Par$k$way 2.0: A Parallel Multilevel Hypergraph Partitioning Tool

Aleksandar Trifunovic and William J. Knottenbelt

Department of Computing, Imperial College London,
South Kensington Campus, London SW7 2AZ, UK
{at701,wjk}@doc.ic.ac.uk

Abstract. We recently proposed a coarse-grained parallel multilevel algorithm for the k-way hypergraph partitioning problem. This paper presents a formal analysis of the algorithm's scalability in terms of its isoefficiency function, describes its implementation in the Par$k$way 2.0 tool and provides a run-time and partition quality comparison with state-of-the-art serial hypergraph partitioners.
The isoefficiency function (and thus scalability behaviour) of our algorithm is shown to be of a similar order as that for Kumar and Karypis' parallel multilevel graph partitioning algorithm. This good theoretical scalability is backed up by empirical results on hypergraphs taken from the VLSI and performance modelling application domains. Further, partition quality in terms of the k-1 metric is shown to be competitive with the best serial hypergraph partitioners and degrades only minimally as more processors are used.

1 Introduction

A hypergraph generalises a graph, such that hyperedges of a hypergraph connect arbitrary, non-empty, sets of vertices. Like graphs, hypergraphs can be used to represent the structure of sparse irregular problems such as data dependencies in distributed databases and component connectivity in VLSI circuits. Hypergraphs may also be partitioned such that a cut metric (a function of the interconnect in a partition) is minimised subject to a load balancing criterion. Hypergraph cut metrics provide a more accurate model than graph partitioning in many cases of practical interest such as the row-wise decomposition of a sparse matrix for parallel matrix–vector multiplication [4].

Algorithms for serial hypergraph partitioning have been studied extensively [9, 2, 14] and tool support exists (e.g. hMeTiS [13] and PaToH [4]). However, these are limited by the computing power and memory available on a single processor. Recently, we proposed the first parallel hypergraph partitioning algorithm [21]. However, while capacity was significantly improved, absolute run times and scalability were poor and partition quality was highly dependent on the structure of the input hypergraph. In [20] we proposed a new coarse-grained algorithm which improved processor utilisation and removed the structural dependency.

In this paper, we introduce an analytical performance model for the asymptotic run time complexity of the new parallel algorithm, derive its isoefficiency function and perform an empirical evaluation of the algorithm's implementation in the tool Par*k*way 2.0. We consider example hypergraphs from two application domains and compare the performance of Par*k*way 2.0 with that of two state-of-the-art serial partitioners in terms of run time and partition quality.

The remainder of this paper is organised as follows. Section 2 outlines serial multilevel hypergraph partitioning. Section 3 describes the parallel algorithm and its scalability analysis. Section 4 presents the experimental evaluation. Section 5 concludes and considers future work.

2 Serial Multilevel Hypergraph Partitioning

A hypergraph $H(V, E)$ is defined as follows. Let V be the set of vertices and E the set of hyperedges, where each hyperedge $e_i \in E$ is a non-empty subset of the vertex set V. The map $f_w : V \rightarrow \mathbf{Z}$ assigns an integer weight w_i to every vertex $v_i \in V$ and the map $f_c : E \rightarrow \mathbf{Z}$ assigns a cost c_i to each hyperedge $e_i \in E$. The *size* of a hyperedge is defined as its cardinality. The sum of the sizes of the hyperedges in a hypergraph, denoted here by m, is referred to as the number of *pins* in the hypergraph.

We formally define the k-way partitioning problem as follows. The goal is to find k disjoint subsets (or parts) V_i, $(i = 0, \ldots, k-1)$ of the vertex set V with corresponding part weights W_i (given by the sum of the constituent vertex weights), such that, given a prescribed balance criterion $0 < \epsilon < 1$,

