

Parallel Multilevel Algorithms for Hypergraph Partitioning

Aleksandar Trifunović* William J. Knottenbelt

*Department of Computing, Imperial College London, South Kensington Campus,
London SW7 2AZ, United Kingdom*

Abstract

In this paper, we present parallel multilevel algorithms for the hypergraph partitioning problem. In particular, we describe schemes for parallel coarsening, parallel greedy k -way refinement and parallel multi-phase refinement. Using an asymptotic theoretical performance model, we derive the isoefficiency function for our algorithms and hence show that they are technically scalable when the maximum vertex and hyperedge degrees are small. We conduct experiments on hypergraphs from six different application domains to investigate the empirical scalability of our algorithms both in terms of runtime and partition quality. Our findings confirm that the quality of partition produced by our algorithms is stable as the number of processors is increased while being competitive with those produced by a state-of-the-art serial multilevel partitioning tool. We also validate our theoretical performance model through an isoefficiency study. Finally, we evaluate the impact of introducing parallel multi-phase refinement into our parallel multilevel algorithm in terms of the trade off between improved partition quality and higher runtime cost.

Key words: parallel hypergraph partitioning, parallel graph partitioning, parallel sparse matrix–vector multiplication, sparse matrix decomposition, load balancing, data partitioning, VLSI circuit design

1 Introduction

Intelligent *a priori* data partitioning enables the efficient parallelisation of many sparse irregular problems by reducing interprocessor communication

* Corresponding author

Email addresses: at701@doc.ic.ac.uk (Aleksandar Trifunović),
wjk@doc.ic.ac.uk (William J. Knottenbelt).

while maintaining computational load balance. Graph and hypergraph partitioning decomposition models have been widely used in this context, in particular in the fields of VLSI circuit design [3,36,46] and matrix decomposition for parallel computation [4,12–15,20,21,50,51]. Hypergraph models are in general preferred to graph models due to their greater expressiveness that overcomes the well documented limits of the graph partitioning approach [12,13,20,31,32].

Serial graph and hypergraph partitioning algorithms have been studied extensively [2,3,5,11,17,23,24,27,33,35,36,38,40,42,44,45]. Many of these are based on the multilevel approach [2,5,33,35,36,38,40], which has three main phases. Firstly, during the coarsening phase, the original (hyper)graph is coarsened to successively smaller (hyper)graphs. Next, during the initial partitioning phase, the smallest (hyper)graph in this sequence is partitioned. Finally, during the uncoarsening phase, this partition is projected back through the sequence of successively larger (hyper)graphs onto the original (hyper)graph, with heuristic refinement applied at each step.

Serial hypergraph partitioning algorithms are limited by the computing power and memory capacity of a single workstation. In this paper, we address this limitation by describing the first parallel algorithms for the hypergraph partitioning problem. Our parallel algorithms are based on the multilevel paradigm, specifically a parallel coarsening algorithm as well as two parallel refinement algorithms – a parallel direct k -way scheme and its parallel multi-phase adaptation. We also demonstrate scalability of the parallel multilevel algorithms under a theoretical performance model and the assumption of low maximum vertex and hyperedge degrees. The parallel coarsening algorithm, the parallel direct k -way refinement algorithm and their theoretical scalability analysis were first described in our earlier conference papers [48,49]. These are presented here in distilled and refined form to make the paper self-contained. We further evaluate the empirical performance of our parallel algorithms on a Beowulf cluster, partitioning a number of hypergraphs from application domains ranging from biology to VLSI circuit design. We test the scalability of our parallel algorithms' runtime and partition quality, and estimate the expected improvement in partition quality when applying parallel multi-phase refinement.

The remainder of this paper is organised as follows. Section 2 outlines the preliminaries and background material on serial hypergraph partitioning algorithms. Section 3 presents our parallel multilevel hypergraph partitioning algorithms and theoretical scalability analysis. Section 4 presents the experimental evaluation and Section 5 concludes.

2 Related Work

2.1 Problem Definition

Formally, a hypergraph is a set system (V, \mathcal{E}) on a set V , here denoted $H(V, \mathcal{E})$, such that $\mathcal{E} \subset \mathcal{P}(V) \setminus \{\emptyset\}$, where $\mathcal{P}(V)$ is the power set of V [7]. We call V the set of *vertices* and \mathcal{E} the set of *hyperedges*. Hypergraphs arising from partitioning problems are weighted, in that a scalar weight is associated with each vertex and hyperedge. When every hyperedge in a hypergraph has cardinality two, the resulting set system is better known as a *graph*. When partitioning problems lead to identical hyperedges in the hypergraph model, we replace these with a single hyperedge whose weight we set to be the sum of the weights of the identical hyperedges.

We say that a hyperedge e (vertex v) is *incident* on a vertex v (hyperedge e) if and only if $v \in e$. Vertices $u, v \in V$ are *adjacent* if and only if there exists a hyperedge $e \in \mathcal{E}$ such that $u \in e$ and $v \in e$. The *degree* of a vertex (hyperedge) is the number of hyperedges (vertices) incident on that vertex (hyperedge). The *vertex-edge incidence matrix* [6] of a hypergraph $H(V, \mathcal{E})$, $V = \{v_1, \dots, v_n\}$ and $\mathcal{E} = \{e_1, \dots, e_m\}$, is the $n \times m$ matrix $\mathbf{A} = (a_{ij})$ with entries

$$a_{ij} = \begin{cases} 1 & \text{if } v_i \in e_j \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

The vertices in a hyperedge are also called its *pins* and the total number of pins in the hypergraph is given by the number of non-zeros in the incidence matrix \mathbf{A} . Hypergraph partitioning seeks a partition of the hypergraph that optimises an objective function subject to balance constraints. A k -way ($k > 1$) *partition* $\Pi \subset \mathcal{P}(V)$ of the hypergraph $H(V, \mathcal{E})$ is a finite collection of subsets of V (or parts), such that $\Pi = \{P_1, \dots, P_k\}$, $P_i \cap P_j = \emptyset$ for all $1 \leq i < j \leq k$ and $\bigcup_{i=1}^k P_i = V$.

In the domain of VLSI Computer-Aided Design, a common problem involves dividing system components into clusters such that cluster interconnect is minimised. The vertices of the hypergraph can be used to represent the components of the circuit and the hyperedges can be used to represent the nets connecting these components [46,3]. Similarly, the decomposition of a sparse matrix across processors for parallel sparse matrix-vector multiplication may be modelled by a number of hypergraph models that correctly quantify the total communication volume [12–15,50,51].

The partitioning objective function $f_o(\Pi)$ is usually defined to be a cut metric on the hyperedges. We say that a hyperedge $e \in \mathcal{E}$ is cut by a partition Π if there exist at least two vertices $v, w \in e$ such that they have been allocated to distinct parts. The number of distinct parts that the vertices of the hyperedge have been allocated to gives the number of parts spanned by the hyperedge. The two objective functions most commonly occurring in hypergraph partitioning applications are the *hyperedge cut* (defined in equation 2) and the $k - 1$ (defined in equation 3) objectives.

$$f_o(\Pi) = \sum_{e_i \in \mathcal{E}, \lambda_i > 1} w(e_i) \quad (2)$$

$$f_o(\Pi) = \sum_{e_i \in \mathcal{E}} (\lambda_i - 1)w(e_i) \quad (3)$$

In equations 2 and 3, λ_i denotes the number of parts spanned by hyperedge e_i under the partition Π , while $w(e_i)$ represents the weight of hyperedge e_i . Note that when a two-way partition is sought (often called (hyper)graph bisection in partitioning literature), the $k - 1$ objective reduces to the (hyper)edge cut objective. We note further that, from a partitioning point of view, it is safe to ignore hyperedges with cardinalities less than two.

The partitioning constraint is defined in terms of part weights. The weight W_i of a part $P_i \in \Pi$ is given by the sum of the weights of its constituent vertices. Given a prescribed balance criterion $0 < \epsilon < 1$, the goal is to find a partition $\Pi = \{P_1, \dots, P_k\}$ such that:

$$W_i < (1 + \epsilon)W_{avg} \quad (4)$$

holds for all $1 \leq i \leq k$, where $W_{avg} = \sum_{i=1}^k W_i/k$.

2.2 Background on Hypergraph Partitioning Algorithms

Finding an optimal hypergraph bisection is NP-Hard [28]. It follows that the problem of finding an optimal k -way partition is also at least NP-Hard. Thus, research effort has been focused on developing polynomial-time heuristic algorithms that give good sub-optimal solutions. Performance of the algorithms in terms of run time and solution quality is usually evaluated using suites of benchmark hypergraphs [1,9]. A k -way partition of a hypergraph $H(V, \mathcal{E})$ is either constructed directly or by the recursive bisection of $H(V, \mathcal{E})$. A comprehensive survey of different heuristic approaches to hypergraph partitioning is presented in [3].

2.2.1 Iterative Improvement Algorithms

In hypergraph partitioning literature, *iterative improvement algorithms* have been preferred to other well-known optimisation techniques such as simulated annealing and genetic algorithms because they have the potential to combine good sub-optimal solutions with fast run times [3]. These begin with a feasible solution and iteratively move to the best neighbouring feasible solution. The algorithms terminate when they reach a feasible solution for which all neighbouring feasible solutions do not improve the objective function. The initial feasible solution can be randomly selected, or greedily constructed around a randomly chosen vertex.

Successful iterative improvement algorithms for hypergraph bisection have been based primarily on the Kernighan-Lin (KL) [42] or Fiduccia-Mattheyses (FM) algorithms [27]. These algorithms rely on a priority queue of vertex moves to greedily select the best vertex move (in the case of the FM algorithm) or the best vertex swap (in the case of the KL algorithm) in terms of the objective function. They proceed in *passes*, during each of which each vertex is moved at most once. Vertex moves resulting in negative gain are also possible, provided that they represent the best feasible move at that point. A pass terminates when none of the remaining vertex moves are feasible. The gain of a pass in terms of the objective function is then computed as the best partial sum of the gains of the individual vertex moves that are made during that pass. The algorithm terminates when the last completed pass does not yield a gain in the objective function. The low computational complexity of the FM algorithm ($O(z)$ per pass, where z is the number of pins in the hypergraph) follows from the way in which the priority queue storing the remaining vertex moves is maintained during a pass. In practice, the FM algorithm converges in a few passes and is thus quoted to run in $O(z)$ time.

