connected vertex to v is chosen for matching, provided that the resulting cluster $v' \in V_{i+1}$ does not exceed a prescribed maximum weight. The matching computation ends when $|V_i|/|V_{i+1}| > r$.

In parallel, each processor p_j traverses its local vertex set $V_i^{p_j}$ in random order, computing vertex matches as in the serial algorithm.

Processor p_j also maintains a *request set* for each of the $p-1$ other processors. If the best match for a local vertex $u \in V_i^{p_j}$ is computed to be a vertex v stored on processor $p_l \neq p_j$, then the vertex u is placed into the request set S_{p_j,p_l}. If another local vertex subsequently chooses u or v as its best match, then it is also added to S_{p_j,p_l}. The local matching computation terminates when the ratio of the number of local vertices $|V_i^{p_j}|$ to the number of local coarse vertices exceeds a prescribed threshold (cf. Equation 3.7). When computing the cardinality of the local coarse vertex set, potential matches with vertices from other processors are included.

Now, with a view to addressing the potential conflicts outlined in Section 4.4, there follows a communication step to resolve the vertex matching requests that span multiple processors. In order to enable a match between two vertices on different processors that make mutual requests to each other, this proceeds in two stages. In the first stage, processor p_j only communicates request sets S_{p_j,p_l} to processor p_l and only receives replies to its requests from p_l if $j > l$, while in the second stage processor p_j only communicates request sets S_{p_j,p_l} to processor p_l and only receives replies to its requests from p_l if $j < l$. This communication pattern is similar to that described in [KK96], in the context of parallel graph partitioning.

The processors concurrently decide to accept or reject matching requests from other processors. Denote by $M^v_{p_j,p_l}$ the set of vertices (possibly consisting of a single vertex) from the remote processor p_j that seeks to match with a local vertex v stored on processor p_l (thus, $S_{p_j,p_l} = \bigcup_x M^x_{p_j,p_l}$). Processor p_l considers the sets $M^v_{p_j,p_l}$ for each of its requested local vertices $v \in V_i^{p_l}$ in turn, handling them as follows:

1. If v is unmatched, matched locally or already matched remotely (during the previous request communication stage), then a match with $M^v_{p_j,p_l}$ is granted to processor p_j if the weight of the combined cluster (including

vertices already matched with v) does not exceed the maximum allowed vertex weight.

2. If v has been sent to a processor p_s, $p_s \neq p_j$, as part of a request for another remote match, then processor p_l informs processor p_j that the match with $M^v_{p_j,p_l}$ has been rejected. This is necessary, since granting this match potentially results in a vertex $w \in V_{i+1}$ that exceeds the maximum allowed vertex weight if the remote match of v with a vertex on processor p_s is granted (as illustrated in Section 4.4). When informed of the rejection by processor p_l, processor p_j will locally match the set $M^v_{p_j,p_l}$ into a single coarse vertex.

Note that in an implementation only the combined weight of the vertices in $M^v_{p_j,p_l}$ and the index of vertex v need to be communicated from processor p_j to processor p_l. The set S_{p_j,p_l} can be received as an array on processor p_l and in our implementation, is processed in random order (because we do not quantify importance of individual matching requests).

Having computed the map $g_i : V_i \rightarrow V_{i+1}$, the coarsening step is completed by constructing $\mathcal{E}_{i+1}$, which is done by applying g_i to every vertex in each $e \in \mathcal{E}_i$. The values of g_i not stored by a processor but required to transform a hyperedge stored on that processor are first communicated by a personalized all-to-all communication. Each processor then applies g_i across each of the $|\mathcal{E}_i|/p$ hyperedges stored on that processor. The removal of duplicate hyperedges in $\mathcal{E}_{i+1}$ and load balancing are done as follows. Processors communicate each hyperedge $e \in \mathcal{E}_{i+1}$ and its weight to the destination processor given by $h(e)$ `mod` p. The processors retain distinct hyperedges, setting their weight to be the sum of the weights of their respective duplicates (if any), since all identical hyperedges will possess the same hash key value and hence will have been communicated to the same processor. The parallel coarsening step concludes with a load-balancing communication of V_{i+1} such that each processor stores $|V_{i+1}|/p$ vertices at the start of the subsequent coarsening step.

4.5.3 Serial Initial Partitioning Phase

It is assumed that the hypergraph $H_c(V_c, \mathcal{E}_c)$ is small enough for a partition to be rapidly computed on a single processor. Thus, the details of this phase are not central to the algorithm. As discussed in Section 4.2.2, in principle, some multiple of p runs of a given serial algorithm could be computed across the processors in parallel, or the k-way partition could be computed via recursive bisection, with the bisections performed in parallel.

4.5.4 Parallel Uncoarsening Phase

Consider the i^{th} multilevel step during the uncoarsening phase, with $H_i(V_i, \mathcal{E}_i)$, $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$ and a k-way partition Π_{i+1} of H_{i+1}. The projection $H_{i+1} \xrightarrow{\Pi_{i+1}} H_i$ is computed in parallel as follows. Let $V_i^{p_j}$ be the subset of V_i stored on processor p_j. For each $v \in V_i^{p_j}$, if $\Pi_{i+1}(g_i(v))$ is not available, it is requested from the processor that is responsible for $g_i(v) \in V_{i+1}$. Then, we set $\Pi_i(v) = \Pi_{i+1}(g_i(v))$.

The projected partition Π_i is refined in parallel using a parallel formulation of the greedy k-way serial refinement algorithm [KK98b] (cf. Section 3.5.4). Our parallel formulation was motivated by the coarse-grained parallel graph partitioning algorithm from [KK97] (cf. Section 3.6.5).

The parallel greedy k-way refinement algorithm proceeds in passes, during each of which a vertex can be moved at most once; however, instead of moving single vertices across a partition boundary, as in the serial algorithm, the parallel algorithm moves sets of vertices (since vertices will be moved concurrently across the processors). Each processor p_j traverses the local vertex set $V_i^{p_j}$ in random order and for each $v \in V_i^{p_j}$, the legal move (if any) leading to the largest positive gain in the objective function is computed. When such moves exist, they are maintained in sets $U_{i,l}^{p_j}$, $i \neq l$, $i, l = 1, \ldots, k$, where i and l denote current and destination parts respectively. In order to reduce the effect of vertex thrashing (cf. Section 4.4.2), the refinement pass proceeds in two stages. During the first stage, only moves from parts of higher index to parts of lower index are permitted

and vice versa during the second stage. Vertices moved during the first stage are locked with respect to their new part in order to prevent them moving back to their original part in the second stage of the current pass. Note that this does not explicitly guarantee that a vertex move will always yield a non-negative gain, but in practice, occurences of vertex thrashing are rarely observed.

As noted in Section 3.6, most parallel graph refinement algorithms attempt to maintain partition balance without explicitly guaranteeing tight enforcement of the partition balance constraint. In contrast, the parallel hypergraph refinement algorithm presented in this section is guaranteed to produce a partition within the balance constraint of Equation 2.3.

The partition balance constraint is maintained as follows. A particular processor is first assigned the status of root processor at the start of refinement. At the beginning of each of the two stages of a pass, all processors know the exact part weights and maintain the balance constraint during the local computation of the sets $U_{i,l}^{p_j}$. The associated weights and gains of all the non-empty sets $U_{i,l}^{p_j}$ are communicated to the root processor which then determines the actual partition balance that results from the moves of the vertices in the sets $U_{i,l}^{p_j}$. If the balance constraint is violated, the root processor determines which of the moves should be taken back to restore the balance and informs the processors containing the vertices to be moved back. This may be simply implemented as a greedy scheme favouring taking back moves of sets with large weight and small gain. Finally, the root processor broadcasts the updated part weights before the processors proceed with the subsequent stage. As in the serial algorithm, the refinement procedure terminates when the overall gain of the most recent pass is not positive. Note that vertices need not explicitly be moved between processors; rather, their part index value can be changed by the processor that stores the vertex, as in [KK96, KK97].

4.5.5 Parallel Multi-phase Refinement

In Section 3.5.4, we described the iterative application of a serial multilevel hypergraph partitioning algorithm within multi-phase refinement. Each such iteration

is called a V-cycle. This section describes a parallel multi-phase refinement algorithm, consisting of the three multilevel phases: the parallel restricted coarsening phase, the serial initial partitioning phase and the parallel refinement phase. The serial initial partitioning and the parallel uncoarsening phases are identical to those described in Section 4.5.3 and Section 4.5.4. The parallel restricted coarsening phase operates as follows.

The parallel restricted coarsening phase takes a partition Π_i of the hypergraph H_i as input. A vertex is only allowed to match with an adjacent vertex that belongs to the same part within the partition, i.e. $v \in V_i$ can match with $u \in V_i$ if, and only if, $\Pi_i(u) = \Pi_i(v)$. To reduce the communication cost of the algorithm, vertices belonging to the same part are collected onto a single processor. The coarsening algorithm then proceeds serially (but concurrently) on each processor. The partition balance criterion in Eq. 2.3 should ensure that computational load balance across the processors during this phase is maintained. The construction of the coarse hypergraph $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$ is done as in the parallel coarsening algorithm from Section 4.5.2.

4.6 Analytical Performance Model

In this section, we present an analytical performance model of the parallel algorithms described in Section 4.5 and derive the asymptotic runtime T_p on p processors by modelling average-case behaviour. A theoretical scalability analysis is then carried out. Assuming an asymptotic runtime of $O(n)$ for the serial multilevel partitioning algorithm (cf. Section 3.5.5), we show conditions under which our parallel algorithm is cost-optimal and derive its isoefficiency function.

We consider the input hypergraph $H(V, \mathcal{E})$ and let $n = |V|$ and $m = |\mathcal{E}|$. Moreover, let l and d denote the average hyperedge cardinality and the average vertex degree of $H(V, \mathcal{E})$, respectively. We assume that $H(V, \mathcal{E})$ has low maximum vertex degree and maximum hyperedge length (i.e. $e_{max} \ll n$ and $d_{max} \ll n$). Consequently, we have that $l \ll n$ and $d \ll n$.

We make the following assumptions about the input hypergraph $H(V, \mathcal{E})$ and the parallel multilevel hypergraph partitioning algorithm from Section 4.5:

1. $H(V, \mathcal{E})$ is sparse and m is of the same order of magnitude as n.
2. The numbers of vertices and the numbers of hyperedges are respectively reduced by constant factors $1 + \nu$ and $1 + \omega$ ($\nu, \omega > 0$) at each coarsening step.
3. The number of parts in the partition sought, k, is small, when compared to n ($k \ll n$).

Because n and m are of the same order of magnitude and are both reduced by constant factors at each multilevel step (assumptions 1 and 2), we need only consider the number of vertices in the asymptotic parallel runtime analysis (since the number of hyperedges at a given multilevel step will be of the same order of magnitude as the number of vertices at that step). Since the number of vertices in the successive coarser hypergraphs is reduced by a factor greater than one at each coarsening step, there are $O(\log n)$ coarsening steps before the coarsest hypergraph $H_c(V_c, \mathcal{E}_c)$ has $\Theta(k)$ vertices (as we assume $k \ll n$).

In the following analysis, let n_i denote the number of vertices in the hypergraph $H_i(V_i, \mathcal{E}_i)$ at the i^{th} multilevel step. The average vertex degree and average hyperedge cardinality of hypergraph H_i are given by l_i and d_i respectively. We note that l_i is bounded above by l and thus $l_i \ll n$ holds for all i. Typically, d_i increases (since in practice the number of hyperedges is usually reduced at a slower rate than the number of vertices by the parallel coarsening algorithm). However, d_i is bounded above by the number of hyperedges in H_i and in the latter stages of the coarsening process, this is an order of magnitude less than n, so that $d_i \ll n$ nevertheless. We assume that d_i remains small throughout the multilevel process.

4.6.1 Performance Model of the Parallel Multilevel Algorithm

The computation and the communication requirements of each phase, T_{cmp} and T_{cmm} respectively, are considered in turn; the asymptotic parallel runtime on p processors, T_p, is then given by the sum of the two ($T_p = T_{cmp} + T_{cmm}$) as described in Section 2.5.2.

Computation time-complexity

Consider first the procedure of assembling the hyperedges incident on the locally stored vertices at the i^{th} multilevel step. Each processor performs $O(n_i l_i/p)$ computation steps in determining destination processors for the locally held hyperedges and then $O(n_i d_i/p)$ computation steps in building a map from its local vertices to the hyperedges incident on these vertices.

Now, consider the parallel coarsening procedure at the i^{th} multilevel step. Here, $O(d_i l_i)$ computation steps are performed in computing matches for each of the $O(n_i/p)$ vertices stored on a processor. Each processor will also potentially perform $O(n_i/p)$ computation steps in resolving matching requests from other processors (assuming that each processor pair exchanges $O(n_i/p^2)$ match requests, which is reasonable if it is assumed that a match with any given vertex is equally likely). Having computed the matching vector, the algorithm constructs $\mathcal{E}_{i+1}$ from $\mathcal{E}_i$. To do this, each processor p_j computes the matching vector values required from other processors and, having obtained them, computes the set $\mathcal{E}_{i+1}^{p_j}$. The former requires $O(n_i l_i/p)$ and the latter $O((n_i l_i \log l_i)/p)$ computation steps. Once a coarse hyperedge is constructed, checking for local duplicate hyperedges in $\mathcal{E}_{i+1}^{p_j}$ is done using a hash table. It takes $O(l_i)$ steps to check for and resolve a possible collision if a duplicate key is found in the table. Since l_i and d_i can be bounded above by small constants, the overall computation requirement during each coarsening step (including assembling incident hyperedges) is $O(n_i/p)$.

During the serial initial partitioning phase, the hypergraph has size $\Theta(k)$ and

can be heuristically partitioned to yield a "good" sub-optimal partition in $O(k^2)$ computation steps, for example, by using one of the multilevel partitioning algorithms described in Section 3.5.

A single uncoarsening step consists of projecting a partition Π_{i+1} of H_{i+1} onto H_i and refining the resulting partition of H_i to obtain Π_i. Projecting a partition at the i^{th} multilevel step involves at most $O(n_i/p)$ computation steps on each processor. Now, consider a single pass of the parallel greedy refinement algorithm at the i^{th} multilevel step, and assume that the refinement algorithm terminates within a small number of passes. Vertex gains are computed concurrently and then rebalancing moves are computed on the root processor, if required. In order to compute the gain for a move of a vertex $v \in V_i$, the algorithm needs to visit all the hyperedges incident on v and determine their connectedness to the source and destination parts of the move. This requires $O(d_i n_i l_i/p)$ computation steps per pass. The rebalancing computation has complexity $O(pk^2)$, since all possible directions of vertex move may have to be considered. Again, as l_i and d_i can be bounded above by small constants, the overall computation requirement during each uncoarsening step (including assembling incident hyperedges) is $O(n_i/p)$.

The overall asymptotic computational complexity, T_{cmp}, of the parallel partitioning algorithm from Section 4.5 is thus given by

$$
\begin{aligned}
T_{cmp} &= \sum_{i=0}^{O(\log n)} \left[O(n_i/p) + O(pk^2)\right] && (4.4)\\
&= \left[\sum_{i=0}^{O(\log n)} O(n(1+\nu)^{-i}/p)\right] + O(pk^2 \log n) && (4.5)\\
&\leq \left[\sum_{i=0}^{\infty} \frac{O(n/p)}{(1+\nu)^i}\right] + O(pk^2 \log n) && (4.6)\\
&= O(n/p) + O(pk^2 \log n) && (4.7)
\end{aligned}
$$

Communication time-complexity

Our attention now shifts to an average-case communication cost analysis, assuming that the underlying parallel architecture is a p-processor hypercube with

bi-directional links and store-and-forward routing. This is a well-known example of a richly-connected parallel architecture [GGKK03]. Again, first consider the procedure of assembling the hyperedges incident to locally held vertices on each processor at the i^{th} multilevel step. This is done using an all-to-all personalized communication. Given $O(n_i/p)$ vertices on each processor, the algorithm will, on average, assemble $O(n_i d_i/p)$ hyperedges on each processor. Since d_i can be bounded above by a small constant and the hyperedges are approximately uniformly distributed across the processors by the hash function, the average message size between any two processors is $O(n_i/p^2)$. An all-to-all personalised communication with this message size can be performed in $O(n_i/p)$ time on a hypercube architecture [GGKK03]. This result is important, since all-to-all communications with message size $O(n_i/p^2)$ occur frequently in different phases of the algorithm.

Now, consider the cost of communicating the required matching vector entries in computing $\mathcal{E}_{i+1}^{p_j}$ from $\mathcal{E}_i^{p_j}$ on each processor p_j and the subsequent load balancing communication of hyperedges of H_{i+1}. Given $O(n_i/p)$ hyperedges in $\mathcal{E}_i^{p_j}$, to construct $\mathcal{E}_{i+1}^{p_j}$, each processor requires $O(n_i l_i/p)$ matching vector entries. Assuming that each of the required entries in g_i (see Section 3.5.2) are on average equally likely to be stored on any of the p processors, the message size between any two processors in the all-to-all communication is on average $O(n_i l_i/p^2)$. In the load balancing communication, the hyperedges in $\mathcal{E}_{i+1}^{p_j}$ are scattered across the p processors with equal probability, thus also giving an average message size in the all-to-all personalized communication of $O(n_i l_i/p^2)$. Hence, given that l_i can be bounded above by a small constant, these all-to-all personalized communications can be done in $O(n_i/p)$ time.

By a similar argument, assuming that a vertex is on average equally likely to match with any other vertex in the hypergraph during each coarsening step, the communication of matching requests and their outcomes is done in $O(n_i/p)$ time. During each coarsening step, a prefix sum computation is required to determine the numbering of the vertices in the coarser hypergraph, which has

time-complexity $O(\log p)$.

During a pass of the parallel refinement algorithm, an additional broadcast of rebalancing moves may be required, as well as a reduction operation to compute the objective function, which have time-complexities $O(k^2 \log p)$ and $O(\log p)$ respectively (since each processor may be required to take moves back in $O(k^2)$ directions).

Arguing as for the computational complexity T_{cmp} over $O(\log n)$ multilevel steps, one can deduce that the overall asymptotic communication cost, T_{cmm}, of the parallel partitioning algorithm is

$$T_{cmm} = O(n/p) + O(k^2 \log p \log n) \tag{4.8}$$

Asymptotic parallel runtime

Setting $T_p = T_{cmp} + T_{cmm}$ and eliminating dominated terms from Equation 4.7 and Equation 4.8, the asymptotic total parallel run time, T_p, is

$$T_p = O(n/p) + O(pk^2 \log n) \tag{4.9}$$

Since the asymptotic time-complexity of the serial algorithm is $O(n)$, it follows that the algorithm is cost-optimal if $p^2k^2 \log n = O(n)$.

Derivation of the isoefficiency function

As $W = O(n)$, from Equation 4.9 and Equation 2.16 in Section 2.5.2, the total overhead, T_o, of the parallel algorithm becomes:

$$T_o = O(p^2k^2 \log W) \tag{4.10}$$

The isoefficiency function is then given by Equation 2.23, from Section 2.5.3:

$$\begin{aligned} W &= O(p^2k^2 \log W) && (4.11)\\ &= O(p^2k^2 \log(p^2k^2 \log W)) && (4.12)\\ &= O(p^2k^2 \log p) + O(p^2k^2 \log k) + O(p^2k^2 \log \log W) && (4.13) \end{aligned}$$

Asymptotically, one can now omit the lower order term involving $\log \log W$ in Equation 4.13 to yield

$$W = O(p^2 k^2 (\log p + \log k)) \tag{4.14}$$

This isoefficiency function implies that when the number of processors is doubled, the problem size needs to increase by a little over a factor of four in order to maintain a constant level of efficiency.

We note that this isoefficiency function is of the same order (in terms of p) as that given in [KK96] for the parallel graph partitioning algorithm (described in Section 3.6.5). However, because hypergraph partitioning is inherently more difficult than graph partitioning, we expect the constants associated with the asymptotically dominating terms in the isoefficiency function of the parallel hypergraph partitioning algorithm to be larger than the equivalent constants for the parallel graph partitioning algorithm's isoefficiency function.

In applications where partitions of size $k = p$ are sought (such as dynamic load-balancing of parallel computations [DBH+05]), the isoefficiency function for the parallel multilevel hypergraph partitioning algorithm becomes $O(p^4)$, making it difficult to maintain constant efficiency with an increasing number of processors. We do note, however, that the k^2 term in Equation 4.14 is derived from the runtime complexity of the rebalancing computation within the parallel refinement algorithm (this is not present in the parallel graph partitioning algorithm [KK96]). In practice, we observe that this rebalancing is required infreqently and in the cases when it is required, significantly fewer than the possible $O(k^2 p)$ moves are actually taken back.

4.6.2 Model of Parallel Multilevel Algorithm with Multi-phase Refinement

Here, the parallel multilevel algorithm (analysed in Section 4.6.1) uses multi-phase refinement, as described in Section 4.5.5.

We assume that parallel V-cycles converge within a small number of iterations. The asymptotic performance model of the algorithm with parallel multi-phase refinement differs from that of the multilevel algorithm analysed in Section 4.6.1 only in the parallel restricted coarsening phase (as described in Section 4.5.5).

