A Parallel Algorithm for Multilevel k-way Hypergraph Partitioning

Aleksandar Trifunovic   William J. Knottenbelt

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

Abstract

In this paper we present a coarse-grained parallel multilevel algorithm for the k-way hypergraph partitioning problem. The algorithm significantly improves on our previous work in terms of run time and scalability behaviour by improving processor utilisation, reducing synchronisation overhead and avoiding disk contention. The new algorithm is also generally applicable and no longer requires a particular structure of the input hypergraph to achieve a good partition quality.

We present results which show that the algorithm has good scalability properties on very large hypergraphs with $\Theta(10^7)$ *vertices and consistently outperforms the approximate partitions produced by a state-of-the-art parallel graph partitioning tool in terms of partition quality, by up to 27%.*

1. Introduction

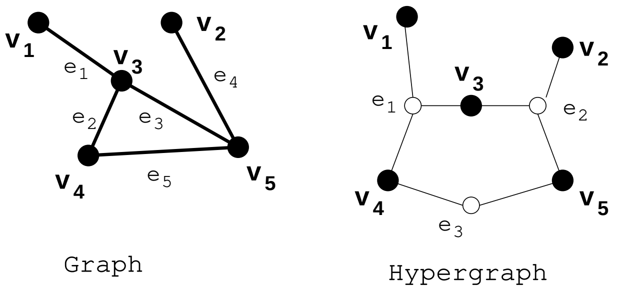

Figure 1. An example graph and a hypergraph

A hypergraph is an extension of a graph data structure in which edges are allowed to connect arbitrary, non-empty sets of vertices (as shown in Fig. 1). Like graphs, hypergraphs can be used to represent the structure of many sparse irregular problems, such as data dependencies in distributed databases and component connectivity in VLSI circuits. Also like graphs, hypergraphs may be partitioned such that a cut metric (a function of the interconnect between parts) is minimised subject to a load balancing criterion. However, hypergraph cut metrics provide a more accurate model than graph partitioning in many cases of practical interest. For example, in the row-wise decomposition of a sparse matrix for parallel matrix-vector multiplication, a hypergraph model provides an exact measure of communication cost, whereas a graph model can only provide an upper bound [5]. It has been shown that, in general, there does not exist a graph model that correctly represents the cut properties of the corresponding hypergraph [12].

Whilst algorithms for sequential hypergraph partitioning have been studied extensively and tool support exists (e.g. hMeTiS [15]), little work has been done in the field of parallel algorithms for partitioning very large hypergraphs. In [21], we proposed the first (to the best of our knowledge) parallel hypergraph partitioning algorithm. However, the partition quality was highly dependent on the structure of the input hypergraph, processors were under-utilised (especially in the later stages of coarsening and in the early stages of the refinement process) and, since it was a disk-based algorithm, there was extensive disk contention. The latter shortcomings led to large absolute parallel run times and poor scalability.

In this paper we present a new coarse-grained parallel algorithm for hypergraph partitioning that proceeds in three phases: parallel coarsening, initial partitioning and parallel refinement. The algorithm does not depend upon the underlying structure of the input hypergraph to produce good quality partitions, fully utilises the available processors and runs entirely from memory.

The remainder of this paper is organized as follows. Section 2 outlines sequential multilevel hypergraph partitioning. Section 3 presents our new parallel algorithm and Section 4 the experimental evaluation. Finally, Section 5 concludes and suggests ideas for further research.

2. Sequential Multilevel Hypergraph Partitioning

Formally, we define a hypergraph $H(V, E)$ as follows. Let V be the set of vertices and E the set of hyperedges, where each hyperedge $e_i \in E$ is a subset of the vertex set V. The map $f_w : V \rightarrow \mathbf{Z}$ associates an integer weight w_i with every vertex $v_i \in V$. In the case of a row-wise sparse-matrix decomposition (as used in parallel sparse-matrix vector multiplication) the weight function is defined by the number of hyperedges incident on each vertex (i.e. the computational load that each row induces on a processor as given by the number of non-zeros in the row). In addition, the map $f_c : E \rightarrow \mathbf{Z}$ associates a cost c_i with each hyperedge $e_i \in E$. This is unity in the case of the row-wise sparse-matrix decomposition since it corresponds to the cost of communicating a vector element across a processor boundary. Furthermore, the *size* of a hyperedge is defined as its cardinality. The sum of the sizes of the hyperedges in a hypergraph is referred to as the number of pins in the hypergraph (reflecting the first application of hypergraph partitioning in VLSI circuit design).