The main disadvantage of the above algorithms is that they make vertex moves based solely on local information (the immediate gain of the vertex move). Enhancements to the basic algorithms that attempt to capture global properties or that incorporate look-ahead have been proposed [23–25,44]. The FM algorithm has also been extended so that it can directly compute a k -way partition [45] (a k -way extension to the KL algorithm was first proposed in the original paper by Kernighan and Lin [42]). In [17], it was observed that this k -way formulation of the FM algorithm is dominated by the FM algorithm implemented in a recursive bisection framework; hence an enhanced k -way algorithm based on a pairwise application of the FM algorithm was proposed.

2.2.2 Multilevel Paradigm

So-called *flat* partitioning algorithms (i.e. those that operate directly on a given hypergraph) suffer substantial degradation in run time and solution quality as the size of the problem increases [3]. Algorithms based on the *multilevel paradigm* are therefore preferred to flat approaches [2,5,10,33,35,36,38,40]. In the multilevel paradigm the original hypergraph $H(V, \mathcal{E})$ is approximated by successively smaller hypergraphs $H_i(V_i, \mathcal{E}_i)$, $i \geq 1$ (such that $|V_i| < |V_j|$ if $i > j$), where the number of vertices in the coarsest approximation has some upper bound α_H (usually a function of k). The construction of these approximations is called the *coarsening phase*. Having computed the set of approximations $\{H_1, \dots, H_c\}$, a partition Π_c of the coarsest hypergraph H_c is computed during the *initial partitioning phase*. This partition is then projected back onto successive finer hypergraphs ($H_c \xrightarrow{\Pi_c} H_{c-1}, \dots, H_{i+1} \xrightarrow{\Pi_{i+1}} H_i, \dots, H_1 \xrightarrow{\Pi_1} H$) and for each intermediate hypergraph (including the original hypergraph H) the partition is further refined using an iterative improvement algorithm during the *uncoarsening* or *refinement phase*.

The coarsening phase consists of a number of steps, during each of which a coarser representation $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$ of the hypergraph $H_i(V_i, \mathcal{E}_i)$ is constructed. This is performed by merging together the vertices of the hypergraph H_i to form vertices of the coarse hypergraph H_{i+1} . We represent this by the map $g_i : V_i \rightarrow V_{i+1}$, where

$$\frac{|V_i|}{|V_{i+1}|} = r_i, r_i > 1 \quad (5)$$

and r_i is the prescribed reduction ratio. It is usual to set r_i to be the same for all i . Given that a set of vertices $A \subset V_i$ maps to a single vertex $v \in V_{i+1}$, the weight of v is set to the sum of the weights of the vertices in A . The map g_i is used to construct \mathcal{E}_{i+1} from \mathcal{E}_i by applying it to every vertex of each hyperedge $e \in \mathcal{E}_i$. When the set $A \subset V_i$ that maps onto a single vertex $v \in V_{i+1}$ represents all the vertices from a hyperedge in \mathcal{E}_i , the corresponding hyperedge in \mathcal{E}_{i+1} will consist of a single vertex. These single vertex hyperedges in \mathcal{E}_{i+1} are discarded, as they will span at most one part. It is possible that a set of (distinct) hyperedges $B \subset \mathcal{E}_i$ yields a set of identical hyperedges $B' \in \mathcal{E}_{i+1}$, in which case B' is replaced by a single hyperedge whose weight is set to be the sum of the weights of the hyperedges in B . The way in which the weights of the vertices and hyperedges in H_{i+1} are calculated ensures that when a partition Π_{i+1} of the hypergraph H_{i+1} is projected onto a partition Π_i of H_i , we have that $f_o(\Pi_{i+1}) = f_o(\Pi_i)$ under both the $k-1$ and hyperedge cut metrics; furthermore, the respective part weights are also preserved.

The coarsening algorithm should ensure that a good partition of a coarser hypergraph H_i , when projected to the original hypergraph H , is also a good

partition relative to the optimal partition of H . Coarsening algorithms are discussed in detail in [3,35]. These try to merge together strongly connected vertices, where connectivity between vertices $u, v \in V_i$ is quantified in terms of the set of hyperedges incident on both u and v . There are also coarsening algorithms that derive the coarse vertex set V_{i+1} by first identifying a set of mutually independent hyperedges (i.e. hyperedges that share no common vertices) from \mathcal{E}_i and then merging together vertices in each hyperedge from this set to form vertices in V_{i+1} [36]. Most of the commonly used coarsening algorithms are implemented within the PaToH serial multilevel hypergraph partitioning tool [16].

The aim of the initial partitioning phase is to construct a good partition of the coarsest hypergraph $H_c(V_c, \mathcal{E}_c)$. Since H_c is assumed to be significantly smaller than H , time spent in the initial partitioning phase should be dominated by time spent in the other phases of the multilevel framework. Multiple randomly-seeded runs of an iterative improvement algorithm are often used [13,35,36,38]. In a graph partitioning context, more computationally expensive initial partitioning algorithms such as spectral partitioning have also been used [5,33]. Multiple partitions constructed during this phase may be propagated to the successively finer hypergraphs because it does not follow that the best partition of the coarsest hypergraph H_c will always result in the best partition of a finer hypergraph H_i , $i < c$ [35].

After projecting the partition Π_{i+1} of H_{i+1} onto Π_i of H_i during the uncoarsening phase, a heuristic refinement algorithm refines Π_i . In the case where a bisection of H_c is projected, a variant of the FM algorithm is usually used [13,35,36,38]. When a k -way partition of H_c (for a general $k > 2$) is projected, a randomized greedy refinement algorithm has been shown to yield partitions of good quality with fast run times [38]. This algorithm also proceeds in passes. During each pass, the set of vertices V_i is traversed in random order. For each vertex $v \in V_i$, the set of neighbouring parts $N(v)$ (such that a part $P \in N(v)$ if and only if there exists a vertex $u \in V_i$ adjacent to v , and $u \in P$) is constructed and gains for moving the vertex to each part $P \in N(v)$ are computed if moving v to part P does not violate the balance constraint on the partition. If there is at least one legal move of v that yields a gain in the objective function, the move resulting in the largest gain is made. Otherwise the vertex v is not moved. The algorithm terminates when the most recently completed pass does not improve the value of the objective function for the partition.

It is possible to utilise the entire multilevel framework to perform further refinement of a partition [36]. This multi-phase approach (also known as *V-cycling*) recursively applies the multilevel algorithm to the current hypergraph and its refined partition. Formally, suppose we have computed a partition Π_i for the hypergraph $H_i(V_i, \mathcal{E}_i)$, in the multilevel sequence. A *restricted* coars-

ening algorithm is applied to H_i , given the partition Π_i , such that vertices $u, v \in V_i$ are allowed to merge together if and only if there exists a part $P \in \Pi_i$ such that $u \in P$ and $v \in P$. This coarsening procedure will construct a new multilevel sequence $\{H_i, H_{i+1}, \dots, H_{c'}\}$. The initial partitioning phase is applied to $H_{c'}$ and partitions are projected through the hypergraphs $(H_{c'} \xrightarrow{\Pi_{c'}} H_{c'-1}, \dots, H_{i+1} \xrightarrow{\Pi_{i+1}} H_i)$, such that after each projection, the partitions are further refined, as in the standard uncoarsening algorithm. Successive calls to multi-phase refinement are terminated when the most recently completed V-cycle has not yielded an improvement in the objective function of the partition.

2.2.3 Concurrent Work on Parallel Hypergraph Partitioning Algorithms

Concurrent to our work, [19] presented a parallel multilevel hypergraph partitioning algorithm that uses a two-dimensional decomposition in which rectangular blocks of the incidence matrix are assigned to processors. Furthermore, the algorithm in [19] partitions the hypergraph by recursive bisection, unlike our algorithm which uses direct k -way partitioning.

Early indications are that this two-dimensional parallel algorithm may be more efficient than our one-dimensional algorithm for hypergraphs that do not have low maximum vertex/hyperedge degrees. This is because most of the global communication in the two-dimensional algorithm involves $O(\sqrt{p})$ processors, whereas the one-dimensional algorithm uses all p processors. On the other hand, the quality of partition produced by our one-dimensional algorithm is generally observed to be better [19]. This may be partly due to the choice of refinement paradigm (recursive bisection versus direct k -way); we note that in principle it should be possible to adapt the k -way parallel refinement algorithm to a two-dimensional distribution of the hypergraph to processors.

3 Parallel Multilevel Partitioning Algorithm

This section describes our parallel hypergraph partitioning algorithm, based on the multilevel paradigm. We assume that the $k-1$ objective is to be minimised, although our approach should also generalise to minimising the hyperedge cut objective function. The target architecture for the parallel algorithm is a distributed-memory, message-passing architecture.

Despite the natural parallelism inherent in the splitting steps of a recursive bisection approach, we chose to parallelise the direct k -way algorithm from [38]. This is because variants of FM bisection refinement require that priority queue data structures are maintained after each vertex move to ensure that the sub-

sequent highest-gain vertex move is easily found. Hence, after each vertex move, all the neighbouring vertices need to be informed of this move and their gains recomputed, leading to high communication overheads in a distributed-memory setting. By contrast, the serial k -way algorithm in [38] has been shown to be competitive when compared to the recursive bisection approach and requires no priority queue data structures.

3.1 Data Distribution

Letting p denote the number of processors, the hypergraph $H(V, \mathcal{E})$ is distributed across the processors as follows. We store $|V|/p$ vertices and $|\mathcal{E}|/p$ hyperedges on each processor. The vertices are allocated to processors contiguously, so that the first $|V|/p$ vertices (in terms of their index) are allocated to the first processor, the next $|V|/p$ 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 randomised allocation of vertices to processors can help to improve load balance. This 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. However, randomisation removes structure that may help to reduce interprocessor communication. Whether or not to use randomised allocation therefore depends on the specific problem instance at hand.

For the first multilevel step, hyperedges are allocated to processors contiguously. In subsequent steps, we also associate a b -bit hash key, with each hyperedge $e \in \mathcal{E}$, computed using a variant of the load balancing hash-function $h : \mathbb{N}^a \rightarrow \mathbb{N}$, from [43], where a is the maximum hyperedge cardinality. This function has the desirable property that for an arbitrary set of hyperedges E , $h(e) \bmod p$, $e \in E$, is near-uniformly distributed. Consequently, in order to ensure an even spread of hyperedges across the processors while preserving the ability to eliminate duplicate hyperedges, each hyperedge e resides on the processor given by $h(e) \bmod p$. 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

$$\mathbb{P}(C(N) \geq 1) = 1 - \mathbb{P}(C(N) = 0) \tag{6}$$

$$= 1 - \frac{M!}{(M-N)!M^N} \tag{7}$$

$$\leq 1 - e^{-\frac{N^2}{2M}} \tag{8}$$

if $N^2 \ll M$, as shown in [43]. Suppose that, for example, $|\mathcal{E}| = 10^8$ and $b = 64$. Then $\mathbb{P}(C(N) \geq 1) \leq 0.0003$ – 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 $e, e' \in \mathcal{E}$, entire sets e and e' can be compared to determine that they are indeed different.

