Parallel Hypergraph Partitioning for Scientific Computing

Karen D. Devine[*], Erik G. Boman[*], Robert T. Heaphy[*],
Rob H. Bisseling[†], and Umit V. Catalyurek[‡]

[*]Sandia National Laboratories
Dept. of Discrete Algorithms and Math.
Albuquerque, NM 87185-1111, USA
{kddevin,egboman,rheaphy}@sandia.gov

[†]Utrecht University
Dept. of Mathemathics
3508 TA Utrecht, The Netherlands
Rob.Bisseling@math.uu.nl

[‡]Ohio State University
Dept. of Biomedical Informatics
Columbus, OH 43210, USA
umit@bmi.osu.edu

Abstract

Graph partitioning is often used for load balancing in parallel computing, but it is known that hypergraph partitioning has several advantages. First, hypergraphs more accurately model communication volume, and second, they are more expressive and can better represent nonsymmetric problems. Hypergraph partitioning is particularly suited to parallel sparse matrix-vector multiplication, a common kernel in scientific computing. We present a parallel software package for hypergraph (and sparse matrix) partitioning developed at Sandia National Labs. The algorithm is a variation on multilevel partitioning. Our parallel implementation is novel in that it uses a two-dimensional data distribution among processors. We present empirical results that show our parallel implementation achieves good speedup on several large problems (up to 33 million nonzeros) with up to 64 processors on a Linux cluster.

1 Introduction

Partitioning and load balancing are important issues in parallel scientific computing. The goal is to distribute data (and work) evenly among processors in a way that reduces communication cost and achieves maximal performance. Graph partitioning has long served as a useful model for load balancing in parallel computing. Data are represented as vertices in a graph, and edges represent dependencies between data. Graph partitioning attempts to minimize the number of cross-edges in the graph between processors, as these result in application communication. It has been shown that for many problems, this cut-edge metric is not an accurate representation of communication cost or volume. On the other hand, *hypergraph models* accurately represent communication volume [3]. A hypergraph $H = (V, E)$ consists of a vertex set V and a set of hyperedges E. (Hyperedges are also called *nets*.) Each hyperedge is a subset of V. In parallel computing, communication is required for a hyperedge whose vertices are in two or more processors. Catalyurek and Aykanat [3] proposed a hypergraph model for sparse matrix-vector multiplication, and showed that the hyperedge cut metric corresponds *exactly* to the communication volume. An important advantage of the hypergraph model is that it can easily represent nonsymmetric and rectangular matrices. For more details on different partitioning models for parallel computing, see [8, 9].

Graph partitioning is frequently used for parallel mesh-based computations such as finite element calculations. Since these applications are typically sparse and often regular, the graph model works quite well. In fact, simple geometric partitioning methods may work almost as well as graph partitioning. In this paper, however, we focus on applications that are non-traditional in some way. Specifically, the sparse matrix representing the problem could be

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

[†]This author's work was mainly performed while visiting the Computer Science Research Institute at Sandia.

[‡]Supported by Sandia contract PO283793.

nonsymmetric, rectangular, semi-dense, or highly irregular. We demonstrate the utility of hypergraph partitioning on test data from a variety of areas, including Markov chains, polymer self-assembly, DNA electrophoresis, electrical circuit simulation, sensor placement, and information retrieval (web search).

An important kernel in many scientific computations is a sparse matrix-vector product. The parallel issue is how to distribute the sparse matrix between the processors. The most common approach is to split the matrix in one dimension (by either rows or columns), assigning approximately even chunks (of rows/columns) to processors. Both variations naturally lead to hypergraph partitioning. In the row-net model, each column corresponds to a vertex and each row corresponds to a hyperedge. For row partitioning, an analogous column-net model can be used. Although simply using hypergraph partitioning gives an improvement over graph partitioning (25-35% reduction is typical [3]), even lower communication volumes can be achieved by going beyond this 1D partitioning. Vastenhouw and Bisseling recently suggested a recursive two-dimensional data distribution known as Mondriaan [17]. Catalyurek and Aykanat have proposed a fine-grain partitioning model [4] where each nonzero in a matrix is independently assigned to a processor. Both these methods rely on hypergraph partitioning as an underlying technique. Software for hypergraph partitioning therefore becomes important.

Several software packages for hypergraph partitioning exist: e.g., PaToH [5], hMETIS [11], Mondriaan [17] (for sparse matrices), and MLpart [2] (for circuits). However, all these packages run in serial. For large-scale parallel applications, partitioning must be performed in parallel. In the following, we describe the design and structure of a parallel hypergraph partitioner we have developed in Sandia's Zoltan toolkit [6, 18], a library of parallel partitioning and load-balancing methods. Our parallel implementation uses a two-dimensional data distribution to reduce communication within the partitioning algorithm. We compare the effectiveness and performance of our parallel hypergraph partitioner with parallel graph partitioners, serial hypergraph partitioners, and a parallel hypergraph partitioner Par$k$way developed by Trifunovic and Knottenbelt [16] largely concurrently with our work.