Recall from Section 4.5.5 that during the uncoarsening phase at multilevel step i, the partition Π_{i+1} is projected onto $H_i(V_i, \mathcal{E}_i)$, yielding partition Π_i. Then, multi-phase refinement is applied to $H_i(V_i, \mathcal{E}_i)$ and Π_i.

The parallel restricted coarsening algorithm first assigns the vertices belonging to the same part in Π_i to the same processor; then, the map g_i is computed concurrently on all processors in parallel, without communication. The remainder of the parallel restricted coarsening algorithm proceeds as the parallel coarsening algorithm described in Section 4.5.2, except that subsequent restricted coarsening steps j, $j > i$, do not involve interprocessor communication during the computation of the map g_j because vertices only match with other locally-stored vertices.

When allocating vertices from $H_i(V_i, \mathcal{E}_i)$ to processors so that all vertices from the same part in Π_i are on the same processor, an all-to-all personalised communication is used. It is assumed that the vertices have an equal probability of being assigned to each of the k parts of the partition Π_i, and assuming a randomised distribution of vertices to processors prior to partitioning, the average message size in the all-to-all communication will be $O(n_i/p^2)$. This all-to-all communication can be done in $O(n_i/p)$ time on a hypercube.

Since the k-way partition satisfies the partitioning constraint of Equation 2.3 and given a sufficiently low variance in vertex weights, each processor will on average hold $O(n_i/p)$ vertices after the above all-to-all communication. The computational requirement of a single step of the parallel restricted coarsening algorithm that computes the map g_i is then $O(n_i d_i l_i/p)$. Since d_i and l_i can be bounded above by small constants, a step of the parallel restricted coarsening algorithm has time-complexity of $O(n_i/p)$.

Given that the initial partitioning phase and the uncoarsening phase of a V-

cycle each have the same time-complexity as the respective phases in the parallel multilevel partitioning algorithm analysed in Section 4.6.1, the overall asymptotic time-complexity of the parallel multilevel partitioning algorithm with multi-phase refinement is of the same order as for the parallel multilevel algorithm in Section 4.6.1. Similarly, the cost-optimality conditions and the isoefficiency function of the parallel multilevel partitioning algorithm with multi-phase refinement follow from Section 4.6.1.

4.7 A New Two-Dimensional Parallel Hypergraph Partitioning Algorithm

Very recently, another approach to parallelism in multilevel hypergraph partitioning was explored by Devine et al. [DBH+06]. The authors use a two-dimensional distribution of the hypergraph $H(V, \mathcal{E})$ across the processors, which is analogous to the two-dimensional graph distribution described in Section 3.6. We present a brief summary of the proposed parallel algorithm, describing only the parallel coarsening and parallel uncoarsening phases. The algorithm is based on the multilevel approach and computes k-way partitions by recursive bisection.

The p processors are logically organised onto an $x \times y$ processor grid (so that $p = xy$), where x and y are the number of processors in the processor row and column, respectively. The hypergraph $H(V, \mathcal{E})$ is represented across the processors by the transpose of its incidence matrix $\mathbf{A}^T$, so that the rows in the matrix $\mathbf{A}^T$ correspond to the hyperedges of $H(V, \mathcal{E})$. $\mathbf{A}^T$ is distributed across the processors in Cartesian fashion, so that each processor is allocated a rectangular submatrix.

4.7.1 Parallel Coarsening Phase

The parallel coarsening algorithm is based on the serial *inner product* coarsening algorithm. An unmatched vertex $v_i \in V$ (corresponding to a column of the

matrix $\mathbf{A}^T$) is matched with another unmatched vertex v_j so that the inner product between columns i and j of $\mathbf{A}^T$ is maximised. This is equivalent to heavy-edge coarsening (cf. Section 3.5.2), since the rows of $\mathbf{A}^T$ correspond to the hyperedges and the s^{th} entry of both column i and j is non-zero if, and only if, vertices v_i and v_j are connected by a hyperedge.

Note that considering all possible vertex matches (column inner products) corresponds to computing $\mathbf{A}\mathbf{A}^T$; however, because the algorithm only considers inner products between two columns that share at least one row with non-zero entries (i.e. the corresponding vertices are connected by at least one hyperedge) and that both columns correspond to unmatched vertices, only a subset of the inner product operations are computed.

The parallel inner product algorithm proceeds in a number of rounds. During each round, every processor selects a random subset of its vertices, called the *candidates*. The candidates are then broadcast across the processor rows (so that matches with adjacent vertices can be computed) and across the processor columns (to ensure that the inner product is computed across the entire processor column). Each processor then computes the partial inner products between its local vertices and the external candidates received during the above broadcast. Having computed all the partial inner products locally, the *actual* inner product values are reduced across the processor columns onto a particular processor, chosen so that load balance is maintained.

The best potential vertex matches (given by the largest inner products) are accumulated on processors in processor row zero. These first greedily determine the best local vertex for each external candidate vertex. The local vertex is then locked, to prevent it subsequently matching with another vertex during the current round. Communication across row zero is then used to find the best global match for each external candidate vertex. The processor that owns the candidate vertex greedily picks the best matching vertex.

During the construction of the coarse hypergraph $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$ (using $H_i(V_i, \mathcal{E}_i)$ and the map $g_i : V_i \rightarrow V_{i+1}$), the parallel coarsening algorithm discards hyper-

edges of size one and removes duplicate hyperedges by computing and comparing hash values of the hyperedges. The hash values are computed by communication across processor rows and compared by communication across processor columns.

4.7.2 Parallel Uncoarsening Phase

Bisseling et al. use a parallel formulation of the FM serial algorithm to refine a projected partition. It performs pairs of passes, until the most recent pass pair fails to improve the partitioning objective or the number of pass pairs exceeds a prescribed limit.

Each pass within a pass pair involves moves in one direction only (either from P_0 to P_1 or vice versa), so that moves in both directions have been considered after the pass pair has been completed. This has the advantage that the overall gain of concurrent vertex moves across different processors is at least the sum of the individual vertex move gains.

The implementation of the parallel algorithm is as follows. In each processor column, the processor storing the largest number of non-zeros is designated as the *vertex mover*. The vertex mover attempts to move vertices from the source part to the destination part, subject to the partition balance constraint. After each vertex move, the vertex mover uses only its local data to update the gains of moves involving (necessarily local) adjacent vertices (in order to avoid inter-processor synchronisation). A local balance constraint is imposed on each vertex mover so that concurrent moves across the processors do not violate the (global) partitioning balance constraint.

4.7.3 Parallel Recursive Bisection

The splitting step in recursive bisection is computed in parallel. The authors note that the hypergraph splitting can be done explicitly, by communicating the two hypergraphs to their respective processors, or implicitly without communication,

by applying the parallel hypergraph partitioning algorithm first to one of the hypergraphs and then to the other hypergraph, using all p processors.

It is noted that explicitly splitting the hypergraph, although initially incurring additional communication, results in a reduction of communication time during recursive bipartitioning. This is because the communication operations involve fewer processors and enable individual processors to have a more global view during partitioning.

4.7.4 Experimental Results

The two-dimensional parallel hypergraph partitioning algorithm was implemented in ANSI C within the `Zoltan` toolkit, using the MPI library for interprocessor communication. The experimental architecture used was a Linux cluster with 236 dual-processor Intel Xeon (3.0 GHz) nodes and a Myrinet-2000 interconnect.

The `Zoltan` parallel multilevel hypergraph partitioner was compared with the serial multilevel hypergraph partitioner `PaToH`, the parallel multilevel hypergraph partitioner `Par`k`way2.0` (described in Chapter 5) and the parallel multilevel graph partitioner `ParMeTiS`. We note that in their experiments, the authors used a partition balance criterion of 10% (equivalent to $\epsilon = 0.1$ in Equation 2.3) and also that `ParMeTiS` was only run on hypergraphs derived from symmetric matrices.

In the experiments, the `Zoltan` parallel hypergraph partitioner exhibits faster runtimes than `Par`k`way2.0`; advantages of the two-dimensional hypergraph distribution over a one-dimensional one are discussed in Section 7.3. However, in general, `Par`k`way2.0` produced partitions of better quality than those produced by `Zoltan`. For completeness, the authors noted that the parallel graph partitioning approach using `ParMeTiS` was faster, but yielded partitions that were of inferior quality when compared to the two parallel hypergraph partitioning approaches.

Chapter 5

Parallel Implementation and Experimental Results

5.1 Introduction

This chapter describes the implementation of the parallel multilevel hypergraph partitioning algorithms from Section 4.5 in the parallel hypergraph partitioning tool Par*k*way2.0 and their experimental evaluation.

The remainder of this chapter is organised as follows. Section 5.2 provides an overview of our implementation of Par*k*way2.0. Section 5.3 then describes the experimental evaluation of the proposed parallel hypergraph partitioning algorithms on hypergraphs from a number of application domains, using Par*k*way2.0.

5.2 Par*k*way2.0: A Parallel Hypergraph Partitioning Tool

5.2.1 Software Architecture

This section describes the parallel hypergraph partitioning tool Par*k*way2.0, which was first presented in [TK04a] and which implements the parallel mul-

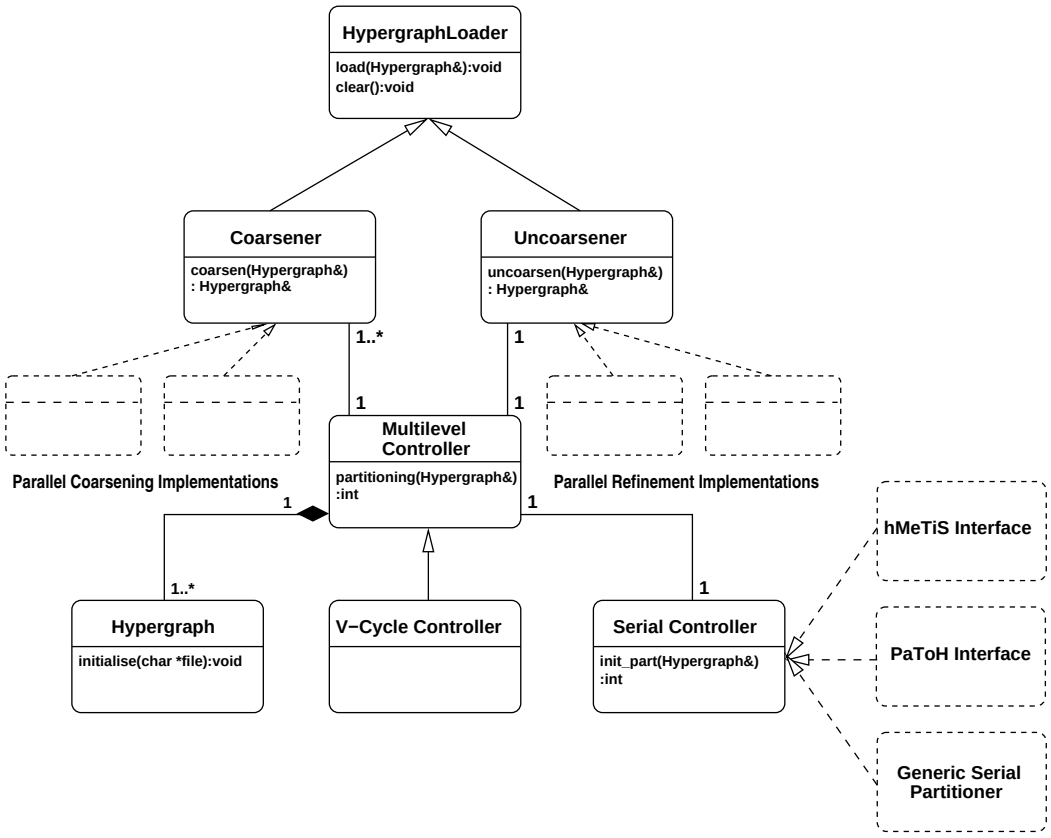

Figure 5.1. High-level diagram of the Par*k*way2.0 software architecture.

tilevel hypergraph partitioning algorithms described in Section 4.5. The tool is written in the C++ language using the Message Passsing Interface (MPI) library [SOHL$^+$98] for interprocessor communication.

MPI is a standardised message-passing library designed to function on a wide variety of distributed-memory parallel computers. The MPI standard is based on the C/C++ and Fortran languages and defines the syntax and the semantics of the library routines; however, in order to preserve portability across different parallel platforms, it does not specify how the routines should be implemented [SOHL$^+$98]. Because it enjoys widespread support from both industry and academia, MPI implementations are available for almost all commercial parallel computers [GGKK03].

The UML-style diagram in Figure 5.1 presents a high-level description of the software architecture of the Par*k*way2.0 tool. A hypergraph $H(V, \mathcal{E})$ is repre-

sented across the processors through the Hypergraph class, with each processor p_i storing sets $V^{p_i} \subset V$ and $\mathcal{E}^{p_i} \subset \mathcal{E}$, such that for all $0 \leq i < j < p$, we have that $V^{p_i} \cap V^{p_j} = \emptyset$ and $\mathcal{E}^{p_i} \cap \mathcal{E}^{p_j} = \emptyset$, while $\bigcup_i V^{p_i} = V$ and $\bigcup_i \mathcal{E}^{p_i} = \mathcal{E}$.

The composition of the set V^{p_i} is such that each processor p_i stores vertices of contiguous index, with p_0 storing the set that includes the vertex with the 0^{th} index. The composition of the set $\mathcal{E}^{p_i}$ is determined by a hash function, as described in Section 4.5.1. Two additional integer arrays store the weights of the vertices in V^{p_i} and the weights of the hyperedges in $\mathcal{E}^{p_i}$.

The input hypergraph may be read from disk or read from memory by Par$k$way2.0. These methods are implemented within the class Hypergraph. When reading the hypergraph from disk using p processors, p files need to be specified; one file to be read by each processor. The file to be read by processor p_i contains the weight and index information of the vertices in V^{p_i} and the weight and incidence information for each hyperedge in $\mathcal{E}^{p_i}$. Par$k$way2.0 also includes a number of routines for converting hypergraphs that are stored in other file formats (e.g. the PaToH [cA01b] and hMeTiS [KK98a] formats) into the Par$k$way2.0 format.

Through the load(Hypergraph&) method, the abstract class HypergraphLoader implements the assembly onto processor p_i of all the hyperedges in $\mathcal{E}$ that are incident on vertices in V^{p_i}. The implementation uses MPI's all-to-all personalised communication routines. The load(Hypergraph&) method is called prior to each multilevel step. The method clear() is used to delete all frontier hyperedges upon the completion of each multilevel step.

The abstract classes Coarsener and Uncoarsener define the parallel coarsening and uncoarsening methods. Specific implementations of these methods are provided in further subclasses. This allows for fast integration of new parallel coarsening and parallel refinement algorithms within the Par$k$way2.0 tool.

The MultilevelController class implements the top-level call to the parallel multilevel hypergraph partitioning algorithm, and maintains pointers to the sequence of approximate hypergraphs (including the original problem instance). The V-CycleController class extends the MultilevelController class by pro-

viding the additional multi-phase refinement capability. It uses an implementation of restricted parallel coarsening, in addition to an orthodox parallel coarsening implementation.

When partitioning the coarsest hypergraph $H_c(V_c, \mathcal{E}_c)$ serially, control of program execution passes to the `SerialController`. Implementations of this abstract class provide interfaces to existing state-of-the-art serial multilevel hypergraph partitioning tools (`PaToH` [cA01b] and `hMeTiS` [KK98a]) as well as a generic serial multilevel recursive bisection partitioner.

5.2.2 Details of the Par*k*way2.0 Implementation

Here we describe the most important aspects of the parallel multilevel hypergraph partitioning algorithm implementation.

Data Distribution

The `load(Hypergraph&)` method, implemented in parallel by `HypergraphLoader`, assembles on each processor p_i all the hyperedges incident on the local vertex set V^{p_i}. We note that hypergraphs from certain application domains contain a small number of hyperedges which have cardinalities significantly larger than the average hyperedge cardinality. For example, hypergraphs derived from web matrices exhibit this characteristic [GZB04].

Such hyperedges are likely to be replicated over many processors during the parallel coarsening and uncoarsening computations in a given multilevel step, since they are incident on a large number of vertices. Moreover, the presence of these large frontier hyperedges will increase the computational load on the respective processors.

We note that it may still be possible to compute a partition of the hypergraph without considering the (small number of) hyperedges with large cardinality [AHK98, cA99]. To this end, Par*k*way2.0 allows user specification of maximum hyperedge length; hyperedges with length greater than the prescribed threshold

do not take part in the multilevel step. During the coarsening phase, these hyperedges will nevertheless be contracted to form hyperedges in $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$.

Par*k*way2.0 also allows the user to specify whether the restriction on large hyperedges should apply during both the coarsening and uncoarsening phases or during the coarsening phase only, or not at all. Note that when large hyperedges are restricted during refinement, the parallel refinement algorithm can only optimise an approximation to the actual value of the partitioning objective.

Parallel Coarsening Algorithm Implementation

Recall from Section 4.5.2 that the serial First Choice coarsening algorithm from [KAKS97, KK98b] was used as the basis for our parallel coarsening algorithm. Now, consider a hypergraph $H_i(V_i, \mathcal{E}_i)$ at stage i in the coarsening process. We have implemented the following vertex connectivity metric between two vertices $u, v \in V_i$, based on those from [HB97, AHK98, KAKS97]:

$$C(u,v) = \frac{1}{w(u)+w(v)} \sum_{\{e \in \mathcal{E}_i : u \in e, v \in e\}} \frac{w(e)}{|e|-1} \tag{5.1}$$

Our parallel coarsening implementation within Par*k*way2.0 enables user modification of the above metric, so that division by $w(u)+w(v)$ (if the user is impartial to large vertex clusters during the coarsening process) and/or division by $|e|-1$ (if the user does not wish to differentiate hyperedge cardinalities) can be omitted from the metric.

Serial Partitioning Implementation

The `SerialController` (cf. Figure 5.1) implements serial methods for producing a partition of the coarsest hypergraph in the multilevel sequence. Currently, we employ three different methods within Par*k*way2.0.

The first of these is a generic implementation of a serial multilevel k-way hypergraph partitioning algorithm that uses recursive bisection. For the remaining

methods, `SerialController` provides interfaces to `HMETIS_PartKway()` (a routine from the `hMeTiS` library [KK98a] that implements the serial multilevel k-way hypergraph partitioning algorithm presented in [KK98b]) and `PaToH_Partition()` (a serial multilevel recursive bisection routine from the `PaToH` library [cA01b]). For serial partitioning by recursive bisection, `PaToH` was preferred to the multilevel recursive bisection variant of `hMeTiS` because in our experiments, `PaToH` was found to produce partitions of comparable quality but with shorter runtimes [TK04a].

Parallel Uncoarsening Algorithm Implementation

During the parallel uncoarsening phase, Par*k*way2.0 allows multiple (distinct) partitions that may have been computed during the serial partitioning phase to be propagated through the uncoarsening phase. This is specified by a user-prescribed percentage value. A value of x means that partitions with objective function within $x\%$ of the best partition at that level will be projected onto the successive finer level. The user may also specify for x to vary with the number of steps completed so far during the uncoarsening phase.

The Par*k*way2.0 implementation of the parallel greedy k-way refinement algorithm also supports a form of early exit from the current pass of the parallel refinement algorithm. A pass of the refinement algorithm may be terminated early if the number of successive vertices visited whose part index has not been changed (i.e. they have not been moved during the pass) exceeds a prescribed threshold, expressed as a percentage of the number of vertices in the hypergraph.

The user may invoke a similar early-exit condition for parallel multi-phase refinement, as well as set explicit limits on the number of passes and parallel V-cycles within a single run of the parallel refinement algorithm.

5.3 Experimental Evaluation

5.3.1 Aims and Objectives

This section presents our experimental evaluation of the parallel multilevel hypergraph partitioning algorithms. To this end, the Par*k*way2.0 tool was applied to hypergraphs from a wide range of application domains. The principal aims of the experiments were (cf. Section 1.1.5):

- To compare the Par*k*way2.0 tool with state-of-the-art serial multilevel hypergraph partitioning tools in terms of the quality of computed partition.
- To evaluate the parallel runtimes of the Par*k*way2.0 tool using state-of-the-art serial multilevel partitioning tools as base-case comparison.
- To evaluate scalability by observing Par*k*way2.0 processor efficiencies in experiments involving hypergraphs from the same application domain.

5.3.2 Experimental Setup

Software

In all experiments, the parallel hypergraph partitioning tool Par*k*way2.0, implementing the algorithms from Section 4.5, was used. When the test hypergraphs were small enough to be partitioned on a single processor, the base-case comparison was provided by the state-of-the-art serial multilevel hypergraph partitioning tools PaToH [cA01b] and hMeTiS [KK98a]. The HMETIS_PartKway() routine from the hMeTiS library was used to provide direct k-way serial multilevel hypergraph partitioning. For serial multilevel hypergraph partitioning by recursive bisection, the PaToH_Partition() routine from the PaToH library was preferred to the hMeTiS equivalent because we observed that it produced partitions of comparable quality with those produced by hMeTiS, but with shorter runtimes (cf. Section 5.2.2).

For test hypergraphs that were too large to be partitioned on a single processor, comparison was provided by the parallel multilevel graph partitioning tool `ParMeTiS` [KSK02]. The parallel graph partitioning algorithm implemented by `ParMeTiS` was described in more detail in Section 3.6. As noted in Section 4.1, in the absence of available parallel hypergraph partitioners, the use of an approximate graph model together with a parallel graph partitioner is currently the only way to partition very large hypergraphs.