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. We refer to hyperedges replicated on multiple processors as *frontier* hyperedges. A map from the local vertices to their adjacent hyperedges is also built (so that both vertex-to-hyperedge and hyperedge-to-vertex maps are available). At the end of the multilevel step, the non-local assembled hyperedges are deleted together with the entire vertex-to-hyperedge map.

Experience suggests that for hypergraphs with small maximum vertex degree, the memory overhead incurred by duplicating frontier hyperedges during a single multilevel step is modest; we report the percentage of the total number of pins of the hypergraph replicated in frontier hyperedges in our experiments in Section 4. We also note that even though vertex degree increases for the coarser hypergraphs (when compared to the original hypergraph $H(V, \mathcal{E})$), the first few hypergraphs in the multilevel sequence are considerably larger than the coarser hypergraphs, and thus the replication of frontier hyperedges is most significant (in terms of the amount of memory used) during the first few levels of the multilevel sequence. Further, memory overhead may be reduced by omitting large hyperedges from the coarsening computation; this has also been proposed in the context of serial partitioning in order to accelerate the coarsening computation [13,16].

3.2 Parallel Coarsening Phase

In this section, we describe our parallel coarsening algorithm (cf. Algorithm 1). At the beginning of each coarsening step, the processors first perform a parallel matching computation in order to construct the map g_i ; this is described in detail later. To construct \mathcal{E}_{i+1} , each processor needs to transform its locally stored $|\mathcal{E}_i|/p$ hyperedges from \mathcal{E}_i using the map g_i . A processor may store a

Algorithm 1 Parallel Coarsening Algorithm

Require: $H(V, \mathcal{E})$

- 1: $i = 0$; $H_0(V_0, \mathcal{E}_0) = H(V, \mathcal{E})$;
 - 2: **repeat**
 - 3: compute $g_i : V_i \rightarrow V_{i+1}$ using parallel vertex matching computation
 - 4: perform all-to-all communication of required values of g_i
 - 5: apply g_i values across locally stored hyperedges
 - 6: compute destination processors for all $e \in \mathcal{E}_{i+1}$ using hash-function h
 - 7: perform all-to-all communication of hyperedges in \mathcal{E}_{i+1}
 - 8: perform load balancing communication of V_{i+1}
 - 9: $i = i + 1$; $c = i$;
 - 10: **until** $|V_i| \leq \alpha_H(k)$ **or** $|V_i|/|V_{i+1}| < r_{\min}$
 - 11: **return** $\{H_1, \dots, H_c\}$
-

hyperedge $e \in \mathcal{E}_i$ with a vertex $v \in e$ for which the processor does not store $g_i(v)$. Thus, required values of g_i 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 $h(e) \bmod p$. Each processor retains 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.

We now describe our parallel vertex matching algorithm, which is based on the First-Choice (FC) or the Heavy Connectivity Clustering (HCC) serial coarsening algorithm [38,13]. Given a hypergraph $H_i(V_i, \mathcal{E}_i)$, the serial algorithm proceeds as follows. The vertices of the hypergraph are visited 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 connectivity metric is computed between pairs of vertices and the most strongly connected vertex to v is chosen for matching, provided that the resulting cluster does not exceed a prescribed maximum weight. The matching computation ends when $|V_i|/|V_{i+1}| > r$, with r the prescribed reduction ratio.

Our parallel matching algorithm is summarised in Algorithm 2. Each processor γ first traverses its local vertex set in random order, computing vertex matches as in the serial algorithm. We have implemented the algorithm with the absorption connectivity metric [2,36]:

$$C(u, v) = \frac{1}{w(u) + w(v)} \sum_{\{e \in \mathcal{E}_i | u \in e, v \in e\}} \frac{w(e)}{|e| - 1} \quad (9)$$

Here $w(e)$ denotes the weight of hyperedge $e \in \mathcal{E}_i$ and $w(u)$ and $w(v)$ the weights of the two vertices u and v respectively. Processor γ also maintains a *request set* for each of the $p - 1$ other processors. If the best match for a local vertex u is computed to be a vertex v stored on processor $\rho \neq \gamma$, then the vertex u is placed into the request set $S_{\gamma, \rho}$. If another local vertex subsequently chooses u or v as its best match, then it is also added to the request set $S_{\gamma, \rho}$. The local matching computation terminates when the ratio of the initial number of local vertices to the number of local coarse vertices exceeds a prescribed threshold (cf. equation 5), or when all the local vertices have been visited. Our definition of local vertices includes clusters formed from tentative matches of local vertices with vertices stored on remote processors, as well as singleton clusters from local vertices being simply copied over to the coarse hypergraph. Note that the processors complete the local matching computation asynchronously (lines 1 and 2 in Algorithm 2) but synchronize prior to the communication step in line 3.

Communication steps resolve the vertex matching requests that span multiple processors (lines 3 to 21 in Algorithm 2). In order to enable a match between two vertices on different processors that make mutual requests to each other, the communication proceeds in two stages. In the first stage, processor γ communicates request sets $S_{\gamma, \rho}$ to processor ρ and then receives replies to its requests from ρ if and only if $\gamma < \rho$, while in the second stage processor γ communicates request sets $S_{\gamma, \rho}$ to processor ρ and receives replies to its requests from ρ if and only if $\gamma > \rho$.

The processors concurrently decide to accept or reject matching requests from other processors. Denote by $M_{\gamma, \rho}^v$ the set of vertices (possibly consisting of a single vertex) from the remote processor γ that seeks to match with a local vertex v stored on processor ρ (thus, $S_{\gamma, \rho} = \bigcup_x M_{\gamma, \rho}^x$). Processor ρ considers the sets for each of its requested local vertices 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 $M_{\gamma, \rho}^v$ is granted to processor γ if the weight of the combined cluster (including vertices already matched with v) does not exceed the maximum allowed vertex weight (lines 6 and 16 in Algorithm 2).
- (2) If v has been sent to a processor μ , $\mu \neq \gamma$, as part of a request for another remote match, then processor ρ informs processor γ that the match with $M_{\gamma, \rho}^v$ has been rejected (lines 8 and 18 in Algorithm 2). This is necessary since granting this match may result in a vertex that exceeds the maximum allowed vertex weight if the remote match of v with a vertex on processor μ is granted.

Algorithm 2 Parallel Matching Computation

Require: $H_i(V_i, \mathcal{E}_i)$

- 1: each processor γ computes matches for its locally stored vertices from V_i
 - 2: each processor γ stores tentative matches with remote vertices in request sets $\{S_{\gamma,j}\}$, $1 \leq j \leq p$ and $j \neq \gamma$
 - 3: γ communicates $S_{\gamma,\rho}$ to processor ρ whenever $\gamma < \rho$
 - 4: **for all** $M_{\gamma,\rho}^v$ received on ρ **do**
 - 5: **if** v is unmatched, or matched locally by ρ **then**
 - 6: v and $M_{\gamma,\rho}^v$ matched, if cluster weight threshold not exceeded
 - 7: **else**
 - 8: match between v and $M_{\gamma,\rho}^v$ is rejected
 - 9: **end if**
 - 10: **end for**
 - 11: processor ρ communicates matching decisions to γ whenever $\rho > \gamma$
 - 12: γ locally matches all $M_{\gamma,\rho}^v$ whenever remote request rejected
 - 13: γ communicates $S_{\gamma,\rho}$ to ρ whenever $\gamma > \rho$
 - 14: **for all** $M_{\gamma,\rho}^v$ received on ρ **do**
 - 15: **if** v is unmatched, matched locally by ρ , or matched remotely **then**
 - 16: v and $M_{\gamma,\rho}^v$ matched, if cluster weight threshold not exceeded
 - 17: **else**
 - 18: match between v and $M_{\gamma,\rho}^v$ is rejected
 - 19: **end if**
 - 20: **end for**
 - 21: ρ communicates matching decisions to γ whenever $\rho < \gamma$
 - 22: γ locally matches all $M_{\gamma,\rho}^v$ whenever remote request rejected
 - 23: vertices $\{v_1, \dots, v_s\}$ from V_i that are matched together are assigned $g_i(v_1) = \dots = g_i(v_s)$
 - 24: **return** $g_i : V_i \rightarrow V_{i+1}$
-

When informed of a match rejection by processor ρ , processor γ will cluster all the vertices in the set $M_{\gamma,\rho}^v$ into a single coarse vertex (lines 11, 12, 21 and 22 in Algorithm 2). Our algorithm admits the possibility that unconnected vertices sharing a common neighbour are matched together during a single coarsening step. To see how this could happen, consider vertices u, v on processor ρ and w on processor γ , with u and w connected to v but not connected to each other. Processor ρ may match u and v together and then put a match request from γ to match v with w , yielding a coarse vertex that encompasses u, v and w if v and w are sufficiently highly connected for the remote match to be granted. This behaviour is allowed because even though u and w are unconnected, in most cases the three vertices u, v and w nonetheless form a natural cluster.

3.3 Serial Initial Partitioning Phase

The parallel coarsening phase terminates when the number of vertices in the coarsest hypergraph is below a pre-specified threshold $\alpha_H(k)$ (in our implementation k times a user-specified constant) or the most recent reduction in the number of vertices $|V_i|/|V_{i+1}|$ does not equal or exceed a minimum required rate r_{\min} . We assume that the coarsest hypergraph $H_c(V_c, \mathcal{E}_c)$ is small enough for a partition to be rapidly computed on a single processor (when compared to the runtimes of the other phases of the algorithm). This is motivated by the observation that the parallel coarsening algorithm usually reduces the original sparse hypergraph by a few orders of magnitude (e.g. from 10^6 vertices and 10^7 pins to 10^3 vertices and 10^4 pins). Currently, our algorithm is not configured to partition the hypergraph in the scenario where the coarsest hypergraph cannot fit into the memory of a single machine. However, we foresee two possible approaches to solving this problem. The first is a parallel initial partition generator, while the second would enforce further clustering of the vertices with the sole purpose of reducing the hypergraph so that it can be partitioned serially on a single machine. In our implementation, $H_c(V_c, \mathcal{E}_c)$ is gathered on all processors and p runs of the serial algorithm are computed concurrently across the processors in parallel. The partition with the lowest cutsize is then projected through the parallel uncoarsening phase.