The formal definition of the k-way partitioning problem is as follows. Find k disjoint subsets V_i, $(i = 0, \ldots, k-1)$ of the vertex set V with part weights W_i $(i = 0, \ldots, k-1)$ (given by the sum of the constituent vertex weights), such that, given a prescribed balance criterion $0 < \epsilon < 1$,

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

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 fuction is the $(k-1)$ metric (as in row-wise sparse-matrix decomposition), the partition cost is given by

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

where λ_i is the number of parts spanned by hyperedge e_i. This formalizes the intuition of the row-wise sparse-matrix decomposition that minimizes the communication cost subject to maintaining a computational load balance.

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 [11]. Thus, research has focused on developing polynomial time heuristic algorithms resulting in good sub-optimal solutions. Because it scales well in terms of run time and solution quality with increasing problem size, the multilevel paradigm is preferred to direct solution approaches. The likelihood of iterative improvement algorithms converging to poor local minima rises significantly with increasing problem size. Alternative direct approaches such as spectral methods are reviewed in more detail in [1]. However, for large problem instances these are usually incorporated within the multilevel framework to preserve realistic run times [2].

The following subsections describe the main phases of the multilevel paradigm in more detail.

2.1. The Coarsening Phase

The coarsening phase approximates the original problem instance via a succession of smaller hypergraphs that maintain as far as possible the structure of the original hypergraph.

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 $f_{merge} : V \rightarrow V_{coarse}$, where

$$\frac{|V|}{|V_{coarse}|} = \quad r, \quad r > 1 \qquad (3)$$

and r is the prescribed reduction ratio. The map f_{merge} is also 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).

It is desirable for the coarsening phase to maintain the natural clusters (highly connected vertices) in the original hypergraph as clusters of coarse vertices in the coarsened hypergraphs. In addition, coarsening aims to reduce the size and number of hyperedges, as well as reducing the *exposed hyperedge cost*, which is defined as the sum of individual hyperedge costs and which represents the upper bound on the cut-size of a partition. The coarsening algorithm has a significant impact on the final solution quality. This is because a poor coarsening algorithm may only allow a partitioning algorithm to explore parts of the solution space where the local minima solutions are of poor quality relative to the global minimum.

Coarsening algorithms are discussed in detail in both [1] and [13]. In our experiments we have found that the *FirstChoice* (hereafter FC) coarsening algorithm [16] and related algorithms [5] result in balanced partitions and fast run times for our case study hypergraphs. The FC coarsening 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 resultant cluster does not exceed a prescribed maximum weight. This condition is imposed to prevent a large imbalance in vertex weights in the coarsest hypergraphs.

A low reduction ratio implies that many coarsening stages may be required, thus increasing the run time of the overall algorithm. On the other hand, a larger reduction ratio may result in a poorer quality of coarsening as vertices may be matched into sub-standard clusters in order to sufficiently reduce the coarse hypergraph. In [13] the author reports that a ratio in the range 1.5–1.8 provides a reasonable balance between run time and solution quality. Our experience is similar with our case study hypergraphs, using ratios in the range 1.5–2.0.

