Hypergraph Sparsification and Its Application to Partitioning

Mehmet Deveci[1,2], Kamer Kaya[1], Ümit V. Çatalyürek[1,3]
The Ohio State University
[1]*Dept. of Biomedical Informatics*
[2]*Dept. of Computer Science and Engineering*
[3]*Dept. of Electrical and Computer Engineering*
{*mdeveci,kamer,umit*}*@bmi.osu.edu*

***Abstract*—The data one needs to cope to solve today's problems is large scale, so are the graphs and hypergraphs used to model it. Today, we have BigData, big graphs, big matrices, and in the future, they are expected to be bigger and more complex. Many of today's algorithms will be, and some already are, expensive to run on large datasets. In this work, we analyze a set of efficient techniques to make "big data", which is modeled as a hypergraph, smaller so that its processing takes much less time. As an application use case, we take the hypergraph partitioning problem, which has been successfully used in many practical applications for various purposes including parallelization of complex and irregular applications, sparse matrix ordering, clustering, community detection, query optimization, and improving cache locality in shared-memory systems. We conduct several experiments to show that our techniques greatly reduce the cost of the partitioning process and preserve the partitioning quality. Although we only measured their performance from the partitioning point of view, we believe the proposed techniques will be beneficial also for other applications using hypergraphs.**

***Keywords*-Hypergraph sparsification; hypergraph partitioning; multi-level approach; identical nets; identical vertices; Jaccard similarity.**

I. Introduction

Using and analyzing large data for practical purposes has always been a fundamental problem. But today, the data one needs to cope with for a major scientific innovation or discovery is immense, distributed, and unstructured. Although recent advancements on computer hardware and storage technologies allow us to gather and store large-scale data, the data itself does not serve science and society. In addition to its size, utilizing this data and finding meaningful patterns inside it require complex algorithms and an immense amount of computing power. Hence, techniques to make the data smaller are always appreciated.

Hypergraphs emerged as a good alternative model for unstructured data for many applications such as parallelization of complex and irregular applications from various domains including parallel scientific computing [1], [2], sparse matrix reordering [3], [4], static and dynamic load balancing [5], social network analysis [6], clustering and recommendation [7], and database design [8], [9], [10]. In many of these applications, once the problem is modeled as a hypergraph, the optimization problem in hand reduces to, sometimes with little variations, the hypergraph partitioning problem. Due to its wide area of applications, a considerable effort has been put into providing tool support for hypergraph partitioning (see hMeTiS [11], MLpart [12], Mondriaan [13], Parkway [14], PaToH [15], UMPa [16] and Zoltan [17]).

There are two main criteria to evaluate the performance of a tool: the quality of the partition with respect to a partitioning metric and the time required to partition the given hypergraph. It has been repeatedly shown that minimizing the partitioning metrics in the literature can significantly increase the performance for many computations. However, the relative importance of these criteria change with respect to the application. If the partitioning time is bigger than the gain in the computation time, it may be better to trade quality for partitioning cost. But one needs to be careful with this trade-off since worsening the quality can also worsen the performance. For example, in a recent study, Akbudak et al. studied different partitioning models to optimize the cache locality for the sparse-matrix vector multiplication kernel (SpMV) [18]. They reported that when the 1D hypergraph model in [19] is used, the partitioning overhead is amortized in 286 SpMV operations. For their 2D model, this number is 1110 but the SpMV performance is 13% better than the 1D case at the same time. Hence, the best model, or even the decision of using a partitioning depends on the subsequent computation. Clearly, if the partitioning process is made faster it will be much more useful for many applications.

In the past, powerful parallel machines were only accessible by a small set of researchers. Today, a High Performance Computing (HPC) system with thousands of processors/cores is now almost an ordinary commodity. However, when the number of processors increases to tens of thousands, which is the case today, the data and task distribution cost via partitioning also increases. Yet the communication between processors and hence the quality of the partition become more important with the increasing size of the systems. Hence, reducing the partitioning cost, especially for a larger system, is a crucial task.

