

## Parallel Partitioning with Zoltan: Is Hypergraph Partitioning Worth It?

Sivasankaran Rajamanickam and Erik G. Boman

**ABSTRACT.** Graph partitioning is an important and well studied problem in combinatorial scientific computing, and is commonly used to reduce communication in parallel computing. Different models (graph, hypergraph) and objectives (edge cut, boundary vertices) have been proposed. Hypergraph partitioning has become increasingly popular over the last decade. Its main strength is that it accurately captures communication volume, but it is slower to compute than graph partitioning. We present an empirical study of the Zoltan parallel hypergraph and graph (PHG) partitioner on graphs from the 10th DIMACS implementation challenge and some directed (nonsymmetric) graphs. We show that hypergraph partitioning is superior to graph partitioning on directed graphs (nonsymmetric matrices), where the communication volume is reduced in several cases by over an order of magnitude, but has no significant benefit on undirected graphs (symmetric matrices) using current parallel software tools.

### 1. Introduction

Graph partitioning is a well studied problem in combinatorial scientific computing. An important application is the mapping of data and/or tasks on a parallel computer, where the goals are to balance the load and to minimize communication [12]. There are several variations of graph partitioning, but they are all NP-hard problems. Fortunately, good heuristic algorithms exist. Naturally, there is a trade-off between run-time and solution quality. In parallel computing, partitioning may be performed either once (static partitioning) or many times (dynamic load balancing). In the latter case, it is crucial that the partitioning itself is fast. Furthermore, the rapid growth of problem sizes in scientific computing dictates that partitioning algorithms must be scalable. The multilevel approach developed in

---

1991 *Mathematics Subject Classification.* Primary 68R10; Secondary 05C65, 68W10, 68Q85.

*Key words and phrases.* graph partitioning, hypergraph partitioning, parallel computing.

We thank the U.S. Department of Energy's Office of Science, the Advanced Scientific Computing Research (ASCR) office, and the National Nuclear Security Administration's ASC program for financial support. Sandia is a multiprogram laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the DOE under Contract No. DE-AC02-05CH11231.

the 1990s [3, 11, 17] provides a good compromise between run-time (complexity) and quality. Software packages based on this approach (Chaco [13], Metis [14], and Scotch [19]) have been extremely successful. Even today, all the major parallel software packages for partitioning in scientific computing (ParMetis [15], PT-Scotch [20], and Zoltan [8, 9]) use variations of the multilevel graph partitioning algorithm.

The 10th DIMACS implementation challenge offers an opportunity to evaluate the current (2012) state-of-the-art in partitioning software. This is a daunting task, as there are several variations of the partitioning problem (e.g., objectives), several software codes, and a large number of data sets. In this paper we limit the scope in the following ways: We only consider parallel software since our focus is high-performance computing. We focus on the Zoltan toolkit since its partitioner can be used to minimize either the edge cut (graph partitioning) or the communication volume (hypergraph partitioning). We include some baseline comparisons with ParMetis, since that is the most widely used parallel partitioning software. We limit the experiments to a subset of the DIMACS graphs. One may view this paper as a follow-up to the 2006 paper that introduced the Zoltan PHG partitioner [9].

Contributions: We compare graph and hypergraph partitioners for both symmetric and unsymmetric inputs and obtain results that are quite different than in [4]. For nonsymmetric matrices we see a big difference in communication volume (orders of magnitude), while there is virtually no difference among the partitioners for symmetric matrices. We exercise Zoltan PHG on larger number of processors than before (up to 1024). We present results for impact of partitioning on an iterative solver. We also include results for the maximum communication volume, which is important in practice but not an objective directly modeled by any current partitioner.

## 2. Models and Metrics

The term “graph partitioning” can refer to several different problems. Most often, it refers to the edge cut metric, though in practice the communication volume metric is often more important. For the latter objective, it is useful to extend graphs to hypergraphs. Here, we review the different models and metrics and explain how they relate.

**2.1. Graph Models.** Given an undirected graph  $G = (V, E)$ , the classic version of graph partitioning is to partition  $V$  into  $k$  disjoint subsets (parts) such that all the parts are approximately the same size and the total number of edges between parts are minimized. More formally, let  $\Pi = \{\pi_0, \dots, \pi_{k-1}\}$  be a balanced partition such that

$$(1) \quad |V(\pi_i)| \leq (1 + \epsilon) \frac{|V|}{k} \quad \forall i,$$

for a given  $\epsilon > 0$ . The edge cut problem (EC) is then to minimize the cut set

$$(2) \quad C(G, \Pi) = \{|(u, v) \in E| \Pi(u) \neq \Pi(v)\}.$$

There are straight-forward generalizations for edge weights (minimize weighted cuts) and vertex weights (balance is weighted).

Most algorithms and software attempt to minimize the edge cut. However, several authors have shown that the edge cut does not represent communication in

parallel computing [4, 12]. A key insight was that the communication is proportional to the *vertices* along the part boundaries, not the cut edges. A more relevant metric is therefore the communication volume, which roughly corresponds to the boundary vertices. Formally, let the communication volume for part  $p$  be

$$(3) \quad comm(\pi_p) = \sum_{v \in \pi(p)} (\lambda(v, \Pi) - 1),$$

where  $\lambda(v, \Pi)$  denotes the number of parts that  $v$  or any of its neighbors belong to, with respect to the partition  $\Pi$ .