2 Preliminaries

The (unweighted) hypergraph partitioning problem is defined as follows: given a hypergraph $H = (V, E)$ and an integer k, partition the vertex set V into k disjoint subsets $V_j, j = 0, \ldots, k-1$, of approximately equal sizes such that a cut metric is minimized. We refer to $P = \{V_0, \ldots, V_{k-1}\}$ as a partitioning and the subsets as partitions. A hyperedge is cut if it contains at least two vertices belonging to different partitions. We seek to minimize the cut metric

$$
\left|\begin{array}{ccccc}
a_{11} & a_{12} & 0 & 0 & a_{15} \\
a_{21} & a_{22} & 0 & 0 & 0 \\
a_{31} & 0 & 0 & a_{34} & 0 \\
0 & 0 & a_{43} & a_{44} & a_{45}
\end{array}\right|
$$

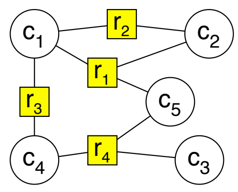


Figure 1. The columns in the sparse matrix A (left) correspond to the vertices (represented as circles) in the hypergraph H (right). The rows in A correspond to hyperedges in H (represented as squares).

$$cuts(H, P) = \sum_{i=0}^{|E|-1} (\lambda_i(H, P) - 1), \tag{1}$$

where $\lambda_i(H, P) \leq k$ is the number of partitions spanned by hyperedge i in the partitioning P. This metric is known as the $(k-1)$-cut; it is important because it accurately reflects communication cost in parallel computing and, in particular, sparse matrix-vector multiplication. When $k = 2$, $cuts(H, P)$ is simply the number of hyperedges cut.

We allow both vertices and hyperedges to have (scalar) weights, since this is more general and important in some applications. In the weighted partitioning problem, the objective is to minimize the weighted cut subject to the partitions having an approximately equal sum of vertex weights.

A hypergraph can also be viewed as a sparse matrix. We use the row-net model, where each row in the matrix corresponds to a hyperedge and each column corresponds to a vertex. Let A be the sparse matrix corresponding to a hypergraph H. Then $a_{ij} = 1$ if vertex j belongs to hyperedge i, and zero otherwise. An example of the row-net model is given in Figure 1.

2.1 Multilevel partitioning methods

Our algorithm follows the well-known multilevel partitioning approach, which has proved successful both for graph partitioning [10, 13] and hypergraph partitioning [3, 11]. The idea is to approximate the hypergraph by a sequence of smaller hypergraphs that reflect the original hypergraph. In *coarsening*, we construct the smaller hypergraphs. In *coarse partitioning*, we partition the smallest hypergraph. In *refinement*, we project a coarse partitioning to a finer (larger) hypergraph and improve the partitioning using a local optimization (refinement) method. We describe our parallel implementation of this multilevel "V-cycle" in the next section.

There are two possible approaches to achieve a k-way partitioning. The first is called direct k-way partitioning, where the multilevel V-cycle is applied once to directly split the hypergraph into k parts. The other is recursive bisection, where the V-cycle partitions the hypergraph into two parts; such bisection is repeated recursively until the desired number of partitions k is reached. By allowing the resulting hypergraphs in each bisection step to have unequal sizes, recursive bisection can support arbitrary k; it is not limited to k being a power of two.

Both approaches are viable; PaToH [5] uses recursive bisection while hMETIS [11] uses direct k-way. Our implementation is based on recursive bisection for arbitrary k. Note that k is a user parameter that may differ from the number of processors p. However, in dynamic load balancing, typically $k = p$.

3 Parallel Hypergraph Partitioning Algorithm

3.1 Data distribution

A major decision for our parallel partitioner is how to distribute the data (the hypergraph or matrix) between processors. Perhaps the most natural options are to divide either the vertices or the hyperedges between processors. These options correspond to distributing the matrix along columns or rows, respectively. We have opted for a third option, namely to divide the matrix along both rows and columns in a way that produces a Cartesian distribution of the matrix. We call this a two-dimensional (2D) layout since each processor is assigned a rectangular submatrix. Conceptually, we think of the processors also as being organized in a 2D fashion, and we will refer to rows and columns of processors. Note that this is only a logical arrangement; the physical interprocessor network may be different.

The main advantage of the 2D layout is that most communication can be done either along rows (horizontally) or along columns (vertically). Suppose we have $p = p_x \times p_y$ processors, where p_x and p_y are the number of processors in a row and a column, respectively. Then only p_x or p_y processors need to participate in collective communication operations. Typically $p_x = O(\sqrt{p})$. Such 2D data distributions have been used successfully for several matrix computations [1, Ch.2,Ch.4].

Another way to view our data layout is that each processor knows only partial information about some vertices and some hyperedges. In contrast to Par$k$way [16] which uses 1D distributions for both vertices and hyperedges, we do not use any type of ghosting, thereby significantly reducing the memory required.

Note that a 2D parallel data distribution was proposed for graph partitioning in [12]. In that case, the vertices were split among $\sqrt{p}$ processors while the adjacency matrix was split among all p processors. It was observed that speedup was limited to $\sqrt{p}$ because the "diagonal processors" became a bottleneck, so those authors later adopted a 1D distribution. We believe a 2D distribution is more suitable for hypergraph partitioning because the most time-consuming parts of the algorithm are distributed among all p processors. There are only a few sub-tasks that are solely vertex-based or solely edge-based and parallel speedup of some of these tasks may be limited to p_x and p_y, respectively; this should not become a bottleneck for the overall algorithm.