2.2. The Initial Partitioning Phase

The coarsest hypergraph is partitioned using a direct partitioning method such as an iterative improvement algorithm. This partitioning is subsequently uncoarsened and refined whilst being projected back through the sequence of successively finer hypergraphs. Because the coarsest hypergraph tends to be significantly smaller than the original problem instance, direct partitioning methods are computationally feasible and the time taken to compute the initial partitioning is usually considerably less than the time taken by the other phases of the multilevel pipeline. As 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

Here the initial partitioning is propagated up through the successively finer hypergraphs and at each step the partition is further refined using heuristic refinement techniques. When the overall k-way partitioning is computed via recursive bisection, the refinement phase consists of a bisection refinement algorithm. Traditionally, iterative improvement algorithms based on the Fiduccia-Mattheyses (FM) algorithm are used. These perform *passes*, during each of which each vertex is moved from its starting part at most once; the best sequence of moves found by the heuristic is actually performed leading to the refined partition. The algorithms operate in $O(m)$ time per pass, where m is the number of pins in the hypergraph, and usually converge within a few passes to a local minimum with respect to the heuristic used [10]. More sophisticated refinement algorithms have been developed, motivated by the idea of escaping from poor local minima [19, 7, 8, 9].

Extending the FM-algorithm to compute a k-way partitioning at each refinement step (as opposed to using recursive bisection) increases both the time complexity of the algorithm and the likelihood that the algorithm terminates at a relatively poor local minimum. However, good results have been reported with a *greedy refinement* algorithm, especially for increasing values of k [16]. The greedy refinement algorithm performs iterations during which the vertices are visited in a random order and moved to the part that results in the largest positive gain. Since the hypergraph is sparse, the algorithm avoids calculating the gain to every other of the $(k-1)$ parts as follows. Vertices are not considered for a move if they are internal to their current part (i.e. all adjacent vertices are also in the same part). Otherwise, the gain of a move is only computed for neighbouring parts (those parts that contain adjacent vertices) if the move to the neighbouring part does not violate the balance constraint. Experiments have showed that the algorithm typically converges after a small number of iterations [16].

A more sophisticated refinement scheme repeats the whole coarsening and refinement process on the refined partition while preserving properties of the partition during the coarsening phase. This type of refinement is called a *V-Cycle* [13, 4] and is a feature of the tool hMeTiS. It attempts to converge to a better solution than would be obtained by simply performing the multilevel pipeline once, but can significantly increase run time.

3. The Parallel Algorithm

This section describes our parallel multilevel partitioning algorithm. As the algorithms that make up the multilevel pipeline are inherently sequential in nature, a coarse-grained formulation is sought. We note that there are two approaches to computing a k-way partitioning: recursive bisection and direct partitioning. We compute the k-way partitioning directly in parallel due to the k-way refinement algorithm having better opportunities for concurrency, since ways to perform the gain update calculations of the FM algorithm in parallel are not readily apparent.

As the coarsest hypergraph should be small enough for the initial partitioning phase to be performed sequentially and multiple runs of the initial partitioning algorithm can be carried out on the available processors in parallel, only the coarsening and refinement phases are parallelised (as in [21]). The parallel multilevel pipeline is illustrated in Fig. 2. The motivation for our parallel formulation comes from the parallel *graph* partitioning algorithm in [14]. The sections below describe the parallel coarsening and refinement phases in more detail.

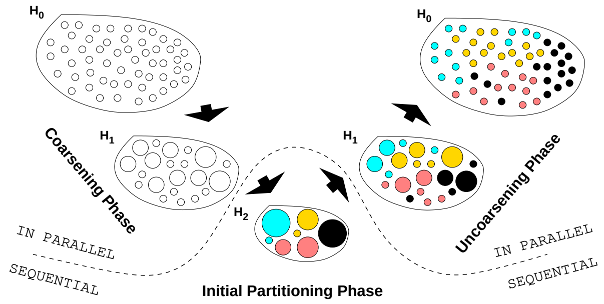

Figure 2. Parallel Multilevel Pipeline

3.1. Parallel Coarsening Phase

We consider the FC algorithm as the basis for our parallel formulation although *hyperedge coarsening* algorithms [13] can also be parallelised in this fashion. If p denotes the number of processors, we store $|E|/p$ hyperedges and $|V|/p$ vertices on each processor. We assume that each processor stores a contiguous set of vertices, although the initial distribution of vertices is not critical because our algorithm allows for vertex matches across processor boundaries.