In all experiments, unless explicitly stated otherwise, the parallel hypergraph partitioning tool `Par`k`way2.0` was configured as follows. The `load(Hypergraph&)` method was configured to consider all hyperedges (rather than omit a small number of large hyperedges, as described in Section 5.2.2).

During the parallel coarsening phase, we used the parallel formulation of the First Choice coarsening algorithm, described in Section 4.5.2. Vertex connectivity was quantified using the metric shown in Equation 5.1. The reduction ratio between successive coarser hypergraphs in the multilevel pipeline, given by Equation 3.7, was set to 1.75.

During the serial initial partitioning phase, we used the `HMETIS_PartKway()` routine with FC coarsening and V-cycle refinement applied to only the final partition. The serial initial partitioning was computed on each processor, so that we performed p serial partitioning runs in total when using p processors. We projected only one partition from the serial initial partitioning phase through the parallel uncoarsening phase.

In the parallel uncoarsening phase, during each run of the parallel greedy k-way refinement algorithm, no early exit was used and no explicit limit was set on the number of passes. We did not use parallel multi-phase refinement.

Test Hypergraphs

The hypergraph test cases for the experiments described in this section are shown in Table 5.1. All of the test hypergraphs, except the ibm hypergraphs, were

Name	Vertices	Hyperedges	Non-zeros	Domain
ibm16	183 484	190 048	778 823	VLSI CAD
ibm17	185 495	189 581	860 036	VLSI CAD
ibm18	210 613	201 920	819 617	VLSI CAD
voting100	249 760	249 760	1 391 617	performance analysis
voting125	541 280	541 280	3 044 557	performance analysis
voting150	778 850	778 850	4 532 947	performance analysis
voting175	1 140 050	1 140 050	6 657 722	performance analysis
voting250	5 218 300	5 218 300	32 986 597	performance analysis
voting300	10 991 400	10 991 400	69 823 797	performance analysis
cage13	445 315	445 315	7 479 343	DNA electrophoresis
cage14	1 505 785	1 505 785	27 130 349	DNA electrophoresis
cage15	5 154 859	5 154 859	99 199 551	DNA electrophoresis
ATTpre2	659 033	659 033	6 384 539	analogue circuits
uk-2002	18 520 486	18 520 486	310 764 149	PageRank analysis

Table 5.1. Significant properties of test hypergraphs.

derived from sparse matrices; they were constructed to accurately model the communication cost of 1D row-wise (or column-wise) decomposition of their corresponding sparse matrix for parallel matrix–vector multiplication, as described in Section 6.3.1. The ibm test hypergraphs were taken from the domain of VLSI CAD. These are the three largest hypergraphs from the ISPD98 Circuit Benchmark Suite [Alp98]. A more detailed description of the hypergraphs and their respective application domains is presented in Appendix A.

Experimental Platform and Configuration

We used the $k-1$ partitioning objective function in our experiments (cf. Definition 2.14). The $k-1$ objective function is used in circuit partitioning because it provides an accurate model of signal delay for nets that span multiple parts [Alp96]. In sparse matrix decomposition for parallel sparse matrix–vector multiplication and dynamic load balancing of parallel computations, the $k-1$ objective correctly quantifies the total communication volume [cA99, DBH$^+$05]. When using the `hMeTiS` tool, we chose to minimise the Sum of External Degrees (SOED) partitioning objective because `hMeTiS` cannot directly optimise the $k-1$ objective. SOED is closely related to the $k-1$ objective and is described in Section 2.3.2.

A balance constraint of 5% ($\epsilon = 0.05$ in Equation 2.3) was used in all of the experiments. Note that this specifies the partitioning constraint for the k-way

partitioning problem. When computing the k-way partition by recursive bisection, we imposed a different balance constraint for bipartitioning, so that the resulting k-way partition would satisfy the 5% balance constraint. Setting the balance constraint on each bipartitioning run to $(1.05/k)^{1/\log_2 k} - 0.5$ ensured that the resulting k-way partition was feasible. Note that this is only necessary for $k > 2$ (the formula yields an incorrect balance constraint when $k = 2$).

The parallel architecture used in all of the experiments consisted of a Beowulf Linux Cluster with 64 dual-processor nodes. However, in our experiments, we were only able to use a 32-processor (16-node) partition, due to configuration limitations and high machine utilisation. Each node in the cluster has two Intel Pentium 4 processors, running at 2GHz with 2GB RAM. The nodes in the cluster are connected by a Myrinet network, which has a peak throughput of 250 MB/s. We note that we were unable to gain access to a hypercube parallel architecture, which was assumed in the theoretical performance model.

All of our reported results (the serial and parallel runtimes and partition quality in terms of the $k - 1$ objective) are averages taken over ten randomised runs of the graph and hypergraph partitioning tools for the appropriate experiment.

5.3.3 Experiments to Evaluate Partition Quality

The purpose of these experiments was to compare the quality of the partitions produced by the parallel multilevel hypergraph partitioning algorithm with the quality of those produced by state-of-the-art serial multilevel hypergraph partitioning tools and an approximate parallel graph partitioning approach.

We noted in Section 4.2 that, in general, serial hypergraph partitioning algorithms are expected to produce slightly better partition quality than their respective parallel formulation. A contributing factor is that the parallel formulations *approximate* the serial algorithms in order to achieve better concurrency.

In our experiments, we make three distinct empirical comparisons. First, the performance of `Par`k`way2.0` is compared with state-of-the-art serial multilevel

tools PaToH and hMeTiS on the smaller test hypergraphs. Then, Par$k$way2.0 is compared with the graph partitioning-based approximation of hypergraph partitioning using the parallel multilevel graph partitioning tool ParMeTiS on large hypergraphs that could not be partitioned on a single processor. Finally, in the third experiment, multi-phase refinement is incorporated within Par$k$way2.0 and comparison is made with Par$k$way2.0 running with the orthodox parallel refinement algorithm. The complete set of experimental results for evaluating partition quality is presented (in tabular form) in Section B.2.

Comparisons Between Par$k$way2.0 and PaToH/hMeTiS

We first describe the settings used in the PaToH and hMeTiS tools for these experiments. The PaToH tool was used with settings for sparse matrices or VLSI hypergraphs, as appropriate [cA01b]. Coarsening was performed using the Heavy Connectivity Clustering (HCC) algorithm, described in Section 3.5.2. During the uncoarsening phase, the Boundary FM bisection algorithm was used, described in Section 3.5.4. The tool hMeTiS was run with the First Choice coarsening algorithm and the serial greedy k-way refinement algorithm, which are described in Section 3.5.2 and Section 3.5.4, respectively.

Figure 5.2 shows how the $k-1$ partitioning objective varies with the number of processors used on the ibm17, ibm18, cage13 and voting175 hypergraphs. The single processor $k-1$ objective value was taken as the minimum of the average values obtained by PaToH and hMeTiS.

We observe that the serial and parallel multilevel hypergraph partitioning algorithms produce partitions of comparable quality. It is significant that there is little variance in partition quality produced by the parallel multilevel partitioning algorithm as the number of processors is increased. A detailed look at the results in Section B.2 confirms this observation across all of the test hypergraphs. In some cases, for example voting250, partition quality improves slightly as the number of processors is increased. We conjecture that this is because running the parallel partitioning algorithm on a large number of processors allows a large

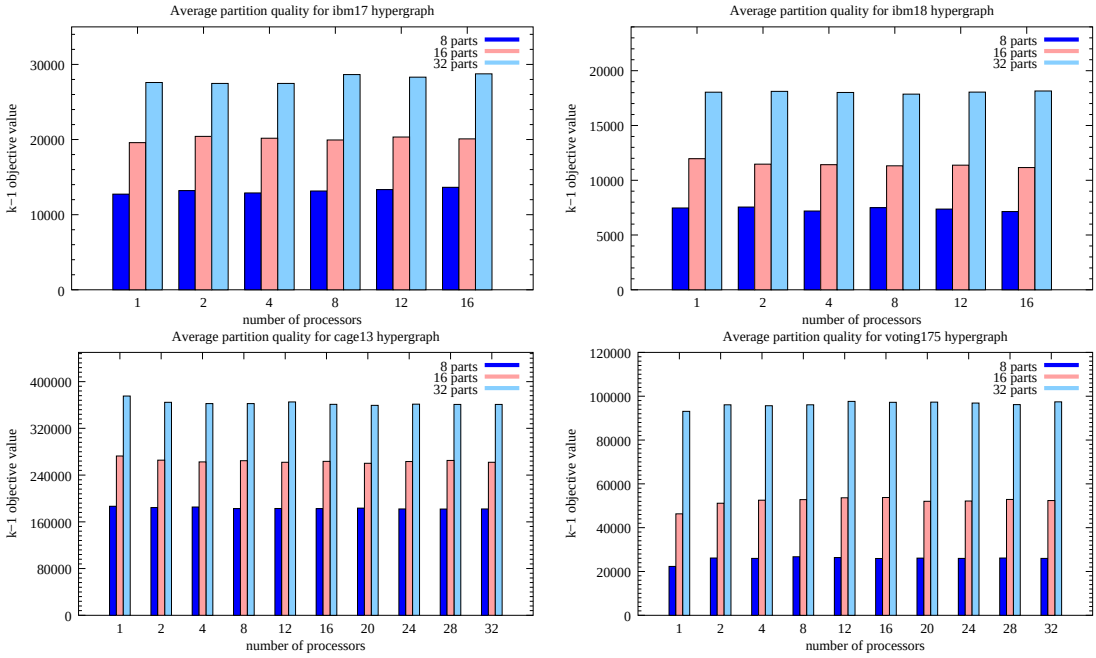

Figure 5.2. Par$k$way2.0 and hMeTiS/PaToH: variation of partition quality with the number of processors used.

number of partitioning runs to be performed during the serial partitioning phase, with no significant additional parallel runtime cost.

Comparisons Between Par$k$way2.0 and ParMeTiS

In these experiments, we compared the performance of Par$k$way2.0 with that of the approximate graph partitioning approach using the ParMeTiS parallel graph partitioning tool on hypergraphs that were too large to be partitioned serially.

The ParMETIS_PartKway() method within the ParMeTiS library was used, running with default parameters. We note that it was not possible to explicitly enforce the partitioning balance constraint on the partitions produced by ParMETIS_PartKway(). However, in practice, we usually observed the partition balance to be within the 5% threshold, although some of the larger partitions ($k = 32$) were found to violate the 5% constraint. The hypergraph-to-graph conversion was computed according to the construction of the graph model for 1D row-wise sparse matrix decomposition described in [cA99].

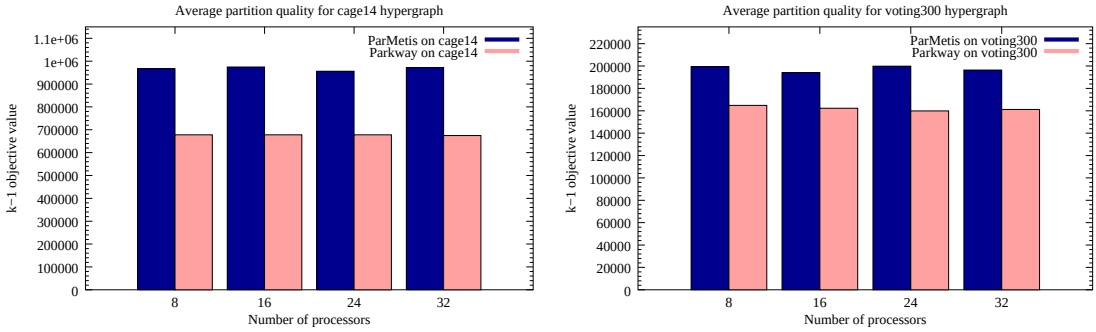

Figure 5.3. Par$k$way2.0 and ParMeTiS: variation of partition quality with the number of processors used for $k = 8$.

Figure 5.3 compares the $k-1$ objective values of 8-way partitions produced by Par$k$way2.0 and ParMeTiS on the cage14 and voting300 hypergraphs. We observe that Par$k$way2.0 consistently outperforms the approximate parallel graph partitioning-based approach using ParMeTiS, with a significant difference in the $k-1$ partitioning objective. This trend can also be seen across other large hypergraphs, as shown in Section B.2. For example, on the uk-2002 hypergraph, we observe that Par$k$way2.0 records improvements of up to 60% over ParMeTiS.

Experiments With Multi-phase Refinement

In this set of experiments, we investigate the effect that parallel multi-phase refinement has on the quality of partition produced by Par$k$way2.0. The parallel multi-phase refinement implementation in Par$k$way2.0 used in these experiments applied parallel V-cycles to each multilevel step during the parallel uncoarsening phase. Before each call to the parallel k-way refinement algorithm within a V-cycle, a random permutation of vertices to processors was carried out, because this was found to improve the partition quality, when compared to using the inherited distribution of vertices to processors. When using the parallel multi-phase refinement algorithm, we note that the partition size is required to be an integer multiple of the number of processors used, so that each processor can be assigned vertices from the same number of parts during the parallel restricted coarsening algorithm.

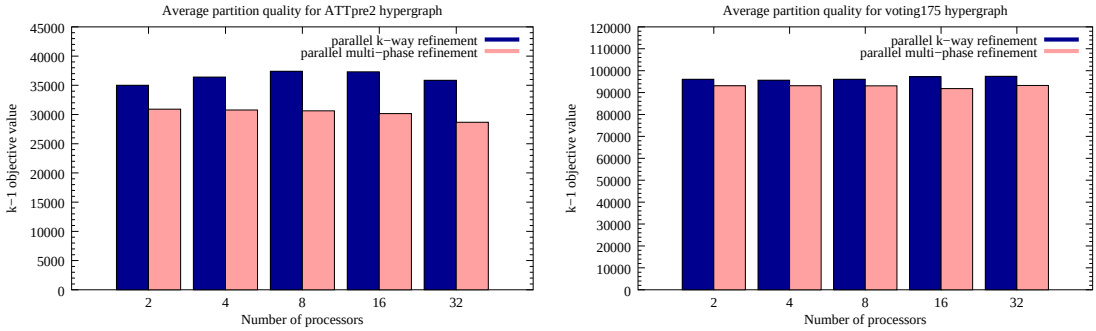

Figure 5.4. Par$k$way2.0: variation of partition quality with the number of processors used, with and without parallel multi-phase refinement.

Figure 5.4 shows the comparison of $k-1$ objective values of partitions produced by Par$k$way2.0 with orthodox parallel k-way refinement and Par$k$way2.0 with parallel multi-phase refinement on the ATTpre2 and voting175 hypergraphs. The benefit of using multi-phase refinement appears to be strongly dependent on the input hypergraph. For example, on the ATTpre2 hypergraph, a significant improvement in partition quality over the orthodox parallel refinement is observed. On the other hand, on the voting175 hypergraph, the improvement achieved is much less significant. We note that the introduction of parallel multi-phase refinement significantly increased the parallel runtime of Par$k$way2.0 (cf. Section 5.3.4).

5.3.4 Experimental Runtime Analysis

Here, we observe the average serial and parallel partitioning runtimes in order to analyse the parallel runtime performance of Par$k$way2.0. The average runtimes for every experimental configuration are shown (in tabular form) in Section B.2. In our speedup observations, base-case (single processor) comparison is provided by the tool PaToH, with settings as described in Section 5.3.3.

Figure 5.5 shows the speedups achieved by Par$k$way2.0, over the PaToH tool on the ibm17, ATTpre2, cage13 and voting175 hypergraphs.

We observe good speedups on the ATTpre2 and voting175 hypergraphs, when

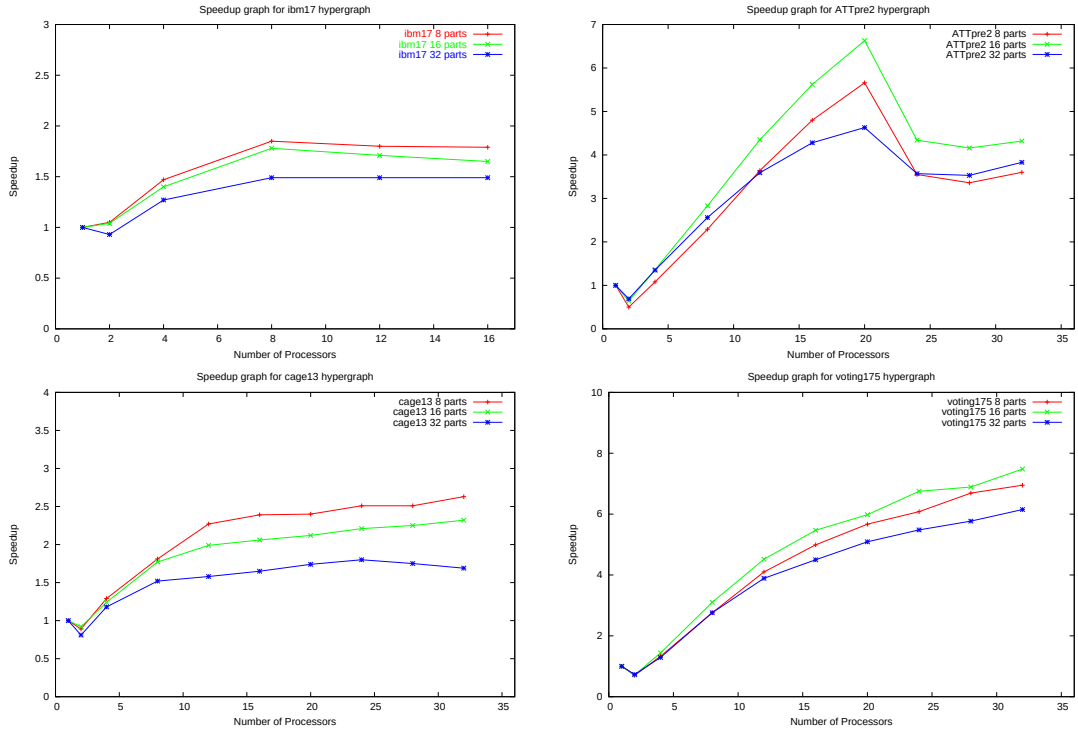

Figure 5.5. Par*k*way2.0: variation of speedup over the PaToH serial base-case with the number of processors used.

less than 24 processors are used. As the number of processors increases for a fixed problem size, the parallel overhead becomes a significant percentage of the runtime and a limiting speedup value over the best serial runtime is reached. In the case of the ATTpre2 hypergraph, the drop in speedup as the number of processors is increased beyond twenty may be attributed to the presence of a small number of hyperedges that are significantly larger than the average hyperedge length (Table A.2 in Appendix A). These potentially connect vertices across many processors and thus, as frontier hyperedges, have to be replicated on many processors during a multilevel step. On the other hand, hyperedges in the voting175 hypergraph have maximum length of seven and we observe that for this hypergraph, the Par*k*way2.0 parallel runtime scales better as the number of processors is increased.

For the cage13 hypergraph, the speedup levels off as the number of processors is increased beyond eight; however, this hypergraph, although sparse, is signif-

icantly more dense than both the ATTpre2 and the voting175 hypergraphs and the parallel overhead associated with data distribution during each multilevel step is correspondingly more significant. In the case of the much smaller ibm17 hypergraph, only a shallow speedup was achieved and a limiting speedup value was attained with only a few processors.

In addition to the above, we make two more important observations. Firstly, in experiments on large hypergraphs, it was observed that parallel multilevel graph partitioning is considerably faster than the parallel multilevel hypergraph partitioning algorithm. This is to be expected because the parallel multilevel hypergraph partitioning algorithm has a potentially significant computation and communication overhead associated with each multilevel step (locally assembling hyperedges incident to local vertices on each processor) which is not present in the serial multilevel hypergraph partitioning algorithm (cf. Section 4.5.1) or necessary for the parallel multilevel graph partitioning algorithms.

Secondly, we observe from Table B.9 in Section B.2 that with the introduction of parallel multi-phase refinement, Par*k*way2.0 exhibits similar parallel runtime scaling behaviour to Par*k*way2.0 with only orthodox parallel refinement. However, the *absolute* parallel runtimes of Par*k*way2.0 with parallel multi-phase refinement are significantly longer than those of Par*k*way2.0 with orthodox parallel refinement.

5.3.5 Empirical Evaluation of Predicted Scalability Behaviour

This section describes experiments carried out to investigate the scalability behaviour of the parallel multilevel hypergraph partitioning algorithm, as predicted by the theoretical performance model in Section 4.6.1.

Recall from Section 2.5.3 that existence of the isoefficiency function should enable a cost-optimal parallel algorithm to maintain a constant level of efficiency as the number of processors is increased, by appropriately increasing the problem

size. In order to be able to investigate this behaviour in the context of parallel hypergraph partitioning, we need to be able to increase the problem size (the number of computation steps taken by the serial multilevel hypergraph partitioning algorithm) by appropriately modifying the input hypergraph as a function of the number of processors used by the parallel multilevel hypergraph partitioning algorithm.

Even though we have derived the parallel multilevel hypergraph partitioning algorithm's isoefficiency function in terms of p, establishing the *exact* relationship between the number of computational steps taken by the serial multilevel partitioning algorithm (the problem size) and the number of processors p is difficult; we instead chose to approximate the problem size by the number of vertices in the hypergraph. In order for this approximation to be meaningful, we consider a family of hypergraphs that exhibit a very similar structure.