3.2 Coarsening

The coarsening phase approximates the original hypergraph via a succession of smaller hypergraphs. When the smallest hypergraph has fewer vertices than some threshold (e.g., 100), the coarsening stops. Several methods have been proposed for constructing coarser representations of graphs and hypergraphs. We consider only methods based on merging pairs of vertices. The issue then becomes how to select vertices to merge together. Intuitively, we wish to merge vertices that are similar and therefore more likely to be in the same partition in a good partitioning. Catalyurek and Aykanat [3] suggested a heavy-connectivity matching, which measures a similarity metric between pairs of vertices. Their preferred similarity metric, which was also adopted by hMETIS [11] and Mondriaan [17], is known as the *inner product*. The inner product between two vertices is defined as the Euclidean inner product between their binary hyperedge incidence vectors, that is, the number of hyperedges they have in common. (Edge weights can be incorporated in a straight-forward way.) Our code also has the option to compute the *cosine* similarity metric, which is a scaled version of the inner product commonly used in information retrieval.

3.2.1 Matching

Given the inner product values, the problem of finding good pairs to merge can be modeled as a maximum-weight matching problem, where the edge weights in the graph are the inner products between vertices from the hypergraph. Previous work indicates that the matching problem does not need to be solved optimally, so quick heuristics are commonly used. We use variations of the greedy strategy (also known as first-choice).

The sequential greedy algorithm works as follows. Pick a (random) unmatched vertex v. For each unmatched neighbor vertex u, compute the inner product $< v, u >$. Select the vertex with the highest non-zero inner product value and match it with v. Repeat until all vertices have been considered. Care must be taken to implement this efficiently; see Algorithm 1.

In parallel, this simple algorithm becomes much more complicated. Each processor knows about only a subset of

Algorithm 1 Serial inner-product matching

```
1:  procedure SERIAL-IPM($H = (V, E)$)
2:      initialize ip[$v$] $\leftarrow$ 0 for $v \in V$
3:      for all unmatched $v \in V$ do
            ▷ Compute all inner products with $v$
4:          for $e \in E$ such that $v \in e$ do
5:              for all unmatched $u \in e$, $u \neq v$ do
6:                  ip[$u$] $\leftarrow$ ip[$u$] + 1
7:          $w \leftarrow$ argmax(ip)
8:          for $e \in E$ such that $v \in e$ do
9:              for all unmatched $u \in e$ do
                    ▷ Reset all inner product values to zero
10:                 ip[$u$] $\leftarrow$ 0
11:         match($v$,$w$) ▷ Match $v$ with best candidate $w$
```

the vertices and the hyperedges. Computing the inner products requires communication. If we consider the hypergraph as a sparse matrix A, we essentially need to compute the matrix product $A^T A$. We use the sparsity of A to compute only entries of $A^T A$ that may be nonzero. Since we use a greedy strategy, we actually compute only a subset of the nonzero entries in $A^T A$.

Even if A is typically very sparse, $A^T A$ may be fairly dense. Therefore we cannot compute all of $A^T A$ at once, but instead compute parts of it in separate *rounds*. In each round, each processor selects a (random) subset of its vertices that we call *candidates*. These candidates are broadcast to all other processors in the processor row. This requires horizontal communication in our 2D layout. Each processor then computes the inner products between its local vertices and the external candidates received. Note that these inner products are only partial inner products; vertical communication along processor columns is required to obtain the full (global) inner products. One could let a single processor within a column accumulate these full inner products, but this processor may run out of memory. So to improve load balance, we accumulate inner products in a distributed way, where each processor is responsible for a subset of the vertices.

At this point, the potential matches in a processor column are sent to the *master row* of processors (row 0). The master row first greedily decides the best local vertex for each candidate. These local vertices are then *locked*, meaning they can match only to the desired candidate (in this round). This locking prevents conflicts between candidates, which could otherwise occur when the same local vertex is the best match for several candidates. Horizontal communication along the master row is used to find the best global match for each candidate. Due to our locking scheme, the desired vertex for each match is guaranteed to be available so no conflicts arise between vertices. The full algorithm is summarized in Algorithm 2.

Observe that the full inner-product matching is computationally intensive and requires several communication phases along both processor rows and columns. Empirically, we observed that the matching usually takes more time than the other parts of the algorithm. We are therefore currently exploring faster, approximate matching methods. We have implemented one such alternative matching method where the matching is limited to pairs of vertices within the same processor column.[1] This way, no horizontal communication is required; only vertical communication to sum the inner products is needed. Communication cost is reduced, but the matching quality is often worse.