3.4 Parallel Uncoarsening Phase

At the beginning of each step of the parallel uncoarsening phase, we have $H_i(V_i, \mathcal{E}_i)$, $H_{i+1}(V_{i+1}, \mathcal{E}_{i+1})$ and a k -way partition Π_{i+1} of H_{i+1} . The projection $H_{i+1} \xrightarrow{\Pi_{i+1}} H_i$ that yields Π_i is computed as follows. Let V_i^γ be the subset of V_i stored on processor γ . For each $v \in V_i^\gamma$ if $\Pi_{i+1}(g_i(v))$ is not available, it is requested from the processor that stores $g_i(v) \in V_{i+1}$. We set $\Pi_i(v) = \Pi_{i+1}(g_i(v))$. The frontier hyperedges are then assembled on each processor, as at the beginning of each parallel coarsening step. We then apply our parallel formulation of the greedy k -way serial refinement algorithm [38].

The parallel refinement algorithm proceeds in passes, during each of which a vertex can be moved at most once; however, instead of moving individual 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 γ traverses its local vertex set in a random order and for each $v \in V_i^\gamma$, 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 $U_{i,j}^\gamma$, $i \neq j$, $i, j = 1, \dots, k$, where i and j denote current and destination parts respectively. In order to reduce possible

conflicts (e.g. movement of vertices in opposing directions such that their individual moves might result in a positive gain in objective function but when both are made, they in fact yield a non-positive gain), 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. We note that this does not guarantee that a sequence of (concurrently made) vertex moves will result in a positive gain. However, in our experiments, this parallel refinement algorithm produced partitions competitive with those produced by state-of-the-art serial tools.

The partition balance constraint (cf. equation 4) is maintained via global communication. The alternative method, which precludes global communication, is to enforce a balance constraint locally on each processor such that the partitioning constraint in equation 4 is maintained overall. However, as the number of processors increases, the local constraints on each processor are likely to become too tight to allow the algorithm to sufficiently explore the feasible solution space; we instead choose to pay the price of possibly higher communication overhead for better refinement algorithm performance.

At the beginning of each of the two stages, the processors know the exact part weights and each processor γ maintains the balance constraint during its local computation of the set $U_{i,j}^\gamma$. The associated weights and gains of all the non-empty sets $U_{i,j}^\gamma$ are communicated to the root processor (chosen arbitrarily) which then determines the actual partition balance that results from the moves of the vertices in the sets $U_{i,j}^\gamma$. If the balance constraint is violated, the root processor determines which of the moves should be taken back and informs the processors containing the vertices to be moved back. This is 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 a pass is not positive. Note that vertices need not be explicitly moved between processors; rather, their part index value can be changed by the processor that stores the vertex.

3.5 Parallel Multi-phase Refinement

This section describes our parallel multi-phase refinement algorithm, which consists of three multilevel phases, namely the parallel restricted coarsening phase, the serial initial partitioning phase and the parallel uncoarsening phase. The serial initial partitioning and the parallel uncoarsening phases are identi-

cal to those described in Sections 3.3 and 3.4.

The parallel restricted coarsening phase takes a partition Π_i of the hypergraph H_i as input; vertices are only allowed to match with a neighbouring 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 $\Pi_i(u) = \Pi_i(v)$. This can be incorporated within the existing parallel coarsening algorithm, since transitivity of the above condition (i.e. $\Pi_i(u) = \Pi_i(v)$ and $\Pi_i(v) = \Pi_i(w)$ imply $\Pi_i(u) = \Pi_i(w)$) ensures that the resolution of remote matching requests will result in the matching together of vertices allocated to the same part. Rather than follow this approach, we exploit additional concurrency afforded by the restriction. Vertices belonging to the same part are collected onto a single processor; this precludes further communication during the restricted coarsening phase since only vertices allocated the same part may match together. Processors then concurrently execute a serial coarsening algorithm (FC/HCC). The partition balance criterion in equation 4 should ensure that computational load balance across the processors is maintained during this phase. The drawback of this scheme is that load balance is only maintained if the number of parts in the partition k is a multiple of the number of processors p and $k \geq p$. We note that this is a not a restriction on parallel multi-phase refinement *per se*. 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 3.2.

3.6 Analytical Performance Model

In this section we present an analytical performance model of our parallel algorithms and derive the average-case asymptotic runtime T_p , assuming that the underlying parallel architecture is a p -processor hypercube. We show that the algorithm is asymptotically scalable when the maximum vertex and hyperedge degrees in the hypergraph are small, and also derive its isoefficiency function [30].

Let $|V| = n$ and suppose that the numbers of vertices and hyperedges in the original hypergraph are of the same order of magnitude, so that we may write $|\mathcal{E}| = \Theta(n)$. Further, let l and d denote the maximum hyperedge degree and the maximum vertex degree of the original hypergraph, and l_i and d_i denote the respective maximum degrees within hypergraph H_i in the multilevel process. We assume that l and d are small constants, so that $l \ll n$ and $d \ll n$, and also assume that the numbers of vertices and hyperedges are respectively reduced by constant factors $1 + \alpha$ and $1 + \omega$ ($\alpha, \omega > 0$) at each coarsening step. We discuss the appropriateness of these assumptions in Section 4. We let $n_i = \max\{|V_i|, |\mathcal{E}_i|\}$ and assume in our analysis that with increasing i these values become small when compared to the numbers of vertices/hyperedges in

the original hypergraph. We have that $l_i \leq l$ for all $0 \leq i \leq c$. We know that in practice d_i is increasing because the number of vertices usually decreases more quickly than the number of hyperedges. However, under our assumptions it remains very small compared to n . This is because $d_i \leq n_i$ for all $i \leq c$ and n_i is decreasing so that $n_i \ll n$ for i close to c .

3.6.1 Performance Model of the Parallel Multilevel Algorithm

We consider the computation and the communication requirements of each phase in turn, given $O(\log n)$ coarsening steps.

The procedure of assembling the hyperedges incident on the locally stored vertices precedes each 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. Here, $O(d_i l_i)$ computation steps are performed in computing a matching 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 we assume 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 γ computes the matching vector values required from other processors and, having obtained them, computes the set \mathcal{E}_{i+1}^γ . 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}^γ 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 are small relative to n_i , the computation requirement during each uncoarsening step (including assembling incident hyperedges) is $O(n_i / p)$. During the latter steps in the coarsening phase, when $b \leq i \leq c$ (for some $b > 1$), we may not have $d_i \ll n_i$. However, here $d_i \leq n_i \ll n$ (as noted earlier), so that the computational requirement of the latter coarsening steps is dominated by the requirement of the first few coarsening steps.

During the serial initial partitioning phase, the hypergraph has size $O(k)$ and can be heuristically partitioned to yield a “good” sub-optimal partition in $O(k^2)$ computation steps [27].

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 involves at most $O(n_i / p)$ computation steps on each processor. We consider a single pass of the parallel greedy refinement algorithm, and assume

that the refinement algorithm terminates in 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 gains for a vertex move, the algorithm needs to visit all the hyperedges incident on that vertex 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 $O(pk^2)$. Arguing as for the coarsening phase, the overall computation requirement during each uncoarsening step (including assembling incident hyperedges) is $O(n_i/p) + O(pk^2)$.

The overall asymptotic computational complexity of our parallel partitioning algorithm is thus given by

$$T_{comp} = \sum_{i=b}^{O(\log n)} \frac{O(n_{i-b})}{p} + O(pk^2 \log n) \quad (10)$$

$$= \sum_{i=b}^{O(\log n)} \frac{O(n)}{p(1 + \min\{\alpha, \omega\})^{i-b}} + O(pk^2 \log n) \quad (11)$$

$$\leq O(n/p) \sum_{i=0}^{\infty} \frac{1}{(1 + \min\{\alpha, \omega\})^i} + O(pk^2 \log n) \quad (12)$$

$$\leq O(n/p) + O(pk^2 \log n) \quad (13)$$

We now shift our attention to an average-case communication cost analysis, assuming that the underlying parallel architecture is a p -processor hypercube with bidirectional links and store-and-forward routing.

Again, we first consider the procedure of assembling the hyperedges incident to locally held vertices on each processor. 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 is a small constant (relative to n_i) for the majority of the multilevel steps 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 [30]. In the multilevel steps involving the coarser hypergraphs (i.e. where $b \leq i \leq c$, for some $b > 1$), we do not necessarily have $d_i \ll n_i$; however, here $d_i \leq n_i \ll n$ and the communication requirements of these steps are dominated by the communication requirements of the multilevel steps that involve the larger hypergraphs.

We now consider the cost of communicating the required matching vector entries in computing the local subset of \mathcal{E}_{i+1} from the local subset of \mathcal{E}_i on each processor and the subsequent load balancing communication of hyperedges of

H_{i+1} . Given $O(n_i/p)$ hyperedges from \mathcal{E}_i on each processor, to construct the corresponding subset of \mathcal{E}_{i+1} , each processor requires $O(n_i l_i/p)$ values from the map g_i (this requirement was discussed in Section 3.2). Assuming that each of the required entries in g_i 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} are scattered across the p processors with equal probability, thus also giving an average message size in the all-to-all personalized communication of $O(n_{i+1} l_{i+1}/p^2) \leq O(n_i l_i/p^2)$. Hence, given that $l_i \ll n_i$, 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 cost of communicating matching requests and their outcomes is done in $O(n_i/p)$ time. During each coarsening step, we also require the computation of prefix sums to determine the numbering of the vertices in the coarser hypergraph, which has complexity $O(\log p)$. During refinement, we require an additional broadcast of rebalancing moves and a reduction operation to compute the cutsize, which have 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, we deduce that the overall average-case asymptotic communication cost of our parallel partitioning algorithm is

$$T_{comm} = O(n/p) + O(k^2 \log p \log n) \quad (14)$$

Eliminating dominated terms from equations 13 and 14, the asymptotic total average-case parallel run time is

$$T_p = O(n/p) + O(pk^2 \log n) \quad (15)$$

As the complexity of the serial algorithm is $O(n)$, it follows that the algorithm is cost-optimal (scalable) for large n as the $O(n/p)$ term dominates the parallel runtime.