We then obtain the following two metrics:

$$(4) \quad CV_{max}(G, \Pi) = \max_p comm(\pi_p)$$

$$(5) \quad CV_{sum}(G, \Pi) = \sum_p comm(\pi_p)$$

In parallel computing, this corresponds to the maximum communication volume for any process and the total sum of communication volumes, respectively.

**2.2. Hypergraph Models.** A *hypergraph*  $H = (V, E)$  extends a graph since now  $E$  denotes a set of hyperedges. An hyperedge is any non-empty subset of the vertices  $V$ . A graph is just a special case of a hypergraph where each hyperedge has cardinality two (since a graph edge always connects two vertices). Hyperedges are sometimes called *nets*, a term commonly used in the (VLSI) circuit design community.

Analogous to graph partitioning, one can define several hypergraph partitioning problems. As before, the balance constraint is on the vertices. Several different cut metrics have been proposed. The most straight-forward generalization of edge cut to hypergraphs is:

$$(6) \quad C(H, \Pi) = \{ \{e \in E\} \mid \Pi(u) \neq \Pi(v) \text{ where } u \in e, v \in e \}.$$

However, a more popular metric is the so-called  $(\lambda - 1)$  metric:

$$(7) \quad CV(H, \Pi) = \sum_{e \in E} (\lambda(e, \Pi) - 1),$$

where  $\lambda(e, \Pi)$  is the number of distinct parts that contain any vertex in  $e$ .

While graphs are restricted to structurally symmetric problems (undirected graphs), hypergraphs make no such assumption. Furthermore, the number of vertices and hyperedges may differ, making the model suitable for rectangular matrices. The key advantage of the hypergraph model is that the hyperedge  $(\lambda - 1)$  cut (CV) accurately models the total communication volume. This was first observed in [4] in the context of sparse matrix-vector multiplication. The limitations of the graph model were described in detail in [12]. This realization led to a shift from the graph model to the hypergraph model. Today, many partitioning packages use the hypergraph model: PaToH [4], hMetis [16], Mondriaan [21], and Zoltan-PHG [9].

Hypergraphs are often used to represent sparse matrices. For example, using row-based storage (CSR), each row becomes a vertex and each column becomes a hyperedge. Other hypergraph models exist: in the “fine-grain” model, each non-zero is a vertex [5]. For the DIMACS challenge, all input is symmetric and given as undirected graphs. Given a graph  $G(V, E)$ , we will use the following derived hypergraph  $H(V, E')$ : for each vertex  $v \in V$ , create an hyperedge  $e \in E'$  that

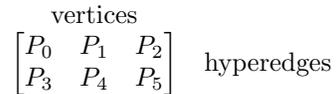
contains  $v$  and all its neighbors. In this case, it is easy to see that  $CV(H, \Pi) = CV_{sum}(G, \Pi)$ . Thus, we do not need to distinguish between communication volume in the graph and hypergraph models.

**2.3. Relevance of the Metrics.** Most partitioners minimize either the total edge cut (EC) or the total communication volume (CV-sum). A main reason for this choice is that algorithms for these metrics are well developed. Less work has been done to minimize the maximum communication volume (CV-max), though in a parallel computing setting this may be more relevant as it corresponds to the maximum communication for any one process.

In order to compare the three metrics and how they correspond to the actual performance we use conjugate gradient (CG) iteration (from the Belos package [1]) as a test case. We used the matrices from the UF sparse matrix collection group of the DIMACS challenge. As the goal is to compare the matrix-vector multiply time in the CG iteration, we used no preconditioner as the performance characteristics will be different depending on the preconditioners. As there is no preconditioner and some of these problems are ill-conditioned the CG iteration might not converge at all, so we report the solve time for 1000 iterations. We compare four different row-based partitionings (on 12 processors): natural (block) partitioning, random partitioning, graph partitioning with ParMetis, and hypergraph partitioning with Zoltan hypergraph partitioner. We only change the data distribution, and do not reorder the matrix, so the convergence of CG is not affected. The results are shown in Table 1. As expected, random partitioning is worst since it just balances the load but has high communication. In all but one case, we see that both graph and hypergraph partitioning beat the simple natural (block) partitioning (which is the default in Trilinos). For the audikw1 test matrix, the time is cut to less than half. For these symmetric problems, the difference between graph and hypergraph partitioning is very small in terms of real performance gains. We will show in Section 4.2 that the partitioners actually differ in terms of the measured performance metrics for three of the problems shown in Table 1. However, the difference in the metrics do not translate to measurable real performance gain in the time for the matrix-vector multiply.

TABLE 1. Solve time (seconds) for 1000 iterations of CG for different row partitioning options.

Matrix Name	Natural	Random	ParMetis	Zoltan PHG
audikw1	62.90	98.58	27.71	27.52
ldoor	22.18	72.09	18.24	18.08
G3_circuit	11.26	25.78	8.13	8.62
af_shell10	20.09	84.51	21.29	21.17
bone010	24.33	84.07	24.92	25.39
geo_1438	25.35	106.36	25.53	25.78
inline_1	22.47	44.57	13.54	13.90
pwtk	4.30	11.88	4.34	4.37

FIGURE 1. Example of the 2D layout for  $2 \times 3$  processes.