In this work, we investigate *hypergraph sparsification* to reduce the partitioning cost. At the high level, we model the sparsification problem as finding the sets of identical (or very similar) sets/nets/vertices in a hypergraph. Let $\mathcal{S}$ be such a set containing distinct elements with identical connectivity information. Hence, the same information is duplicated $|\mathcal{S}|$ times in the hypergraph. Duplicate information causes two main problems for a task: First, the same information is processed several times, and this redundancy usually worsens the efficiency of the task. Second, if the purpose of this task is optimization, the redundant (or almost redundant) information makes the search space larger and finding good (or optimal) solutions harder. The action we take to remove such information is using a single *representative* instead of all the elements in $\mathcal{S}$. The representative can be either an existing element of $\mathcal{S}$ or can be artificially created according to a criterion by using the properties of $\mathcal{S}$'s elements.

In particular, here we investigate cheap hypergraph sparsification heuristics: identical-net, identical-vertex, and similar-net removal. Although identical-net removal has been used in partitioning before [15], [20], [21], to the best of our knowledge, there is no work which analyzes its effectiveness in detail. Even though, some implementations exist in widely-used partitioning tools such has PaToH and Zoltan, it is disabled in PaToH, since on average, the existing code does not amortize itself in PaToH's recursive bisection framework, and we show that Zoltan's sort based approach can be improved. Hence, the algorithms designed and analyzed in this work can be used in other tools, including PaToH and Zoltan, for faster partitioning. Furthermore, there is no previous work which analyzes the effectiveness of the similarity-based sparsification techniques we propose in this paper for hypergraph based data.

We carefully design a set of experiments to analyze the effectiveness of the proposed sparsification techniques, their integration to the multi-level approach, and their efficient implementation. For the experiments, we use a multi-level K-way partitioning tool UMPa (pronounced as "Oompa") [16] and minimize the total communication volume which is the classical partitioning objective used in practice. Our experiments show that when the hypergraphs in the multi-level framework are sparsified, one can obtain partitions with similar quality in significantly less time.

The rest of the paper is organized as follows: Section II gives the notation and background for hypergraph partitioning, and Section III describes the proposed sparsification heuristics. The experimental results are presented in Section IV. Section V concludes the paper.

II. Background

A. Hypergraph partitioning

A hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$ is defined as a set of vertices $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. A net $n \in \mathcal{N}$ is a subset of vertices and the vertices in n are called its *pins*. The number of pins of a net is called the *size* of it, and the *degree* of a vertex is equal to the number of nets it is connected to. In this paper, we will use $\mathsf{pins}[n]$ and $\mathsf{nets}[v]$ to represent the pin set of a net n and the set of nets vertex v is connected to, respectively. Vertices can be associated with weights, denoted with $\mathsf{w}[\cdot]$, and nets can be associated with costs, denoted with $\mathsf{c}[\cdot]$. The total number of pins of the hypergraph is denoted by $\rho = \sum_{n\in\mathcal{N}} |\mathsf{pins[n]}|$.

A K*-way partition* of a hypergraph $\mathcal{H}$ is denoted as $\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ where parts are pairwise disjoint (i.e., $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \le k < \ell \le K$), each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$ (i.e., $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \le k \le K$), union of K parts is equal to $\mathcal{V}$ (i.e., $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$).

Let W_k denote the total vertex weight in $\mathcal{V}_k$ (i.e., $W_k = \sum_{v\in\mathcal{V}_k} \mathsf{w}[v]$) and W_{avg} denote the weight of each part when the total vertex weight is equally distributed (i.e., $W_{avg} = (\sum_{v\in\mathcal{V}} \mathsf{w}[v])/K$). If each part $\mathcal{V}_k \in \Pi$ satisfies the *balance criterion*

$$W_k \le W_{avg}(1+\varepsilon), \quad \text{for } k = 1,2,\ldots,K \qquad (1)$$

we say that Π is ϵ*-balanced* where ε represents the maximum allowed imbalance ratio.