To derive the isoefficiency function (for a discussion of isoefficiency see e.g. [30]), we note that the complexity of the serial multilevel algorithm (and thus problem size W) is $O(n)$ when $l \ll n$. From equation 15, the total overhead T_o of the parallel algorithm becomes:

$$T_o = O(p^2 k^2 \log W) \quad (16)$$

The isoefficiency function is then given by

$$\begin{aligned}
W &= KT_o(W, p) \\
&= O(p^2 k^2 \log W) \\
&= O(p^2 k^2 \log(p^2 k^2 \log W)) \\
&= O(p^2 k^2 \log p) + O(p^2 k^2 \log k) + O(p^2 k^2 \log \log W)
\end{aligned}$$

and thus, omitting the lower order terms,

$$W = O(p^2 k^2 (\log p + \log k)) \tag{17}$$

We note that this isoefficiency function is of the same order as that given in [39] for the parallel graph partitioning algorithm implemented in the ParMeTiS tool [41].

3.6.2 Model of Algorithm with Parallel Multi-phase Refinement

Since we assume that parallel multi-phase refinement converges in a small number of iterations, the performance model of this algorithm differs from that of the multilevel algorithm analysed in Section 3.6.1 only in the parallel restricted coarsening phase described in Section 3.5.

A single parallel restricted coarsening step consists of two stages. In the first stage, the vertices belonging to the same part in Π_i are assigned to the same processor; in the second stage the coarsening algorithm computes the map g_i concurrently on all processors in parallel, without communication. The vertices are then allocated to processors using an all-to-all personalised communication. We assume that the vertices have an equal probability of being assigned to each of the k parts of the partition and thus, assuming a random distribution of vertices to processors prior to partitioning, the average message size in the all-to-all communication will be $O(n_i/p^2)$ so that the all-to-all communication can be done in $O(n_i/p)$ time on a hypercube. Since the k -way partition is subject to the balancing constraint of equation 4, each processor will on average hold $O(n_i/p)$ vertices during a parallel restricted coarsening step (assuming a low variance in vertex weights). The computational requirement of a single step of the parallel restricted coarsening algorithm is $O(n_i d_i l_i / p)$, followed by the construction of the coarse hyperedge set \mathcal{E}_{i+1} , as in the standard multilevel algorithm. Thus, since $d_i \ll n_i$ and $l_i \ll n_i$ for the dominating multilevel steps (i.e. those involving the largest hypergraphs), the asymptotic runtime and the isoefficiency function of the parallel multilevel algorithm with parallel multi-phase refinement are the same as those of the parallel multilevel algorithm analysed in Section 3.6.1.

4 Experimental Evaluation

4.1 Implementation and Test Environment

The three phases of our parallel multilevel k -way partitioning algorithm have been implemented in C++ using the Message Passing Interface (MPI) standard [47], thus forming the Parkway 2.1 parallel hypergraph partitioning tool. For the initial partitioning phase, our implementation provides an interface to the `HMETIS_PartKway()` routine from the hMeTiS [37] library; this is used for serial partitioning when the coarsest hypergraph from the parallel coarsening phase has less than $200 \times k$ vertices.

The architecture used in all experiments reported below consists of a Beowulf Linux Cluster with 64 dual-processor nodes, although we were restricted to 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 of RAM. The nodes are connected by a Myrinet network with a peak throughput of 250 MB/s. The serial base cases reported below (generated by the PaToH [16] tool) were run as single processor jobs on this cluster.

4.2 Test Hypergraphs

The hypergraphs used in our experimental evaluation have been derived either from large sparse matrices that arise in scientific numerical computations, or models of a physical system (such as a VLSI circuit). For the sparse matrices, the corresponding hypergraphs represent the 1D row-wise decomposition of the sparse matrix for parallel matrix–vector multiplication, as described in [13]. Thus, a given sparse matrix defines the incidence matrix of its corresponding hypergraph, except that non-zeros are added to the entries of the main diagonal of the incidence matrix wherever the sparse matrix has a zero on the main diagonal. The weights of the vertices in the hypergraph are set to be the number of non-zeros in the corresponding row of the sparse matrix, while weights of the hyperedges are set to unity. Finally, single-vertex hyperedges are removed, since they will not contribute to the value of the $k-1$ metric of the computed partition.

Table 1 shows the main characteristics of the test hypergraphs. The first three matrices were obtained from the University of Florida Sparse Matrix Collection [18]. The entries in the **Stanford** matrix represent the link structure between URLs within the Stanford University domain. This matrix has been used in PageRank calculations in [29]. **ATTpre2** represents a set of linear equa-

Table 1

Main characteristics of test hypergraphs

Name	#vertices	#hyperedges	#non-zeros	Domain
Stanford	281 903	281 903	2 594 400	PageRank analysis
ATTpre2	659 033	659 033	6 384 539	analog circuits
cage13	445 315	445 315	7 479 343	DNA electrophoresis
bcsstk32	44 609	44 609	2 014 701	Structural engineering
ibm18	210 613	201 920	819 617	VLSI circuit
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
voting200	1 597 373	1 597 373	9 320 912	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
voting350	16 156 700	16 156 700	102 822 650	performance analysis

Table 2

Average, 90th, 95th and 100th percentiles of hyperedge degrees and vertex weights of test hypergraphs. A value for the x^{th} percentile means that x percent of vertex weights/hyperedge degrees in the hypergraph are less than or equal to that value.

Name	Hyperedge degrees				Vertex weights			
	avg	90%	95%	max	avg	90%	95%	max
Stanford	9.20	20	32	256	8.20	1 883	19 377	38 602
bcsstk32	45.2	54	60	216	45.2	54	60	216
ATTpre2	9.04	14	21	745	9.69	18	628	628
cage13	16.8	24	26	39	16.8	22	27	39
ibm18	4.06	8	14	66	1	1	1	1
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
voting200	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
voting350	6.36	7	7	7	6.36	7	7	7

tions that are solved during a harmonic balance analysis of a large non-linear analog circuit [26]. The **cage13** matrix is a transition matrix from a model of DNA electrophoresis, where the entries represent the probability of changes between polymer configurations, for polymers of length 13 [34]. The **bcsstk32** matrix was obtained from the Rutherford-Boeing collection [22] and originates from a problem in structural engineering. The **ibm18** matrix represents a VLSI circuit and was obtained as part of the ISPD98 Circuit Benchmark suite [1]. Finally, the **voting** matrices are transition matrices derived from a semi-Markov performance model of an electronic voting system with failures and repairs, for various numbers of voters (100 to 350). Entries in the matrix reflect the rates at which the system moves from one system configuration to another [8].

4.3 Empirical Scalability Evaluation

In this set of experiments, we set out to empirically validate the asymptotic scalability analysis from Section 3.6. Two experiments are performed. In the first, we compute speedups relative to the PaToH serial partitioning tool and also investigate partition quality as the number of processors is increased. In the second, we investigate the empirical validity of our scalability analysis by observing whether processor efficiency is maintained with an increasing number of processors, given that the problem size is increased according to our derived isoefficiency function.

For these experiments, our parallel algorithm was configured as follows. The reduction ratio (cf. equation 5) was set to 1.75. We set the partitioning objective to SOED (sum of external degrees) in `HMETIS_PartKway()` (although we report the resulting $k-1$ objective values) and ran it with the First-Choice coarsening option and the V-Cycle best intermediate refinement option. The SOED metric is computed using equation 3, but using λ_i instead of $\lambda_i - 1$ to scale the hyperedge weight so that every hyperedge contributes to the cut-size. We used it because `HMETIS_PartKway()` cannot directly minimize the $k - 1$ objective and the two metrics are very closely related. During the parallel uncoarsening phase, only the best partition from the serial partitioning phase was projected. Partitions with a balance constraint of 5% were sought ($\epsilon = 0.05$ in equation 4). The tool PaToH [16] provided the serial base-case comparison and was run with default parameters and the `PATOH_CONPART` ($k-1$ metric) partitioning objective. To ensure that the final partition satisfies the balance constraint of equation 4 with $\epsilon = 0.05$ when computed by recursive bisection using PaToH¹, the balance constraint on each individual bisection was set to $((1 + \epsilon)/k)^{1/\log_2 k} - 0.5$.

Table 3 shows the overhead (measured as the proportion of the total number of pins replicated across the processors and shown in bold, as well as our simulated overhead figures for contiguous and random allocation of hyperedges to processors for the initial hypergraph only) associated with assembling frontier hyperedges and the average imbalances in the hypergraph distribution across the processors prior to the load balancing communications. We note that the number of frontier pins is significantly reduced when the contiguous rather than random hyperedge allocation is used for the initial hypergraph; this may be because the incidence matrices have a substantial amount of non-zeros around the main diagonal. The average frontier hyperedge overhead is low for the `ATTpre2`, `voting175` and `bcsstk32` hypergraphs, but is more significant for the `cape13`, `ibm18` and `Stanford` hypergraphs, the difference being accounted

¹ We note that the most recent release of PaToH accepts a single global balance constraint rather than a constraint on each individual bisection.

for by factors such as maximum vertex/hyperedge degrees and the degree of locality in the hypergraph structure. The observed hyperedge and vertex ratios justify the load balancing communication at each multilevel step; for example, an average ratio of 1.05 at each of 8 multilevel steps potentially leads to a final ratio of $1.05^8 = 1.48$ without explicit load balancing.

Tables 6 and 7 in the Appendix summarise the observed performance of *Parkway2.1* when executed on the benchmark hypergraphs with a varying number of processors ($p \geq 2$). The PaToH tool provides the serial base case and the runtimes and cutsizes shown are the averages of ten runs.

Figure 1 shows the speedups over the PaToH serial base case for each hypergraph. For *voting175* and *ATTpre2*, speedup increases with the number of processors; in the case of *ATTpre2*, it then drops off when more than 24 processors are used. A possible explanation may be the presence of a small number of hyperedges that are significantly larger than the average hyperedge length (cf. Table 2). These may cause imbalanced all-to-all communications, with a small number of messages that are significantly larger than the rest. The speedups for *Stanford*, *bcsstk32* and *ibm18* are shallow, while there is no absolute speedup for *cake13*. These observations are consistent with the frontier hyperedge overheads shown in Table 3. It is not surprising that we observe best speedups for the *voting175* hypergraph; in terms of sparsity and low maximum vertex and hyperedge degree, it most closely fits the assumptions of our performance model in Section 3.6.