### 3. Overview of the Zoltan Hypergraph Partitioner

Zoltan was originally designed as a toolkit for dynamic load-balancing [8]. It included several geometric partitioning algorithms, plus interfaces to external (third-party) graph partitioners, such as ParMetis. Later, a native parallel hypergraph partitioner (PHG) was developed [9] and added to Zoltan. While PHG was designed for hypergraph partitioning, it can also be used for graph partitioning but it is not optimized for this use case. (Note: “PHG” now stands for Parallel Hypergraph and Graph partitioner.) Zoltan also supports other combinatorial problems such as graph ordering and graph coloring [2].

Zoltan PHG is a parallel multilevel partitioner, consisting of the usual *coarsening*, *initial partitioning*, and *refinement* phases. The algorithm is similar to the serial partitioners PaToH [4], hMetis [16] and Mondriaan [21], but Zoltan PHG is parallel (based on MPI) so can run on both shared-memory and distributed-memory systems. Note that Zoltan can partition data into  $k$  parts using  $p$  processes, where  $k \neq p$ . Neither  $k$  nor  $p$  need be powers of two. We briefly describe the algorithm in Zoltan PHG, with emphasis on the parallel computing aspects. For further details on PHG, we refer to [9]. The basic algorithm remains the same, though several improvements have been made over the years.

**3.1. 2D Data Distribution.** A novel feature of Zoltan PHG is that internally, the hypergraph is mapped to processes in a 2D block (checkerboard) fashion. That is, the processes are logically mapped to a  $p_x$  by  $p_y$  grid, where  $p = p_x p_y$ . The hypergraph is partitioned accordingly, when viewed as a sparse matrix (Fig. 1). We do not attempt an optimal 2D Cartesian (checkerboard) distribution: The best known algorithm requires multiconstraint hypergraph partitioning [6], which is even harder than the partitioning problem we wish to solve.

The goal of this design is to reduce communication within the partitioner itself. Instead of expensive all-to-all or any-to-any communication, all communication is limited to process rows or columns. Thus, the collective communication is limited to communicators of size  $p_x$  or  $p_y$ , which is  $O(\sqrt{p})$  for squarish configurations. The drawback of this design is that there are more synchronization points than if an 1D distribution had been used. Furthermore, neither vertices nor hyperedges have unique owners, but are spread over multiple processes. This made the 2D parallel implementation quite complex and challenging. 2D data distributions have recently been used in several applications, such as sparse matrix-vector multiplication in eigensolvers [22]. SpMV is a fairly simple kernel to parallelize. 2D distributions are still rarely used in graph algorithms, probably due to the complexity of implementation and the lack of payoff for small numbers of processors.

**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. The most popular methods merge pairs of vertices, but one can also aggregate more than two vertices at a time. 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 [4] suggested a heavy-connectivity matching, which measures a similarity metric between pairs of vertices. Their preferred similarity metric, which was also adopted by hMETIS [16] and Mondriaan [21], is known as the *inner product* or simply, *heavy connectivity*. 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.) Zoltan PHG also uses the heavy-connectivity (inner-product) metric in the coarsening. Originally only pairs of vertices were merged (matched) but later vertex aggregation (clustering) that allows more than two vertices to be merged was made the default as it produces slightly better results.

Previous work have shown that greedy strategies work well in practice so optimal matching based on similarity scores (inner products) is not necessary. The sequential greedy algorithm works as follows. Pick a (random) unmatched vertex  $v$ . For each unmatched neighbor vertex  $u$ , compute the inner product  $\langle v, u \rangle$ . Select the vertex with the highest non-zero inner product value and match it with  $v$ . Repeat until all vertices have been considered. If we consider the hypergraph as a sparse matrix  $A$ , we essentially need to compute the matrix product  $A^T A$ . We can use the sparsity of  $A$  to compute only entries of  $A^T A$  that may be nonzero. Since we use a greedy strategy, we save work and compute only a subset of the nonzero entries in  $A^T A$ . This strategy has been used (successfully) in several serial partitioners.

With Zoltan's 2D data layout, this fairly simple algorithm becomes much more complicated. Each processor knows about only a subset of the vertices and the hyperedges. Computing the inner products requires communication. 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 the 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 given in [9].

Observe that the full heavy connectivity 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. Potentially, one could save substantial time in the coarsening phase by using a cheaper heuristic that gives preference to local data. We have experimented with several such strategies, but the faster run time comes at the expense of the partitioning quality. Therefore, the default in Zoltan PHG is heavy connectivity aggregation, which was also used in our experiments.

After the matching or aggregation has been computed, we build the coarser hypergraph by merging matched vertices. Note that hyperedges are not contracted, leading to unsymmetry. The matrix corresponding to the hypergraph becomes more rectangular at every level of coarsening. The number of hyperedges is only reduced in two ways: (a) hyperedges that become internal to a coarse vertex are simply deleted, and (b) identical hyperedges are collapsed into a single hyperedge with adjusted weight.

**3.3. Initial Partitioning.** The coarsening stops when the hypergraph is smaller than a certain threshold. Since the coarsest hypergraph is small, we replicate it on every process. Each processor runs a randomized greedy algorithm to compute a different partitioning. We then evaluate the desired cut metric on each processor and pick the globally best partitioning, which is broadcast to all processes.

**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) [18] and Fiduccia–Mattheyses (FM) [10]. 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, Zoltan PHG uses an FM-like approach but made some changes to accommodate the 2D data layout.