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

holds $\forall i = 0, \ldots, k-1$ and an objective function over the hyperedges is minimized. Here W_{avg} denotes the average part weight. If the objective function is the *hyperedge cut* metric, then the partition cost (or cut-size) is given by the sum of the costs of hyperedges that span more than one part. Alternatively, when the objective function is the k-1 metric, the partition cost is given by

$$P_{cost} = \sum_{i=0}^{|E|-1} (\lambda_i - 1)c_i \quad (2)$$

where λ_i is the number of parts spanned by hyperedge e_i. Computing the optimal bisection of a hypergraph under the hyperedge cut metric (and hence the k-1 metric since $k = 2$ for a bisection) is known to be NP-complete [10]. Thus, research has focused on developing polynomial-time heuristic algorithms resulting in good sub-optimal solutions. The k-way partition is typically computed either directly or by recursive bisection. As it scales well in terms of run time and solution quality with increasing problem size, the *multilevel* paradigm is preferred to solely *flat* approaches because the likelihood of flat heuristic algorithms converging to poor local minima rises significantly with increasing problem size. Flat heuristic algorithms such as spectral bisection and simulated annealing methods are reviewed in more detail in [2]. Note that flat approaches can also be used at the coarsest levels of the multilevel framework. The following subsections briefly describe the main phases of the multilevel paradigm.

2.1 The Coarsening Phase

The original hypergraph is approximated via a succession of smaller hypergraphs that maintain its structure as accurately as possible. A single coarsening step is performed by merging the vertices of the original hypergraph together to form vertices of the coarse hypergraph, denoted by a map $f_{merge} : V \to V_{coarse}$, where

$$\frac{|V|}{|V_{coarse}|} = r, \, r > 1 \tag{3}$$

and r is the prescribed reduction ratio. The map f_{merge} is used to transform the hyperedges of the original hypergraph to the hyperedges of the coarse hypergraph. Single vertex hyperedges in the coarse hypergraph are discarded as they cannot contribute to the cut-size of a partition of the coarse hypergraph. When multiple hyperedges map onto the same hyperedge of the coarse hypergraph, only one of the hyperedges is retained, with its cost set to be the sum of the costs of the hyperedges that mapped onto it (thus preserving the cut-size properties of the original hypergraph). Coarsening algorithms are discussed in detail in both [2] and [12].

2.2 The Initial Partitioning Phase

The coarsest hypergraph is partitioned using a flat partitioning method such as an iterative improvement algorithm. As the coarsest hypergraph is significantly smaller than the original hypergraph, flat partitioning methods are computationally feasible and the time taken to compute the initial partition is usually considerably less than the time taken by the other phases of the multilevel pipeline. Since heuristic algorithms are typically used, the best solution out of a number of runs is chosen as the starting point for the uncoarsening phase.

2.3 The Uncoarsening Phase

The initial partition is propagated up through the successively finer hypergraphs and at each step the partition is further refined using heuristic refinement techniques. When the k-way partition is computed via recursive bisection, the refinement phase consists of bisection refinement, typically based on the Fiduccia-Mattheyses (FM) algorithm [9]. Conversely, when it is computed directly, the *greedy refinement* algorithm [14] has been shown to perform well, especially with increasing values of k. Both refinement algorithms typically converge within a few *passes*, during each of which each vertex is moved at most once. More sophisticated refinement algorithms have been developed, motivated by the idea of escaping from poor local minima [18, 6–8, 12, 3].

3 Parallel Algorithm and Performance Model

This section briefly reviews our parallel multilevel partitioning algorithm and then presents its analytical performance model. Since the algorithms that make up the multilevel pipeline are inherently serial in nature, we sought a coarse-grained formulation of the multilevel k-way partitioning algorithm [14]. This was chosen over the recursive bisection algorithm as the k-way refinement algorithm has better opportunities for concurrency than variants of the FM bisection refinement, where ways to perform the gain update calculations in parallel are not readily apparent.

Only the coarsening and refinement phases are parallelised since the coarsest hypergraph should be small enough for the initial partition to be rapidly computed serially. Multiple runs of the initial partitioning algorithm are carried out on the available processors in parallel with the best partition selected for further refinement. The initial partitioning phase commences when the coarsest hypergraph has approximately $100 \times k$ vertices.

3.1 Data Distribution

If p denotes the number of processors, we store the hypergraph across the processors by storing $|E|/p$ hyperedges and $|V|/p$ vertices on each processor. We assume that each processor initially stores a set of contiguous vertices (in terms of their indices), although a random initial allocation of vertices to processors can be supported via a reassignment of vertex indices. In addition, for each of these vertices the processor stores the index of the corresponding vertex in the coarse hypergraph and the part index of the vertex.