Figure 2 plots the partition quality delivered by *Parkway2.1* in terms of the ratio of its cutsize ($k - 1$ metric) to the cutsize of the PaToH serial base case. We observe that the partitioning performance of our parallel algorithm closely matches and often exceeds that of PaToH. We conjecture two reasons for this. Firstly, within our parallel multilevel framework, many partitioning runs are performed on the coarsest hypergraph (concurrently on each processor). Thus, it is likely that our uncoarsening phase begins with a superior initial partition. Secondly, our parallel algorithm uses direct k -way partitioning whereas PaToH uses recursive bisection. For larger values of k , the direct k -way partitioning approach may be better than recursive bisection because it maintains a global view when making vertex moves, rather than a divide-and-conquer approach. In addition, the recursive bisection approach may be further constrained by the need to enforce tight balance constraints on each individual bisection run. These results also suggest that the partition quality provided by our parallel algorithm is maintained as the number of processors increases.

The existence of the isoefficiency function should enable our parallel algorithm to maintain a constant level of efficiency as the number of processors is increased, by appropriately increasing the problem size [30]. That is to say, with our isoefficiency function $O(p^2 \log p)$ (for fixed partition size), if the number

Table 3

Vertex and hyperedge distribution statistics for varying numbers of processors. The frontier pins column shows the total number of pins duplicated in frontier hyperedges across all processors for all multilevel steps in bold, expressed as a multiple of total number of pins in all the hypergraphs in the multilevel sequence. Also shown in parentheses are simulated overhead figures for a contiguous (c) and random (r) allocation of hyperedges to processors for the initial hypergraph only; note that vertices are always allocated contiguously to processors. The hyperedge ratio and vertex ratio columns are the average ratios of the maximum and minimum numbers of hyperedges/vertices of a hypergraph on individual processors before load balancing. These latter two averages are computed over all multilevel steps.

ATTpre2	frontier pins	hyperedge ratio	vertex ratio
2	0.07 (0.05c,0.53r)	1.01	1.01
4	0.18 (0.16c,0.87r)	1.01	1.02
8	0.34 (0.30c,1.13r)	1.03	1.04
16	0.45 (0.33c,1.25r)	1.07	1.06
32	0.59 (0.39c,1.35r)	1.17	1.11
voting175	frontier pins	hyperedge ratio	vertex ratio
2	0.05 (0.01c,0.50r)	1.01	1.04
4	0.12 (0.03c,0.77r)	1.02	1.04
8	0.21 (0.06c,0.93r)	1.03	1.04
16	0.36 (0.12c,1.05r)	1.04	1.05
32	0.57 (0.25c,1.21r)	1.05	1.05
cage13	frontier pins	hyperedge ratio	vertex ratio
2	0.59 (0.27c,0.69r)	1.00	1.03
4	1.21 (0.72c,1.29r)	1.00	1.04
8	2.19 (1.60c,2.27r)	1.01	1.07
16	3.38 (2.69c,3.46r)	1.03	1.10
32	4.52 (3.56c,4.42r)	1.04	1.11
ibm18	frontier pins	hyperedge ratio	vertex ratio
2	0.83 (0.90c,0.90r)	1.01	1.05
4	1.93 (2.09c,2.90r)	1.02	1.07
8	3.23 (3.43c,3.43r)	1.05	1.12
16	4.56 (4.80c,4.80r)	1.11	1.17
32	4.59 (5.94c,5.93r)	1.21	1.21
Stanford	frontier pins	hyperedge ratio	vertex ratio
2	0.90 (0.92c,0.96r)	1.02	1.04
4	2.37 (2.49c,2.62r)	1.04	1.05
8	4.57 (4.94c,5.20r)	1.06	1.05
16	7.77 (8.34c,8.76r)	1.08	1.07
32	10.9 (12.2c,12.8r)	1.11	1.09
bcsstk32	frontier pins	hyperedge ratio	vertex ratio
2	0.17 (0.15c,0.58r)	1.01	1.01
4	0.29 (0.26c,0.95r)	1.04	1.02
8	0.43 (0.36c,1.19r)	1.07	1.03
16	0.67 (0.54c,1.44r)	1.13	1.07
32	0.97 (0.75c,1.70r)	1.22	1.16

of processors is increased from p to p' , then the problem size must increase by a factor $\psi_p = (p'^2 \log p') / (p^2 \log p)$ in order to maintain the efficiency achieved using p processors. To validate this empirically, we make use of the voting

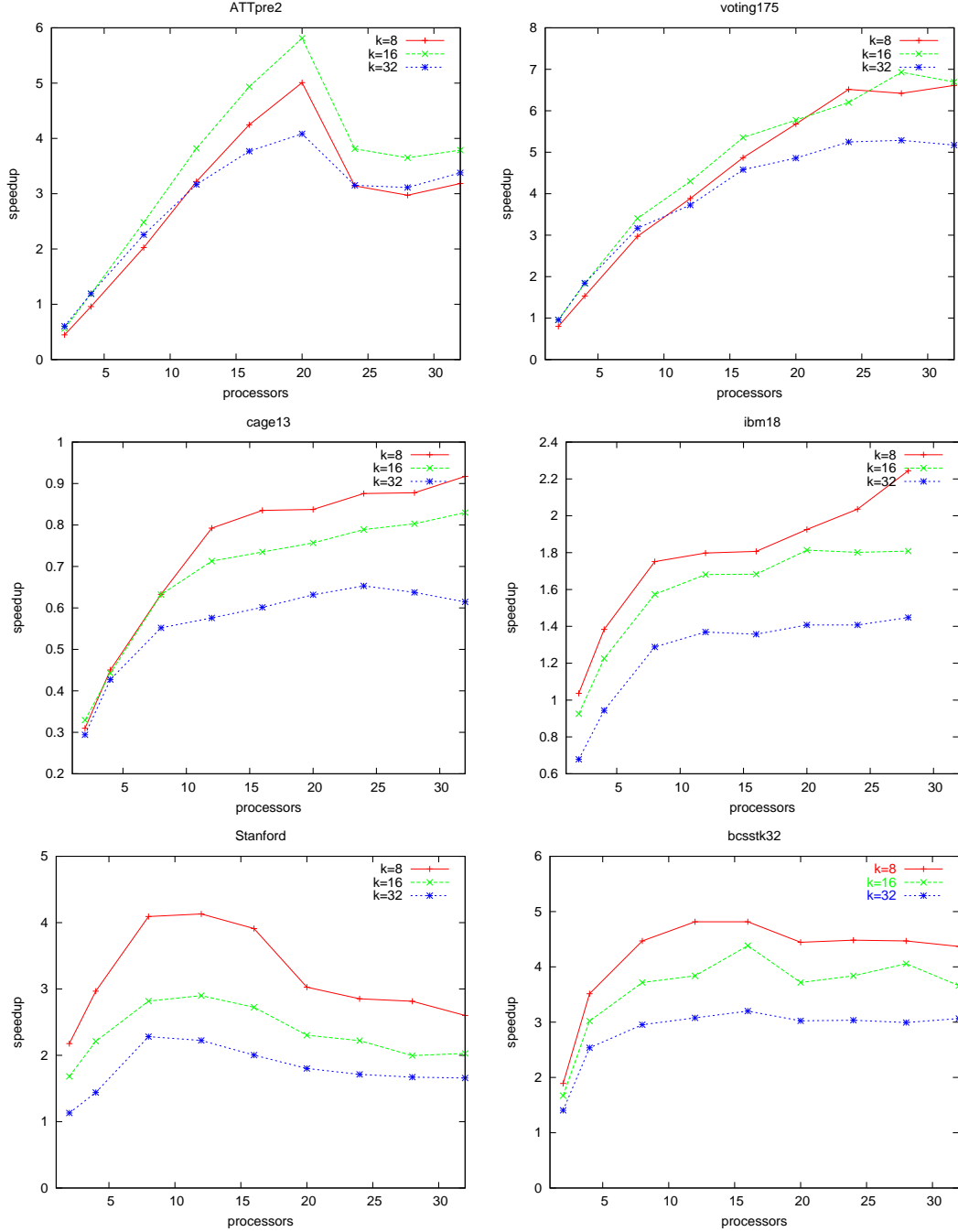


Fig. 1. Parkway2.1: speedup results using PaToH as base-case.

family of hypergraphs that cover a wide range of problem sizes and share a similar structure that satisfies our assumptions of low maximum hyperedge and vertex degrees. We take as our base case the `voting100` hypergraph partitioned using 2 processors, and represent the problem size as the number of pins in the hypergraph. For higher values of p , we compute the required ideal problem size and choose the `voting` hypergraph with the number of pins that most closely matches this number. We stop at $p = 10$, since above this value