Since Zoltan PHG uses recursive bisection, only two-way refinement is needed. The main challenge with the 2D layout is that each vertex is shared among several processes, making it difficult to compute gain values and also to decide which moves to actually perform (as processes may have conflicting local information). The strategy used in PHG is a compromise between staying faithful to the FM algorithm and accommodating more concurrency in the 2D parallel setting. See [9] for further details. Although the refinement in PHG works well on moderate number of processors, the quality will degrade for very large number of processes.

**3.5. Recursive Bisection.** Zoltan PHG uses recursive bisection to partition into  $k$  parts. Note that  $k$  can be any integer greater than one, and does not need to be a power of two. Also, Zoltan can run on  $p$  processes, where  $k \neq p$ . However, the typical use case is  $k = p$ .

An important design choice in the recursive bisection is whether the data is left in-place or moved onto separate subsets of processors. The first approach avoids some data movement but the latter reduces communication in the partitioner and allows more parallelism. Initial experiments indicated that moving the data

and splitting into independent subproblems gave better performance, so this is the default in Zoltan PHG.

**3.6. PHG as a Graph Partitioner.** PHG was designed as a hypergraph partitioner but can also do graph partitioning since a graph is just a special case of a hypergraph. When PHG is used as a graph partitioner, each hyperedge is of size two. When we coarsen the hypergraph, only vertices are coarsened, not hyperedges. This means that the symmetry of graphs is destroyed already after the first level of coarsening. We conjecture that PHG is not particularly efficient as a graph partitioner because it does not take advantage of the special structure of graphs (in particular, symmetry and constant size hyperedges). Still, we believe it is interesting (and fair) to compare PHG as a graph partitioner because it uses exactly the same code as the hypergraph partitioner, so any performance difference is due to the model not the implementation.

## 4. Experiments

**4.1. Software, Platform, and Data.** Our primary goal is to study the behavior of Zoltan PHG as a graph and a hypergraph partitioner, using different objectives and a range of data sets. We use Zoltan 3.5 (Trilinos 10.8) and ParMetis 4.0 as a reference for all the tests. The compute platform was mainly Hopper, a Cray XE6 at NERSC. Hopper has 6,384 compute nodes, each with 24 cores (two 12-core AMD MagnyCours) and 32 GB of memory. The graphs for the tests are from five test families of the DIMACS collection that are relevant to the computational problems we have encountered at Sandia. Within each family, we selected some of the largest graphs that were not too similar. In addition we picked four other graphs, two each from the street networks and clustering instances (which also happened to be road networks), to compile our diverse 22 test problems.

The graphs are partitioned into 16, 64, 256, and 1024 parts. In the parallel computing context, this covers everything from a multicore workstation to a medium-sized parallel computer. Except where stated otherwise, the partitioner had the same number of MPI ranks as the target number of parts.

Zoltan uses randomization, so results may vary slightly from run to run. However, for large graphs, the random variation is relatively small. Due to limited compute time on Hopper, each partitioning test was run only once. Even with the randomization, it is fair to draw conclusions based on several data sets, though one should be cautious about overinterpreting any single data point.

**4.2. Zoltan vs. ParMetis.** In this section, we compare Zoltan’s graph and hypergraph partitioning with ParMetis’s graph partitioning. We partition the graphs into 256 parts with 256 MPI processes. The performance profile of the three metrics – total edge cut (EC), the maximum communication volume (CV-max) and the total communication volume (CV-sum) – for the 22 matrices is shown in Figure 2.

The main advantage of the hypergraph partitioners is the ability to handle unsymmetric problems and to reduce the communication volume for such problems directly (without symmetrizing the problems). However, all the 22 problems used for the comparisons in Figure 2 are symmetric problems from the DIMACS challenge set. We take this opportunity to compare graph and hypergraph partitioners even for symmetric problems.

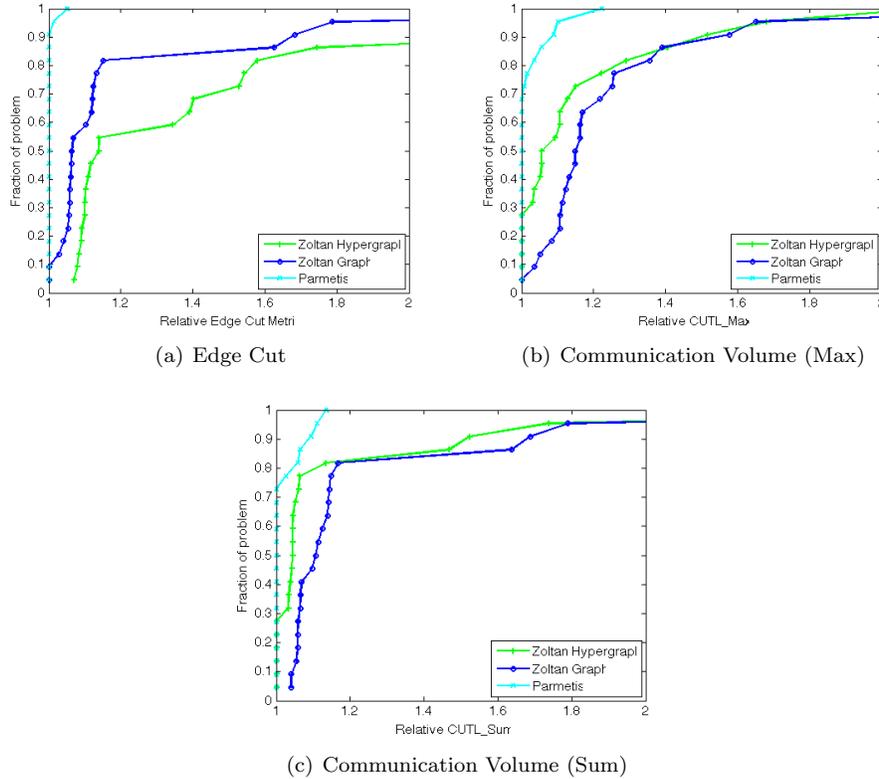


FIGURE 2. **Zoltan Vs Parmetis:** Comparing Zoltan’s partitioning with graph and hypergraph model with Parmetis for symmetric problems for 256 parts and 256 MPI processes.