For a K-way partition Π, a net that has at least one pin (vertex) in a part is said to *connect* that part. The number of parts connected by a net n, i.e., *connectivity*, is denoted as λ_n. A net n is said to be *uncut* (*internal*) if it connects exactly one part (i.e., $\lambda_n = 1$), and *cut* (*external*), otherwise (i.e., $\lambda_n > 1$).

There are various cutsize definitions [22] for hypergraph partitioning. The one that will be used in this work, which is shown to accurately model the total communication volume [1], is called the *connectivity* metric and defined as:

$$conn_{\mathcal{H}}(\Pi) = \sum_{n\in\mathcal{N}} \mathsf{c}[n](\lambda_n - 1)\ . \qquad (2)$$

In this metric, each cut net n of the hypergraph $\mathcal{H}$ contributes $\mathsf{c}[n](\lambda_n - 1)$ to the cutsize. The hypergraph partitioning problem can be defined as the task of finding a balanced partition Π with K parts such that $conn_{\mathcal{H}}(\Pi)$ is minimized. This problem is also NP-hard [22].

B. K-way partitioning and multi-level framework

Arguably, the multi-level approach [23] is the most successful heuristic for the hypergraph partitioning problem. Although, it has been first proposed for recursive-bisection based graph partitioning, it also works well for hypergraphs [1], [11].

In the multi-level approach, a given hypergraph is coarsened to a much smaller one, a partition is obtained on the smallest hypergraph, and that partition is projected to the original hypergraph. These three phases will be called the coarsening, initial partitioning, and uncoarsening phases, respectively. The coarsening and uncoarsening phases have multiple levels. In a coarsening level, similar vertices are merged to make the hypergraph smaller. In the corresponding uncoarsening level, the merged vertices are split, and the

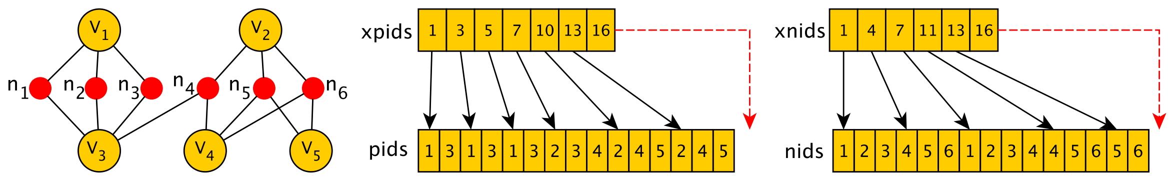


Figure 1. A simple hypergraph with 6 nets and 5 vertices with the data structures used in the implementation.

partition of the coarser hypergraph is refined for the finer one.

Most of the multi-level partitioning tools used in practice are based on recursive bisection (RB). In RB, the multi-level approach is used to partition a given hypergraph into two. Each of these parts is further partitioned into two recursively until K parts are obtained in total. Another approach is using the direct K-way scheme with the multi-level framework, which first coarsens the hypergraph, then directly obtains a K-way partition of the smallest hypergraph in the initial partitioning phase, and refines it while uncoarsening the hypergraph. Hence, to partition a hypergraph, a K-way partitioner has only one coarsening, one initial partitioning, and one uncoarsening phase, where an RB-based partitioner has $K-1$ of them. On the other hand, the initial partitioning and uncoarsening phases are much cheaper for RB since a K-way partitioner needs a K-way partition in the initial partitioning scheme and a K-way refinement at each level of the uncoarsening phase. Several works show that, the direct K-way approach can be successfully used within a multi-level framework [14], [16], [21] and it can produce higher quality results. The only caveat is that the partitioning time of a direct K-way approach increases more rapidly with increasing K, in comparison to the RB-based partitioner. In this work, we use a multi-level K-way partitioning tool UMPa [16] and apply the proposed sparsification heuristics not only to the original hypergraph but also to the coarser hypergraphs obtained in the coarsening phase to further reduce the execution time.

C. Data Structures for Hypergraphs

For efficient access, hypergraphs are usually stored and utilized using two pairs of arrays which represent the hypergraph w.r.t. its nets (i.e., $\mathsf{pins}[]$) and vertices (i.e., $\mathsf{nets}[]$), respectively.