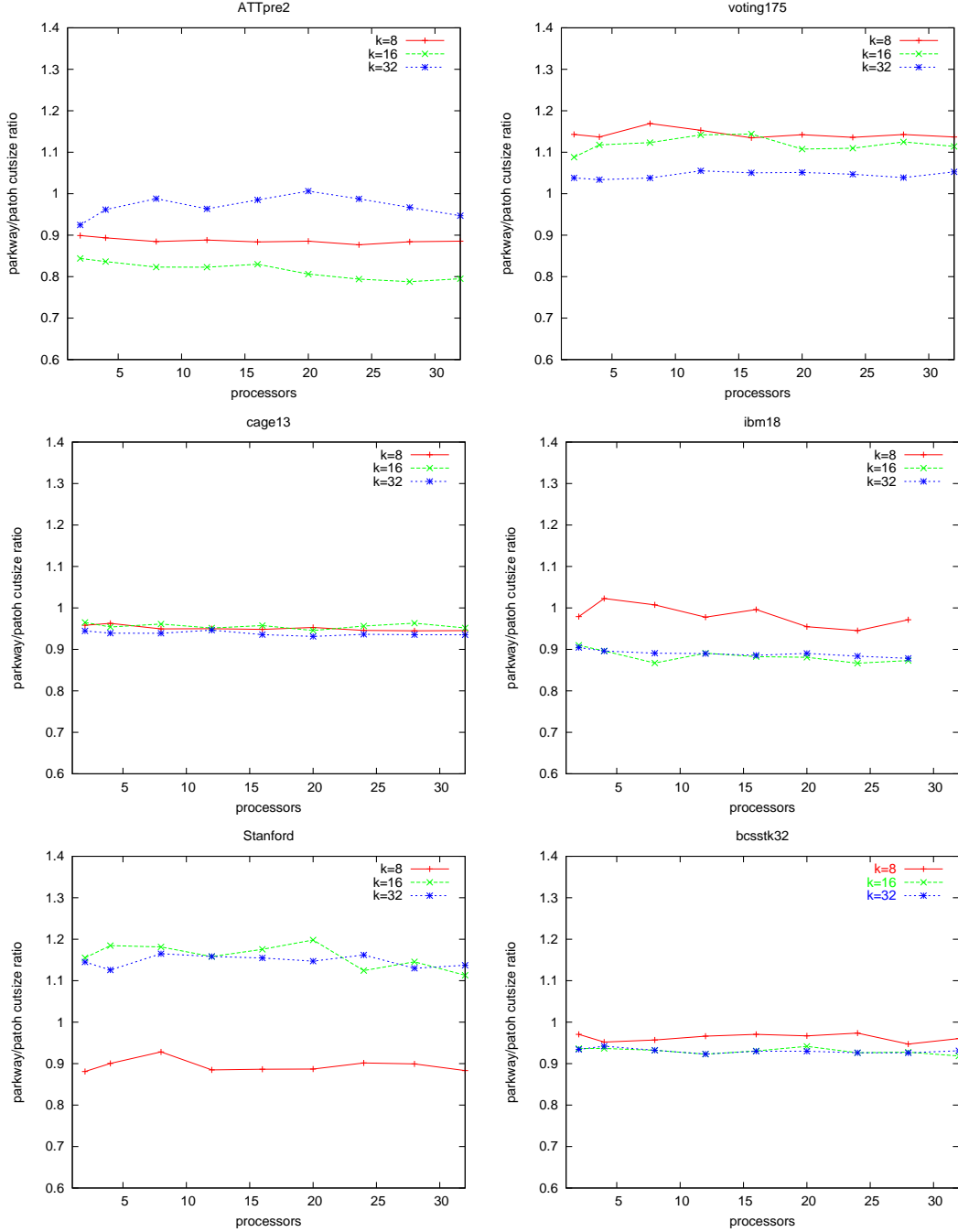


Fig. 2. Parkway2.1: plots of Parkway/PaToH partition quality ratio (in terms of $k - 1$ metric) against p .

the per-processor memory requirement becomes a limiting factor.

In order to compute parallel efficiencies on hypergraphs that could not be partitioned on a single processor, we approximate serial runtimes by fitting a linear regression model to a log-log plot of the observed serial PaToH runtimes, as shown in equation 18. Here η is the residual term representing the variation

Table 4

Processor efficiencies in experiments when the problem size is increased according to the isoefficiency function, given a prescribed increase in the number of processors.

	$p = 2$	$p = 4$	$p = 6$	$p = 8$	$p = 10$
hypergraph	voting100	voting200	voting250	voting300	voting350
ideal ψ_p	1	8	23	48	83
actual ψ_p	1	6.7	24	50	74
$E(k = 8)$	0.43	0.39	0.33	0.33	0.34
$E(k = 16)$	0.43	0.47	0.42	0.45	0.46
$E(k = 32)$	0.32	0.50	0.46	0.48	0.41

between $\log n$ and $\log T_s$ not explained by the linear model.

$$\log T_s = \alpha + \beta \log n + \eta \quad (18)$$

The R^2 value of this regression is well over 0.99, meaning that this model explains more than 99% of the variation between (the log of) PaToH runtimes and the (log of the) number of vertices in the hypergraph n . The full table of observed and extrapolated serial runtimes for $k = 8$, $k = 16$ and $k = 32$ are presented in Table 8 in the Appendix.

Table 4 presents the efficiencies observed for $k = 8$, $k = 16$ and $k = 32$ as the number of processors increases, given that the problem size is also increased according to our isoefficiency function. Here, the ‘‘actual ψ_p ’’ row refers to the actual increase in the problem size, given that we chose the **voting** hypergraph which best approximates the required increase. We note that efficiency remains relatively stable, as predicted by our scalability analysis.

4.4 Empirical Partition Quality Improvement Through Parallel Multi-phase Refinement

We conducted a further set of experiments to compare empirically the quality of partitions produced by the parallel multilevel algorithm implemented as in Section 4.3 with those produced by the parallel multilevel algorithm utilising parallel multi-phase refinement, as described in Section 3.5.

During the parallel uncoarsening phase, our implementation employed parallel multi-phase refinement at each multilevel step. Before each call to the parallel k -way refinement algorithm, a random permutation of vertices to processors was performed, as it was found to improve the performance of the parallel multi-phase refinement algorithm. Each processor generates a pseudorandom vector (whose length equals the number of locally stored vertices) of integers, each from 1 to p (inclusive). Each processor then sends the vertices to the processor given by the corresponding value in the random vector. The remainder of the coarsening and refinement parameters were implemented as in Section 4.3, and again partitions with balance criterion of 5% were sought.

Table 5

Parkway2.1 with parallel multi-phase refinement: runtime (in seconds) and partition quality ($k-1$ metric) on smaller hypergraphs. The figures in brackets are percentage cutsizes improvements over Parkway2.1 without parallel multi-phase refinement. The ‘-’ denote configurations of p and k that cannot be handled by our implementation of the parallel multi-phase refinement algorithm.

p	Partition Size(k)					
	8		16		32	
voting175	time	cutsizes	time	cutsizes	time	cutsizes
2	753.0	24 252(7.2%)	968.4	48 222(5.7%)	1 030	93 125(3.1%)
4	479.4	24 422(6.0%)	881.6	48 166(8.3%)	1 020	93 121(2.7%)
8	318.8	24 553(8.2%)	560.9	48 714(7.7%)	418.0	93 104(3.1%)
16	-	-	301.0	48 435(9.9%)	436.4	91 842(5.5%)
32	-	-	-	-	147.5	93 243(4.3%)
ATTpre2	time	cutsizes	time	cutsizes	time	cutsizes
2	155.0	8 684(-1.1%)	172.9	16 748(3.0%)	216.8	30 917(12%)
4	82.13	8 504(0.0%)	105.5	16 534(3.4%)	140.9	30 771(16%)
8	47.08	8 503(-1.0%)	62.26	16 440(2.4%)	108.4	30 634(18%)
16	-	-	49.45	16 095(4.7%)	81.73	30 162(19%)
32	-	-	-	-	64.89	28 675(20%)
cage13	time	cutsizes	time	cutsizes	time	cutsizes
2	1 392	177 736(3.6%)	1 798	256 145(3.6%)	2 875	352 296(3.4%)
4	1 129	181 881(1.9%)	1 688	257 594(1.9%)	2 511	349 765(3.5%)
8	1 065	176 894(3.2%)	1 425	254 349(3.9%)	2 379	350 416(3.3%)
16	-	-	1 381	253 155(4.0%)	2 102	350 881(2.8%)
32	-	-	-	-	1 672	351 058(2.7%)

Since vertices from the same part are allocated to the same processor when using the parallel multi-phase refinement algorithm, our implementation was restricted to configurations of p and k where k is an integer multiple of p to ensure a good load balance.

Table 5 shows the runtimes and partition cutsizes obtained using parallel multi-phase refinement for the `voting175`, `ATTpre2` and `cage13` hypergraphs. Across the hypergraphs, the average percentage improvements in cutsizes are 3.1%, 4.9% and 8.0% for $k = 8$, $k = 16$ and $k = 32$ respectively. We expect that the increase in the number of feasible partitions as partition size increases explains the greater observed average improvements. However, we observe that in a few cases, cutsizes actually degrades (due to the randomized nature of the algorithm); in addition the runtimes of the parallel algorithm with parallel multi-phase refinement are an order of magnitude slower than when multi-phase refinement is not used. Thus, except for situations where the quality of partition is the main objective and runtime is not a consideration, it is unlikely that parallel multi-phase refinement would be preferred to the “vanilla” k -way parallel refinement algorithm.

5 Conclusion and Future Work

In this paper, we have presented the first parallel algorithms for the hypergraph partitioning problem and have shown the scalability for certain classes of hypergraph under a theoretical performance model. Specifically, we have enabled large hypergraphs with small maximum vertex and hyperedge degrees to be partitioned in a scalable manner, something that had hitherto not been achieved. In the context of the multilevel paradigm, we presented a parallel coarsening algorithm based on the serial FC/HCC algorithms [38,13] and two parallel refinement algorithms, based on serial greedy k -way [38] and serial multi-phase [36] refinement respectively.

The algorithms were empirically evaluated on hypergraphs representing sparse matrix problems from the domains of DNA electrophoresis, analog circuits, structural engineering, VLSI circuit design, PageRank computations and performance analysis. We observed reasonable speedups on hypergraphs with small maximum hyperedge degrees and/or a high degree of structural locality, and partitions that were competitive with those produced by the state-of-the-art serial multilevel partitioner PaToH [16]. The isoefficiency study conducted on the `voting` family of hypergraphs validated our theoretical performance model.

Using the parallel multi-phase refinement algorithm instead of the parallel greedy k -way algorithm, we were able to achieve even better partition quality over the basic parallel k -way refinement algorithm (with average improvements of 3.1%, 4.9% and 8.0% for $k = 8$, $k = 16$ and $k = 32$ respectively), albeit at a significantly larger runtime cost.

There are a number of directions that we can see for future work. In particular, the algorithms in this paper are designed to be applied to general hypergraphs representing sparse irregular problems and as such have not been optimised for particular application domains. We anticipate that an improvement in runtime and also partition quality would be possible if the algorithms were tailored to take advantage of any inherent domain-specific structure in the hypergraphs to achieve a better data distribution among the processors and guide the coarsening and refinement algorithms. Secondly, it is clear that frontier hyperedge overhead is the main limiting factor in terms of the scalability of our algorithms. A different approach to data distribution could be helpful in this respect. While here we have considered a data distribution scheme analogous to one-dimensional row-wise decomposition for parallel sparse matrix-vector multiplication, concurrent work presented in [19] uses a two-dimensional distribution. Experimental results suggest that this distribution results in lower overheads when partitioning hypergraphs with larger maximum vertex and hyperedge degrees.