3.2.2 Contraction

After a matching (pairing of vertices) has been computed, we build the coarser hypergraph by merging matched vertices. Matched vertices in the finer hypergraph become a single vertex in the coarser hypergraph, with its vertex weight equal to the sum of the fine vertices' weights. The new coarse vertex is a member of each hyperedge that contained at least one of its fine vertices. This merging reduces the number of vertices by the number of matches. To further reduce both memory requirements and run time, we take two steps to reduce the number of hyperedges. First, we discard all hyperedges of size one, as they cannot contribute to the cut metric (1). Second, we collapse identical hyperedges into a single hyperedge. Two hyperedges are identical if they contain the same vertices. Such a comparison can be done efficiently using a hash function based on the vertices in a hyperedge; edges with different hash values are not identical. Since our hyperedges are distributed, computing the hash function requires horizontal communication. For hyperedges with identical hash values, we compare their vertex lists to determine whether the hyperedges are truly identical; this step requires vertical communication. When hyperedges are collapsed, the new edge gets an edge weight equal to the sum of all the edges it represents. In this way, the cut metric is preserved, so the coarse problem is equivalent to the case with no edge removal.

3.3 Coarse Partitioning

The coarsening stops when the hypergraph is small. Since the coarse hypergraph is small, we replicate it on every processor. Each processor runs a randomized greedy algorithm to compute a different partitioning into k partitions. (For the recursive bisection algorithm, we use $k = 2$.) We then evaluate the cut metric (1) on each processor and pick the globally best partitioning.

[1]Due to space limitations, we do not present results for this method here.

Algorithm 2 Parallel inner-product matching

```
procedure PARALLEL-IPM($H = (V, E)$)                                   ▷ $H$ is the local part of the hypergraph
    $rounds \leftarrow 8 \times p_x$                                    ▷ $p_x$ is the #processors in a processor row
    $ncand \leftarrow |V|/(2 \times rounds)$                            ▷ each match pairs 2 vertices
    for $k \leftarrow 1$ to $rounds$ do
        $C' \leftarrow ncand$ unmatched candidate vertices in $my\_processor\_column$
        Broadcast $C'$ and their columns (hyperedges) to all processors in $my\_processor\_row$
        $C \leftarrow$ all received candidates
        for $v \in C$ do                                                ▷ Compute all local inner products with $v$
            initialize ip[$u$,$v$] $\leftarrow 0$ for $u \in V$
            for $e \in E$ such that $v \in e$ do
                for all *unmatched* $u \in e$, $u \notin C'$ do
                    ip[$u$,$v$] $\leftarrow$ ip[$u$,$v$] + 1            ▷ ip[$u$,$v$] is a local inner product
        for $v \in C$ do
            For all ip[$u$,$v$]$> 0$, send ip[$u$,$v$] to row ($v$ mod $p_y$)
        Receive partial inner products, ip[$u$,$v$].
        for $v \in C$, where $v$ mod $p_y$ = $my\_processor\_row$ do
            For all received ip[$u$,$v$], gip[$u$,$v$] $\leftarrow$ gip[$u$,$v$] + ip[$u$,$v$] ▷ Sum received values to global inner product gip
            Send gip[$*$,$v$] to *master row* (row 0)
        if $my\_processor\_row$ = 0 then                                ▷ master row
            for each candidate $v \in C$ do
                Select *unmatched* local vertex $w$ with highest gip[$v$,$w$]
                Store ($v, w$,gip[$v, w$]) in array $local\_best$
            Compute $global\_best$ from $local\_best$ by AllReduce communication along *master row*
            for each local candidate $v \in C'$ do
                if $(v, w) \in global\_best$ then
                    Match $v$ and $w$.
```

3.4 Refinement

The refinement phase takes a partition assignment projected from a coarser hypergraph and improves it using a local optimization method. The most successful refinement methods are variations of Kernighan–Lin (KL) [15] and Fiduccia–Mattheyses (FM) [7]. These are iterative methods that move (or swap) vertices from one partition to another based on *gain* values, that is, how much the cut weight decreases by the move. While greedy algorithms are often preferred in parallel because they are simpler and faster, they generally do not produce partition quality as good as KL/FM. Thus, we have adopted an FM-like approach.

We have implemented a parallel two-way ($k = 2$) refinement heuristic based on FM. The algorithm performs multiple *pass-pairs* until either a predefined *pass limit* is reached or no further improvement is achieved in the last pass-pair. Each pass-pair consists of two consecutive passes where, on each pass, vertices from alternating partitions are moved to the other partition. Doing one-directional moves on each pass guarantees that none of the concurrent moves adversely affects the gain of vertices in other processors. In other words, if a set of vertices is moved to the other partition, the actual reduction in the cut metric (1) is at least the sum of the gains of the vertices moved.

Our local refinement is performed with the goal of improving balance and cuts within processor columns and, thus, in the global partitioning. At the beginning of each pass, even though the partitioning satisfies the global balance constraints, local balances on each processor column might violate the balance constraint. Based on this initial distribution, we adjust the move-feasibility constraints on each processor to guarantee that no moves violate the global balance constraint. Each processor contributes to the computation of vertex gains at the beginning of a pass. Within each processor column, the processor with the largest number of nonzeros is selected as the *vertex mover*. The vertex mover tries to move all vertices from the source partition to the destination partition without violating the balance constraint. After moving each vertex, the vertex mover updates the gain values of the adjacent vertices using only its local data. Although this scheme deviates from the original FM algorithm, it allows each vertex mover to work concurrently without any synchronization. By selecting the processor with the largest number of nonzeros, we make more informed vertex moves.

We have observed that our parallel method produces quite good partitionings, but it is possible that its effectiveness will decrease for very large numbers of processors due to its local perspective. In future versions, we will explore other parallel refinement schemes.