With each hyperedge we associate a b-bit hash-key, computed using a variant of the load balancing hash-function f_2 from [17]. It has the desirable property that $f_2(e) \ \mathtt{mod}\ p$ is near-uniformly distributed, independent from the input hyperedge e. Consequently, to ensure an even spread of hyperedges across the processors, each hyperedge e resides on the processor given by $f_2(e) \ \mathtt{mod}\ p$. To calculate the probability of collision, assume that f_2 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. Then,

$$\begin{aligned} \mathbb{P}(C(N) \geq 1) &= 1 - \mathbb{P}(C(N) = 0) && (4) \\ &= 1 - \frac{M!}{(M-N)!M^N} && (5) \\ &\leq e^{\frac{-N^2}{2M}} && (6) \end{aligned}$$

if $N^2 << M$, as shown in [17]. We have $b = 64$ and $N = |E|$. This ensures that the probability of collisions occurring is remote – for example, when $|E| = 10^8$, $\mathbb{P}(C(N) \geq 1) \leq 0.0003$ – and thus facilitates rapid hyperedge comparison.

At the beginning of every multilevel step, each processor assembles the set of hyperedges that are adjacent to its locally held vertices using an all-to-all personalised 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. Frontier hyperedges may be replicated on multiple processors, but only for the hypergraph used in the current multilevel step. Experience suggests that the memory overhead incurred by duplicating frontier hyperedges is modest (see Table 3).

3.2 Parallel Coarsening Phase

We parallelised the *First Choice* (FC) [14] serial coarsening algorithm. Briefly, the serial algorithm proceeds as follows. The vertices of the hypergraph are visited in a random order. For each vertex v_i, all vertices (both those already matched and those unmatched) that are connected via hyperedges incident on v_i are considered for matching with v_i. A connectedness metric is computed between pairs of vertices and the most strongly connected vertex to v_i is chosen for the matching, provided that the resulting cluster does not exceed a prescribed maximum weight. This condition is imposed to prevent a large imbalance in vertex weights in the coarsest hypergraph.

In parallel, each processor i traverses the local vertex set V_i in random order, computing the vertex matchings as prescribed by the FC algorithm. Each processor also maintains request sets to the $p-1$ remote processors. If the best match for a local vertex v becomes a vertex w stored on processor j, $i \neq j$, then the vertex v is placed into the request set $S_{i,j}$. If another local vertex subsequently chooses v or w as its best match then it is also added to the request set $S_{i,j}$. 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. Eq. 3). When computing the cardinality of the local coarse vertex set, we include the potential matches with vertices from other processors.

Each processor i then communicates its request sets to the other processors, including the weights of the vertices that are involved in the matching request. The processors concurrently decide to accept or reject matching requests from other processors. Denote by $M^w_{i,j}$ the set of vertices (possibly consisting of a single vertex) from the remote processor i that seeks to match with a local vertex w stored on processor j (thus, $S_{i,j} = \bigcup_x M^x_{i,j}$). Processor j considers these sets for each of its requested local vertices in turn, handling them as follows:

1. If w is unmatched, matched locally or already matched remotely, then a match with $M^w_{i,j}$ is granted to processor i if the weight of the combined cluster (including vertices already matched with w) does not exceed the maximum allowed coarse vertex weight.
2. If w has been sent to a processor l, $l \neq i$, as part of a request for another remote match, then processor j informs processor i that the match with $M^w_{i,j}$ has been rejected. This is necessary since granting this match may otherwise result in a coarse vertex that exceeds the maximum allowed coarse vertex weight, if the remote match of w with a vertex on processor l is granted. When informed of the rejection by processor j, processor i will locally match the set $M^w_{i,j}$ into a single coarse vertex.

In order to enable a match between two vertices on remote processors that make requests to each other, we communicate the request sets in two stages. In the first stage, processor i communicates request sets $S_{i,j}$ to processor j and receives replies to its requests from j if $i > j$, while in the second stage processor i communicates request sets $S_{i,j}$ to processor j and receives replies to its requests from j if $i < j$. Note that only the combined weight of the vertices in $M^w_{i,j}$ and the index of vertex w need to be communicated from processor i to processor j, further reducing the communication requirements. The sets $M^w_{i,j}$ are received as an array on processor j and are processed in random order.

The coarsening step is completed by contracting the hyperedges of the finer hypergraph onto the hyperedges of the coarse hypergraph. Each processor contracts the $|E|/p$ locally stored hyperedges. The matching vector values for vertices not stored locally are assembled using an all-to-all personalised communication. The removal of duplicate coarse hyperedges on remote processors and load balancing is done as follows. Processors communicate each hyperedge e and its cost to the destination processor given by $f_2(e) \texttt{ mod } p$. Each processor retains distinct hyperedges, setting their cost to be the sum of the costs of their respective duplicates (if any). The parallel coarsening step concludes with a load-balancing communication of coarse vertices such that each processor has $|V_{coarse}|/p$ local vertices at the start of the subsequent coarsening step.