References

- [1] C.J. Alpert. The ISPD98 Circuit Benchmark Suite. In *Proc. International Symposium of Physical Design*, pages 80–85, April 1998.
- [2] 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.
- [3] C.J. Alpert and A.B. Kahng. Recent Directions in Netlist Partitioning. *Integration, the VLSI Journal*, 19(1–2):1–81, 1995.
- [4] C. Aykanat, A. Pinar, and U.V. Catalyurek. Permuting Sparse Rectangular Matrices into Block-Diagonal Form. *SIAM Journal on Scientific Computing*, 25(6):1860–1879, 2004.
- [5] 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.
- [6] C. Berge. *Hypergraphs*. North Holland, 1986.
- [7] B. Bollobás. *Combinatorics*. Cambridge University Press, 1986.
- [8] 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 2003.
- [9] 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.
- [10] T.N. Bui and C. Jones. Finding Good Approximate Vertex and Edge Partitions is NP-Hard. *Information Processing Letters*, 42(3):153–159, May 1992.
- [11] 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.
- [12] U.V. Catalyurek and C. Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix–Vector Multiplication. In *Lecture Notes in Computer Science*, volume 1117, pages 75–86, 1996.
- [13] U.V. Catalyurek and C. Aykanat. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999.
- [14] U.V. Catalyurek and C. Aykanat. A Fine-grain Hypergraph Model for 2D Decomposition. In *Proc. 15th IEEE International Parallel and Distributed Processing Symposium*, San Francisco, CA, 2001.

- [15] U.V. Catalyurek and C. Aykanat. A Hypergraph-partitioning Approach for Coarse-Grain Decomposition. In *Proc. ACM/IEEE Supercomputing*, Denver, 2001.
- [16] U.V. Catalyurek and C. Aykanat. *PaToH: Partitioning Tool for Hypergraphs, Version 3.0*, 2001.
- [17] 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.
- [18] T. Davis. University of Florida Sparse Matrix Collection, March 2005. <http://www.cise.ufl.edu/research/sparse/matrices>.
- [19] K.D. Devine, E. Boman, R. Heaphy, R. Bisseling, and U.V. Catalyurek. Parallel Hypergraph Partitioning for Scientific Computing. In *Proc. 20th IEEE International Parallel and Distributed Processing Symposium*, 2006.
- [20] K.D. Devine, E.G. Boman, R.T. Heaphy, B.A. Hendrickson, J.D. Teresco, J. Faik, J.E. Flaherty, and L.G. Gervasio. New Challenges in Dynamic Load Balancing. *Applied Numerical Mathematics*, 52(2–3):133–152, 2005.
- [21] N.J. Dingle, W.J. Knottenbelt, and P.G. Harrison. Uniformization and Hypergraph Partitioning for the Distributed Computation of Response Time Densities in Very Large Markov Models. *Journal of Parallel and Distributed Computing*, 64(8):908–920, August 2004.
- [22] I.S. Duff, R.G. Grimes, and J.G. Lewis. The Rutherford-Boeing sparse matrix collection. Technical Report TR/PA/97/36, CERFACS, Toulouse, France, 1997.
- [23] 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.
- [24] 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.
- [25] S. Dutt and H. They. Partitioning Around Roadblocks: Tackling Constraints with Intermediate Relaxations. In *Proc. 1997 IEEE/ACM International Conference on Computer-Aided Design*, pages 350–355, Nov 1997.
- [26] P. Feldmann, R. Melville, and D. Long. Efficient frequency domain analysis of large nonlinear analog circuits. In *Proc. IEEE Custom Integrated Circuits Conference*, Santa Clara, CA, 1996.
- [27] 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.
- [28] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

- [29] D. Gleich, L. Zhukov, and P. Berkhin. Fast parallel PageRank: A linear system approach. Technical Report YRL-2004-038, Institute for Computation and Mathematical Engineering, Stanford University, 2004.
- [30] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.
- [31] 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.
- [32] B.A. Hendrickson and T.G. Kolda. Graph Partitioning Models for Parallel Computing. *Parallel Computing*, 26:1519–1534, 2000.
- [33] B.A. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proc. ACM/IEEE Conference on Supercomputing*, 1995.
- [34] 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.
- [35] G. Karypis. Multilevel Hypergraph Partitioning. Technical Report #02-25, University of Minnesota, 2002.
- [36] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on VLSI Systems*, 7(1):69–79, 1999.
- [37] G. Karypis and V. Kumar. *hMeTiS: A Hypergraph Partitioning Package, Version 1.5.3*. University of Minnesota, 1998.
- [38] G. Karypis and V. Kumar. Multilevel k -way Hypergraph Partitioning. Technical Report #98-036, University of Minnesota, 1998.
- [39] G. Karypis and V. Kumar. A Parallel Algorithm for Multilevel Graph Partitioning and Sparse Matrix Ordering. *Journal of Parallel and Distributed Computing*, 48:71–95, 1998.
- [40] 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.
- [41] G. Karypis, K. Schloegel, and V. Kumar. *ParMeTiS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.0*. University of Minnesota, 2002.
- [42] B.W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49:291–307, February 1970.
- [43] W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College, London, United Kingdom, 2000.
- [44] B. Krishnamurthy. An Improved min-cut Algorithm for Partitioning VLSI Networks. *IEEE Transactions on Computers*, 33(C):438–446, 1984.

- [45] L.A. Sanchis. Multiple-way Network Partitioning with Different Cost Functions. *IEEE Transactions on Computers*, 42(22):1500–1504, 1993.
- [46] 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.
- [47] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1998.
- [48] A. Trifunovic and W. Knottenbelt. Parkway2.0: A Parallel Multilevel Hypergraph Partitioning Tool. In *Proc. 19th International Symposium on Computer and Information Sciences*, volume 3280 of *Lecture Notes in Computer Science*, pages 789–800. Springer, 2004.
- [49] A. Trifunovic 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.
- [50] B. Ucar 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.
- [51] 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.

6 Appendix

Table 6

Parkway2.1 (for $p > 1$) and PaToH (for $p = 1$) runtimes (in seconds) and partition quality ($k-1$ metric) for ATTPre2, voting175 and cage13 hypergraphs. All results are averages over ten runs.

p	Partition size(k)					
	8		16		32	
ATTPre2	time	cutsz	time	cutsz	time	cutsz
1	28.7	9564	37.3	19331	46.3	37217
2	63.9	8598	66.4	17270	76.3	35008
4	29.7	8545	31.3	17111	38.7	36416
8	14.1	8460	15.0	16839	20.4	37410
12	8.85	8496	9.74	16836	14.6	36480
16	6.71	8452	7.54	16981	12.2	37304
20	5.69	8470	6.40	16498	11.3	38098
24	9.07	8385	9.76	16248	14.6	37395
28	9.59	8456	10.2	16121	14.8	36611
32	8.95	8469	9.82	16264	13.7	35858
voting175	time	cutsz	time	cutsz	time	cutsz
1	44.4	22864	59.0	47000	72.3	92555
2	55.3	26135	61.6	51139	75.1	96078
4	28.9	25991	32.1	52531	39.0	95679
8	14.9	26735	17.3	52787	22.7	96072
12	11.4	26357	13.7	53662	19.3	97654
16	9.1	25942	11.0	53766	15.7	97204
20	7.8	26120	10.2	52053	14.8	97307
24	6.8	25970	9.5	52154	13.7	96898
28	6.9	26127	8.5	52869	13.6	96160
32	6.7	25990	8.8	52363	13.9	97428
cage13	time	cutsz	time	cutsz	time	cutsz
1	94.1	192506	118	275262	139	385922
2	300	184435	355	265638	470	364558
4	207	185317	264	262630	323	362390
8	147	182751	185	264586	250	362355
12	118	182878	164	261919	240	365280
16	112	182544	159	263569	230	361083
20	111	183431	155	260250	219	359392
24	106	182042	148	263237	212	361352
28	106	181889	146	265108	217	360998
32	102	181974	141	262001	225	360909

Table 7

Parkway2.1 (for $p > 1$) and PaToH (for $p = 1$) runtimes (in seconds) and partition quality ($k-1$ metric) for ibm18, Stanford and bcsstk32 hypergraphs. All results are averages over ten runs.

p	Partition size(k)					
	8		16		32	
ibm18	time	cutsz	time	cutsz	time	cutsz
1	11.4	7 625	13.6	13 176	15.2	19 329
2	11.0	7 466	14.7	11 988	22.4	17 487
4	8.24	7 799	11.1	11 810	16.1	17 321
8	6.51	7 681	8.64	11 425	11.8	17 218
12	6.34	7 455	8.09	11 737	11.1	17 200
16	6.31	7 596	8.08	11 627	11.2	17 121
20	5.92	7 279	7.50	11 607	10.8	17 208
24	5.60	7 208	7.55	11 419	10.8	17 078
28	5.08	7 407	7.52	11 503	10.5	16 977
32	5.23	7 520	7.54	11 633	10.7	17 300
Stanford	time	cutsz	time	cutsz	time	cutsz
1	29.2	37 147	32.3	71 545	40.5	140 492
2	20.3	32 724	29.3	82 642	55.3	160 882
4	14.9	33 450	22.3	84 749	43.4	158 175
8	10.8	34 482	17.5	84 525	27.4	163 690
12	10.7	32 872	17.0	82 855	28.1	162 741
16	11.3	32 928	18.1	84 106	31.2	162 232
20	14.6	32 947	21.4	85 727	34.7	161 170
24	15.5	33 496	22.2	80 448	36.5	163 276
28	15.7	33 404	24.7	81 962	37.4	158 733
32	17.0	32 807	24.3	79 629	37.7	159 784
bcsstk32	time	cutsz	time	cutsz	time	cutsz
1	5.20	4 569	7.10	8 215	8.80	13 154
2	2.75	4 435	4.25	7 692	6.26	12 288
4	1.48	4 350	2.35	7 693	3.47	12 392
8	1.11	4 373	1.91	7 657	2.98	12 262
12	1.08	4 415	1.85	7 581	2.86	12 140
16	1.08	4 435	1.62	7 644	2.75	12 231
20	1.17	4 417	1.91	7 635	2.91	12 231
24	1.16	4 448	1.85	7 608	2.90	12 184
28	1.11	4 327	1.75	7 621	2.94	12 181
32	1.19	4 388	1.94	7 545	2.87	12 248

Table 8

PaToH runtimes on the voting hypergraphs. For voting100 to voting200 the runtimes are averages over ten runs. Remaining runtimes are approximated by a log-log least squares regression over the observed runtimes. The regression α is the zero intercept and the regression β is the slope of the line. The quantities in brackets denote the standard errors of the parameters.

Hypergraph	$k = 8$	$k = 16$	$k = 32$
regression R^2	0.999	0.999	0.997
regression α	-12.22(0.26)	-12.24(0.31)	-12.08(0.50)
regression β	1.146(0.02)	1.168(0.02)	1.171(0.04)
voting100	7.90	10.1	12.2
voting125	18.0	23.2	27.6
voting150	28.4	37.5	46.0
voting175	44.4	59.0	72.3
voting200	65.1	86.5	107
voting250	246	339	417
voting300	578	810	997
voting350	899	1 270	1 570