We used the family of hypergraphs constructed from transition matrices given by a semi-Markov model of a voting system [BDKW03]. The size of the transition matrix, and thus the size of the hypergraph, depends on the number of voters in the model. As can be seen from Table 5.3.5, the voting hypergraphs used in the experiments exhibit a very similar structure and range in size from 250 000 to 11 000 000 vertices. The voting hypergraphs were also chosen because they possess small maximum and average vertex degrees and hyperedge lengths, which is consistent with the analytical performance model used to derive the isoefficiency function in Section 4.6.

From Equation 4.14, the isoefficiency function of the parallel hypergraph partitioning algorithm for a constant partition size is $O(p^2 \log p)$; we assume a small constant term in the isoefficiency function. As the number of processors increases, we increase problem size by computing partitions for succesive larger voting hypergraphs, according to Table 5.3.

In these experiments, Par$k$way2.0 was run with settings as described in Section 5.3.2, except that during the uncoarsening phase, the number of passes of the parallel k-way refinement algorithm was restricted to at most four.

	Hyperedge lengths				Vertex weights			
Name	**avg**	**90%**	**95%**	**max**	**avg**	**90%**	**95%**	**max**
voting100	5.57	7	7	7	5.57	7	7	7
voting125	5.62	7	7	7	5.62	7	7	7
voting150	5.82	7	7	7	5.82	7	7	7
voting175	5.84	7	7	7	5.84	7	7	7
voting250	6.32	7	7	7	6.32	7	7	7
voting300	6.35	7	7	7	6.35	7	7	7

Table 5.2. Average, 90th, 95th and 100th percentiles of hyperedge length and vertex weight of the `voting` hypergraphs.

p	**Hypergraph**	**Number of vertices**
2	voting100	249 760
3	voting125	541 280
4	voting175	1 140 050
8	voting250	5 218 300
11	voting300	10 991 400

Table 5.3. Par*k*way2.0 isoefficiency experimental configuration: the hypergraphs and the numbers of processors used.

We used the serial multilevel tool PaToH, with settings as described in Section 5.3.3. Figure 5.6 shows PaToH runtimes across the voting100, voting125, voting150 and voting175 hypergraphs. We note that the graph increases superlinearly with increasing problem size; this may be partly due to effects of hierarchical memory.

In order to compute parallel efficiencies on hypergraphs that could not be partitioned on a single processor, we constructed approximations for the serial runtimes by fitting a linear regression model to a log-log plot of the existing serial runtimes. Specifically, we observed an approximate linear relationship between the natural logarithm of the number of vertices in a voting hypergraph and the natural logarithm of the PaToH serial runtime (see Figure 5.7). This yields expected serial PaToH runtimes for the two larger hypergraphs, namely voting250 and voting300. These, together with the *actual* PaToH runtimes, were used to compute processor efficiencies for parallel hypergraph partitioning configurations described in Table 5.3. Full results from the experiments are presented in Sec-

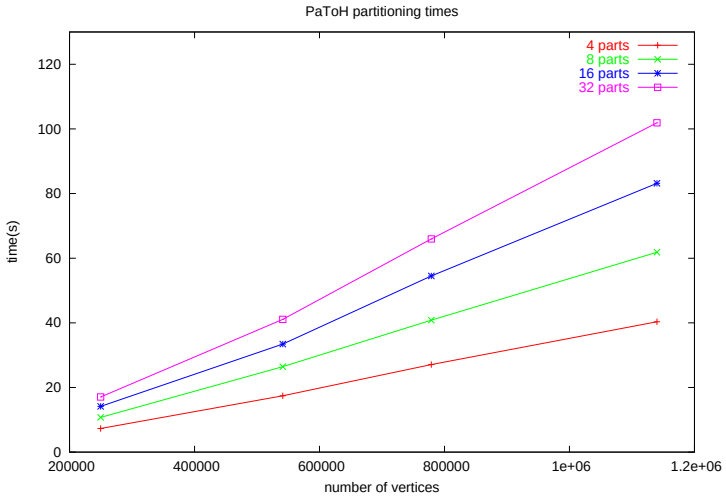

Figure 5.6. `PaToH`: variation of runtimes on the voting hypergraphs.

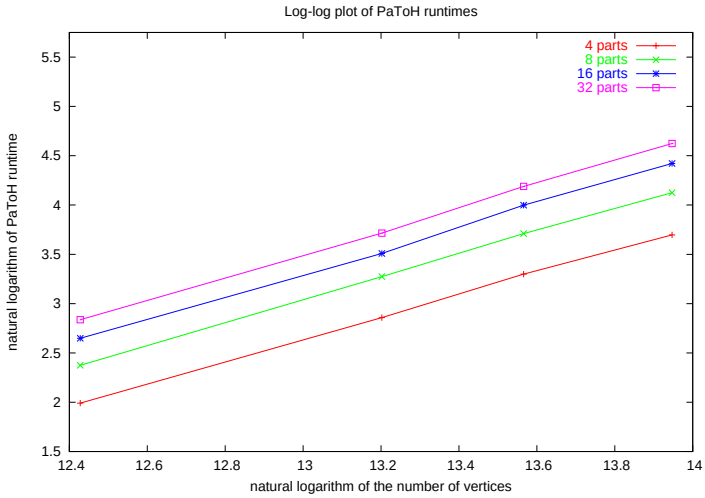

Figure 5.7. `PaToH`: the log-log plot of the variation of runtimes on the voting hypergraphs.

tion B.2.

Figure 5.8 shows the relationship between `Par`k`way2.0` processor efficiency and the number of processors used, when the number of parts in the partition sought is fixed, with values $k = 4$, $k = 8$, $k = 16$ and $k = 32$. The processor efficiency is expected to remain relatively constant for different processor/problem size configurations. For the 4-way and 8-way partitions, we observe a relatively steady but shallow descrease in efficiency as the number of processors used increases. The parallel computation of the 8-way partition exhibits a significantly larger

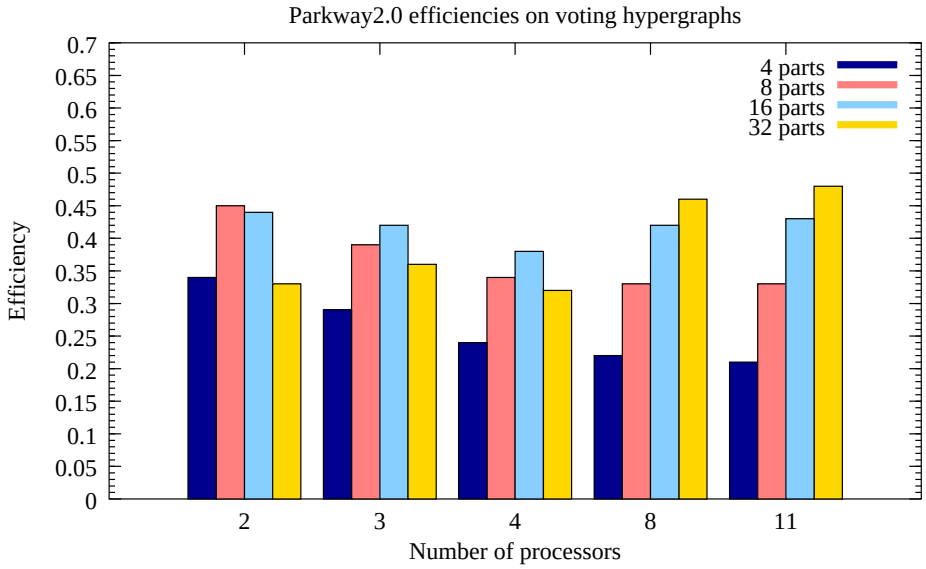

Figure 5.8. Par*k*way2.0: variation of processor efficiency with the number of processors used on the voting hypergraphs.

processor efficiency than the parallel computation of the 4-way partition. In the case of 16-way partitions, the observed processor efficiency remains relatively constant. Finally, for the 32-way partitions, we observe an increase in efficiency, as the number of processors increases.

Although some of our observations are based on extrapolated serial runtimes, we note an increasing efficiency of our parallel multilevel hypergraph partitioning algorithm with increasing partition size. This may be partly because serial partitioning is done by multilevel recursive bisection, while in parallel, partitions are computed directly; a runtime advantage in favour of the direct partitioning method as the size of the partition increases is usually observed [KK98b].

Finally, from the observed processor efficiencies, we conclude that the experiments do not provide any evidence to reject our hypothesis that the isoefficiency function of our parallel multilevel hypergraph partitioning algorithm is $W(p) = O(p^2 \log p)$.

Chapter 6

Application to Parallel PageRank Computation

This chapter presents the application of parallel hypergraph partitioning to accelerate parallel PageRank computation. PageRank is extensively used by internet search engines as a quantitative measure of the relative importance of web pages. At the core of the PageRank computation is an iterative eigensystem solver, whose kernel operation is sparse matrix–vector multiplication [PBMW99, HK03, GZB04]. We apply state-of-the-art hypergraph-based models for sparse matrix decomposition to make this kernel operation more efficient in a parallel context. We apply the parallel hypergraph partitioning tool Par$k$way2.1 to parallel PageRank computation using a number of web matrices in the public domain. Par$k$way2.1 is an optimised version of the Par$k$way2.0 parallel hypergraph partitioning tool described in Section 5.2.

The remainder of this chapter is organised as follows. Section 6.1 provides a brief introduction to the random surfer model giving rise to the PageRank metric. Section 6.2 introduces parallel sparse matrix–vector multiplication, while Section 6.3 discusses hypergraph-based models for sparse matrix decomposition in the context of parallel sparse matrix–vector multiplication. Finally, Section 6.4 describes the application of Par$k$way2.1 to accelerate parallel iterative PageRank computation on a number of web matrices from the public domain.

6.1 The PageRank Algorithm

6.1.1 Introduction

The PageRank metric is a widely-used hyperlink-based estimate of the relative importance of web pages. Its computation was originally outlined by Page and Brin [PBMW99]; later Kamvar et al. [KHMG03b] presented a more rigorous formulation that differs from the original. However, apart from the respective treatment of web pages with no outgoing links, the difference is largely superficial [de 04]. Here, we consider the Kamvar et al. formulation.

Two intuitive explanations are offered for PageRank [KHMG03b]. The first presents PageRank as an analogue of citation theory: that is, an out-link from a web page w to a web page w' is an indication that w' may be "important" to the author of w. Many such links into w', especially from pages that are themselves "important", should raise the importance of w' relative to other web pages. More specifically, the importance that is propagated from w to w' should be proportional to the importance of w and inversely proportional to the number of out-links from w. This account of PageRank is still incomplete as it does not take into account any form of user *personalisation*, or how to deal with pages with no outgoing links.

The second conceptual model of PageRank is called the *random surfer* model. Consider a surfer who starts at a web page and picks one of the links on that page at random. On loading the next page, this process is repeated. If the surfer encounters a page with no outgoing links, then (s)he chooses to visit a random page. During normal browsing, the user may also decide, with a fixed probability, not to choose a link from the current page, but instead to jump at random to another page. In the latter case, to support both unbiased and personalised surfing behaviour, the model allows for the specification of a probability distribution of target pages.

The PageRank of a page is considered to be the (steady-state) probability that the surfer is visiting a particular page after a large number of click-throughs.

Calculating the steady-state probability vector corresponds to finding a maximal eigenvector of the modified web graph transition matrix. As shown in Section 6.1.3 below, this can be done via an iterative numerical method based on sparse matrix–vector multiply operations.

6.1.2 Random Surfer Model

In the random surfer model, the web is represented by a directed graph $G(V, \mathcal{E})$, with web pages forming the set of vertices, V, and the links between web pages forming the set of directed edges, $\mathcal{E}$. If a link exists from page u_i to page u_j then $(u_i \rightarrow u_j) \in \mathcal{E}$.

To represent the following of hyperlinks, we construct a transition matrix $\mathbf{P}$ from the web graph, setting:

$$p_{ij} = \begin{cases} \frac{1}{\deg(u_i)} & : \text{if } (u_i \rightarrow u_j) \in \mathcal{E} \\ 0 & : \text{otherwise} \end{cases}$$

where $\deg(u)$ is the out-degree of vertex u, i.e. the number of outbound links from page u. From this definition, it can be seen that if a page has no out-links, then this corresponds to a zero row in the matrix $\mathbf{P}$.

To represent the surfer's jumping from pages with no outgoing links, we construct a second matrix $\mathbf{D} = \mathbf{d}\mathbf{p}^T$, where $\mathbf{d}$ and $\mathbf{p}$ are both column vectors and:

$$d_i = \begin{cases} 1 & : \text{if } \deg(u_i) = 0 \\ 0 & : \text{otherwise} \end{cases}$$

and $\mathbf{p}$ is the personalisation vector representing the probability distribution of destination pages when a random jump is made. Typically, this distribution is taken to be uniform, i.e. $p_i = 1/n$ for an n-page graph ($1 \leq i \leq n$). But it need not be – many distinct personalisation vectors may be used to represent different classes of user with different web browsing patterns. This flexibility comes at a cost: for each distinct personalisation vector, another PageRank calculation is required.

Putting together the surfer's following of hyperlinks and his/her random jumping from *cul de sac* pages yields the matrix $\mathbf{P}' = \mathbf{P} + \mathbf{D}$, such that $\mathbf{P}'$ is a transition matrix of a discrete-time Markov chain (DTMC).

To represent the surfer's decision not to follow any of the current page links, but to instead jump to a random web page, we construct a *teleportation* matrix $\mathbf{E}$, where $e_{ij} = p_j$ for all i, i.e. this random jump is also dictated by the personalisation vector.

Incorporating this matrix into the model gives:

$$\mathbf{A} = c\mathbf{P}' + (1 - c)\mathbf{E} \tag{6.1}$$

where $0 < c < 1$, and c represents the probability that the user chooses to follow one of the links on the current page, i.e. there is a probability of $(1 - c)$ that the surfer randomly jumps to another page instead of following links on the current page.

This definition of $\mathbf{A}$ avoids two potential problems. The first is that $\mathbf{P}'$, although a valid DTMC transition matrix, is not necessarily irreducible (i.e. it might have more than one strongly connected subset of states) and aperiodic. Taken together, these are a sufficient condition for the existence of a unique steady-state distribution. Now, provided $p_i > 0$ for all $1 \leq i \leq n$, irreducibility and aperiodicity are trivially guaranteed.

The second problem relates to the rate of convergence of power method iterations used to compute the steady-state distribution. This rate depends on the reciprocal of the modulus of the subdominant eigenvalue (λ_2) [Ste94]. For a general $\mathbf{P}'$, $|\lambda_2|$ may be very close to 1, resulting in a very poor rate of convergence. However, it has been shown in [HK03] that in the case of matrix $\mathbf{A}$, $|\lambda_2| \leq c$, thus guaranteeing a good rate of convergence for the widely taken value of $c = 0.85$.

The unique PageRank vector, $\boldsymbol{\pi}$, can now be defined to be the steady-state vector or the maximal eigenvector that satisfies:

$$\boldsymbol{\pi}\mathbf{A} = \boldsymbol{\pi} \tag{6.2}$$

6.1.3 Power Method Solution

Having constructed $\mathbf{A}$, the naïve attempt to finding the PageRank vector of Equation 6.2 uses a direct power method approach:

$$\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)}\mathbf{A} \tag{6.3}$$

where $\mathbf{x}^{(i)}$ is the i^{th} iterate towards the PageRank vector, $\boldsymbol{\pi}$. However, given that $\mathbf{A}$ is a (completely) dense matrix and the large (and increasing) size of the web, it is clear that this is not a practical approach. Fortunately, the PageRank algorithm, as cited in [KHMG03a] for instance, reduces Equation 6.3 to a series of sparse vector–matrix operations on the original $\mathbf{P}$ matrix.

In particular, transforming Equation 6.3 gives:

$$\begin{aligned}
\mathbf{x}^{(i+1)} &= \mathbf{x}^{(i)}\mathbf{A} && (6.4)\\
&= c\mathbf{x}^{(i)}\mathbf{P}' + (1-c)\mathbf{x}^{(i)}\mathbf{E} && (6.5)\\
&= c\mathbf{x}^{(i)}\mathbf{P} + c\mathbf{x}^{(i)}\mathbf{D} + (1-c)\mathbf{x}^{(i)}(\mathbf{1}\mathbf{p}^T) && (6.6)
\end{aligned}$$

Now $\mathbf{x}^{(i)}\mathbf{D} = (\|\mathbf{x}^{(i)}\|_1 - \|\mathbf{x}^{(i)}\mathbf{P}\|_1)\mathbf{p}^T$, where $\|\mathbf{a}\|_1 = \sum_j |a_j|$ is the 1-norm of $\mathbf{a}$ and further $\|\mathbf{a}\|_1 = \mathbf{1}^T\mathbf{a}$ if $a_j \geq 0$ for all j. It can be shown inductively that $\|\mathbf{x}^{(i)}\|_1 = 1$ for all i (if $\|\mathbf{x}^{(0)}\|_1 = 1$), so:

$$\begin{aligned}
\mathbf{x}^{(i+1)} &= c\mathbf{x}^{(i)}\mathbf{P} + c(1 - \|\mathbf{x}^{(i)}\mathbf{P}\|_1)\mathbf{p}^T + (1-c)(\mathbf{x}^{(i)}\mathbf{1})\mathbf{p}^T && (6.7)\\
&= c\mathbf{x}^{(i)}\mathbf{P} + (1 - c\|\mathbf{x}^{(i)}\mathbf{P}\|_1)\mathbf{p}^T && (6.8)
\end{aligned}$$

This leads to the algorithm shown in Algorithm 4. In practice, the power method iterations are computed in parallel. This is because parallel computation in general yields faster per-iteration runtimes than serial computation and more importantly, because the sparse matrix $\mathbf{P}$ is too large to be stored within the memory of a single processor (e.g. in 2003, the World Wide Web contained at least 8.9 billion webpages [WD04]).

Algorithm 4 Iterative computation of PageRank via the Power Method

Require: $0 < \epsilon \ll 1$

1: $i := 0$
2: $\mathbf{x}^{(i)} := \mathbf{p}^T$
3: **repeat**
4: $\quad \mathbf{y} := c\mathbf{x}^{(i)}\mathbf{P}$
5: $\quad \omega := 1 - \|\mathbf{y}\|_1$
6: $\quad \mathbf{x}^{(i+1)} := \mathbf{y} + \omega\mathbf{p}^T$
7: $\quad i := i + 1$
8: **until** $\|\mathbf{x}^{(i)} - \mathbf{x}^{(i-1)}\|_1 < \epsilon$

6.2 Parallel Sparse Matrix–Vector Multiplication

We consider parallel sparse matrix–vector multiplication in the context of a distributed memory parallel architecture. Let $\mathbf{Ax} = \mathbf{b}$ be the sparse matrix–vector product to be computed in parallel on p distributed processors that are connected by a network. A parallel algorithm for sparse matrix–vector multiplication with an arbitrary non-overlapping distribution of the matrix and the vectors across the processors has the following general form [VB05]:

1. Each processor sends its components x_j to those processors that possess a non-zero a_{ij} in column j.

2. Each processor computes the products $a_{ij}x_j$ for its non-zeros a_{ij} and adds the results for the same row index i. This yields a set of contributions b_{is}, where s is the processor identifier $0 \leq s < p$.

3. Each processor sends its non-zero contributions b_{is} to the processor that is assigned vector element b_i.

4. Each processor adds the contributions received for its components b_i, giving $b_i = \sum_{s=0}^{p-1} b_{is}$.

Efficient parallel sparse matrix–vector multiplication requires intelligent *a priori* partitioning of the sparse matrix non-zeros across the processors to ensure that

interprocessor communication during stages 1 and 3 is minimised and computational load balance is achieved across the processors (cf. Section 1.1.3).

We note that the computational requirement of step 2 dominates that of step 4; henceforth we assume that the computational load of the entire parallel sparse matrix–vector multiplication algorithm can be represented by the computational load induced during step 2 only.

The decomposition of the sparse matrix to the p processors may be one-dimensional or two-dimensional (Cartesian). In a one-dimensional decomposition, entire rows (or columns) of the matrix $\mathbf{A}$ are allocated to processors. In literature, this is called a row (or column) block distribution [GGKK03]. Note that a one-dimensional row-wise decomposition has the effect of making the communication step 3 in the parallel sparse matrix–vector multiplication pipeline redundant; in the one-dimensional column-wise decomposition, step 1 is redundant.

A two-dimensional sparse matrix decomposition involves the allocation of individual sparse matrix non-zeros to processors. Note that a one-dimensional sparse matrix decomposition is a special case of a two-dimensional sparse matrix decomposition.

6.3 Hypergraph Models for Sparse Matrix Decomposition

Recently, a number of hypergraph-based models for parallel sparse matrix–vector multiplication that correctly model total communication volume (the aggregate size of all messages) and per-processor computational load have been proposed [cA99, cA01a, UA04, VB05]. These have addressed the shortcomings implicit in traditional graph models, which can, in general, only approximate the total communication volume [Hen98]. In [cA99], a hypergraph-based model for one-dimensional decomposition of the sparse matrix is proposed. The hypergraph-based models in [cA01a, UA04, VB05] are two-dimensional.