1) $xpids$-$pids$ pair gives the net view. The sizes of $xpids$ and $pids$ are $|\mathcal{N}|+1$ and ρ, respectively. The array $pids$ stores the pin ids, and $xpids[i]$ points to the start location in $pids$ for each net i. The last value of $xpids$ is equal to ρ. That is $pids[xpids[i],\ldots,xpids[i+1]-1]$ contains the pin ids for net i.
2) $xnids$-$nids$ pair gives the vertex view. The sizes of $xnids$ and $nids$ are $|\mathcal{V}|+1$ and ρ, respectively. The array $nids$ stores the net ids, and $xnids[i]$ points to the start location in $nids$ for each vertex i. The last value of $xnids$ is equal to ρ. That is $nids[xnids[i],\ldots,xnids[i+1]-1]$ contains the net ids for vertex i.

In multi-level partitioning context, these arrays are generated not only for the original hypergraph, but also for each coarse hypergraph too. Hence, in any part of the partitioner, accesses to $\mathsf{pins}[n]$ and $\mathsf{nets}[v]$ of a net n and a vertex v can be handled very efficiently. A toy hypergraph and the corresponding data structures are given in Figure 1.

III. Sparsification Techniques for Hypergraphs

In this section, we materialize this sparsification approach for the problem of scalable hypergraph partitioning. However, the heuristics and algorithms proposed here can be used for other problems and applications.

A. Identical-Net Removal

Although we start with the description of the identical-net removal heuristic, the techniques described in this subsection are also valid for the other heuristics in this work. The identicality of the nets depends on the set of vertices they are connected to.

Definition 1: Two nets n_i and n_j are *identical* if $\mathsf{pins}[n_i]$ and $\mathsf{pins}[n_j]$ are the same.

Given a hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$, let $\{\mathcal{N}_1,\mathcal{N}_2,\ldots,\mathcal{N}_\kappa\}$ be the partition of the net set $\mathcal{N}$ such that two nets in $\mathcal{N}$ are in the same subset $\mathcal{N}_\ell$, $1\leq\ell\leq\kappa$, if and only if they are identical. The identical-net removal (INR) process generates a smaller hypergraph $\mathcal{H}'=(\mathcal{V},\mathcal{N}')$ where $\mathcal{N}'=\{n'_1,n'_2,\ldots,n'_\kappa\}$ and each net n'_ℓ corresponds to $\mathcal{N}_\ell\subseteq\mathcal{N}$ for $1\leq\ell\leq\kappa$. The net n'_ℓ is also called *representative* net of the nets in $\mathcal{N}_\ell$, and its pin set is defined as

$$\mathsf{pins}'[n'_\ell]\leftarrow\mathsf{pins}[n]\text{ s.t. }n\in\mathcal{N}_\ell, \tag{3}$$

and the net set of each vertex $v\in\mathcal{H}'$ contains the representatives having v in their pin set.

In hypergraph partitioning context, if two nets n_i and n_j are identical they are either both internal or both external with respect to a partition Π. Furthermore, $\lambda_{n_i}=\lambda_{n_j}$. Since each net's contribution to the connectivity-1 metric given in (2) is proportional to its cost, the cost of a representative $n'_\ell \in \mathcal{N}'$ is computed as

$$\mathsf{c}'[n'_\ell] \leftarrow \sum_{n \in \mathcal{N}_\ell} \mathsf{c}[n]. \tag{4}$$

Let Π be a K-way partition of the vertex set $\mathcal{V}$. The contribution of net $n'_\ell \in \mathcal{N}'$ to the cutsize $conn_{\mathcal{H}'}(\Pi)$ is equal to the sum of contributions of the nets in $\mathcal{N}_\ell$ to $conn_{\mathcal{H}}(\Pi)$ since the connectivity $\lambda_{n'_\ell}$ of the representative is equal to the connectivity of identical nets. That is, the cutsize metrics, $conn_{\mathcal{H}}(\Pi) = conn_{\mathcal{H}'}(\Pi)$, are equal. Hence, an optimal partition Π for the sparsified hypergraph $\mathcal{H}'$ is also optimal for the original $\mathcal{H}$.