In terms of the edge cut metric ParMetis does better than Zoltan for 20 of the matrices and Zoltan’s graph model does better for just two matrices. However, Zoltan’s graph model is within 15% of ParMetis’s edge cuts for 82% of the problems (see Figure 2(a)). The four problems that cause trouble to Zoltan’s graph model are the problems from the street networks and clustering instances.

In terms of the CV-sum metric Zoltan’s partitioning with the hypergraph model, is able to do better than Zoltan’s graph model in all the instances, and is better than ParMetis for 33% of the problems, and is within 6% or better of CV-sum of the ParMetis for another 44% of the problems (see Figure 2(c)). Again the street networks and the clustering instances are the ones that cause problems for the hypergraph partitioning. In terms of the CV-max metric Zoltan’s hypergraph partitioning is better than the other two methods for 27% of the problems, and within 15% of the CV-max for another 42% of the problems (see Figure 2(b)).

From our results, we can see that even for symmetric problems hypergraph partitioners can perform nearly as well as (or even better than) the graph partitioners depending on the problems and the metrics one cares about. We also note that three of these 22 instances (af\_shell10, audikw1 and G3\_circuit) come from the

same problems we used in Section 2.3 and Zoltan does better in one problem and ParMetis does better on other two problems in terms of the CV-max metric. In terms of EC metric ParMetis does better for all these four problems. However, as we can see from Table 1 the actual solution time is slightly better when we use the hypergraph partitioning for the three problems irrespective of which method is better in terms of the metrics we compute. To be precise, we should again note that the differences in actual solve time between graph and hypergraph partitioning are minor for those three problems. We would like to emphasize that we are not able to observe any difference in the performance of the actual application when the difference in the metrics is a small percentage. We study the characteristics of Zoltan's graph and hypergraph partitioning in the rest of this paper.

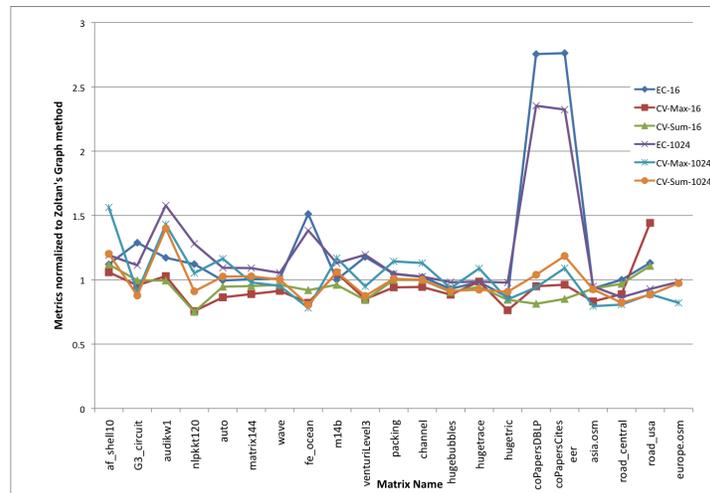


FIGURE 3. Comparing Zoltan's quality metrics with graph and hypergraph models for 16 and 1024 parts.

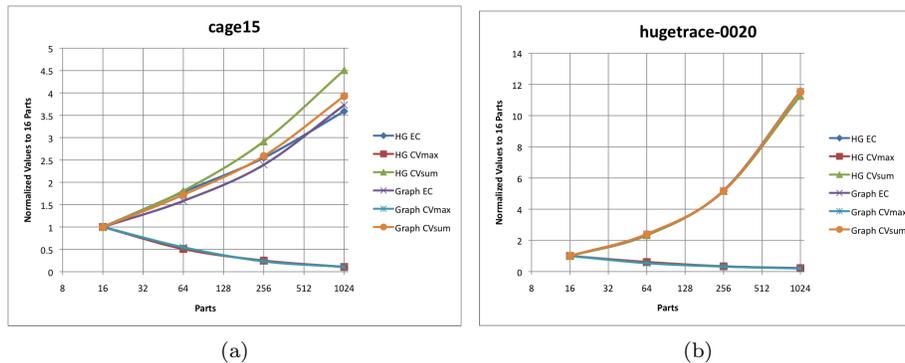


FIGURE 4. Comparing Zoltan's partitioning with graph and hypergraph (HG) model quality metrics for different part sizes.