At the beginning of each coarsening step, the hyperedges adjacent to the locally held vertices are assembled at each processor using an all-to-all personalised communication. Then, each processor i visits vertices from 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 chooses v or w as its best match then it is also added to the request set $S_{i,j}$. Note that a local vertex on processor i can request a match with at most one other vertex not stored on processor i. 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.

Now, each processor i communicates its request sets to the other processors, including the weights of the vertices that are involved in the matching request. The processors then 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 seek to match with a local vertex w stored on processor 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 k, $k \neq i$, 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 k is also 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.

Now consider the case where vertex v on processor i requests a match with vertex w on processor j and vice versa. This would be expected if the two vertices are strongly connected; in this case it is desirable to match them together into a single coarse vertex. However, under the scheme above, the remote match would be rejected by both processors since both v and w are involved in remote requests. We resolve this by communicating 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 $i < j$. During each stage, we aggregate the communication of the request sets and also aggregate the communication of the replies. These are sent as all-to-all personalised communications. 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 processed in order of increasing i. A randomized scheme may also be implemented, whereby this array is traversed in random order.

Having computed the vertex matching vector across the processors, the coarsening step is completed by contracting the hyperedges of the finer hypergraph, thus creating the hyperedges of the coarse hypergraph. Each processor contracts the $|E|/p$ locally stored hyperedges. Note that in the sequential algorithm duplicate coarse hyperedges are replaced by a single coarse hyperedge whose weight is the sum of the weights of the finer hyperedges that are contracted onto it. We minimise the communication requirements of this step and load balance the hyperedge computations by using a probabilistic hash function to associate a 64-bit key with each hyperedge. The hash function is a 64-bit variant of that given in [18] and has the desirable property that $key \bmod p$ scatters the hash-keys across the processors with a near-uniform distribution, independently from the nature of the hyperedge input. In order to eliminate duplicate hyperedges, processors communicate the hyperedge hash-keys and weights to the destination processor given by $key \bmod p$. The destination processor retains only one copy of the hyperedge, setting its cost to be the sum of the costs of all duplicates. The parallel coarsening step concludes with a communication of coarse vertices so that each processor has $|V_{coarse}|/p$ local vertices at the start of the subsequent coarsening step.

3.2. Initial Partitioning Phase

During this phase the initial partition is computed on the coarsest hypergraph. Each processor concurrently performs the sequential partitioning algorithm and the best partitioning is selected for further refinement. Because the coarsest hypergraph is small (typically of the order of $100 \times k$ vertices, where k is the number of desired parts), this phase does not contribute significantly to the run time of the parallel algorithm.

3.3. Parallel Uncoarsening Phase

At the beginning of the parallel uncoarsening phase, the partition of the coarse hypergraph is used to initialise the partition of the current hypergraph. Then, the hyperedges adjacent to the locally held vertices are assembled at each processor using an all-to-all personalised communication.

Like the sequential greedy k-way refinement algorithm [16], our parallel refinement algorithm proceeds in passes. However, instead of moving single vertices across a partition boundary one at a time, the parallel algorithm moves sets of vertices. The processors then perform gain computations for each of their local vertices in parallel. In addition, each processor maintains sets of moved vertices $U_{i,j}$, $i \neq j, i, j = 0, \ldots, k-1$. These sets contain the local vertices whose moves from their current part i to the destination part j result in a positive gain in cut-size. 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. Vertex sets are moved from overweight parts, subject to the balance constraint. Finally, the root processor broadcasts the updated part weights before the processors proceed with the subsequent stage. Note that, given at least one non-empty set $U_{i,j}$ with positive gain on some processor, the stage is guaranteed to yield a positive gain overall. The communication and computation on the root processor should not affect scalability since the number of vertices moved during any one pass is usually significantly less that the number of vertices in the hypergraph. This is to be expected since we are refining what is already a good partition and few vertex moves will result in a positive gain in cut-size. As in the sequential algorithm, the refinement procedure terminates when the overall gain of a pass is not positive.