3.3 Parallel Uncoarsening Phase

Firstly, the partition of the coarse hypergraph is used to initialise the partition of the finer hypergraph. Processors scan the local vertex list of the finer hypergraph and if the part index value of the corresponding coarse vertex is not available, it is requested from the relevant processor. Our parallel refinement algorithm then proceeds in passes; however, instead of moving single vertices across a partition boundary as in the serial algorithm, the parallel algorithm moves sets of vertices. The processors traverse the local vertex set in random order and compute the best move for each vertex. The best moves resulting in positive gain are maintained in sets $U_{i,j}$, $i \neq j$, $i, j = 0, \ldots, k-1$, where i and j denote current and destination parts respectively. In order to prevent vertex thrashing, 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. The balance constraint on part weights (cf. Eq. 1) is maintained as follows. At the beginning of each of the two stages, the processors know the exact part weights and maintain the balance constraint during the local computation of the sets $U_{i,j}$. The associated weights and gains of all the non-empty sets $U_{i,j}$ are communicated to the root processor which then determines the actual partition balance that results from the moves of the vertices in the sets $U_{i,j}$. If the balance criterion 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. Currently, 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 are not explicitly moved between processors; rather, their part index value is changed by the processor that stores the vertex.

3.4 Analytical Performance Model

Suppose that $|V| = n$ and $|E| = \Theta(n)$. Let h and d denote the average hyperedge size and the average vertex degree of the original hypergraph respectively. In our analysis, we assume that $h << n$, $d << n$ and that the numbers of vertices and hyperedges are respectively reduced by constant factors $1+\upsilon$ and $1+\omega$ $(\omega, \upsilon > 0)$ at each coarsening step. We consider the computation and the communication requirements in turn, assuming $O(\log n)$ coarsening steps.

During each coarsening step, $O(dh)$ computation steps are performed for matching each vertex and $O(h \log h)$ computation steps in contracting each hyperedge. Once a coarse hyperedge is constructed, checking for local duplicate hyperedges is done using a hash table. It takes $O(h)$ steps to check for and resolve a possible collision if a duplicate key is found in the table. Thus, the computation requirement during each coarsening step is $O(n/p)$. At the initial partitioning phase, the hypergraph has size $O(k)$ and can be partitioned in $O(k^2)$ time. During each pass of a refinement step, we compute the vertex gains concurrently and then compute rebalancing moves on the root processor if required. In order to compute the gains for a vertex, we need to visit all the hyperedges incident on each vertex and determine the connectedness to the source and destination parts. These computations have complexity $O(n/p)$ per pass. The rebalancing computation has complexity $O(pk^2)$. As the number of passes during a refinement stage is a small constant, the overall asymptotic computational complexity is given by

$$
\begin{aligned}
T_{comp} &= O(n/p)\left(\sum_{i=0}^{\log n}\frac{1}{(1+\upsilon)^i} + \sum_{i=0}^{\log n}\frac{1}{(1+\omega)^i}\right) + O(pk^2\log n) && (7)\\
&\le O(n/p)\left(\sum_{i=0}^{\infty}\frac{1}{(1+\upsilon)^i} + \sum_{i=0}^{\infty}\frac{1}{(1+\omega)^i}\right) + O(pk^2\log n) && (8)\\
&\le O(n/p) + O(pk^2\log n) && (9)
\end{aligned}
$$

In the following communication cost analysis, we assume the underlying parallel architecture to be a p-processor hypercube. During both the coarsening and refinement stages the hyperedges adjacent to the locally held vertices are assembled at each processor using an all-to-all personalised communication. The required matching vector entries during coarsening and the required entries of the partition vector during refinement are assembled in the same fashion. We will compute an average-case time for hyperedge communication. As each processor stores $O(n/p)$ vertices, it requires $O(n/p)$ adjacent hyperedges (since d is a small constant). Thus, each remote processor will on average contribute $O(n/p^2)$ of its hyperedges, resulting in message size of $O(n/p^2)$ in the all-to-all personalised communication. This is a reasonable assumption since the hash function scatters the hyperedges randomly across the processors with a near-uniform distribution. An all-to-all personalised communication with this message size can be performed in $O(n/p)$ time [11]. During coarsening, 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 cut-size, 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 overall computational complexity, we deduce that the overall asymptotic communication cost is