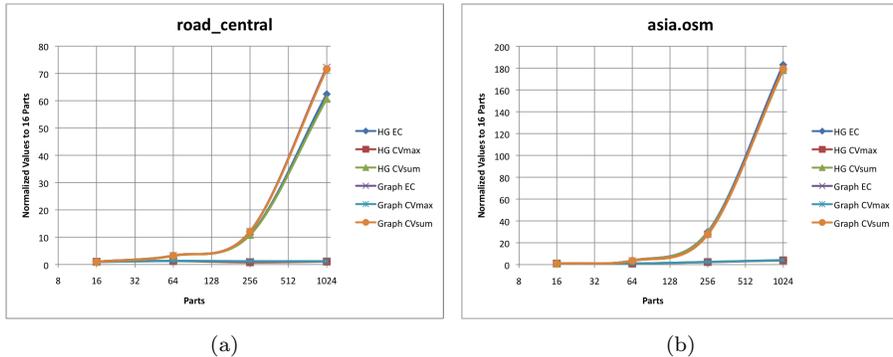


FIGURE 5. Comparing Zoltan’s partitioning with graph and hypergraph (HG) model quality metrics for different part sizes.

**4.3. Zoltan Graph vs. Hypergraph model.** We did more extensive experiments on the symmetric problems with the graph and hypergraph partitioning of Zoltan. For each of the test problems from we compute the three metrics (EC, CV-max, CV-sum) for part sizes 16, 1024. All the experiments use the same number of MPI processes as the part sizes. Figure 3 shows the three metrics for hypergraph partitioning normalized to graph partitioner results for both 16 and 1024 parts. The results show that based on the EC metric, Zoltan’s graph partitioning is the best for most problems. In terms of the CV-sum metric the hypergraph partitioning fares better. Neither of the algorithms optimize, CV-max metric and as expected the results are mixed for this metric. The results for 64 and 256 parts were not different from the results presented in Figure 3 and are not presented here.

Figure 4 shows the change in the partitioning quality with respect to the three metrics for both graph and hypergraph partitionings for two problems – cage15 and hugetrace-0020. The metrics are normalized with respect to the values for the 16 parts case in these figures. These results are for the “good” problems and from the results we can see why we call these problems the “good” problems – EC and CV-sum go up by a factor of 3.5 to 4.5 when going from 16 parts to 1024 parts. In contrast, we also show the change in the metrics from one problem from the street networks and clustering set each (road\_central and asia.osm) in Figure 5. Note that for the some of these problems the metrics scale with similar values that the lines overlap in the graph. These second set of problems are challenging for both our graph and hypergraph partitioners as EC and CV-max go up by a factor 60-70 going from 16-1024 processes (for road.central). The changes in these values are mainly because of the structure of the graphs.

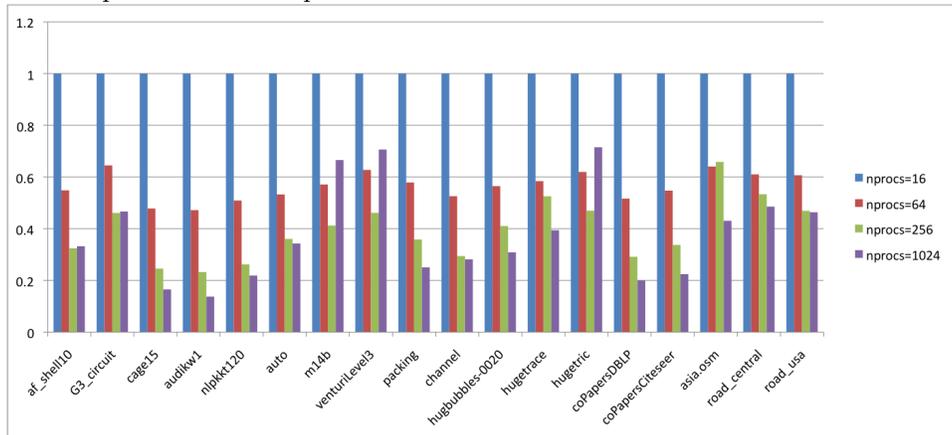
**4.4. Zoltan scalability.** Many of Zoltan’s users use Zoltan within their parallel applications dynamically, where the number of parts equals the number of MPI processes. As a result it is important for Zoltan to have a scalable parallel hypergraph partitioner. We have made several improvements within Zoltan over the past few years and we evaluate our parallel scalability for the DIMACS problems instances in this section. Note that having a parallel hypergraph partitioner also enables us to solve large problems that does not fit into the memory of a compute node. However, we were able to partition all the DIMACS instances except the

matrix europe.osm with 16 cores. We omit the europe.osm matrix and three small matrices from the Walshaw group that get partitioned within two seconds even with 16 cores, from these tests. The scalability results for the rest of the 18 matrices are shown in Figure 6. We normalize the time for all the runs with time to compute 16 parts. Note that even though the matrix size remains the same, this is not a traditional strong scaling test as the number of parts increases linearly with the number of MPI processes. Since the work for the partitioner grows, it is unclear what “perfect scaling” would be, but we believe this is a reasonable experiment as it reflects a typical use case.

Even with the increase in the amount of work for large matrices like cage15 and hugebubbles-0020 we see performance improvements as we go to 1024 MPI processes. However, for smaller problems like the auto or m14b the performance remains flat (or degrades) as we go from 256 MPI processes to 1024 MPI processes.

The scalability of Zoltan’s graph partitioners is shown in Figure 7. We see that the graph partitioner tends to scale well for most problems. Surprisingly, the PHG hypergraph partitioner is faster than our graph partitioner in terms of actual execution time for several of the problems. This may in part be due to the fact that there are only  $n$  hyperedges in the hypergraph model compared to  $m$  edges in the graph model. Recall that PHG treats graphs as hypergraphs, without exploiting the special structure.