4. Experimental Results

4.1. Implementation and Test Environment

The three phases of our parallel multilevel k-way partitioning algorithm were implemented in the C++ language using the Message Passing Interface (MPI) standard [20] as follows:

1. Parallel coarsening is performed using our parallel formulation of the FC coarsening algorithm described in Section 3.1.
2. The initial partitioning phase commences when the coarsest hypergraph has $100 \times k$ vertices. We interface with the sequential partitioning routine `HMETIS_PartKway()` from the hMeTiS library [15]. This is used to compute several initial partitionings concurrently across the processors, with the best selected for parallel refinement.
3. Finally, the output partition vector given by `HMETIS_PartKway()` is refined in parallel using the parallel k-way refinement algorithm described in Section 3.3.

The hMeTiS library [15] was used for base-case sequential comparison, i.e. to provide the run time and cut-size of the best sequential algorithm. We used the `HMETIS_PartKway()` k-way partitioning routine from the hMeTiS library instead of `HMETIS_PartRecursive()`, as the latter was not able to consistently produce partitions within the balance constraints. `HMETIS_PartKway()` also provides a like-for-like comparison to our parallel algorithm since it too implements the greedy k-way refinement algorithm from [16]. As recommended in [16], we set the partitioning objective to SOED (the sum of external degrees) and switched off the V-Cycle feature to obtain a fair run time comparison with the parallel algorithm (which does not perform V-Cycling).

When a hypergraph instance was too large to be partitioned on a single workstation, a suitable comparison for our algorithm was provided by the state-of-the-art parallel graph partitioning tool ParMeTiS [17], in the absence of other parallel hypergraph partitioning tools. Although the optimisation objectives for the two tools are different, ParMeTiS is used because such parallel graph partitioners are currently the only feasible way to obtain reasonable partitions for many large problems, as noted in [21, 6].

The architecture used in all the experiments consisted of a Beowulf Linux Cluster with 64 dual processor nodes. 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. Case Study

The parallel algorithm was experimentally evaluated on hypergraph representations of transition matrices derived from a high-level Semi-Markov model of a voting system, a full description of which can be found in [3]. In this system, voters cast votes through polling units which in turn register votes with all available central voting units. Both polling units and central voting units can suffer breakdowns, from which there is a soft recovery mechanism. If, however, all the polling or voting units fail, then, with high priority, a failure recovery mode is instituted to restore the system to an operational state. The number of voters, polling units and central voting servers is configurable, and each combination of these parameters results in a sparse transition matrix of a different size, as shown in Table 1.

The aim of the analysis is to find the response time density and/or quantiles of the time taken for a certain number of voters to successfully register their votes. This requires the numerical inversion of the Laplace Transform of the response-time density, which in turn requires the solution of many thousands of sets of linear equations with the same non-zero sparsity pattern. The kernel operation in such solvers is parallel matrix-vector multiplication and hypergraph partitioning is used to reduce the amount of inter-processor communication in a row-wise decomposition. Note that the hypergraph partitioning need only be performed once, but is reused several thousand times. Thus, the quality of the resulting partition is key to scalability.

The hypergraph representing the transition matrix of the Voting System with 175 customers (with 1 140 050 vertices) can be partitioned on a single workstation, while the hypergraph representations of transition matrices from the Voting System with 250 and 300 customers (with 5 218 300 and 10 991 040 vertices respectively) are too large for sequential partitioning. To convert the matrix into the appropriate input files, we use the transformations presented in [5]. The complexity of these transformations (into inputs for both the parallel hypergraph and parallel graph tools) is $O(m)$, where m is the number of non-zero elements in the matrix.

The maximum partition imbalance in Eq. 1 was set to $\epsilon = 0.05$ and results (for both run time and cut-size) were averaged over five runs since the partitioning algortihms are randomized (e.g. randomly visiting the vertices during coarsening). In Tables 2 and 3 our parallel implementation is denoted by Par$k$way 1.1.