3.5 Recursive Bisection Data Splitting

After a multilevel V-cycle computes a bipartitioning of the hypergraph, we split the hypergraph into two subsets (one for each partition), and apply the multilevel algorithm recursively to each subset. There are two options for managing data during parallel recursive bisection. One option is to leave the data in place, and let all processors first work on one subset of the hypergraph, then the other subset. The advantage of this approach is that no data movement is required.

An alternative approach is to split the processors into two subsets, and move the hypergraph corresponding to each subset (after the first bisection step) onto separate sets of processors. This allows each half of the processors to work on independent subproblems simultaneously. There is a cost associated with remapping (moving) the hypergraph data, but the communication cost within the partitioning is reduced because the communication becomes more local. Moreover, each processor in the subset gets a larger percentage of the split hypergraph and, thus, a more complete view of it, resulting in higher quality. Figure 2 illustrates processor and hypergraph splitting during recursive bisection with $p = 6$. In this example, partitioning of the hypergraph is illustrated with color-coded vertices (circles) and hyperedges (squares). Red and blue circles represent vertices assigned to partitions 0 and 1, respectively. Yellow and magenta squares represent un-cut and cut hyperedges. During splitting, vertices of partition 0 are moved to four processors (P_{11}, P_{12}, P_{21} and P_{22}); vertices of partition 1 are moved to the remaining 2 processors (P_{31}, P_{32}). Uncut hyperedges are preserved. But to accurately measure cut size in further bisections, each cut hyperedge is split into two hyperedges: one connecting vertices in partition 0, and one connecting vertices in partition 1. If this process yields hyperedges of size one, they are discarded, as they cannot contribute to the cut metric in further bisections.

We have tested both strategies in our code. Splitting the processors both reduced execution time and improved quality. Thus, we made splitting the default option and used it in all experiments reported here.

4 Experimental Results

Our parallel hypergraph code is part of the Zoltan [6, 18] toolkit for load balancing and parallel data management, which is open-source and freely available. It has been implemented in ANSI C and uses MPI for communication. Only collective communication calls are implemented in MPI directly; unstructured (point-to-point) communication is performed in BSP-like [1] supersteps via Zoltan's communication layer.

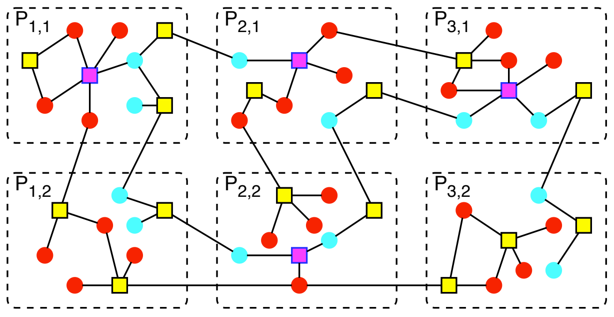
We ran our tests on a Linux cluster at Sandia that has dual-processor Intel Xeon (3.0 GHz) nodes. Its interconnect network is Myrinet-2000.

4.1 Test data and software

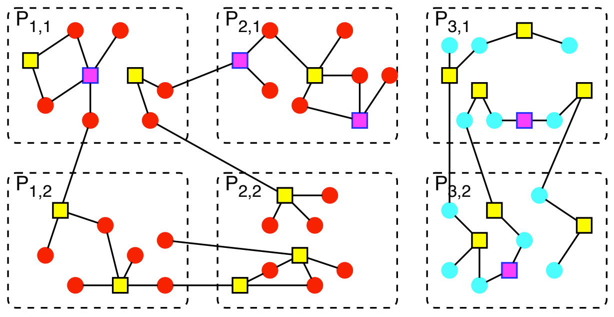
We collected a set of test problems from a variety of applications. A summary of the test hypergraphs and matrices is given in Table 1. The matrices 2DLipidFMat and polyDFT are from the Tramonto density functional theory code and represent self-assembly of lipid bilayers and polymers, respectively; cage14 is a DNA electrophoresis model; d256 is a random matrix in which each vertex has degree 256; ibm18 is the largest test problem in the ISPD98 circuit benchmark suite; roads2 is a mixed-integer linear programming matrix from sensor placement; StanfordBerkeley is a matrix representing web links between Stanford's and Berkeley's web sites; StanfordBerkeleyT is the transpose of the StanfordBerkeley matrix; tbdlinux is a term-by-document matrix from a Linux manual; voting250 represents a Markov transition matrix from a model of voters and polling stations; and the xyce680 problem represents an ASIC model. The xyce680 problem comes in two variations: a "base" version and a "stripped" version where dense rows and columns in the matrix have been removed. For all experiments, we partitioned matrix columns with unit weights per column.

The partitioners we used were ParMETIS 3.1 [14] (parallel graph partitioner), PaToH 3.0 [5] (serial hypergraph partitioner), Par$k$way 2.0 [16] (parallel hypergraph partitioner), and our own parallel hypergraph partitioner implemented in Zoltan. Although Zoltan has many different partitioners, we use the name Zoltan in this paper to denote its parallel hypergraph partitioner. Par$k$way can use PaToH or hMETIS as its coarse partitioner; we used PaToH with Par$k$way as it was faster and gave cut quality at least as good as hMETIS. In all experiments, we requested a load balance tolerance of 10%; that is, $\text{load}_{\max}/\text{load}_{avg} \leq 1.1$. All partitioners managed to satisfy this condition, so we do not report the actual load imbalances. All data is based on the average of 25 runs. Par$k$way results for some problems are based on fewer than 25 iterations due to their long run times.