FIGURE 6. Scalability of Zoltan Hypergraph Partitioning time for DIMACS challenge matrices normalized to the time for 16 MPI processes and 16 parts.



**4.5. Partitioning on a single node to improve quality.** As discussed before, Zoltan can compute a partitioning statically with different number of MPI processes than number of parts. One strategy to obtain better partitioning quality is therefore to partition for  $p$  parts on  $k$  cores, where  $k < p$ . This often results in better quality than the dynamic approach where  $k = p$ . However, the users have to retain the partition in this case for future use. We evaluate this case for the symmetric matrices from the DIMACS collection for just the hypergraph partitioning. We compute 1024 parts with 24 MPI processes. The assumption is that the user will be willing to devote one compute node to compute the partition

FIGURE 7. Scalability of Zoltan Graph Partitioning time for DIMACS challenge matrices normalized to the time for 16 MPI processes and 16 parts.

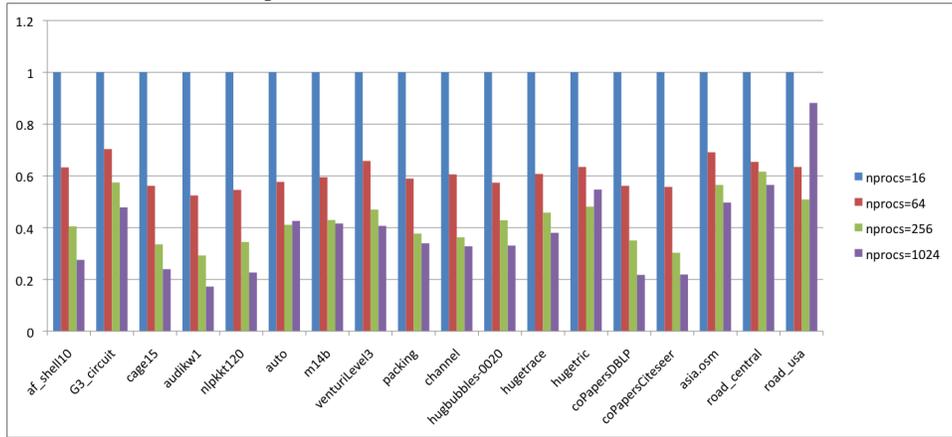
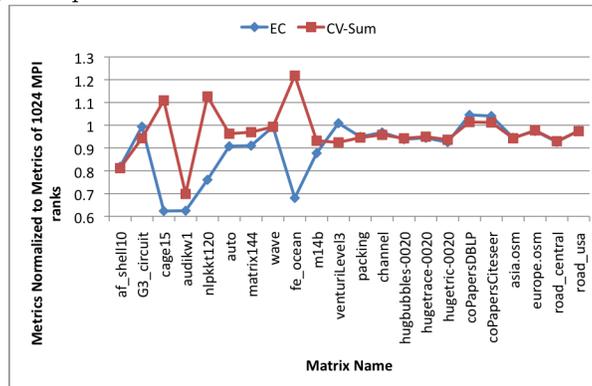


FIGURE 8. Improvement in the partitioning quality when computing 1024 parts with 24 MPI ranks instead of 1024 MPI ranks.



he needs. The results of these experiments for the 22 DIMACS graphs in Figure 8. On an average the edge cuts gets reduced by 10% and the CV-sum gets reduced by 4% when partitioners 1024 parts with just 24 MPI processes instead of using the 1024 MPI processes. This confirms our conjecture that using fewer cores (MPI processes) gives higher quality results, and raises the possibility of using shared-memory techniques to improve the quality in the future.

**4.6. Nonsymmetric Data (Directed Graphs).** The 10th DIMACS implementation challenge includes only undirected graphs, corresponding to structurally symmetric matrices. This clearly favors graph partitioners. Many real-world problems are nonsymmetric, e.g., web link graphs, term-by-document matrices, and

circuit simulation. For such applications, it is well known in the partitioning community that it is better to work directly on the original (nonsymmetric) problem [4, 12]. Remarkably, applications people who are not experts still overwhelmingly use a graph partitioner with symmetrization ( $A + A^T$  or  $A^T A$ ) and apply the result to the original unsymmetric problem. The difference in terms of the communication volume is presumed to be small. We compare hypergraph partitioning on  $A$  against graph partitioning on the symmetrized graph/matrix. We measure the communication volume on the original nonsymmetric problem, since this typically corresponds to the communication cost for the user and show order of magnitudes difference. For these experiments we partitioned the matrix rows, but results for column partitioning were similar.

These experiments were run on a 12-core workstation. We ran Zoltan on 12 MPI processes and partitioned into 12 parts. The test matrices were taken from the UF collection [7], and vary in their degree of symmetry from 0 to 95%. We see from Table 2 that hypergraph partitioning directly on  $A$  gives communication volume at least one order of magnitude smaller than graph partitioning on the symmetrized version in half the test cases. This is substantially different from the 30 – 38% average reduction observed in [4]. We arranged the matrices in decreasing degree of symmetry. Observe that hypergraph partitioning performs relatively better on the highly nonsymmetric matrices. Also note that there is essentially no difference in quality between Zoltan PHG as a graph partitioner and ParMetis for these cases. We conjecture the difference is negligible because the error made in the model by symmetrizing the matrix is far greater than differences in the implementation.