$$T_{comm} = O(n/p) + O(k^2 \log p \log n) \tag{10}$$

Eliminating dominated terms from equations 9 and 10, the parallel run time of the multilevel partitioning algorithm is

$$T_p = O(n/p) + O(pk^2 \log n) \tag{11}$$

As the complexity of the serial algorithm is $O(n)$, we deduce that the isoefficiency function is $W = O(k^2 p^2 (\log p + \log k))$. Thus, if the number of processors is doubled and the number of parts is kept constant, the input problem size must increase by a factor of just over 4 to maintain a given level of efficiency. This isoefficiency function is of the same order as that given in [15] for the parallel graph partitioning algorithm implemented in the ParMeTiS tool [16].

4 Experimental Results

4.1 Implementation and Test Environment

The three phases of our parallel multilevel k-way partitioning algorithm were implemented in C++ using the Message Passing interface (MPI) standard [19], thus forming the Par$k$way 2.0 tool. It is an optimised version of the first Par$k$way implementation [20]. Par$k$way 2.0 interfaces with the `HMETIS_PartKway()` routine from the hMeTiS [13] library for the initial partitioning phase when the coarsest hypergraph from the parallel coarsening phase has less than $100 \times k$ vertices. The best partition obtained by p serial runs of `HMETIS_PartKway()` in parallel is then passed to the parallel uncoarsening phase.

Base-case serial comparison was provided by the state-of-the-art hypergraph partitioning tools khMeTiS [13] and PaToH [5]. Like Par$k$way 2.0, khMeTiS is a direct k-way partitioner implementing the `HMETIS_PartKway()` routine and thus can be compared fairly with our tool. For comparison with the recursive bisection algorithm, PaToH was preferred to the recursive bisection variant hMeTiS [13] because it produced partitions of comparable quality at significantly faster run times.

Table 1. Characteristics of the test hypergraphs

Hypergraph	#vertices	#hyperedges	#pins	min	max	avg	variance
Voting 175	1 140 050	1 140 050	6 657 722	2	7	5.84	3.37
ibm 16	183 484	190 048	778 823	2	40	4.10	13.06
ibm 17	185 495	189 581	860 036	2	36	4.54	16.57
ibm 18	210 613	201 920	819 617	2	66	4.06	15.71

The architecture used in all the experiments consisted of a Beowulf Linux Cluster with 64 dual-processor nodes (although we only had access to a 32-processor partition due to configuration limitations and high machine utilisation). Each node has two Intel Xeon 2.0GHz processors and 2GB of RAM. The nodes are connected by a Myrinet network with a peak throughput of 250 MB/s.

4.2 Empirical Evaluation

We evaluated our parallel algorithm on hypergraphs from the domain of performance modelling and VLSI circuit design. **Voting 175** is the hypergraph representation of a transition matrix derived from a high-level semi-Markov model of a voting system with 175 voters. It has an almost lower-triangular structure typical of transition matrices from the domain of performance modelling [21, 17]. The three largest hypergraphs from the ISPD98 Circuit Benchmark Suite [1] (**ibm16–ibm18**) were also used in the evaluation. The main characteristics of the test hypergraphs are shown in Table 1, where **min** and **max** denote the minimum and maximum hyperedge length respectively while **avg** and **variance** denote the average and variance of hyperedge length. We sought partitions with a 5% imbalance according to Eq. 1. When computed by recursive bisection using the PaToH tool, this meant that the maximum imbalance factor on each bisection was set to $(1.05/k)^{1/\log_2 k} - 0.5$ in order to enforce the 5% balance criterion in the final partition. Since the k-1 metric was evaluated, we set the partitioning objective to SOED (sum of external degrees) in khMeTiS and `HMETIS_PartKway()` [14] while for PaToH we used settings for sparse matrices or VLSI hypergraphs as appropriate [5]. The V-Cycle feature was turned off for all experimental runs as it was observed to provide only a marginal increase in partition quality at a large run time cost. The coarsening reduction ratio from Eq. 3 was set to 2.0 in the Par$k$way 2.0 tool. The results were averaged over ten runs for each parameter configuration. Table 2 presents the results of our experiments. The parallel implementation achieves a ten-fold speedup over the fastest serial time on the larger Voting 175 hypergraph with 32 processors, as seen in Fig. 1. We observe absolute speedups over the PaToH base-case when four or more processors are used and a near-linear speedup trend as the number of processors increases. The latter supports the scalability behaviour predicted by our analytical performance model. On the VLSI hypergraphs, good speedups are harder to achieve because the communication overhead in the parallel algorithm is more significant given the small problem sizes. However, in general, absolute run times decrease as p increases. In terms of partition quality, we note that our parallel algorithm outperforms the serial partitioners on almost all the VLSI benchmark