6.3.1 One-Dimensional Sparse Matrix Decomposition

In our description, all non-zeros in a row of the matrix are allocated to the same processor. A similar column-wise model follows from considering the allocation of all non-zeros in a column of the matrix to the same processor. These one-dimensional hypergraph-based models were first proposed in [cA99].

The hypergraph model $H(V, \mathcal{E})$ for the row-wise decomposition of a sparse $n \times n$ matrix $\mathbf{A}$ is constructed by interpreting $\mathbf{A}$ as the incidence matrix of the hypergraph $H(V, \mathcal{E})$ (cf. Section 2.1). The rows of the matrix $\mathbf{A}$ form the set of vertices, V, in the hypergraph $H(V, \mathcal{E})$ and the columns form the set of hyperedges, $\mathcal{E}$. That is, if $a_{ij} \neq 0$, then hyperedge $e_j \in \mathcal{E}$, defined by column j of the matrix $\mathbf{A}$, contains vertex $v_i \in V$. The weight of vertex $v_i \in V$ is given by the number of non-zero elements in row i of the matrix $\mathbf{A}$, representing the computational load induced by assigning row i to a processor. The weights of each hyperedge are set to unity.

The allocation of the rows of the matrix $\mathbf{A}$ to p processors for parallel sparse matrix–vector multiplication corresponds to a p-way partition Π of the hypergraph $H(V, \mathcal{E})$. The computational load on each processor i is given by the number of scalar multiplications performed on that processor during stage 2 of the general parallel sparse matrix–vector multiplication pipeline. This quantity is given by the number of non-zeros of the matrix $\mathbf{A}$ allocated to processor i, which is in turn given by the weight of part P_i.

The vector elements x_i and b_i are allocated to the processor that is allocated row i of the matrix $\mathbf{A}$. There remains one further condition that the hypergraph model must satisfy to ensure that the $k-1$ metric objective function on partition Π exactly represents the total communication volume incurred during a single parallel sparse matrix–vector multiplication (in this case stage 1 in the pipeline only). It is required that, for all $1 \leq i \leq n$, $v_i \in e_i$ holds. If this is not the case for some $1 \leq i' \leq n$, then we add $v_{i'}$ to hyperedge $e_{i'}$ in the hypergraph model. The weight of $v_{i'}$ is not modified.

Upon construction, the hypergraph model $H(V, \mathcal{E})$ is partitioned to minimise the $k-1$ objective function subject to a partitioning balance constraint; this directly corresponds to minimising communication volume during parallel sparse matrix–vector multiplication while maintaining a computational load balance.

6.3.2 Two-Dimensional Sparse Matrix Decomposition

The two-dimensional sparse matrix decomposition takes a more general approach, no longer imposing the restriction of allocating entire rows (or columns) of the sparse matrix $\mathbf{A}$ to the same processor (as the one-dimensional decomposition does). Instead, an arbitrary distribution of matrix non-zeros to processors is considered. Compared to a one-dimensional row-wise decomposition, this may introduce additional communication operations during stage 3 in the general parallel sparse matrix–vector multiplication pipeline. We describe the fine-grained hypergraph model introduced in [cA01a].

The hypergraph model $H(V, \mathcal{E})$ is constructed as follows. Each $a_{ij} \neq 0$ is modelled by a vertex $v \in V$, so that a p-way partition Π of the hypergraph $H(V, \mathcal{E})$ will correspond to an assignment of matrix non-zeros across p processors.

In order to define the hyperedges in the hypergraph model, we consider the cause of communication between processors in stages 1 and 3 of the parallel sparse matrix–vector multiplication pipeline. In stage 1, the processor with non-zero a_{ij} requires vector element x_j for computation during stage 2. This results in a communication of x_j to the processor assigned a_{ij} if x_j is assigned to a different processor than a_{ij}. The dependence between non-zeros in column j of matrix $\mathbf{A}$ and vector element x_j can be modelled by a hyperedge, whose constituent vertices are the non-zeros of column j of the matrix $\mathbf{A}$. So that the communication volume associated with communicating vector element x_j is given by $\lambda_j - 1$, where λ_j denotes the number of parts spanned by the column j hyperedge, it is required that the column j hyperedge contains the vertex corresponding to non-zero a_{jj}. If a_{jj} is zero in the matrix $\mathbf{A}$, a "dummy" vertex with zero weight corresponding to a_{jj} is added. The fact that this vertex has weight zero means that its allocation

to a processor will have no bearing on the processor's computational load, while the exact communication volume during stage 1 is modelled correctly.

During stage 3, the processor assigned vector element b_i requires the value of the inner product of row i of the matrix $\mathbf{A}$ with the vector $\mathbf{x}$. A communication between processors is induced if matrix non-zero a_{ij} is assigned to a different processor from vector entry b_i. The dependence between non-zeros in row i of matrix $\mathbf{A}$ and vector element b_i can be modelled by a hyperedge, whose constituent vertices are the non-zeros of row i of the matrix $\mathbf{A}$. This is analogous to modelling the communication of stage 1 with column hyperedges and likewise, dummy vertices corresponding to a_{ii} are added to row hyperedge i if the value of a_{ii} in matrix $\mathbf{A}$ is zero.

Upon construction, the hypergraph model $H(V, \mathcal{E})$ is then partitioned into p parts, minimising the $k-1$ objective subject to a partition balance constraint. As for the one-dimensional hypergraph model in Section 6.3.1, this corresponds to minimising communication volume during parallel sparse matrix–vector multiplication while maintaining the computational load balance.

Thus far, the vector entries have not been explicitly allocated to processors. We assume that the i^{th} components of vectors $\mathbf{x}$ and $\mathbf{b}$ are allocated to the same processor. This is also called *symmetric* vector partitioning in literature and is usually necessary when, during an iterative computation, the output vector $\mathbf{b}$ is used as the input vector in the following iteration. Non-symmetric vector partitioning (with an arbitrary distribution of vector elements to processors) is discussed in more detail in [UA04, BM05, VB05].

The overall communication volume during the parallel sparse matrix–vector multiplication will be correctly modelled by the two-dimensional hypergraph model, provided that the vector elements are allocated to processors in the following fashion. Consider the vector element with index i:

1. If both the row i hyperedge and the column i hyperedge are not cut, then assign vector elements x_i and b_i to the processor assigned vertices from row i and column i hyperedges.

2. If the row i hyperedge is cut and the column i hyperedge is not cut, then assign vector elements x_i and b_i to the processor assigned vertices from column i hyperedge.

3. If the row i hyperedge is not cut and the column i hyperedge is cut, then assign vector elements x_i and b_i to the processor assigned vertices from row i hyperedge.

4. If both the row i hyperedge and the column i hyperedge are cut, then let R_i denote the set of processors that contain row i hyperedge elements and let C_i denote the set of processors that contain column i hyperedge elements. Since either $a_{ii} \neq 0$ or there exists a "dummy" vertex in the row i and column i hyperedges corresponding to a_{ii}, the set $T_i = R_i \cap C_i$ is non-empty and vector elements x_i and b_i may be assigned to any of the processors in T_i.

With the additional freedom in the assignment of vector elements to processors given by case 4 above, it may be possible to further decrease the *maximum* number of messages as well as the volume of communication sent or received by an individual processor, while keeping the overall communication volume constant. Such vector partitioning is discussed in detail in [BM05].

6.4 Case Study: Parallel PageRank Computation

This section describes a case-study application of the hypergraph models for sparse matrix decomposition to parallel iterative PageRank computation. We consider a parallel formulation of the iterative PageRank algorithm described in Section 6.1.3, which has the parallel computation of the sparse matrix–vector product $\mathbf{x}^{(i)}\mathbf{P}$ as the kernel operation during each iteration i.

When distributing this algorithm, it is important to distribute the sparse matrix–vector calculation of $\mathbf{x}^{(i)}\mathbf{P}$ in such a way so as to balance computational load

as evenly as possible across the processors and minimise communication overhead between processors. Here, we investigate the use of hypergraph models for sparse matrix decomposition and use the parallel hypergraph partitioning tool `Par`*k*`way2.1` to compute partitions of the hypergraph models in parallel.

Although our discussion is in the context of power method solution, there is nothing to prevent the application of the hypergraph partitioning-based techniques to other iterative linear system solvers with a sparse matrix–vector multiplication kernel, such as the Krylov subspace methods proposed in [GZB04]. Furthermore, our approach does not preclude the application of power method acceleration techniques, for example those proposed in [KHMG03b].

6.4.1 Experimental Setup

Sparse Matrix Decomposition Strategies

Four sparse matrix decomposition strategies were evaluated in the case-study. The two hypergraph decomposition methods of Section 6.3 (here referred to as one-dimensional and two-dimensional models) were tested against two purely load balancing methods.

In the cyclic row-striping matrix decomposition, the non-zeros of the matrix $\mathbf{A}$ in the row with index i are assigned to the processor $i \bmod p$. Vector elements x_i and b_i are also allocated to processor $i \bmod p$. This ensures that each processor is allocated the same number (± 1) of rows of matrix $\mathbf{A}$ and vector elements of $\mathbf{x}$ and $\mathbf{b}$. However, this scheme does not take into account the distribution of the non-zeros within the rows.

The load balancing scheme, presented in [GZB04], and hereafter referred to as the GleZhu scheme, attempts to balance the number of non-zeros across the processors, while assigning consecutive rows of the matrix $\mathbf{A}$ to each processor. A threshold value $\tau_p = (w_n n + w_\eta \eta)/p$ is computed, where n is the number of rows and η the number of non-zeros in the matrix. The parameters w_n and w_η are both set to unity in [GZB04] and in the case-study experiments. Starting

WebGraph	rows	columns	non-zeros
Stanford	281 903	281 903	2 594 228
Stanford_Berkeley	683 446	683 446	8 262 087
india-2004	1 382 908	1 382 908	16 917 053

Table 6.1. The main characteristics of the web matrices.

with row index zero and processor p_0, the load-balancing algorithm then assigns consecutive rows of matrix $\mathbf{A}$ and consecutive elements of vectors $\mathbf{x}$ and $\mathbf{b}$ to each processor p_i, maintaining the value of $\tau_i = w_n n_i + w_\eta \eta_i$, where n_i is the number of rows and η_i the number of non-zeros assigned thus far to processor p_i. When τ_i exceeds τ_p, the algorithm begins to assign subsequent rows to processor p_{i+1}.

Case Study Web Matrices

Our experiments were performed on three publicly available web graphs. Each web graph was generated from a crawl of a particular domain or combination of domains and is represented by a sparse matrix $\mathbf{A}$ with non-zero a_{ij} whenever there exists a link from page i to page j. The Stanford and Stanford_Berkeley web graphs were obtained from the University of Florida Sparse Matrix Collection [Dav05] and represent lexically ordered crawls of the Stanford and combined Stanford/Berkeley domains respectively. The india-2004 web graph represents a breadth-first crawl of the .in domain, conducted in 2004, obtained from the UbiCrawler public data set [BCSV04]. The main characteristics of the web matrices are given in Table 6.1.

The hypergraph partitioning-based models for sparse matrix decomposition give rise to the following hypergraphs, whose characteristics are shown in more detail in Appendix A. Stanford-1D and Stanford-2D are the 1-dimensional and 2-dimensional hypergraph models for sparse matrix decomposition of the Stanford web matrix. Similarly, the Stfd_Bkly-1D and Stfd_Bkly-2D hypergraphs correspond to the Stanford_Berkeley web matrix, while the india-2004-1D and india-

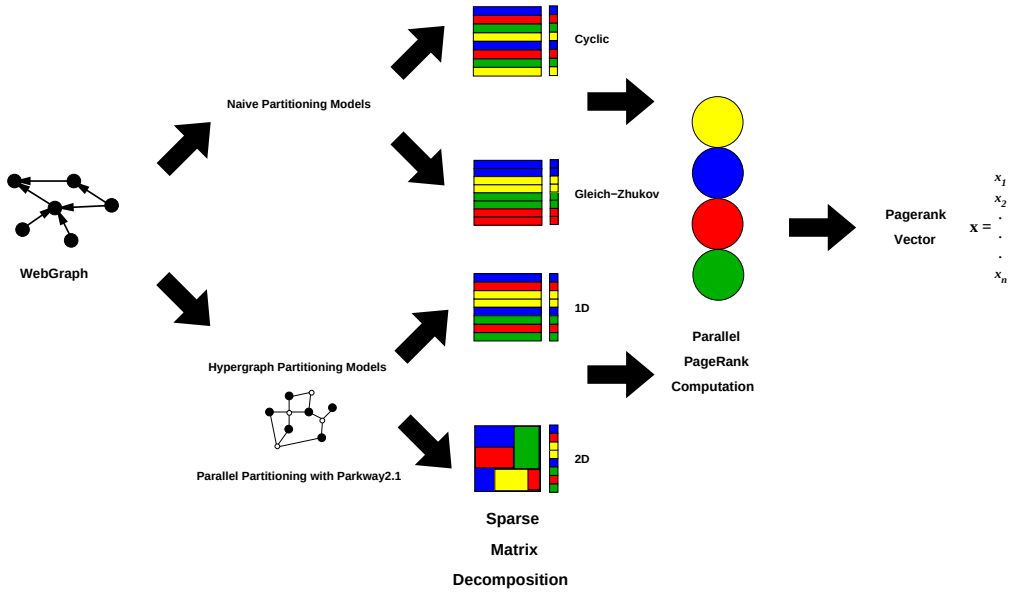

Figure 6.1. Parallel PageRank computation pipeline.

2004-2D hypergraphs correspond to the india-2004 web matrix.

Experimental Platform and Configuration

The parallel PageRank computation pipeline is shown in Figure 6.1. Taking the web-graph matrix $\mathbf{A}$ as input, a decomposition of this matrix across p processors is performed using either one of the hypergraph partitioning-based models (i.e. 1D or 2D) or one of the purely load balancing row-wise decomposition methods (i.e. cyclic or GleZhu). The hypergraph partitioning-based schemes compute a p-way partition of the hypergraph representation of the sparse web matrix using the parallel hypergraph partitioning tool `Par`k`way2.1`, which is an optimised version of the tool described in Section 5.2.

`Par`k`way2.1` was configured for the experiments as follows. A 5% partition balance constraint was imposed, equivalent to setting $\epsilon = 0.05$ in Equation 2.3. The parallel coarsening and uncoarsening settings were as used in experiments in Chapter 5 and are described in Section 5.3.2. No multi-phase refinement was used and the number of passes of the parallel k-way refinement algorithm at each multilevel step was restricted to at most four.

The computed hypergraph partition was then used to allocate the rows (in the case of 1D partitioning) or the non-zeros (in the case of 2D partitioning) of the web matrix to the processors. Finally, the algorithm described in Section 6.1.3 was used to compute the PageRank vector for the matrix, with all matrix–vector and vector operations performed in parallel. The criterion of convergence for the PageRank calculation was taken to be 10^{-8} and convergence was computed using the L_1 norm.

The architecture used in all the experiments consisted of a Beowulf Linux Cluster with 8 dual processor nodes. Each node has two Intel Pentium 4 3.0GHz processors and 2GB RAM. The nodes are connected by a gigabit Ethernet network. The algorithms were implemented in C++ using the Message Passing Interface (MPI) standard [SOHL$^+$98].

6.4.2 Experimental Results

For each matrix decomposition method, we observed the following measures of communication cost during each parallel PageRank iteration:

- The total communication volume (the total volume of all messages sent)
- The number of messages sent
- The maximum total communication volume of messages sent and received by a single processor

The purely load balancing matrix decomposition approaches did not attempt to minimise the metrics above. The one- and two-dimensional hypergraph-based models aimed to minimise the overall communication volume. In a row-wise decomposition, the number of messages sent during parallel sparse matrix–vector multiplication is at most $p(p-1)$. In a two-dimensional decomposition, the number of messages is at most $2p(p-1)$.

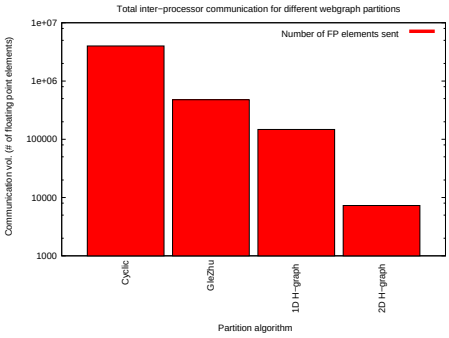

Figure 6.2. Total per-iteration communication volume for 16-processor Stanford_Berkeley PageRank computation (note log scale on communication volume axis).

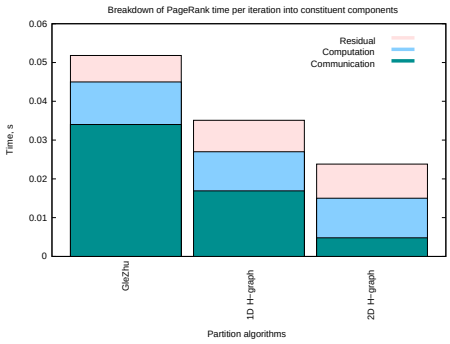

Figure 6.3. Per-iteration execution time for 16-processor Stanford_Berkeley PageRank computation.

The complete set of experimental results is presented in Section B.3. We note that due to numerical inaccurracies (truncation and roundoff), the number of iterations varied slightly across the different methods.

We observe that the hypergraph partitioning-based sparse matrix decomposition schemes attracted a significantly lower overall communication overhead than the two purely load-balancing schemes. The two-dimensional hypergraph model was the most effective at reducing overall communication volume, although this did not always translate into a lower PageRank per-iteration time, on account of the higher number of messages sent and the relatively high message start-up cost on the gigabit PC cluster.

Figure 6.2 displays the total per-iteration communication volume for each partitioning algorithm (note the log scale on the y axis). It shows that the GleZhu technique had a lower communication overhead than the naïve cyclic partitioning, as might have been expected. It also shows that, when compared to the GleZhu method, hypergraph partitioning reduced communication volume by an order of magnitude for one-dimensional hypergraph partitioning and by 2 orders of magnitude for two-dimensional hypergraph partitioning.

Figure 6.3 shows the overall PageRank iteration time for GleZhu, one-dimensional and two-dimensional hypergraph-partitioning based decompositions of the Stan-

ford_Berkeley web matrix on the 16-processor cluster. The *computation* label refers to the time taken to compute a single $\mathbf{Ax} = \mathbf{b}$ iteration. The *communication* label represents the time taken in communication when performing a single $\mathbf{Ax} = \mathbf{b}$ iteration. The *residual* label corresponds to the time taken by the remaining computation and communication operations during an iteration of the parallel PageRank computation. The results for the cyclic technique are not shown as they are orders of magnitude larger and the main interest here is in comparing the GleZhu sparse matrix decomposition method (as the best currently used alternative) with the hypergraph-based methods. It is clear that the overall PageRank iteration time was dictated by the communication overhead incurred in performing the distributed $\mathbf{Ax} = \mathbf{b}$ calculation. As might have been expected, the computation element and the residual of the PageRank computation (those calculations not involving the distributed matrix–vector multiplication) of the algorithm contributed an (approximately) fixed cost to the overall iteration time. We observe that one-dimensional and two-dimensional hypergraph partitioning successfully reduced the communication overhead by factors of 2 and 6 respectively. This reduction resulted in a decrease in the overall PageRank per-iteration time by 50% in the 2-dimensional case, when compared to the GleZhu method.

We note that, contrary to intuition, in some cases computation times did vary significantly, depending on decomposition method used. It is conjectured that this occurred because no attempt was made to optimise the caching behaviour of the parallel PageRank solver. As a consequence, the GleZhu method (which assigns consecutive vector elements to processors) had a good cache hit rate; conversely the cyclic method (which assigns vector elements on a striped basis) suffered a poor cache hit rate.

Finally, we observe that the partitioning overhead for hypergraphs arising from the one-dimensional model was significantly lower than the partitioning overhead for hypergraphs arising from the two-dimensional model. There are a number of reasons for this. Firstly, the more sophisticated nature of the two-dimensional hypergraph model yields a significantly larger hypergraph for partitioning than

the one-dimensional hypergraph model. Secondly, our parallel multilevel hypergraph partitioning tool `Par`*k*`way2.1` has not yet been optimised for partitioning hypergraphs arising from two-dimensional models of parallel sparse matrix–vector multiplication; such hypergraphs possess a particular structure, where each vertex is incident on exactly two hyperedges.

Chapter 7

Conclusion

7.1 Summary of Achievements

The main contributions of this thesis are the first parallel algorithms for multilevel hypergraph partitioning. These significantly increase capacity, enabling partitioning of very large hypergraphs, and achieve good speedups over existing state-of-the-art serial partitioners, without sacrificing partition quality.

Our explorations into parallel hypergraph partitioning began in Section 4.3 with an application-specific high-capacity disk-based parallel hypergraph partitioning algorithm. This algorithm was developed to exploit the particular structure found in hypergraphs derived from Markov and semi-Markov transition matrices that are constructed by a breadth-first traversal of the state-transition graphs. The high capacity of this disk-based parallel algorithm enabled very large hypergraphs (with $\Theta(10^7)$ vertices) to be partitioned for the first time. In our experiments with the parallel implementation `Par`k`way1.0`, our parallel algorithm was shown to consistently outperform an approximate approach based on parallel graph partitioning (using the state-of-the-art tool `ParMeTiS`), by up to 27% in terms of the $k-1$ partitioning objective function. However, our parallel algorithm exhibited long runtimes and poor scalability due to relatively poor processor utilisation and slow disk access times; the number of processors was also restricted to (positive) powers of two only.