Because graph partitioners operate on undirected graphs, we ran ParMETIS only on symmetric problems. Nonsymmetric matrices can be symmetrized into (undirected) graphs, but it has been shown [3] that graph partitioning gives a worse approximation to communication volume for nonsymmetric than symmetric problems, so the hypergraph model is clearly preferable in such cases.



(a) Before split



(b) After split

Figure 2. An illustration of splitting for p=k=6. For the sake of illustration, assume that the first bisection has a ratio of 4 to 2, and processors are also split into two by the same ratio. Circles represent vertices (columns), and squares represent hyperedges (rows).

Name	$\|V\|$ (cols)	$\|E\|$ (rows)	pins (nonzeros)	Sym.	Application Area
2DLipidFMat	4,368	4,368	5,592,344	Yes	Lipid bilayer self-assembly
cage14	1,505,785	1,505,785	27,130,349	Yes	DNA electrophoresis
d256	100,000	100,000	25,600,000	No	Random matrix
ibm18	210,613	201,920	819,697	No	VLSI design
polyDFT	46,176	46,176	3,690,048	No	Polymer self-assembly
roads2	1,284,498	1,683,554	7,713,905	No	Sensor placement
StanfordBerkeley	683,446	683,446	7,583,376	No	Web-links
StanfordBerkeleyT	683,446	683,446	7,583,376	No	Transpose of StanfordBerkeley
tbdlinux	112,757	20,167	2,157,675	No	Information retrieval
voting250	5,218,300	5,218,300	32,986,597	No	Markov process
Xyce680b	682,862	682,862	3,871,773	Yes	VLSI design
Xyce680s	682,712	682,712	2,329,176	Yes	VLSI design

Table 1. Test hypergraphs and matrices.

4.2 Comparison of partitioners

First, we consider the case where we fix the number of partitions $k = 64$ and the number of processors $p = 1$ and $p = 64$. We show in Tables 2 and 3 the cut metric (1) and partitioning times for each test matrix. Results marked with "-" indicate unsuccessful runs, typically due to insufficient memory. For $p = 1$, we compare Zoltan's hypergraph partitioner with PaToH and METIS. METIS was applied only to symmetric test matrices. With PaToH, the processor had insufficient memory to partition the cage14, voting250, and d256 matrices. Thus, we exclude voting250 and d256 from Table 2, as Zoltan was the only method to partition them. For $p = 64$, we compare Zoltan's hypergraph partitioner with Par$k$way and ParMETIS. ParMETIS was applied only to symmetric test matrices. Par$k$way ran out of memory for the d256, StanfordBerkeley, and Xyce680b matrices; we exclude d256 from Table 3 because Zoltan was the only method that partitioned it.

As expected, hypergraph partitioners generally give higher quality (i.e., lower cut metric (1)) than graph partitioners. This higher quality translates directly into reduced communication volume for operations such as matrix-vector multiplication. Furthermore, we see that Zoltan's partition quality is competitive with the serial hypergraph partitioner PaToH. Par$k$way sometimes produces fewer cuts than Zoltan. However, Zoltan runs are typically much faster than Par$k$way.

The Xyce680 problems display a dramatic difference between the base and the stripped version. Zoltan and PaToH have implemented special handling of dense hyperedges; we applied this same handling to Par$k$way's input as well. It is much more difficult for a graph partitioner to detect such structure, and we see that ParMETIS ran slowly and showed fluctuating quality for the base version. It ran more quickly on the stripped version.

4.3 Scalability

We examine the scalability of Zoltan, Par$k$way, and ParMETIS on a subset of our test problems. We select cage14, 2DLipidFMat and Xyce680s, as their symmetry allows us to apply ParMETIS to them. Additionally, these test problems represent well the range of performance we

		Hypergraph cut metric normalized w.r.t. Zoltan			Partitioning time in seconds		
Matrix	Norm. Value	METIS	PaToH	Zoltan	METIS	PaToH	Zoltan
2DLipidFMat	107,736	1.82	1.29	1.00	7	17	109
cage14	1,550,618	1.15	-	1.00	26	-	829
ibm18	25,787		1.01	1.00		14	16
polyDFT	82,420		0.96	1.00		34	36
roads2	5,681		0.65	1.00		30	66
StanfordBerkeley	25,676		0.89	1.00		33	2489
StanfordBerkeleyT	158,064		1.20	1.00		84	45
tbdlinux	232,895		1.00	1.00		29	73
Xyce680b	726,663	1.23	0.94	1.00	913	1650	36
Xyce680s	25,521	2.12	0.83	1.00	17	12	29

Table 2. Comparison of Zoltan hypergraph partitioning with METIS graph partitioning and PaToH hypergraph partitioning with $k = 64$ on one processor.