Table 2. Partitioning results for four sample hypergraphs. Here p is the number of processors used and time is the average of ten serial/distributed run times

		Partition Size					
p	**Tool**	**8**		**16**		**32**	
ibm16		**time (s)**	**cut-size avg (best)**	**time (s)**	**cut-size avg (best)**	**time (s)**	**cut-size avg (best)**
1	PaToH	8.20	10 036 (9 381)	9.67	15 565 (14 536)	10.77	22 906 (22 394)
1	*k*hMeTiS	8.66	8 651 (7 696)	11.58	13 719 (13 214)	15.52	20 713 (20 216)
2	Par*k*way 2.0	10.17	8 472 (8 125)	12.23	13 882 (13 639)	17.52	21 605 (21 114)
4	Par*k*way 2.0	6.71	8 221 (7 931)	10.02	13 813 (13 305)	13.16	21 250 (20 979)
8	Par*k*way 2.0	4.96	8 357 (7 865)	7.70	13 615 (13 269)	13.50	20 961 (20639)
16	Par*k*way 2.0	5.07	7 974 (7 713)	7.37	13 374 (13 245)	10.35	20 747 (20 388)
ibm17		**time (s)**	**cut-size avg (best)**	**time (s)**	**cut-size avg (best)**	**time (s)**	**cut-size avg (best)**
1	PaToH	9.62	12 731 (11715)	11.74	19 586 (18 774)	13.37	27 597 (26 959)
1	*k*hMeTiS	11.02	13 181 (12 636)	14.99	19 797 (18 700)	20.57	28 476 (27 567)
2	Par*k*way 2.0	11.26	12 194 (11767)	14.63	19 390 (19 059)	20.76	25 932 (25 448)
4	Par*k*way 2.0	7.40	12 173 (11 846)	11.71	18 669 (17 978)	15.82	26 164 (26 013)
8	Par*k*way 2.0	5.63	11 834 (11 489)	9.30	18 574 (18 188)	15.96	25 696 (25 453)
16	Par*k*way 2.0	6.11	11 477 (11411)	8.88	18 661 (18 336)	13.20	25 649 (25 328)
ibm18		**time (s)**	**cut-size avg (best)**	**time (s)**	**cut-size avg (best)**	**time (s)**	**cut-size avg (best)**
1	PaToH	8.76	12 169 (11 415)	10.34	17 340 (16 318)	11.79	23 252 (22 454)
1	*k*hMeTiS	9.91	7 973 (7 465)	12.93	12 084 (11 190)	17.75	18 271 (17 510)
2	Par*k*way 2.0	11.15	8 440 (7 806)	13.64	13 544 (11 947)	20.05	18 307 (17 803)
4	Par*k*way 2.0	7.58	8 076 (7 677)	9.33	12 760 (12 213)	14.58	17 445 (17 078)
8	Par*k*way 2.0	5.36	8 376 (7 560)	7.90	11 806 (11 430)	10.99	17 845 (17 226)
16	Par*k*way 2.0	5.48	7 181 (6 837)	7.63	11 317 (11 096)	10.55	17 007 (16 858)
Voting 175		**time (s)**	**cut-size avg (best)**	**time (s)**	**cut-size avg (best)**	**time (s)**	**cut-size avg (best)**
1	PaToH	41.40	22 863 (22 191)	54.30	46 496 (45 960)	67.13	93 045 (92 654)
1	*k*hMeTiS	53.76	25 387 (24 600)	58.39	50 588 (49 246)	67.92	95 072 (94 352)
2	Par*k*way 2.0	66.30	26 227 (25 605)	74.60	52 876 (51 673)	90.57	97 715 (97 043)
4	Par*k*way 2.0	30.41	26 230 (25 785)	35.78	53 031 (52 313)	46.18	98 201 (97 819)
8	Par*k*way 2.0	14.66	26 406 (26 160)	17.37	53 207 (52 973)	30.81	97 534 (96 832)
16	Par*k*way 2.0	6.57	26 671 (26 548)	9.68	53 013 (52 160)	19.93	98 000 (97 078)
32	Par*k*way 2.0	4.10	26 570 (25 786)	5.55	53 411 (52 679)	11.18	98 082 (97 217)