This preliminary experience led to the main contribution of this thesis, which is a general parallel multilevel hypergraph partitioning algorithm that imposes no restrictions on the input hypergraph (cf. Section 4.5). It can compute k-way partitions for $k > 1$, independent of the number of processors used, p. The algorithm uses a one-dimensional distribution of the hypergraph across the processors. At every multilevel step, the hyperedges incident on the local vertices are assembled at each processor. Conflicts that may result from making coarsening and refinement decisions concurrently across many processors are avoided through a two-stage communication schedule. A hash function ensures that computational load balance is achieved at each multilevel step by inducing an evenly-spread allocation of hyperedges to processors. We have also developed a parallel formulation of multi-phase refinement during the uncoarsening phase. This parallel refinement algorithm recursively applies the multilevel approach to a partition in order to further improve partition quality.

Section 4.6 presented an analytical average-case performance model for our general parallel algorithm. Under a number of assumptions, including that the input hypergraph has small maximum vertex and hyperedge degrees and that the parallel coarsening algorithm reduces the hypergraph by a constant factor at each multilevel step, we showed that the algorithm is cost-optimal on a hypercube parallel architecture. We also showed that the algorithms are technically scalable, by deriving their isoefficiency function. The isoefficiency function $O(p^2k^2(\log p+\log k))$ is of the same order in p as that of Karypis and Kumar's parallel graph partitioning algorithm.

Chapter 5 described the implementation of the general parallel multilevel hypergraph partitioning algorithm from Section 4.5 in the parallel tool `Par`k`way2.0`. We also presented an extensive experimental evaluation of the general parallel multilevel hypergraph partitioning algorithm, using `Par`k`way2.0`. The experiments were performed on hypergraphs from a wide range of application domains, including VLSI Computer-Aided Design, Markovian performance modelling and DNA electrophoresis; partition quality was evaluated using the $k-1$ objective

function. The parallel architecture used in the experiments was a Beowulf Linux cluster with a Myrinet interconnect.

On hypergraphs that were small enough to be partitioned serially, Par$k$way2.0 was shown to produce partitions of comparable quality with those produced by the state-of-the-art serial tools PaToH and hMeTiS. On larger hypergraphs, Par$k$way2.0 consistently outperformed the parallel graph partitioner ParMeTiS, with improvements of up to 60% in the $k-1$ partitioning objective function.

The partitioning runtimes from the experiments were used to investigate speedups achieved by Par$k$way2.0 over state-of-the-art serial tools. We used the serial tool PaToH as the base-case comparison in speedup calculations because it has been highly optimised for fast runtimes. Par$k$way2.0 exhibited good speedups on hypergraphs with small average and maximum vertex degrees. On hypergraphs with higher maximum-, but small average-vertex degrees, shallower speedups were observed.

Another set of experiments investigated the scalability behaviour of the general parallel multilevel hypergraph partitioning algorithm, as predicted by its isoefficiency function. For a scalable parallel algorithm, the isoefficiency function specifies how the problem size should be increased in order to maintain a constant level of efficiency, as the number of processors used increases.

We used a family of hypergraphs derived from transition matrices generated by a semi-Markov model of a voting system, because these possess a very similar structure, and thus, proportionally similar partitioning overheads. As the number of processors used was increased, we increased the problem size by choosing the appropriate input hypergraph, according to the algorithm's isoefficiency function.

For computation of processor efficiency, the PaToH serial multilevel partitioner was used for hypergraphs that could be partitioned serially. For hypergraphs that we could not partition serially, we used linear regression on a log-log plot of the PaToH runtimes against problem size for the smaller hypergraphs to extrapolate the serial runtimes.

Across the different partition sizes, we observed no consistent increasing or de-

creasing trend in efficiency, as the number of processors and problem size are increased in accordance with the isoefficiency function. The experiments are thus consistent with our hypothesised isoefficiency function.

In Chapter 6, we presented the application of the optimised parallel multilevel hypergraph partitioning tool Par*k*way2.1 to parallel PageRank computation on a gigabit ethernet Linux cluster. Hypergraph partitioning-based sparse matrix decomposition methods for parallel sparse matrix–vector multiplication were used within a parallel iterative PageRank solver for the first time.

We investigated the use of both one-dimensional (row-wise) and two-dimensional fine-grained hypergraph partitioning-based sparse matrix decomposition methods and compared them against a recently-proposed purely load-balancing partitioning and a simple row-wise cyclic partitioning. The comparison was performed on three publicly available webgraphs that were constructed from crawls of actual web domains. The hypergraphs constructed by the models were partitioned in parallel using Par*k*way2.1. This partition was then used to distribute the web matrix across the processors, prior to the parallel iterative PageRank computation.

The hypergraph partitioning-based methods were observed to outperform the purely load-balancing method in terms of per-iteration runtime by up to 50%, while both of these methods dominated the cyclic row-wise decomposition method.

7.2 Applications

In this section, we outline possible applications of our contributions across a number of hypergraph partitioning application domains.

The parallel algorithms presented in this thesis have significantly increased capacity enabling the partitioning of very large hypergraphs. As discussed in Section 1.1.3, large-scale static and dynamic load balancing of parallel computations give rise to large-scale hypergraph partitioning problems. As an illustration of

the latter, in Chapter 6, we investigated the application of parallel hypergraph partitioning to parallel PageRank computation.

In general, parallel computations that require repeated parallel sparse matrix–vector multiplication, such as iterative linear system solvers, could achieve faster runtimes and better scalability through the use of parallel hypergraph partitioning. A striking example of this is response time analysis in Markov and semi-Markov performance models of concurrent systems. Here, a large number of sparse systems of linear equations are solved; each sparse system of linear equations is solved using a parallel iterative method, for which the kernel operation is parallel sparse matrix–vector multiplication [BDKW03]. Because the systems of linear equations all have the same non-zero sparsity pattern, hypergraph partitioning is computed only once, but reused many thousands of times.

We noted in Section 1.1.4 that direct implementation of programs written using high-level languages into reconfigurable hardware has recently attracted considerable research effort. The resulting circuits are mapped using hypergraph partitioning and as the complexity of programs to be compiled into reconfigurable hardware increases, high-capacity parallel hypergraph partitioning will be required.

7.3 Future Work

In this section, we identify a number of possible directions for future work, based on our work presented in this thesis. Future work may involve identifying improvements within the existing parallel multilevel pipeline or developing new application-specific and general algorithms.

We first outline various aspects of our parallel multilevel algorithm that require further investigation:

1. Multi-phase refinement: consider its use at coarser levels of the parallel multilevel pipeline only, while using parallel greedy k-way refinement at levels where the hypergraph is larger.

2. Reduction ratio: consider varying the reduction ratio during the coarsening phase, rather than using the same value at each coarsening step.

3. Runtime and partition quality tradeoff: investigate this by imposing limits on the number of passes during the parallel refinement phase, the size of the coarsest hypergraph and the number of multilevel steps.

4. Recursive bisection: investigate a recursive bisection multilevel formulation instead of the currently-used direct multi-way partitioning formulation.

We noted in Chapter 5 and Chapter 6 that the performance of our parallel hypergraph partitioning algorithm depends on the structure of the input hypergraph. We have identified two related areas requiring further attention:

1. Hypergraphs with a small number of large hyperedges induce a large communication overhead during each multilevel step as a result of having to replicate these large hyperedges on many processors. Future work should aim to reduce the impact of this overhead.

2. Hypergraphs arising from two-dimensional models of parallel sparse matrix–vector multiplication have a particular structure that may be exploited by a parallel partitioning algorithm; that is, each vertex in the hypergraph is incident on exactly two hyperedges.

Finally, the two-dimensional parallel multilevel hypergraph partitioning algorithm described in Section 4.7 has the advantages that it does not replicate hyperedges across the processors and that the number of processors taking part in all-to-all communication operations is proportional to $\sqrt{p}$ (rather than p, as in our parallel multilevel hypergraph partitioning algorithm). In conjuction with experience from our one-dimensional parallel multilevel algorithm, developing scalable parallel hypergraph partitioning algorithms based on the two-dimensional data distribution is a promising direction for future work.

Appendix A

Test Hypergraphs

This appendix describes in detail the test hypergraphs that have been used in this thesis. The test hypergraphs were taken from a number of application domains.

The following hypergraphs have been constructed from sparse matrices obtained from the University of Florida Sparse Matrix Collection [Dav05]. ATTpre2 was derived from a matrix representing a set of linear equations that are solved during the harmonic balance analysis of a large non-linear analogue circuit [FML96]. The cage hypergraphs were derived from transition matrices describing a model of DNA electrophoresis, where the entries represent the probability of changes between polymer configurations, for various polymer lengths (13, 14, 15) [HBB02]. These hypergraphs have all been constructed using the hypergraph model for one-dimensional row-wise decomposition, described in Section 6.3.1.

The voting hypergraphs were derived from transition matrices of a semi-Markov performance model of an electronic voting system with failures and repairs, for various numbers of voters [BDKW03]. Entries in a transition matrix reflect the rates at which the system moves from one system configuration to another. These have also been constructed using the hypergraph model for one-dimensional row-wise decomposition.

The following hypergraphs have been derived from web matrices. The entries in each web matrix represent the link structure between URLs within the domain.

Name	Vertices	Hyperedges	Non-zeros	Domain
ibm16	183 484	190 048	778 823	VLSI CAD
ibm17	185 495	189 581	860 036	VLSI CAD
ibm18	210 613	201 920	819 617	VLSI CAD
voting100	249 760	249 760	1 391 617	performance analysis
voting125	541 280	541 280	3 044 557	performance analysis
voting150	778 850	778 850	4 532 947	performance analysis
voting175	1 140 050	1 140 050	6 657 722	performance analysis
voting250	5 218 300	5 218 300	32 986 597	performance analysis
voting300	10 991 400	10 991 400	69 823 797	performance analysis
cage13	445 315	445 315	7 479 343	DNA electrophoresis
cage14	1 505 785	1 505 785	27 130 349	DNA electrophoresis
cage15	5 154 859	5 154 859	99 199 551	DNA electrophoresis
ATTpre2	659 033	659 033	6 384 539	analogue circuits
Stanford-1D	281,903	281 903	2 594 400	PageRank analysis
Stanford-2D	2 594 400	563 806	5 188 800	PageRank analysis
Stfd_Bkly-1D	683 446	683 446	8 266 822	PageRank analysis
Stfd_Bkly-2D	8 266 822	1 366 892	16 533 644	PageRank analysis
india-2004-1D	1 382 908	1 382 908	17 922 551	PageRank analysis
india-2004-2D	17 922 551	2 765 816	35 845 102	PageRank analysis
uk-2002	18 520 486	18 520 486	310 764 149	PageRank analysis

Table A.1. Significant properties of test hypergraphs.

The lexically ordered crawls of the Stanford and combined Stanford/Berkeley domains yield four hypergraphs, corresponding to two hypergraph-based sparse matrix decomposition models (one-dimensional and two-dimensional) for each domain, as described in Section 6.3. The hypergraphs are Stanford-1D, Stanford-2D and Stanford_Berkeley-1D, Stanford_Berkeley-2D respectively. Both of these web matrices were also obtained from the University of Florida Sparse Matrix Collection [Dav05].

The uk-2002 and india-2004 hypergraphs have been derived from web matrices representing a 2002 web crawl of the `.uk` domain and a 2004 crawl of the `.in` domain, respectively. Both were obtained from [BCSV04]. These web matrices have been used in PageRank calculations in [GZB04]. Note that the india-2004-1D and india-2004-2D hypergraphs correspond to hypergraph models for one-dimensional and two-dimensional decomposition of the .in web matrix, respectively. The uk-2002 hypergraph was constructed according to the hypergraph model for one-dimensional row-wise sparse matrix decomposition.

The ibm16, ibm17 and ibm18 hypergraphs are the three largest hypergraphs from the ISPD98 Circuit Benchmark Suite [Alp98]. They represent the correspond-

	Hyperedge lengths				Vertex weights			
Name	**avg**	**90%**	**95%**	**max**	**avg**	**90%**	**95%**	**max**
ATTpre2	9.04	14	21	745	9.69	18	628	628
cage13	16.8	24	26	39	16.8	22	27	39
cage14	18.0	25	27	41	18.0	23	29	41
cage15	19.2	27	29	47	19.2	25	31	47
voting100	5.57	7	7	7	5.57	7	7	7
voting125	5.62	7	7	7	5.62	7	7	7
voting150	5.82	7	7	7	5.82	7	7	7
voting175	5.84	7	7	7	5.84	7	7	7
voting250	6.32	7	7	7	6.32	7	7	7
voting300	6.35	7	7	7	6.35	7	7	7
ibm16	4.10	9	12	40	1	1	1	1
ibm17	4.54	11	13	36	1	1	1	1
ibm18	4.06	8	14	66	1	1	1	1
Stanford-1D	9.20	20	32	256	8.20	1,883	19 377	38 606
Stanford-2D	9.20	16	25	38 607	0.89	1	1	1
Stfd_Bkly-1D	12.10	27	37	250	11.10	2 675	44 037	83 448
Stfd_Bkly-2D	12.10	21	35	83 449	0.92	1	1	1
india-2004-1D	12.96	26	35	7 753	12.23	1 948	9 206	21 866
india-2004-2D	12.96	21	31	21 866	0.94	1	1	1
uk-2002	16.9	17	36	194 943	16.1	111	208	2 450

Table A.2. Average, 90th, 95th and 100th percentiles of hyperedge length and vertex weight of the test hypergraphs.

ing benchmark circuits with unit cell areas (yielding unit vertex weights in the hypergraphs, as described in Section 1.1.3).

Table A.1 shows the most significant properties of the test hypergraphs, namely their respective numbers of vertices, hyperedges and non-zeros in the incidence matrix. Table A.2 shows percentiles corresponding to the vertex weights and hyperedge lengths of the test hypergraphs.

Appendix B

Summary of Experimental Results

This appendix presents a detailed summary of results from the experiments described in this thesis. All of the results shown (i.e. both partitioning runtimes and objective function values achieved) are averages, taken from ten runs of the respective partitioners. We note that in all experiments, the $k-1$ partitioning objective function was used. The parallel hypergraph partitioning algorithms proposed in this thesis were implemented within the different versions of the Par$k$way parallel partitioner.

The remainder of this appendix is organised as follows. Section B.1 presents experimental results evaluating the Par$k$way1.0 disk-based parallel multilevel hypergraph partitioning tool. These are discussed in more detail in Section 4.3.5. Section B.2 shows the results of experiments using the Par$k$way2.0 parallel multilevel hypergraph partitioning tool. These are discussed in more detail in Section 5.3. Finally, Section B.3 presents the results from the application of Par$k$way2.1, an optimised version of Par$k$way2.0, to parallel PageRank computation in Section 6.4.

Partition size	voting250 results using 4 processors			
	Par*k*way1.0		ParMeTiS	
	$k-1$ objective	time(s)	$k-1$ objective	time(s)
8	91 511	1 309	117 354	25
16	182 206	1 393	249 415	27
32	354 561	1 495	402 681	32
64	525 856	1 777	610 597	33
total:	1 154 134	5 974	1 380 047	117

Table B.1. Par*k*way1.0 and ParMeTiS: variation in partition quality and runtime for the voting250 hypergraph.

B.1 Experiments Using Par*k*way1.0

This section presents the average $k-1$ objective function values and runtimes in experiments performed using the Par*k*way1.0 tool. Par*k*way1.0 implements the application-specific disk-based parallel multilevel hypergraph partitioning algorithm described in Section 4.3.

In the experiments described in Section 4.3.5, Par*k*way1.0 was compared to approximations produced by a graph model using the state-of-the-art parallel multilevel graph partitioning tool ParMeTiS [KSK02].

Partition size	voting300 results using 8 processors			
	Par*k*way1.0		ParMeTiS	
	$k-1$ objective	time(s)	$k-1$ objective	time(s)
16	322 737	4 827	442 387	85
32	529 763	4 762	687 659	61
64	874 652	5,007	1 033 312	80
total:	1 727 152	14 596	2 163 358	246

Table B.2. Par*k*way1.0 and ParMeTiS: variation in partition quality and runtime for the voting300 hypergraph.

	Partition Size					
p	**8**	**16**	**32**	**8**	**16**	**32**
	ATTpre2			voting175		
1 (khm)	9 799	20 829	41 245	25 245	49 997	94 550
1 (pat)	10 036	20 462	32 936	22 328	46 324	93 088
2	8 598	17 270	35 008	26 135	51 139	96 078
4	8 545	17 111	36 416	25 991	52 531	95 679
8	8 460	16 839	37 410	26 735	52 787	96 072
12	8 496	16 836	36 480	26 357	53 662	97 654
16	8 452	16 981	37 304	25 942	53 766	97 204
20	8 470	16 498	38 098	26 120	52 053	97 307
24	8 385	16 248	37 395	25 970	52 154	96 898
28	8 456	16 121	36 611	26 127	52 869	96 160
32	8 469	16 264	35 858	25 990	52 363	97 428
	cage13			ibm16		
1 (khm)	186 566	275 542	378 718	8 651	13 719	20 713
1 (pat)	190 282	272 624	375 378	8 012	13 323	20 990
2	184 435	265 638	364 558	8 843	13 628	20 782
4	185 317	262 630	362 390	8 456	14 072	20 971
8	182 751	264 586	362 355	8 401	13 714	20 699
12	182 878	261 919	365 280	8 384	13 531	20 323
16	182 544	263 569	361 083	8 542	13 712	20 208
20	183 431	260 250	359 392	-	-	-
24	182 042	263 237	361 352	-	-	-
28	181 889	265 108	360 998	-	-	-
32	181 974	262 001	360 909	-	-	-
	ibm17			ibm18		
1 (khm)	13 181	19 797	28 476	7 973	12 084	18 271
1 (pat)	12 731	19 586	27 597	7 470	11 967	18 038
2	13 196	20 420	27 466	7 555	11 473	18 114
4	12 900	20 176	28 561	7 192	11 415	18 013
8	13 153	19 940	28 655	7 496	11 321	17 862
12	13 334	20 340	28 303	7 368	11 373	18 049
16	13 647	20 087	28 752	7 140	11 159	18 150

Table B.3. Par*k*way2.0 and PaToH/hMeTiS: variation of partition quality and the number of processors on hypergraphs that were small enough to be partitioned serially.

B.2 Experiments Using Par*k*way2.0

This section presents the average $k-1$ objective function values and runtimes in the experiments using the Par*k*way2.0 tool described in Chapter 5.2. Par*k*way2.0 implements the parallel multilevel hypergraph partitioning algorithms described in Section 4.5. For the purpose of the experimental evaluation, the following

	Partition Size					
p	**8**	**16**	**32**	**8**	**16**	**32**
	voting250			voting300		
4	90 097	184 606	347 024	-	-	-
8	91 236	181 442	348 208	164 817	327 519	593 010
16	92 233	184 184	347 083	162 246	326 545	590 840
24	90 620	181 286	348 644	159 848	324 384	592 926
32	90 864	180 529	347 975	161 196	322 932	594 465
	cage14			cage15		
4	668 788	958 857	1 237 890	-	-	-
8	678 092	961 587	1 248 730	1 707 530	2 419 960	3 328 120
16	677 753	955 970	1 249 470	1 770 500	2 440 040	3 318 620
24	677 723	953 470	1 243 310	1 730 530	2 433 260	3 336 230
32	674 826	945 271	1 245 700	1 756 520	2 439 560	3 324 270
	uk-2002					
16	217 107	302 274	387 122	-	-	-
24	217 400	298 641	378 878	-	-	-
32	212 252	295 966	376 774	-	-	-

Table B.4. Par*k*way2.0: variation of partition quality and the number of processors for hypergraphs that were too large to be partitioned serially.

partitioners were also used:

- Serial multilevel hypergraph partitioning tool `hMeTiS` [KK98a]; `khm` refers to the `HMETIS_PartKway()` direct k-way multilevel partitioning routine.
- Serial multilevel hypergraph partitioning tool `PaToH` [cA01b]; `pat` refers to the `PaToH_Partition()` recursive bisection partitioning routine.
- Parallel multilevel graph partitioning tool `ParMeTiS` [KSK02].

The settings for the partitioning tools used in these experiments are described in Sections 5.3.2, 5.3.3 and 5.3.4.

Table B.3 and Table B.10 show the experimental results for those hypergraphs that could be partitioned serially on a single processor. For multiple processors, the parallel multilevel hypergraph partitioning tool Par*k*way2.0 was used.

Table B.4 and Table B.7 show Par*k*way2.0 results for those hypergraphs that were too large to be partitioned serially. Table B.5 and Table B.8 show the ex-

perimental results achieved by the graph-based approximation using the parallel multilevel graph partitioning tool ParMeTiS on the same hypergraphs.

For the experimental results shown in Table B.6 and Table B.9, the Par*k*way2.0 tool used parallel multi-phase refinement, described in Section 4.5.5. The experiments were carried out on hypergraphs that could be partitioned serially.