Note that some of the problems in the 22 symmetric test problems were originally unsymmetric problems (like citeseer and DBLP data) but were symmetrized for graph partitioning. We do not have the unsymmetric versions of these problems so we could not use those here.

TABLE 2. Comparison of communication volume (CV-sum) for nonsymmetric and the corresponding symmetrized matrices. PHG was used as hypergraph partitioner on  $A$  and as a graph partitioner on  $A_{sym} \equiv (A + A^T)$

Matrix	dim. ( $\times 10^3$ )	avg. deg.	symmetry	PHG on $A$	PHG on $A_{sym}$	ParMetis on $A_{sym}$
torso3	259	17.1	95%	27,083	48,034	51,193
stomach	213	14.2	85%	15,128	20,742	21,619
rajat21	411	4.6	76%	112,273	174,717	158,296
amazon0312	400	8.0	53%	81,957	846,011	851,793
web-stanford	281	8.2	28%	2,307	543,446	543,547
twotone	120	9.9	24%	6,364	19,771	20,145
wiki-Talk	2,394	2.1	14%	0	53,009	–
hamrle3	1,447	3.8	0%	18,748	1,446,541	1,447,388

## 5. Conclusions

We have evaluated the parallel performance of Zoltan PHG, both as a graph and hypergraph partitioner on test graphs from the DIMACS challenge data set. We also made comparisons to ParMetis, a popular graph partitioner. We observed that

ParMetis consistently obtained best edge cut (EC), as we expected. Surprisingly, ParMetis also obtained lower communication volume (CV) in lot of the symmetric problems. This raises the question: Is hypergraph partitioning worth it? A key advantage of hypergraph partitioning is that it accurately minimizes communication volume [4, 12]. It appears that the superiority of the hypergraph model is not reflected in current software. We believe that one reason Zoltan PHG does relatively poorly on undirected graphs, is that symmetry is not preserved during coarsening, unlike graph partitioners. Future research should consider hypergraph partitioners with symmetric coarsening, to try combine the best of both methods.

We further showed that hypergraph partitioners are superior to graph partitioners on nonsymmetric data. The reduction in communication volume can be one or two orders of magnitude. This is a much larger difference than previously observed. This may in part be due to the selection of data sets, which included some new areas such as weblink matrices. A common approach today is to symmetrize a nonsymmetric matrix and partition  $A + A^T$ . We demonstrated this is often a poor approach, and with the availability of the PHG parallel hypergraph partitioner in Zoltan, we believe many applications could benefit from using hypergraph partitioners without any symmetrization.

Our results confirm that it is important to use a hypergraph partitioner on directed graphs (nonsymmetric matrices). However, for naturally undirected graphs (symmetric matrices) graph partitioners perform better. If a single partitioner for all cases is desired, then Zoltan-PHG is a reasonable universal partitioner.

### Acknowledgements

We thank Karen Devine for helpful discussions.

### References

- [1] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist. Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 20(3):241–255, 2012.
- [2] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring. *Scientific Programming*, 20(2), 2012.
- [3] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.
- [4] 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.
- [5] Ü. Çatalyürek and C. Aykanat. A fine-grain hypergraph model for 2d decomposition of sparse matrices. In *Proc. IPDPS 8th Int'l Workshop on Solving Irregularly Structured Problems in Parallel (Irregular 2001)*, April 2001.
- [6] Ü. Çatalyürek and C. Aykanat. A hypergraph-partitioning approach for coarse-grain decomposition. In *Proc. Supercomputing 2001*. ACM, 2001.
- [7] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.
- [8] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002.
- [9] K.D. Devine, E.G. Boman, R.T. Heaphy, R.H. Bisseling, and U.V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

- [10] 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.
- [11] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95*. ACM, December 1995.
- [12] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26:1519 – 1534, 2000.
- [13] Bruce Hendrickson and Robert Leland. The Chaco user’s guide, version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, 1993.
- [14] G. Karypis and V. Kumar. METIS: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Dept. Computer Science, University of Minnesota, 1995. <http://www.cs.umn.edu/~karypis/metis>.
- [15] G. Karypis and V. Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. Technical Report 97-060, Dept. Computer Science, University of Minnesota, 1997. <http://www.cs.umn.edu/~metis>.
- [16] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: Applications in VLSI domain. In *Proc. 34th Design Automation Conf.*, pages 526 – 529. ACM, 1997.
- [17] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20(1):359–392, 1998.
- [18] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, 1970.
- [19] F. Pelligrini. SCOTCH 3.4 user’s guide. Research Rep. RR-1264-01, LaBRI, Nov. 2001.
- [20] F. Pelligrini. PT-SCOTCH 5.1 user’s guide. Research rep., LaBRI, 2008.
- [21] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.
- [22] Andy Yoo, Allison H. Baker, Roger Pearce, and Van Emden Henson. A scalable eigensolver for large scale-free graphs using 2d graph partitioning. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, pages 63:1–63:11, New York, NY, USA, 2011. ACM.

SANDIA NATIONAL LABORATORIES, ALBUQUERQUE, NEW MEXICO, USA.  
*E-mail address:* [srajama@sandia.gov](mailto:srajama@sandia.gov)

SANDIA NATIONAL LABORATORIES, ALBUQUERQUE, NEW MEXICO, USA.  
*E-mail address:* [egboman@sandia.gov](mailto:egboman@sandia.gov)