Since INR is not expected to change the quality of the final partition reducing the overhead of the net-removal process is of utmost importance. The naive algorithm for detecting and removing identical nets requires $\mathcal{O}(V^2)$ pair-wise comparisons of the pin sets. The number of these comparisons can be greatly reduced by using a simple checksum function,

$$\text{Cs1}(n) = \sum_{i \in \mathsf{pins}[n]} i, \tag{5}$$

as shown by Algorithm 1, INRSRT. According to Definition 1, if two nets are identical their checksums must be equal. Hence, the inequality of the checksums can be used as a *witness* for two nets being non-identical. On the other hand, such a witness is not sufficient to identify *false-positive* net pairs who have the same checksum value but are not identical.

INRSRT first computes the checksum values of all $n \in \mathcal{N}$. Second, it sorts the nets with respect to these values. And third, it examines only the nets with the same csum value at a time to see if they are identical. In INRSRT, during the examination for a net $n \in \mathcal{N}$, rep[n] is set to an existing representative id (line 4) which is added to $\mathcal{N}'$ after a previous examination. If such a representative does not exist, at line 5, rep[n] will be equal to a new id, and a new representative will be added to $\mathcal{H}'$. The pin set and cost of the representative are assigned according to (3) and (4), respectively. This algorithm has been used in the literature with similar motivations such as detecting identical vertices in graphs [24] and identifying supervariables in a nested-dissection based matrix reordering scheme [25]. It has also been used to remove identical nets [20], [21].

We say that a *collision* exists when $\mathsf{csum}[n_i] = \mathsf{csum}[n_j]$ for two non-identical nets n_i and n_j. If there are no collisions, i.e., no false positives, line 3 of INRSRT will be executed at most once for each net $n \in \mathcal{N}$. Since each identicality check for a net n costs $\mathcal{O}(|\mathsf{pins}[n]|)$, the total cost due to 3 is $\mathcal{O}(\rho)$. Hence, with a good checksum function, which only creates a negligible amount of conflicts, the total complexity is $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}| + \rho)$. We evaluate the effectiveness of a checksum function w.r.t. two criteria:

Algorithm 1: INRSRT

```
  Data: $\mathcal{H} = (\mathcal{V}, \mathcal{N}), \mathsf{c}, \mathsf{pins}$
  Output: $\mathcal{H}' = (\mathcal{V}, \mathcal{N}'), \mathsf{c}', \mathsf{pins}'$
  $\mathcal{N}' \leftarrow \emptyset$
  for each $n \in \mathcal{N}$ do
1     csum[n] $\leftarrow$ Cs1$(n)$
  $\sigma \leftarrow$ the permutation of the nets in the increasing order of csum values
  $prev \leftarrow -1$
  $r \leftarrow 1$
  for $i = 1$ to $|\mathcal{N}|$ do
      $n \leftarrow \sigma[i]$
      if $prev \neq$ csum[n] then
          $\mathcal{R} \leftarrow \{n'_r\}$
          $r \leftarrow r + 1$
          $prev \leftarrow$ csum[n]
      else
          rep[n] $\leftarrow r$
2         for each $n'_\ell \in \mathcal{R}$ do
3             if pins$[n'_\ell]$ = pins[n] then
4                 rep[n] $\leftarrow \ell$
                  c'$[n'_r] \leftarrow$ c'$[n'_r]$ + c$[n]$
                  break
          $c \leftarrow$ rep[n]
5         if $c = r$ then
              $\mathcal{R} \leftarrow \mathcal{R} \cup \{n'_r\}$
              $\mathcal{N}' \leftarrow \mathcal{N}' \cup \{n'_r\}$
              c'$[n'_r] \leftarrow$ c$[n]$
              pins'$[n'_r] \leftarrow$ pins$[n]$
              $r \leftarrow r + 1$
  $\mathcal{H}' \leftarrow (V, \mathcal{N}')$
```

- *False-positive cost*: The number of pairwise comparisons (line 3) during the course of INRSRT for two non-identical nets.
- *Checksum occupancy*: The average number of distinct representatives having the same checksum value. When no false-positives exist, the occupancy is one. Otherwise, it is greater.