		Hypergraph cut metric normalized w.r.t. Zoltan $p = 1$			Partitioning time in seconds		
Matrix	Norm. Value	ParMETIS	Par$k$way	Zoltan	ParMETIS	Par$k$way	Zoltan
2DLipidFMat	107,736	1.90	1.08	1.05	5	466	4
cage14	1,550,618	1.26	0.97	1.02	2	2899	79
ibm18	25,787		1.12	1.08		55	5
polyDFT	82,420		0.90	1.03		93	3
roads2	5,681		0.68	1.13		17	19
StanfordBerkeley	25,676		-	0.86		-	805
StanfordBerkeleyT	158,064		1.02	1.36		144	16
tbdlinux	232,895		1.05	1.07		565	73
voting250	511,609		1.03	1.04		266	117
Xyce680b	726,663	1.15	-	0.99	1027	-	13
Xyce680s	25,521	2.20	0.89	1.07	13	10	11

Table 3. Comparison of Zoltan hypergraph partitioning with ParMETIS graph partitioning and Par$k$way hypergraph partitioning with $k = 64$ on 64 processors.

observe for the entire test suite.

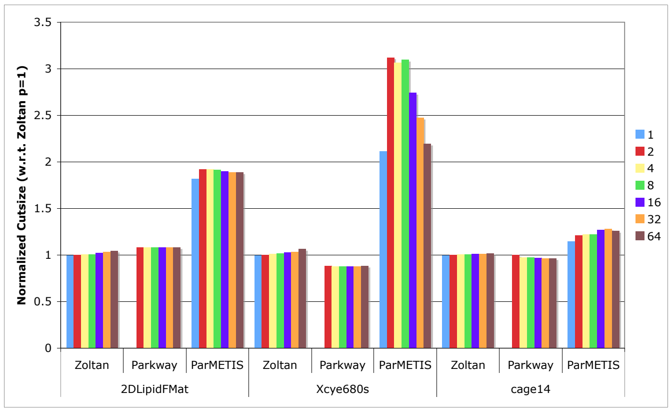
For each test problem, we set $k = 64$, and we vary p from $p = 1$ to 64. We partition each matrix with Zoltan, Par$k$way, and ParMETIS. (Par$k$way does not run with $p = 1$, so Par$k$way results are shown only for $p > 1$.) In Figure 3, we show the cut metric (1) normalized with respect to the cut metric for Zoltan with $p = 1$. We observe that Zoltan's partition quality remains stable with increasing p.

In Figure 3, we also show the execution times for each partitioner. Again, we see that while Zoltan produces decompositions with lower communication volume than ParMETIS, it is typically slower than ParMETIS. Par$k$way sometimes obtains even fewer cuts but takes a longer time. The run time of Zoltan generally decreases with p but scalability is not perfect. Its scalability is comparable to or better than that of ParMETIS and Par$k$way, respectively.

4.4 2D processor configuration

We test the effectiveness of Zoltan's 2D processor configuration with experiments using various 1D and 2D processor configurations. By default, Zoltan's 2D processor layout is as square as possible; that is, $p_x \approx p_y$. A square layout may not be optimal for each phase of the algorithm (e.g., matching, coarsening, refinement), as each phase has different communication requirements in the x and y dimensions. Our approach assumes that distributions with $p_x \approx p_y$ work well on average.

Choosing $p = 64$, we experimented with different values of p_x and p_y. We set $k = 2$ to ensure that the specified ratio could be maintained throughout the computation; for $k > 2$, the recursive bisection would require some variance in the processor ratios. Results for several test matrices and configurations of $p_x \times p_y$ are shown in Figure 4. We observe that rectangular configurations with p_x small but greater than one generally worked best. Interestingly, we



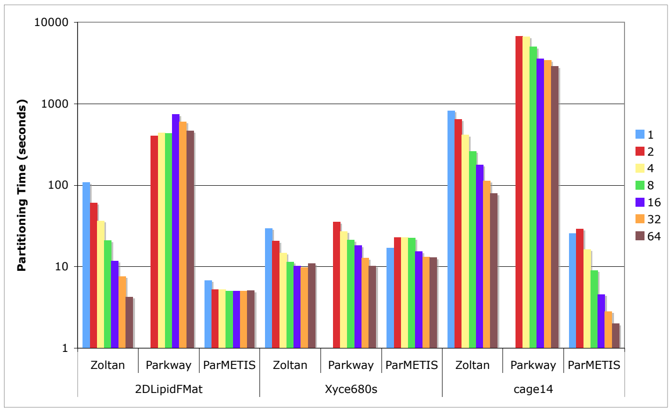


Figure 3. A comparison of partition quality (left) and execution time (right) for Zoltan, Par$k$way, and ParMETIS with $k = 64$ on $p = 1$ to 64 processors.

observe that the "natural" strategy of distributing vertices among processors and maintaining full connectivity information for each vertex ($p_y = 1$) worked least well. This strategy is typical in parallel graph partitioners. Distributing hyperedges among processors while maintaining full vertex information for each hyperedge ($p_x = 1$) was the better 1D layout on most problems, but generally less efficient than a 2D approach.