An empirical evaluation of the parallel hypergraph partitioning algorithms' isoefficiency function was described in Section 5.3.5. The serial results using the PaToH multilevel partitioner are shown in Table B.11 and Table B.12. These include the extrapolated serial runtimes on hypergraphs that were too large to be partitioned serially, as described in Section 5.3.5.

The parallel multilevel hypergraph partitioning tool Par*k*way2.0 was configured as described in Section 5.3.5. The Par*k*way2.0 results are shown in Table B.13 and Table B.14. Table B.15 shows the calculated processor efficiencies for Par*k*way2.0, based on the PaToH serial runtimes from Table B.11.

	Partition Size					
p	**8**	**16**	**32**	**8**	**16**	**32**
	voting250			voting300		
4	112 788	237 915	399 692	-	-	-
8	113 728	234 113	406 717	199 398	410 645	695 402
16	109 923	234 113	406 717	194 006	403 825	682 900
24	109 181	221 080	400 740	199 810	400 474	681 524
32	108 171	220 157	399 971	196 580	398 681	683 293
	cage14			cage15		
4	927 047	1 283 558	1 615 524	-	-	-
8	966 840	1 247 109	1 613 363	2 721 964	3 767 945	4 821 755
16	974 369	1 336 449	1 695 092	2 692 714	3 774 712	4 989 808
24	954 539	1 362 100	1 751 141	2 662 354	3 862 317	5 132 662
32	978 009	1 350 387	1 730 139	2 744 563	3 811 813	5 194 172
	uk-2002					
32	530 630	657 702	753 548	-	-	-

Table B.5. ParMeTiS: variation of partition quality and the number of processors on hypergraphs that were too large to be partitioned serially.

	Partition Size					
p	**8**	**16**	**32**	**8**	**16**	**32**
	voting175			ATTpre2		
2	24 252	48 222	93 125	8 684	16 748	30 917
4	24 422	48 166	93 121	8 504	16 534	30 771
8	24 553	48 714	93 104	8 503	16 440	30 634
16	-	48 435	91 842	-	16 095	30 162
32	-	-	93 243	-	-	28 675
	cage13			ibm16		
2	177 736	256 145	352 296	7 860	13 197	19 943
4	181 881	257 594	349 765	7 824	13 098	19 769
8	176 894	254 349	350 416	8 179	13 011	19 775
16	-	253 155	350 881	-	13 126	19 597
32	-	-	351 058	-	-	-
	ibm17			ibm18		
2	12 266	18 463	26 318	6 951	10 896	17 150
4	12 622	18 672	26 528	6 896	10 670	17 325
8	12 444	19 045	27 097	6 885	10 689	17 266
16	-	19 382	27 065	-	10 838	17 264

Table B.6. Par$k$way2.0 with parallel multi-phase refinement: variation of partition quality and the number of processors on hypergraphs that were small enough to be partitioned serially.

B.3 Experimental Results From the PageRank Case Study

This section presents the experimental results from the case-study presented in Chapter 6. Here, the parallel multilevel hypergraph partitioning tool Par$k$way2.1 (an optimised version of Par$k$way2.0) was applied as a pre-processing step in the parallel iterative PageRank computation pipeline.

Partitions computed by Par$k$way2.1 were used to distribute the web matrix across the processors for parallel PageRank computation according to the hypergraph partitioning-based models, described in Section 6.3. This was compared to sparse matrix decompositions produced by purely load-balancing methods. The experimental setup and discussion of results are presented in Section 6.4.

Table B.16, Table B.17 and Table B.18 present results of the experiments on the Stanford, Stanford_Berkeley and india-2004 web graphs, respectively.

	Partition Size					
p	**8**	**16**	**32**	**8**	**16**	**32**
	voting250			voting300		
4	330.9	413.9	490.0	-	-	-
8	176.9	204.9	304.6	402.1	526.5	574.1
16	91.29	116.1	170.2	217.4	292.5	383.7
24	65.68	82.38	113.1	162.7	193.8	243.8
32	54.95	66.91	86.81	140.9	170.1	197.1
	cage14			cage15		
4	1 002	1 557	2 019	-	-	-
8	815.2	1 157	1 559	4 118	5 169	7 090
16	641.6	828.3	1 170	3 188	4 270	5 256
24	522.1	703.2	1 063	2 603	3 430	4 597
32	453.9	631.4	986.7	2 322	3 054	4 093
	uk-2002					
16	10 882	11 199	11 460	-	-	-
24	8 181	8 457	8 439	-	-	-
32	7 188	7 421	7 428	-	-	-

Table B.7. Par$k$way2.0: variation of runtime and the number of processors on hypergraphs that were too large to be partitioned serially.

The following statistics were also recorded, for the combination of different web graph models being run on 4, 8 and 16 processor clusters using the 4 distinct partitioning algorithms:

- **decomposition time** (time taken to prepare the partition for each of the different partitioning algorithms)
- **number of iterations** (number of iterations to convergence of the distributed PageRank algorithm)
- **per iteration times** (average time for a single PageRank iteration)
- **Ax = b time** (average time to perform a single $\mathbf{Ax} = \mathbf{b}$ iteration)
- **Ax = b comp. time** (time taken by the local computation in an $\mathbf{Ax} = \mathbf{b}$ iteration)
- **Ax = b comm. time** (time taken by the interprocessor communication in an $\mathbf{Ax} = \mathbf{b}$ iteration)

	Partition Size					
p	**8**	**16**	**32**	**8**	**16**	**32**
	voting250			voting300		
4	10.67	10.88	11.11	-	-	-
8	5.58	5.69	5.86	12.11	12.31	12.55
16	3.23	3.32	3.39	7.03	7.18	7.28
24	2.69	2.83	2.83	5.44	5.54	5.61
32	2.41	2.43	2.51	5.01	5.16	5.20
	cage14			cage15		
4	10.36	11.43	12.37	-	-	-
8	6.16	6.53	7.09	22.74	24.35	25.79
16	3.66	3.82	4.02	13.58	14.16	14.84
24	2.90	2.98	3.11	9.35	9.82	10.25
32	2.45	2.55	2.59	8.01	8.30	8.68
	uk-2002					
32	337.4	353.7	382.5	-	-	-

Table B.8. `ParMeTiS`: variation of runtime and the number of processors on hypergraphs that were too large to be partitioned serially.

- **max non-zeros per proc.** (maximum number of non-zeros allocated per processor)
- **max vector elems per proc.** (maximum number of vector elements allocated per processor)
- **max per proc. comm. vol.** (maximum communication volume sent and received by a processor)
- **total comm. vol.** (total communication volume of number of floating point elements sent in a single PageRank iteration)

	Partition Size					
p	**8**	**16**	**32**	**8**	**16**	**32**
	voting175			ATTpre2		
2	753.0	968.4	1 030	155.0	172.9	216.8
4	479.4	881.6	1 020	82.13	105.5	140.9
8	318.8	560.9	418.0	47.08	62.26	108.4
16	-	301.0	436.4	-	49.45	81.73
32	-	-	147.5	-	-	64.89
	cage13			ibm16		
2	1 392	1 798	2 875	45.88	70.17	94.81
4	1 129	1 688	2 511	38.85	58.09	80.91
8	1 065	1 425	2 379	37.69	50.61	68.72
16	-	1 381	2 102	-	43.48	68.95
32	-	-	1 672	-	-	-
	ibm17			ibm18		
2	66.22	94.69	137.1	50.13	70.58	110.3
4	50.10	79.26	95.62	36.59	64.42	84.62
8	60.02	71.73	95.98	35.59	57.08	70.82
16	-	57.21	96.34	-	43.90	69.30

Table B.9. Par$k$way2.0 with parallel multi-phase refinement: variation of runtime and the number of processors on hypergraphs that were small enough to be partitioned serially.

	Partition Size					
p	**8**	**16**	**32**	**8**	**16**	**32**
	ATTpre2			voting175		
1 (`khm`)	28.5	33.0	41.3	73.0	78.8	90.4
1 (`pat`)	32.2	42.4	52.3	59.2	77.6	95.5
2	63.89	66.39	76.33	84.64	108.1	130.7
4	29.74	31.26	38.70	44.64	54.03	74.10
8	14.07	14.98	20.42	21.45	25.01	34.54
12	8.85	9.74	14.56	14.45	17.18	24.55
16	6.71	7.54	12.23	11.86	14.18	21.23
20	5.69	6.40	11.29	10.44	12.98	18.78
24	9.07	9.76	14.63	9.74	11.50	17.42
28	9.59	10.19	14.81	8.85	11.26	16.55
32	8.95	9.82	13.65	8.52	10.38	15.54
	cage13			ibm16		
1 (`khm`)	283.3	397.7	506.5	8.66	11.58	15.52
1 (`pat`)	267.0	327.2	380.0	14.50	17.10	19.00
2	300.7	355.1	470.0	14.26	16.59	20.51
4	207.0	264.3	323.4	10.03	12.42	15.29
8	147.3	185.1	250.4	8.05	10.14	12.92
12	117.6	164.1	240.0	8.07	10.29	12.60
16	111.6	159.2	229.8	8.90	10.57	12.93
20	111.3	154.6	218.8	-	-	-
24	106.4	148.3	211.6	-	-	-
28	106.2	145.7	216.8	-	-	-
32	101.6	141.0	224.9	-	-	-
	ibm17			ibm18		
1 (`khm`)	11.02	14.99	20.57	9.91	12.93	17.75
1 (`pat`)	9.62	11.74	13.37	16.50	19.04	21.17
2	17.47	20.63	25.78	15.67	18.28	22.69
4	13.00	16.00	19.88	11.25	13.61	16.72
8	10.17	12.81	17.22	8.92	10.67	14.20
12	9.49	12.70	15.94	9.19	11.12	14.23
16	9.99	13.06	16.26	9.23	11.51	14.21

Table B.10. Par$k$way2.0 and PaToH/hMeTiS: variation of runtime and the number of processors on hypergraphs that were small enough to be partitioned serially.

Hypergraph	$k = 4$	$k = 8$	$k = 16$	$k = 32$
voting100	7.32	10.74	14.14	17.07
voting125	17.43	26.42	33.43	41.06
voting150	27.08	40.84	54.50	65.96
voting175	40.34	61.84	83.17	101.9
voting250	227	364	496	614
voting300	527	862	1190	1480

Table B.11. PaToH: variation of runtime on voting hypergraphs (including extrapolated runtimes for voting250 and voting300).

Hypergraph	$k = 4$	$k = 8$	$k = 16$	$k = 32$
voting100	4 040	7 924	16 729	31 268
voting125	7 251	14 609	32 880	54 957
voting150	8 715	18 152	41 676	74 827
voting175	11 218	22 285	48 517	94 855

Table B.12. PaToH: variation of partition quality on voting hypergraphs.

Hypergraph	p	$k = 4$	$k = 8$	$k = 16$	$k = 32$
voting100	2	10.88	12.03	16.25	25.48
voting125	3	19.75	22.44	26.68	37.62
voting175	4	41.44	45.11	58.25	79.89
voting250	8	131.8	137.5	147.9	167.1
voting300	11	231.4	237.0	251.7	280.0

Table B.13. Par$k$way2.0: variation of runtime on voting hypergraphs.

Hypergraph	p	$k = 4$	$k = 8$	$k = 16$	$k = 32$
voting100	2	4 498	9 157	18 303	31 848
voting125	3	8 145	16 378	33 136	55 208
voting175	4	12 569	25 970	52 870	95 313
voting250	8	45 614	91 938	183 954	349 634
voting300	11	81 141	163 573	334 346	593 446

Table B.14. Par$k$way2.0: variation of partition quality on voting hypergraphs.

Hypergraph	p	$k = 4$	$k = 8$	$k = 16$	$k = 32$
voting100	2	0.34	0.45	0.44	0.33
voting125	3	0.29	0.39	0.42	0.36
voting175	4	0.24	0.34	0.38	0.32
voting250	8	0.22	0.33	0.42	0.46
voting300	11	0.21	0.33	0.43	0.48

Table B.15. Par$k$way2.0: variation of processor efficiencies on voting hypergraphs.

$p = 4$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	15.2	1 181
iterations	83	83	85	88
per iteration time(s)	0.2153	0.1681	0.0699	0.0762
Ax = b time(s)	0.2028	0.1621	0.0583	0.0657
Ax = b comp. time(s)	0.0607	0.0390	0.0551	0.0599
Ax = b comm. time(s)	0.1427	0.1237	0.0035	0.0058
messages	12	12	12	19
max non-zeros per proc.	614 346	583 653	607 030	601 362
max vector elems per proc.	70 476	73 611	90 601	87 253
max per proc. comm. vol.	304 442	267 683	12 344	1 318
total comm. vol.	601 964	530 420	13 849	1 399
$p = 8$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	13.2	1 061
iterations	79	79	83	86
per iteration time(s)	0.1854	0.1473	0.0443	0.0465
Ax = b time(s)	0.1716	0.1415	0.0318	0.0365
Ax = b comp. time(s)	0.0425	0.0169	0.0269	0.0309
Ax = b comm. time(s)	0.1299	0.1253	0.0055	0.0056
messages	56	56	44	64
max non-zeros per proc.	326 891	297 854	303 515	299 503
max vector elems per proc.	35 238	38 962	49 443	55 398
max per proc. comm. vol.	255 053	231 233	31 564	1 660
total comm. vol.	989 071	894 098	34 221	2 285
$p = 16$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	18.3	543.1
iterations	75	76	79	81
per iteration time(s)	0.1810	0.1446	0.0515	0.0513
Ax = b time(s)	0.1614	0.1377	0.0347	0.0353
Ax = b comp. time(s)	0.0532	0.0182	0.0242	0.0277
Ax = b comm. time(s)	0.1094	0.1203	0.0116	0.0076
messages	240	240	147	207
max non-zeros per proc.	192 857	155 898	151 757	151 236
max vector elems per proc.	17 619	21 208	31 215	28 221
max per proc. comm. vol.	186 331	173 525	39 820	2 214
total comm. vol.	1 364 285	1 325 808	74 137	4 307

Table B.16. Stanford web graph parallel PageRank computation results.

$p = 4$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	22.9	5 169
iterations	84	87	89	89
per iteration time(s)	0.4596	0.0618	0.0353	0.0377
Ax = b time(s)	0.4341	0.0527	0.0253	0.0264
Ax = b comp. time(s)	0.0632	0.0237	0.0239	0.0244
Ax = b comm. time(s)	0.3714	0.0293	0.0018	0.0019
messages	12	12	12	20
max non-zeros per proc.	1 977 527	1 906 240	1 990 554	1 989 151
max vector elems per proc.	170 862	188 568	204 129	243 758
max per proc. comm. vol.	810 530	112 101	6 432	2 023
total comm. vol.	1 605 286	165 765	6 648	2 081
$p = 8$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	18.4	3 304
iterations	80	85	85	84
per iteration time(s)	0.4616	0.0458	0.0285	0.0246
Ax = b time(s)	0.4376	0.0395	0.0202	0.0167
Ax = b comp. time(s)	0.0774	0.0123	0.0136	0.0130
Ax = b comm. time(s)	0.3578	0.0276	0.0071	0.0038
messages	56	56	42	62
max non-zeros per proc.	1 063 001	961 340	994 257	994 592
max vector elems per proc.	85 431	115 805	131 713	142 253
max per proc. comm. vol.	727 768	129 977	35 117	2 620
total comm. vol.	2 744 682	269 095	45 132	3 479
$p = 16$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	18.8	1 842
iterations	76	85	85	83
per iteration time(s)	0.5955	0.0518	0.0351	0.0238
Ax = b time(s)	0.5549	0.0443	0.0271	0.0150
Ax = b comp. time(s)	0.1435	0.0110	0.0101	0.0102
Ax = b comm. time(s)	0.4132	0.0340	0.0169	0.0048
messages	240	178	129	165
max non-zeros per proc.	627 253	510 616	497 659	497 055
max vector elems per proc.	42 716	73 665	78 873	69 754
max per proc. comm. vol.	548 922	120 589	80 112	3 242
total comm. vol.	4 002 962	478 162	147 590	7 302

Table B.17. Stanford_Berkeley web graph parallel PageRank computation results.

$p = 4$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	557.5	13 480
iterations	81	84	84	85
per iteration time(s)	0.7577	0.1142	0.0762	0.0781
Ax = b time(s)	0.7094	0.0972	0.0537	0.0528
Ax = b comp. time(s)	0.1243	0.0501	0.0526	0.0506
Ax = b comm. time(s)	0.5856	0.0475	0.0015	0.0022
messages	12	12	11	24
max non-zeros per proc.	4 346 286	4 319 031	4 431 469	4 264 282
max vector elems per proc.	345 727	381 623	501 669	557 602
max per proc. comm. vol.	1 326 626	147 078	2 110	1 901
total comm. vol.	2 646 280	223 467	2 428	3 018
$p = 8$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	280.9	11 360
iterations	77	81	83	81
per iteration time(s)	0.8489	0.0756	0.0458	0.0444
Ax = b time(s)	0.7985	0.0641	0.0290	0.0292
Ax = b comp. time(s)	0.1455	0.0251	0.0276	0.0263
Ax = b comm. time(s)	0.6537	0.0395	0.0024	0.0028
messages	56	56	46	105
max non-zeros per proc.	2 196 083	2 165 349	2 218 547	2 185 533
max vector elems per proc.	172 864	204 069	335 547	309 293
max per proc. comm. vol.	1 214 716	105 491	3 248	2 996
total comm. vol.	4 800 997	266 447	4 758	5 867
$p = 16$	**Cyclic**	**GleZhu**	**1D hypergraph**	**2D hypergraph**
decomposition time(s)	Neg.	Neg.	157.3	7 857
iterations	74	81	79	80
per iteration time(s)	0.9548	0.0577	0.0396	0.0405
Ax = b time(s)	0.8755	0.0455	0.0229	0.0255
Ax = b comp. time(s)	0.2797	0.0207	0.0194	0.0198
Ax = b comm. time(s)	0.5987	0.0257	0.0045	0.0055
messages	240	240	154	306
max non-zeros per proc.	1 124 363	1 126 092	1 110 174	1 091 597
max vector elems per proc.	86 432	122 143	182 236	198 703
max per proc. comm. vol.	928 783	88 210	4 486	3 896
total comm. vol.	7 237 257	313 198	14 433	11 684

Table B.18. india-2004 web graph parallel PageRank computation results.

Bibliography

[AHK95] C.J. Alpert, J-H. Huang, and A.B. Kahng. Recent directions in netlist partitioning. *Integration, the VLSI Journal*, 19(1–2):1–81, 1995.

[AHK96] C.J. Alpert, L.W. Hagen, and A.B. Kahng. A hybrid multilevel/genetic approach for circuit partitioning. In *Proc. 5th ACM/SIGDA Physical Design Workshop*, pages 100–105, 1996.

[AHK97] C.J. Alpert, J-H. Huang, and A.B. Kahng. Multilevel circuit partitioning. In *IEEE/ACM Design Automation Conference*, pages 752–757, 1997.

[AHK98] C.J. Alpert, J-H. Huang, and A.B. Kahng. Multilevel circuit partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 17(8):655–666, 1998.

[AK94a] C.J. Alpert and A.B. Kahng. A general framework for vertex orderings, with applications to netlist clustering. In *IEEE/ACM Int. Conf. on Computer Aided Design*, pages 63–67, 1994.

[AK94b] C.J. Alpert and A.B. Kahng. Multi-way partitioning via spacefilling curves and dynamic programming. In *IEEE/ACM Int. Conf. on Computer Aided Design*, pages 652–657, 1994.

[Alp96] C.J. Alpert. *Multi-way Graph and Hypergraph Partitioning*. PhD thesis, University of California Los Angeles, 1996.

[Alp98] C.J. Alpert. The ISPD98 circuit benchmark suite. In *Proc. International Symposium of Physical Design*, pages 80–85, April 1998.

[Amd67] G.M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[Are00] S. Areibi. An integrated genetic algorithm with dynamic hill climbing for VLSI circuit partitioning. In *Genetic and Evolutionary Computation Conference (GECCO–2000)*, pages 97–102, Las Vegas, USA, July 2000.

[AV00] S. Areibi and A. Vannelli. Tabu search: A meta heuristic for netlist partitioning. *VLSI Design Journal*, 11(3):259–283, 2000.

[AV03] S. Areibi and A. Vannelli. Tabu search: Implementation and complexity analysis for netlist partitioning. *ISCA International Journal of Computers and Their Applications*, 10(4):211–232, December 2003.

[AY04] S. Areibi and Z. Yang. Effective memetic algorithms for VLSI design = genetic algorithms + local search + multi-level clustering. *Journal of Evolutionary Computations, Special Issue on Memetic Evolutionary Algorithms*, 12(3):327–353, May 2004.

[AYM+04] S.N. Adya, M.C. Yildiz, I.L. Markov, P.G. Villarrubia, P.N. Parakh, and P.H. Madden. Benchmarking for large-scale and beyond. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 23(4):472–488, April 2004.

[Bar95] S.T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proc. 1995 ACM/IEEE Supercomputing Conference*, pages 602–625, 1995.

[BB99] R. Battiti and A.A. Bertossi. Greedy, prohibition and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, 1999.

[BCLS87] T. Bui, S. Chaudhuri, T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behaviour. *Combinatorica*, 7(2):171–191, 1987.

[BCSV04] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software – Practice and Experience*, 34(8):711–726, July 2004.