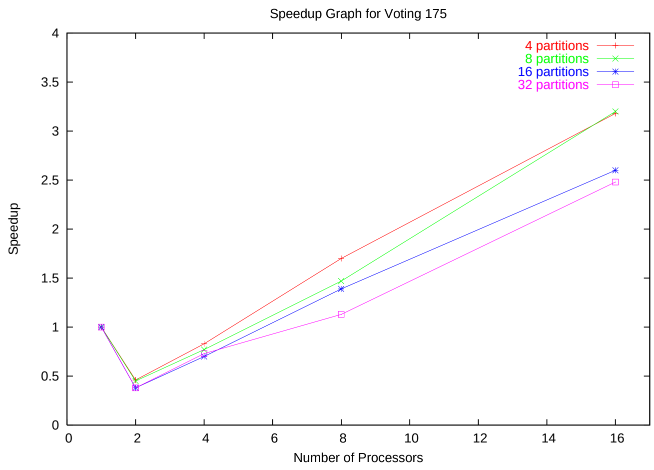

Figure 3. Speedup results on Voting Model with 175 customers

Table 2 shows the run times of the algorithms on the test hypergraphs, while the speedup on Voting 175 is summarised in Fig. 3. The additional communication and computation overhead incurred by the parallel algorithm, as well as the highly-optimised nature of the base case we used (hMeTiS has been in development since 1997), means that we observe a slowdown for a small number of processors. However, speedups are obtained when more than four processors are used. Furthermore, the scalable nature of our algorithm is reflected in the linear speedup trend observed in computing partitions with 4, 8, 16 and 32 parts.

Table 3 presents the average $(k - 1)$ metric cut-sizes for the test hypergraphs. We see that where comparison with the sequential algorithm `HMETIS_PartKway()` is possible, our algorithm produces partitions of comparable quality, although the quality of the partitions deteriorates slightly as the number of processors used increases.

Where the hypergraphs are too large for

Hypergraph	Model Parameters	#Vertices	#Hyperedges	#Pins
Voting 175	175/45/5	1 140 050	1 140 050	6 657 722
Voting 250	250/60/10	5 218 300	5 218 300	32 986 597
Voting 300	300/80/10	10 991 040	10 991 040	69 823 797

Table 1. Characteristics of the test hypergraphs

	Par$k$way 1.1				`HMETIS_PartKway()`**/ParMeTiS**			
#Processors	#Parts				#Parts			
Voting 175	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**
1	-	-	-	-	50.69	53.41	57.91	67.60
2	109.1	119.5	152.2	176.4	4.36	4.36	4.51	4.59
4	61.3	69.2	83.0	92.9	2.40	2.46	2.53	2.61
8	29.8	31.9	41.4	60.0	1.48	1.49	1.56	1.61
16	15.9	16.7	22.3	27.3	1.18	1.10	1.16	1.18
Voting 250	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**
4	560.2	594.2	636.9	782.5	12.4	12.6	13.0	13.5
8	294.9	309.6	356.5	376.1	7.30	7.44	7.88	7.92
16	148.8	159.8	169.6	229.5	4.98	5.05	5.34	5.54
Voting 300	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**
8	-	997.4	1 084.8	1 239.5	-	16.0	16.6	17.2
16	-	496.6	538.0	608.3	-	10.6	11.1	11.7
32	-	252.9	292.4	325.1	-	8.9	9.3	9.9

Table 2. Partitioning times in seconds

	Par$k$way 1.1				`HMETIS_PartKway()`**/ParMeTiS**			
#Processors	#Parts				#Parts			
Voting 175	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**
1	-	-	-	-	12 907	25 459	50 052	94 762
2	12 803	25 777	50 979	97 461	16 146	32 535	68 189	113 744
4	12 833	25 570	52 079	98 067	15 897	34 800	67 016	115 356
8	13 139	26 260	53 019	98 113	16 302	33 073	68 633	114 917
16	13 057	26 762	52 579	99 282	15 955	32 528	68 872	116 765
Voting 250	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**
4	46 378	92 026	183 104	330 472	53 671	117 354	249 415	402 681
8	46 221	93 910	185 881	356 093	54 333	111 781	228 134	407 163
16	46 155	91 986	188 179	358 162	55 014	107 488	221 735	394 156
Voting 300	**4**	**8**	**16**	**32**	**4**	**8**	**16**	**32**
8	-	169 970	334 139	612 285	-	193 720	442 387	687 659
16	-	170 810	335 624	616 313	-	196 339	401 573	689 444
32	-	164 552	332 739	613 084	-	196 803	395 332	689 992

Table 3. Partitioning $k-1$ cut-sizes

`HMETIS_PartKway()` to partition sequentially, our parallel multilevel hypergraph partitioning algorithm outperforms the parallel graph partitioning tool ParMeTiS in terms of the $(k-1)$ metric on all the test configurations, by between 11% and 27%.