Table 3. Maximum number of hypergraph pins on a processor after hyperedges adjacent to local vertices have been assembled

	Number of Processors					
Hypergraph	**1**	**2**	**4**	**8**	**16**	**32**
Voting 175	6 657 722	3 369 040	1 718 498	886 076	469 206	260 828
ibm 16	778 823	692 001	539 420	377 743	243 514	-
ibm 17	860 036	776 561	622 477	445 314	291 184	-
ibm 18	819 617	734 599	571 752	403 543	263 631	-

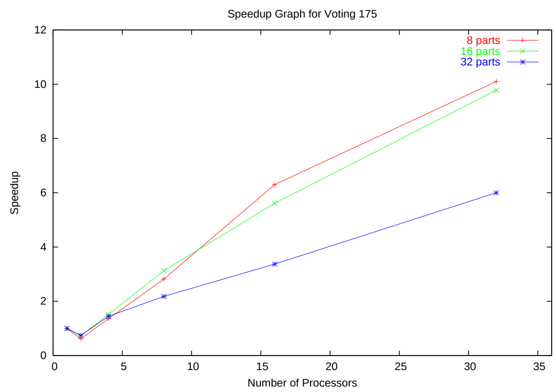

Fig. 1. Speedup results on Voting 175 hypergraph using PaToH as base-case.

hypergraphs. This may be because Par$k$way 2.0 utilises all processors during the initial partitioning phase, enabling it to select the best quality partition from many more candidate runs than is possible on a single processor. In addition, this superior initial partition quality helps to maintain overall partition quality as the number of processors is increased. Hypergraphs arising from the performance modelling domain exhibit regularity and are more amenable to recursive bisection (divide and conquer) solution methods than the more irregular VLSI benchmark hypergraphs. This may explain the slightly higher partition quality achieved by PaToH over the direct k-way partitioners. We note that the partition quality of Par$k$way 2.0 is comparable to that produced by khMeTiS for the Voting 175 model. Finally, Table 3 shows the maximum number of hypergraph pins per processor after frontier hyperedges adjacent to local vertices have been assembled in the multilevel steps involving the original hypergraph.

5 Conclusion

This paper has presented an analytical performance model for our recently proposed parallel multilevel hypergraph partitioning algorithm. By deriving the isoefficiency function from the performance model we have shown that our algorithm is scalable in a technically correct sense. This has been empirically confirmed by running our parallel tool on hypergraphs taken from two different application domains.

In the future, we aim to apply our tool to an even wider range of application domains, for example bioinformatics and computational grids. We will also investigate parallel formulations of the recursive bisection algorithm.

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. Recent Directions in Netlist Partitioning. *Integration, the VLSI Journal*, 19(1–2):1–81, 1995.
3. 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.
4. 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.
5. U.V. Catalyurek and C. Aykanat. *PaToH: Partitioning Tool for Hypergraphs, Version 3.0*, 2001.
6. 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.
7. 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.
8. 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, Nov 1997.
9. 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.
10. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.
11. A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison-Wesley, 2nd edition, 2003.
12. G. Karypis. Multilevel Hypergraph Partitioning. Technical Report #02-25, University of Minnesota, 2002.
13. G. Karypis and V. Kumar. *hMeTiS: A Hypergraph Partitioning Package, Version 1.5.3*. University of Minnesota, 1998.
14. G. Karypis and V. Kumar. Multilevel k-way Hypergraph Partitioning. Technical Report #98-036, University of Minnesota, 1998.
15. 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.
16. G. Karypis, K. Schloegel, and V. Kumar. *ParMeTiS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.0*. University of Minnesota, 2002.
17. W.J. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College, London, United Kingdom, 2000.
18. B. Krishnamurthy. An Improved min-cut Algorithm for Partitioning VLSI Networks. *IEEE Transactions on Computers*, 33(C):438–446, 1984.
19. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*. MIT Press, Cambridge, Massachussets, 2nd edition, 1998.
20. 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*, University College Cork, Ireland, July 2004.
21. A. Trifunovic 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.