[BDHK03] J.T. Bradley, N.J. Dingle, P.G. Harrison, and W.J. Knottenbelt. Performance queries on semi-Markov stochastic Petri Nets with an extended continuous stochastic logic. In *Proc. 10th International Workshop on Petri Nets and Performance Models (PNPM'03)*, pages 62–71, Urbana-Champaign IL, USA, September 2nd–5th 2003.

[BDKW03] J.T. Bradley, N.J. Dingle, W.J. Knottenbelt, and H.J. Wilson. Hypergraph-based parallel computation of passage time densities in large semi-Markov models. In *Proc. 4th International Conference on the Numerical Solution of Markov Chains (NSMC'03)*, pages 99–120, Urbana-Champaign IL, USA, September 2nd–5th 2003.

[BDKW04] J.T. Bradley, N.J. Dingle, W.J. Knottenbelt, and H.J. Wilson. Hypergraph-based parallel computation of passage time densities in large semi-Markov models. *Linear Algebra and Its Applications*, 386:311–334, July 2004.

[BGG+99] D. Boley, M. Gini, R. Gross, E-H. Han, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, and J. Moore. Partitioning-based clustering for web document categorization. *Decision Support Systems*, 27(3):329–341, 1999.

[BHJL89] T. Bui, C. Heigham, C. Jones, and T. Leighton. Improving the performance of the Kernighan-Lin and Simulated Annealing graph bisection algorithms. In *Proc. 26th ACM/IEEE Conference on Design Automation*, pages 775–778, 1989.

[BKdT05] J.T. Bradley, W.J. Knottenbelt, D.V. de Jager, and A. Trifunović. Hypergraph partitioning for faster parallel PageRank computation. In *Proc. 2nd European Performance Evaluation Workshop*, volume 3670 of *LNCS*, pages 155–171. Springer, September 2005.

[BM96] T. Bui and B. Moon. Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, 45(7):841–855, July 1996.

[BM05] R.H. Bisseling and W. Meesen. Communication balancing in parallel sparse matrix–vector multiplication. *Electronic Transactions on Numerical Analysis: Special Issue on Combinatorial Scientific Computing*, 21:47–65, 2005.

[Bol86] B. Bollobás. *Combinatorics*. Cambridge University Press, 1986.

[BPR+97] R.F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J.J. Dongarra. The Matrix–Market: a web resource for test matrix collections. In R.F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137. Chapman & Hall, 1997. `http://www.math.nist.gov/MatrixMarket/index.html`.

[Brg93] F. Brglez. A D&T special report on ACM/SIGDA design automation benchmarks: Catalyst or anathema. *IEEE Design and Test*, 10(3):87–91, 1993.

[Bro41] R.L. Brooks. On colouring the nodes of a network. *Proc. Cambridge Phil. Soc.*, 37:194–197, 1941.

[BS94] S.T. Barnard and H.D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, April 1994.

[BVCG04] M. Budiu, G. Venkataramani, T. Chelcea, and S.C. Goldstein. Spatial computation. In *Proc. Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 14–26, Boston, MA, 2004.

[cA99] U.V. Çatalyürek and C. Aykanat. Hypergraph partitioning-based decomposition for parallel sparse matrix–vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.

[cA01a] U.V. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *Proc. 8th International Workshop on Solving Irregularly Structured Problems in Parallel*, San Francisco, USA, April 2001.

[cA01b] U.V. Çatalyürek and C. Aykanat. *PaToH: Partitioning Tool for Hypergraphs, Version 3.0*, 2001.

[Cer85] V. Cerny. A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm. *J. Optimization Theory Applications*, 45(1):41–51, 1985.

[CKM99a] A.E. Caldwell, A.B. Kahng, and I.L. Markov. Hypergraph partitioning for VLSI CAD: Methodology for reporting and new results. In *Proc. ACM/IEEE Design Automation Conference*, pages 349–354, June 1999.

[CKM99b] A.E. Caldwell, A.B. Kahng, and I.L. Markov. Hypergraph partitioning with fixed vertices. In *Proc. ACM/IEEE Design Automation Conf.*, pages 355–360, June 1999.

[CKM00a] A.E. Caldwell, A.B. Kahng, and I.L. Markov. Improved algorithms for hypergraph bipartitioning. In *Proc. 2000 ACM/IEEE Conference on Asia South Pacific Design Automation*, pages 661–666, January 2000.

[CKM00b] A.E. Caldwell, A.B. Kahng, and I.L. Markov. *UCLA Physical Design Tools Release*. University of California, Los Angeles, `http://nexus6.cs.ucla.edu/software/PDtools/tar.gz/`, 2000.

[CKR+97] J.A. Chandy, S. Kim, B. Ramkumar, S. Parkes, and P. Banerjee. An evaluation of parallel Simulated Annealing strategies with application to standard cell placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 16(4):398–410, April 1997.

[CL98] J. Cong and S.K. Lim. Multiway partitioning with pairwise movement. In *Proc. ACM/IEEE International Conference on Computer Aided Design*, pages 512–516, San Jose, CA, November 1998.

[CL00] J. Cong and S.K. Lim. Edge separability-based circuit clustering with application to circuit partitioning. In *Asia South Pacific Design Automation Conference*, pages 429–434, 2000.

[CLL+97] J. Cong, H.P. Li, S.K. Lim, T. Shibuya, and D. Xu. Large-scale circuit partitioning with loose/stable net removal and signal flow based clustering. In *Proc. IEEE International Conference on CAD*, pages 441–446, San Jose, CA, November 1997.

[CRX03] J. Cong, M. Romesis, and M. Xie. Optimality, scalability and stability study of partitioning and placement algorithms. In *Proc. 2003 International Symposium on Physical Design*, pages 88–94, 2003.

[CS93] J. Cong and M. Smith. A parallel bottom-up clustering algorithm with applications to circuit partitioning in VLSI design. In *Proc. 30th ACM/IEEE Design Automation Conference*, pages 755–760, 1993.

[CSZ94] P.K. Chan, M.D.F. Schlag, and J.Y. Zien. Spectral k-way ratio-cut partitioning and clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, 13(9):1088–1096, September 1994.

[DA97] A. Dasdan and C. Aykanat. Two novel multiway circuit partitioning algorithms using relaxed locking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(2):169–177, February 1997.

[Dav05] T. Davis. University of Florida sparse matrix collection, 2005. `http://www.cise.ufl.edu/research/sparse/matrices`.

[DBH+05] K.D. Devine, E.G. Boman, R.T. Heaphy, B.A. Hendrickson, J.D. Teresco, J.Faik, J.E. Flaherty, and L.G. Gervasio. New challenges in dynamic load balancing. *Appl. Numer. Math.*, 52(2–3):133–152, 2005.

[DBH+06] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Çatalyürek. Parallel hypergraph partitioning for scientific computing. In *IEEE International Parallel and Distributed Processing Symposium*, 2006. Accepted for publication.

[DD96a] S. Dutt and W. Deng. A probability-based approach to VLSI circuit partitioning. In *Proc. 33rd Annual Design Automation Conference*, pages 100–105, June 1996.

[DD96b] S. Dutt and W. Deng. VLSI circuit partitioning by cluster-removal using iterative improvement techniques. In *Proc. 1996 IEEE/ACM International Conference on Computer-Aided Design*, pages 194–200, Nov 1996.

[DD99] S. Dutt and W. Deng. Probability-based approaches to VLSI circuit partitioning. Technical report, University of Illinois at Chicago, 1999.

[DD00] S. Dutt and W. Deng. Probability-based approaches to VLSI circuit partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(5):534–549, May 2000.

[DD02] S. Dutt and W. Deng. Cluster-aware iterative improvement techniques for partitioning large VLSI circuits. *ACM Transactions on Design Automation of Electronic Systems*, 7(1):91–121, January 2002.

[de 04] D.V. de Jager. PageRank: Three distributed algorithms. M.Sc. thesis, Department of Computing, Imperial College London, September 2004.

[DPHL95] P. Diniz, S.J. Plimpton, B.A. Hendrickson, and R. Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proc. 7th SIAM Conf. on Parallel Processing for Scientific Computing*, pages 615–620, 1995.

[DT97] S. Dutt and H. Theny. Partitioning around roadblocks: Tackling constraints with intermediate relaxations. In *Proc. 1997 IEEE/ACM International Conference on Computer-Aided Design*, pages 350–355, November 1997.

[Dut93] S. Dutt. New faster Kernighan-Lin type graph-partitioning algorithms. In *Proc. IEEE International Conference on Computer Aided Design*, pages 138–143, 1993.

[EC99] C.K. Eem and J. Chong. An efficient iterative improvement technique for VLSI circuit partitioning using hybrid bucket structures. In *Proc. Asia and South Pacific Design Automation Conference*, pages 73–76, January 1999.

[Fie73] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal*, 23(98):298–305, 1973.

[Fie75a] M. Fiedler. Eigenvectors of acyclic matrices. *Czechoslovak Mathematical Journal*, 25(100):607–618, 1975.

[Fie75b] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its applications to graph theory. *Czechoslovak Mathematical Journal*, 25(100):619–633, 1975.

[FM82] C.M. Fiduccia and R.M. Mattheyses. A linear time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[FML96] P. Feldmann, R. Melville, and D. Long. Efficient frequency domain analysis of large nonlinear analog circuits. In *Proc. IEEE Custom Integrated Circuits Conference*, pages 461–464, Santa Clara, CA, 1996.

[Gaz93] H. Gazit. *Synthesis of Parallel Applications*, chapter 1, pages 197–214. Morgan Kaufmann Publishers, 1993.

[GGKK03] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.

[GHR95] R. Greenlaw, H.J. Hoover, and W.L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, 1995.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[Glo89] F. Glover. Tabu search – part 1. *ORSA Journal of Computing*, 1(3):190–206, 1989.

[GZ87] J.R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. *International Journal of Parallel Programming*, 16(6):498–513, 1987.

[GZB04] D. Gleich, L. Zhukov, and P. Berkhin. Fast parallel PageRank: A linear system approach. Tech. Rep. YRL-2004-038, Institute for

Computation and Mathematical Engineering, Stanford University, 2004.

[Haj88] B. Hajek. Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13(2):311–329, 1988.

[Hal70] K.M. Hall. An r-dimensional quadratic placement algorithm. *Management Sci.*, 17(3):219–229, 1970.

[HB97] S. Hauck and G. Borriello. An evaluation of bipartitioning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8):849–866, August 1997.

[HBB02] A. Van Heukelum, G.T. Barkema, and R.H. Bisseling. DNA electrophoresis studied with the cage model. *Journal of Computational Physics*, 180(1):313–326, July 2002.

[Hen98] B.A. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes. In *Proc. Irregular'98*, volume 1457 of *LNCS*, pages 218–225. Springer, 1998.

[HHK97] L.W. Hagen, J-H. Huang, and A.B. Kahng. On implementation choices for iterative improvement partitioning algorithms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(10):1199–1205, 1997.

[HK92] L. Hagen and A. Kahng. A new approach to effective circuit clustering. In *Proc. IEEE/ACM International Conference on CAD*, pages 422–427, 1992.

[HK00] B.A. Hendrickson and T. Kolda. Partitioning rectangular and structurally nonsymmetric sparse matrices for parallel processing. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, 2000.

[HK03] T.H. Haveliwala and S.D. Kamvar. The second eigenvalue of the Google matrix. Stanford Database Group Tech. Rep. 2003–20, Computational Mathematics, Stanford University, March 2003.

[HKKM97] E-H. Han, G. Karypis, V. Kumar, and B. Mobasher. Clustering in a high-dimensional space using hypergraph models. Technical Report 97063, University of Minnesota, 1997.

[HL93] B.A. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, SANDIA Natl. Laboratories, 1993.

[HL94] B.A. Hendrickson and R. Leland. The chaco user's guide. Technical Report SAND94-2692, SANDIA Natl. Laboratories, 1994.

[HL95] B.A. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Stat. Comput.*, 16(2):452–469, 1995.

[Hol75] J. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[HR95] M.T. Heath and P. Raghavan. A cartesian parallel nested dissection algorithm. *SIAM Journal of Matrix Analysis and Applications*, 16(1):235–253, 1995.

[IWW93] E. Ihler, D. Wagner, and F. Wagner. Modeling hypergraphs by graphs with the same mincut properties. *Information Processing Letters*, 45(4):171–175, March 1993.

[KA05] M. Koyutürk and C. Aykanat. Iterative improvement-based declustering heuristics for multi-disk databases. *Information Systems*, 30(1):47–70, 2005.

[Kah98] A. Kahng. Futures for partitioning in physical design. In *Proc. ACM/IEEE intl. Symp. on Physical Design*, pages 190–193, 1998.

[KAKS97] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. In *Proc. 34th Annual ACM/IEEE Design Automation Conference*, pages 526–529, 1997.

[Kar02] G. Karypis. Multilevel hypergraph partitioning. Technical Report 02–025, University of Minnesota, 2002.

[KH99] W.J. Knottenbelt and P.G. Harrison. Distributed disk-based solution techniques for large Markov models. In *Proceedings of the 3rd International Meeting on the Numerical Solution of Markov Chains (NSMC '99)*, pages 58–75, Zaragoza, Spain, September 1999.

[KHMG03a] S.D. Kamvar, T.H. Haveliwala, C.D. Manning, and G.H. Golub. Exploiting the block structure of the web for computing PageRank. Stanford Database Group Tech. Rep. 2003–17, Computational Mathematics, Stanford University, March 2003.

[KHMG03b] S.D. Kamvar, T.H. Haveliwala, C.D. Manning, and G.H. Golub. Extrapolation methods for accelerating PageRank computations. In *Twelfth International World Wide Web Conference*, pages 261–270, Budapest, Hungary, May 2003. ACM.

[KJV83] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[KK95] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical Report 95–064, University of Minnesota, 1995.

[KK96] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Technical Report 96–036, University of Minnesota, 1996.

[KK97] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[KK98a] G. Karypis and V. Kumar. *hMeTiS: A Hypergraph Partitioning Package, Version 1.5.3*. University of Minnesota, 1998.

[KK98b] G. Karypis and V. Kumar. Multilevel k-way hypergraph partitioning. Technical Report 98-036, University of Minnesota, 1998.

[KK98c] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[KK99] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1999.

[KKM04] J-P. Kim, Y-H. Kim, and B-R. Moon. A hybrid genetic approach for circuit bipartitioning. In *Proc. Genetic and Evolutionary Computation Conference (GECCO)*, volume 3102 of *LNCS*, pages 1054–1064. Springer, 2004.

[KL70] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 2(49):291–307, 1970.

[KL87] S. Kauffman and S. Levin. Toward a general theory of adaptive walks on rugged landscapes. *J. Theoret. Biol.*, 128:11–45, 1987.

[Kno00] W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College, London, United Kingdom, 2000.

[Kri84] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Transactions on Computers*, 33(C):438–446, 1984.

[KSK02] G. Karypis, K. Schloegel, and V. Kumar. *ParMeTiS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.0*. University of Minnesota, 2002.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. Wiley-Teubner, 1990.

[LR88] T. Leighton and S. Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. In *Proc. IEEE Sym. Foundations of Computer Science*, pages 422–431, 1988.

[Lub86] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM J. Computing*, 15(4):1036–1053, 1986.

[Mis04] M. Mishra. Scalable defect tolerance beyond the SIA roadmap. In *14th Intl. Conf. Field Programmable Logic and Application*, volume 3203 of *LNCS*, pages 1181–1182. Springer, 2004.

[NI92] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Math.*, 5(1):54–66, 1992.

[NORL87] A. Nour-Omid, A. Raefsky, and G.A. Lyzenga. Solving finite element equations on concurrent processors. In *Proc. Symposium on Parallel Computations and their Impact on Mechanics*, pages 209–227, 1987.

[Pap94] C.H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PBMW99] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Stanford Database Group Tech. Rep. 1999–66, Stanford University, November 1999.

[PSL90] A. Pothen, H.D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11(3):430–452, 1990.

[Rag95] P. Raghavan. Parallel ordering using edge-contraction. Technical Report CS–95–293, University of Tennessee, 1995.

[RM03] A. Ramani and I.L. Markov. Combining two local search approaches to hypergraph partitioning. In *Proc. Intl. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 1546–1548, 2003.

[RS93] K. Roy and C. Sechen. A timing driven n-way chip and multi-chip partitioner. In *Proc. IEEE Intl. Conf. Computer-Aided Design*, pages 240–247, 1993.

[Saa04] Y.G. Saab. An effective multilevel algorithm for bisecting graphs and hypergraphs. *IEEE Transactions on Computers*, 53(6):641–652, 2004.

[San89] L.A. Sanchis. Multiple-way network partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989.

[San93] L.A. Sanchis. Multiple-way network partitioning with different cost functions. *IEEE Transactions on Computers*, 42(22):1500–1504, 1993.

[Sim91] H.D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2/3):679–693, 1991.

[SK72] D.G. Schweikert and B.W. Kernighan. A proper model for the partitioning of electrical circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 57–62, 1972.

[SKC94] B. Selman, H. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proc. 12th Natl. Conf. on Artificial Intelligence (AAAI)*, pages 337–343, 1994.

[SNK95] T. Shibuya, I. Nitta, and K. Kawamura. SMINCUT: VLSI placement tool using min-cut. *Fujitsu Scientific and Technical Journal*, pages 197–207, 1995.

[SO99] J. Schwarz and J. Ocenasek. Experimental study: Hypergraph partitioning based on the simple and advanced genetic algorithms BMDA and BOA. In *Proc. of the Mendel 1999 Conference*, pages 124–130, Brno, CZ, 1999.

[SOHL+98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*. MIT Press, Cambridge, Massachussets, 2nd edition, 1998.

[SS93] W. Sun and C. Sechen. Efficient and effective placements for very large circuits. In *Proc. IEEE Intl. Conf. Computer-Aided Design*, pages 170–177, 1993.

[Ste94] W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

[SW91] J. Savage and M. Wloka. Parallelism in graph partitioning. *Journal of Parallel and Distributed Computing*, 12(3):257–272, 1991.

[TK04a] A. Trifunović 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 *LNCS*, pages 789–800. Springer, 2004.

[TK04b] A. Trifunović and W.J. Knottenbelt. A parallel algorithm for multilevel k-way hypergraph partitioning. In *Proc. 3rd International Symposium on Parallel and Distributed Computing*, pages 114–121, University College Cork, Ireland, July 2004.

[TK04c] A. Trifunović and W.J. Knottenbelt. Towards a parallel disk-based algorithm for multilevel k-way hypergraph partitioning. In *Proc. 5th Workshop on Parallel and Distributed Scientific and Engineering Computing*, Santa Fe, NM, USA, April 2004.

[TK06a] A. Trifunović and W. Knottenbelt. Parallel multilevel algorithms for hypergraph partitioning. *Journal of Parallel and Distributed Computing*, 2006. Submitted for publication.

[TK06b] A. Trifunović and W. Knottenbelt. Towards a parallel disk-based algorithm for multilevel k-way hypergraph partitioning. *International Journal of Computational Science and Engineering*, 2006. Accepted for publication.

[TYAS94] M. Toyonaga, S-T. Yang, T. Akino, and I. Shirakawa. A new approach of fractal-dimension based module clustering for VLSI layout. In *IEEE International Symposium on Circuits and Systems*, pages 185–188, 1994.

[UA04] B. Uçar and C. Aykanat. Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix–vector multiples. *SIAM Journal on Scientific Computing*, 25(6):1837–1859, 2004.

[VB05] B. Vastenhouw and R.H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix–vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[VCS00] P. Verplaetse, J. Van Campenhout, and D. Stroobandt. On synthetic benchmark generation methods. In *Proc. Intl. Symp. On Circuits and Systems (ISCAS)*, pages 213–216, 2000.

[Vis93] U. Vishkin. A case for the PRAM as a standard programmer's model. In *Proc. Workshop on Parallel Architectures and Their Ef-*

ficient Use: State of the Art and Perspectives, volume 678 of *LNCS*, pages 11–19, Padeborn, 1993. Springer.

[WA98] S. Wichlund and E.J. Aas. On multilevel circuit partitioning. In *Proc. IEEE International Conf. on Computer-Aided Design*, pages 505–511, 1998.

[Wal02] C. Walshaw. *The Parallel JOSTLE Library User's Guide*. University of Greenwich, 2002. Version 3.0.

[WC89] Y.-C. Wei and C.-K. Cheung. Towards efficient hierarchical designs by ratio-cut partitioning. In *Proc. IEEE Intl. Conf. Computer-Aided Design*, pages 298–301, 1989.

[WC99] C. Walshaw and M. Cross. Parallel optimisation algorithms for multilevel mesh partitioning. Technical Report 99/IM/44, University of Greenwich, 1999.

[WCE95] C. Walshaw, M. Cross, and M. G. Everett. Dynamic mesh partitioning: A unified optimisation and load-balancing algorithm. Technical Report 95/IM/06, University of Greenwich, Greenwich, London, 1995.

[WCE97] C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997.

[WD04] Y. Wang and D.J. DeWitt. Computing PageRank in a distributed internet search system. In *Proc. 30th Conf. Very Large Databases*, pages 420–431, Toronto, 2004.

[YCL92] C-W. Yeh, C-K. Cheng, and T-T.Y. Lin. A probabilistic multicommodity flow solution to circuit clustering problems. In *Proc. IEEE Intl. Conf. Computer-Aided Design*, pages 428–431, 1992.