5. Conclusion

We have presented a parallel multilevel k-way partitioning algorithm that improves substantially on our previous work by improving processor utilisation and reducing communication and synchronisation overheads. The algorithm features novel schemes for resolving conflicts that arise during the parallel coarsening process and for reducing vertex thrashing during parallel refinement.

The algorithm has been implemented and our initial results on several test hypergraphs from the performance analysis domain demonstrate good scalability properties and parallel partition quality comparable with the state-of-the-art sequential partitioning tool hMeTiS. Further, partition quality comfortably exceeds those produced by the parallel graph partitioning tool ParMeTiS, by up to 27%.

Future work will include developing an analytical performance model to compute the algorithm's isoefficiency function. This can be used to determine what increase in problem size is necessary to maintain constant efficiency, given an increase in the number of processors used). This would formally complement our empirical evidence that the algorithm is scalable. We would also like to apply our algorithm to other application domains, such as VLSI circuit partitioning. Finally, in addition to further enhancing the current parallel algorithm, we aim to develop a parallel formulation of the recursive multilevel bisection partitioning algorithm (as opposed to the direct k-way scheme considered here).

References

[1] C. Alpert, J. Huang, and A. Kahng. Recent Directions in Netlist Partitioning. *Integration, the VLSI Journal*, 19(1–2):1–81, 1995.

[2] S. Barnard and H. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice and Experience*, 6(2):101–117, April 1994.

[3] J. Bradley, N. Dingle, W. Knottenbelt, and H. 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.

[4] A. Caldwell, A. Kahng, and I. Markov. Improved Algorithms for Hypergraph Bipartitioning. In *Proc. 2000 Conference on Asia South Pacific Design Automation*, pages 661–666. ACM/IEEE, January 2000.

[5] U. 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.

[6] N. Dingle, W. Knottenbelt, and P. Harrison. Uniformization and Hypergraph Partitioning for the Distributed Computation of Response Time Densities in Very Large Markov Models. *Journal of Parallel and Distributed Computing*, 2004. (To appear).

[7] 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.

[8] 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.

[9] 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.

[10] C. Fiduccia and R. Mattheyses. A Linear Time Heuristic For Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[11] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., 1979.

[12] E. Ihler, D. Wagner, and F. Wagner. Modeling Hypergraphs by Graphs with the same Mincut Properties. *Information Processing Letters*, 45:171–175, March 1993.

[13] G. Karypis. Multilevel Hypergraph Partitioning. Technical Report #02-25, University of Minnesota, 2002.

[14] G. Karypis and V. Kumar. A Coarse-Grain Parallel Formulation of Multilevel k-way Graph Partitioning Algorithm. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[15] G. Karypis and V. Kumar. *hMeTiS: A Hypergraph Partitioning Package, Version 1.5.3*. University of Minnesota, 1998.

[16] G. Karypis and V. Kumar. Multilevel k-way Hypergraph Partitioning. Technical Report #98-036, University of Minnesota, 1998.

[17] G. Karypis, K. Schloegel, and V. Kumar. *ParMeTiS: Parallel Graph Partitioning and Sparse Matrix Ordering Library, Version 3.0*. University of Minnesota, 2002.

[18] W. Knottenbelt. *Parallel Performance Analysis of Large Markov Models*. PhD thesis, Imperial College, London, United Kingdom, 2000.

[19] B. Krishnamurthy. An Improved min-cut Algorithm for Partitioning VLSI Networks. *IEEE Transactions on Computers*, 33(C):438–446, 1984.

[20] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*. MIT Press, Cambridge, Massachussets, 2nd edition, 1998.

[21] A. Trifunovic and W. 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.