5 Conclusions and Future Work

We have described the design of a parallel multilevel hypergraph (or sparse matrix) partitioner that runs on distributed-memory computers. We have shown that the partition quality is similar to that of serial hypergraph partitioners (PaToH), and in most cases it runs much faster than the recent similar code Par$k$way. Our code gives fairly good parallel speedup without compromising partition quality on a Linux cluster with up to 64 processors. Our 2D data distribution gives better performance than the standard 1D layout.

As future work, we want to improve the parallel scalability so that partitioning will be fast even on machines with thousands of processors. In order to achieve that, we will investigate faster matching and coarsening methods. We do not necessarily need to do the complete inner product matching (which requires a lot of communication); other simpler (approximate) and faster variations are under development.

Another potential source of improvement is better load balancing within the partitioner. This is a chicken-and-egg problem, since hypergraph partitioning is the best way to compute a good data distribution! However, simpler heuristics to balance the work may prove useful. Finally, we plan to add new features to our partitioner, in particular, dynamic repartitioning and multi-constraint partitioning.

Acknowledgments

We thank Bruce Hendrickson for his advice and guidance in this project, and Robert Preis for his help developing a serial partitioner that preceded the present work. We also thank Alex Trifunovic for providing the Par$k$way code and helping us use it. We thank Tim Davis, Laura Frink, Mike Heroux, Rob Hoekstra, and Michael Collins for providing test matrices. And we thank Aaron Becker for simplifying our experiment procedures.

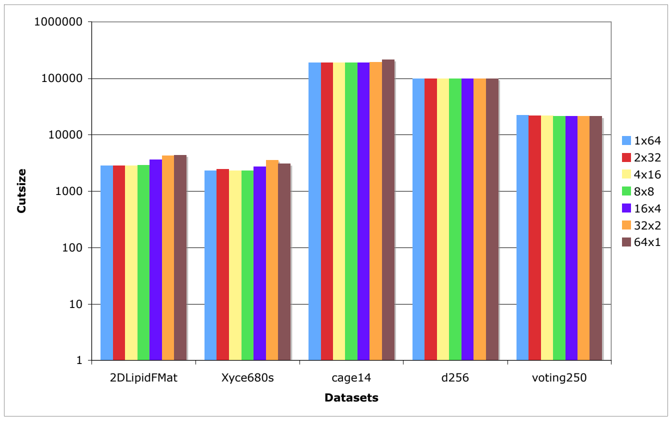
References

[1] R. H. Bisseling. *Parallel Scientific Computing: A structured approach using BSP and MPI*. Oxford University Press, 2004.

[2] A. E. Caldwell, A. B. Kahng, and I. L. Markov. Improved algorithms for hypergraph bipartitioning. In *Proc. Asia and South Pacific Design Automation Conf.*, pages 661–666. IEEE Press/ ACM Press, 2000.

[3] Ü. Çatalyürek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Dist. Systems*, 10(7):673–693, 1999.

[4] Ü. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proc.*



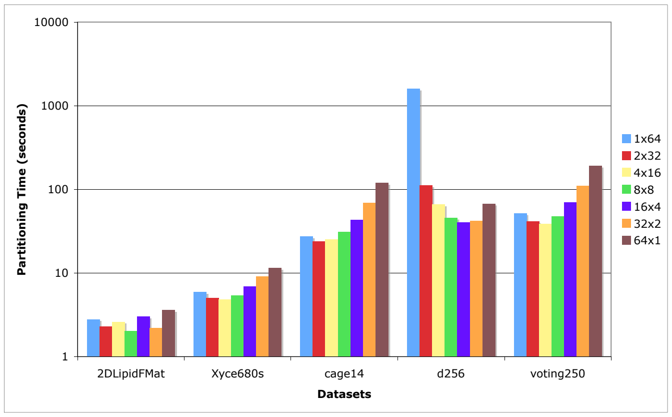


Figure 4. A comparison of partition quality (left) and execution time (right) for Zoltan using various processor configurations $p_x \times p_y$ with $k = 2$.

IPDPS 8th Int'l Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001), April 2001.

[5] U. V. Çatalyürek and C. Aykanat. *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0.* Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at http://bmi.osu.edu/∼umit/software.htm, 1999.

[6] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, and C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.

[7] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, 1982.

[8] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519 – 1534, 2000.

[9] B. Hendrickson and T. G. Kolda. Partitioning rectangular and structurally nonsymmetric sparse matrices for parallel computation. *SIAM Journal on Scientific Computing*, 21(6):2048–2072, 2000.

[10] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95*. ACM, December 1995.

[11] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: application in VLSI domain. In *Proc. 34th Conf. Design Automation*, pages 526 – 529. ACM, 1997.

[12] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95, 1998.

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), 1999.

[14] G. Karypis, K. Schloegel, and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library, version 3.1. Technical report, Dept. Computer Science, University of Minnesota, 2003. `http://www-users.cs.umn.edu/~karypis/metis/parmetis/download.html`.

[15] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.

[16] A. Trifunovic and W. J. Knottenbelt. Parkway 2.0: A parallel multilevel hypergraph partitioning tool. In *Proc. 19th International Symposium on Computer and Information Sciences (ISCIS 2004)*, volume 3280 of *LNCS*, pages 789–800. Springer, 2004.

[17] 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.

[18] Zoltan: Data management services for parallel applications. http://www.cs.sandia.gov/Zoltan/.