When all κ representatives have the same checksum value, the occupancy is κ, and the false-positive cost is $\mathcal{O}(\kappa|\mathcal{N}|)$. Hence, the total worst case complexity is $\mathcal{O}(|\mathcal{V}| \log |\mathcal{V}| + \kappa\rho)$. Furthermore, if κ is also $\mathcal{O}(|\mathcal{N}|)$, i.e., all nets are distinct and have the same checksum value, the total cost of line 3 will be huge. Fortunately, hypergraphs from real life are random enough to avoid such pathological cases even with a simple checksum function such as Cs1. Still, it is better in practice to reduce the number of collisions to make the identical-net removal faster. Yet again, using a simple function is also a good practice since the cost of the checksum computation will be less which may be important for some applications.

As described above, INRSRT sorts the nets with respect to their checksums. This operation can be considered as using the witnesses to eliminate a huge amount of pairwise

Algorithm 2: INRMEM

```
Data: $\mathcal{H} = (\mathcal{V}, \mathcal{N}), \mathsf{c}, \mathsf{pins}$
Output: $\mathcal{H}' = (\mathcal{V}, \mathcal{N}'), \mathsf{c}', \mathsf{pins}'$
$\mathcal{N}' \leftarrow \emptyset$
for $i$ from 1 to $q$ do
    $first[i] = -1$
for $i$ from 1 to $|\mathcal{N}|$ do
    $next[i] = -1$
1   $\mathsf{csum}[i] \leftarrow \text{Cs1}(n_i) \mod q$
$r \leftarrow 1$
for $i$ from 1 to $|\mathcal{N}|$ do
    $c \leftarrow first[\mathsf{csum}]$
    if $c = -1$ then
        $first[\mathsf{csum}[i]] = i$
        $c \leftarrow i$
    while $c \neq i$ do
        $\ell \leftarrow \mathsf{rep}[n_c]$
2       if $\mathsf{pins}'[n'_\ell] = \mathsf{pins}[n_i]$ then
            $\mathsf{c}'[n'_\ell] \leftarrow \mathsf{c}'[n'_\ell] + \mathsf{c}[n_i]$
            break
        else
            if $next[c] = -1$ then
3               $next[c] = i$
4       $c \leftarrow next[c]$
    if $c = i$ then
5       $\mathcal{N}' \leftarrow \mathcal{N}' \cup \{n'_r\}$
        $\mathsf{c}'[n'_r] \leftarrow \mathsf{c}[n_i]$
6       $\mathsf{pins}'[n'_r] \leftarrow \mathsf{pins}[n_i]$
        $\mathsf{rep}[n_i] \leftarrow r$
        $r \leftarrow r + 1$
$\mathcal{H}' \leftarrow (V, \mathcal{N}')$
```

comparisons. However, when a good checksum function with a small occupancy and false-positive cost is used, the sorting operation can be the dominant factor considering the complexity of the rest of the algorithm. In this work, we propose to use another, *hash-based*, approach given in Algorithm 2, INRMEM that trades memory for performance. The data structure we use contains two arrays, $first$ and $next$, of sizes q and $|\mathcal{N}|$, respectively, to store the representative information. Here, q is the first prime number greater than $|\mathcal{N}|$. INRMEM first initializes the entries in $first$ to -1 and starts to traverse the nets from n_1 to $n_{|\mathcal{N}|}$. For each net n_i, by using the $first$ array, it checks if there exist a net n_c s.t. $c < i$ and $\text{Cs1}(n_c) \equiv \text{Cs1}(n_i) \pmod q$. If such a net exists the corresponding entry in $first$ will be c, and -1, otherwise. In the former case, starting from n_c, the algorithm initiates a set of consecutive checks (line 2) to eliminate false positives for n_i by following the links in the $next$ array (line 4). Similar to INRSRT, if n_i is identical to an existing representative n'_ℓ its contribution is added to $\mathsf{c}'[n'_\ell]$. If no such representative exists, a new one is created (line 5) and $next$ is updated accordingly (line 3). With a good checksum function, the total complexity of INRMEM is $\mathcal{O}(\rho)$, since the check to distinguish false positives is executed only once for most of the nets.

Partition	Checksum
$\mathcal{N}_1 = \{n_1, n_2, n_3\}$	$\text{Cs1}(n'_1) \mod q = 4$
$\mathcal{N}_2 = \{n_4\}$	$\text{Cs1}(n'_2) \mod q = 2$
$\mathcal{N}_3 = \{n_5, n_6\}$	$\text{Cs1}(n'_3) \mod q = 4$

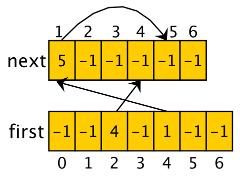


Figure 2. Identical-net removal via INRMEM on a toy hypergraph in Figure 1 with 6 nets (i.e., $q = 7$). On the left, the 3 subsets of identical nets and the corresponding representatives' Cs1 mod q values are given. On the right, the $first$-$next$ data structure at the end of the process is shown.

Since the net information is stored in sets, the order of the set elements should not affect the csum value, i.e., the checksum function should be *order independent*. Having collision reduction in mind, in this work, we investigate two simple order-independent checksum functions:

$$\text{Cs2}(n) = \sum_{i \in \mathsf{pins}[n]} i^2, \tag{6}$$

$$\text{Cs3}(n) = \sum_{i \in \mathsf{pins}[n]} i^3. \tag{7}$$

These functions can be integrated to the process by modifying only lines 1 of INRSRT and INRMEM, respectively. Another checksum function we used is *MurmurHash*, a fast but order dependent hash function with good collision properties [26]. Since the function is order dependent, we need to sort the pin set of each net in the hypergraph as a prerequisite for INRSRT and INRMEM.

B. Identical-Vertex Removal

The connectivity information of hypergraphs can be viewed from two different perspectives: nets and vertices. Here we investigate a hypergraph sparsification technique from the vertices' point of view. The identicality definition of the vertices is similar to that of the nets:

Definition 2: Two vertices v_i and v_j are *identical* if $\mathsf{nets}[v_i]$ and $\mathsf{nets}[v_j]$ are the same.

Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, let $\{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_\kappa\}$ be the partition of the vertex set $\mathcal{V}$ such that two vertices in $\mathcal{V}$ are in the same subset $\mathcal{V}_\ell$, $1 \leq \ell \leq \kappa$, if and only if they are identical. The identical-vertex removal process generates a smaller hypergraph $\mathcal{H}' = (\mathcal{V}', \mathcal{N})$ where $\mathcal{V}' = \{v'_1, v'_2, \ldots, v'_\kappa\}$ and each vertex $v'_\ell \in \mathcal{V}'$ corresponds to a $\mathcal{V}_\ell$ for $1 \leq \ell \leq \kappa$. The vertex v'_ℓ is also called a *representative* vertex for vertices in $\mathcal{V}_\ell$. The net set and weight of a representative $v'_\ell \in \mathcal{V}'$ are defined as

$$\mathsf{nets}'[v'_\ell] \leftarrow \mathsf{nets}[v] \text{ s.t. } v \in \mathcal{V}_\ell,$$

$$\mathsf{w}'[v'_\ell] \leftarrow \sum_{v \in \mathcal{V}_\ell} \mathsf{w}[v].$$

After vertex removal, the pin set of each net $n \in \mathcal{H}'$ contains the representative vertices having n in their net set.

Unlike the case for identical-nets, the optimality of the partitioning is not preserved for identical-vertex removal: Let Π and Π' be two optimal partitions for $\mathcal{H}$ and $\mathcal{H}'$, respectively. Then we have $conn_{\mathcal{H}}(\Pi) \leq conn_{\mathcal{H}'}(\Pi')$. The equality does not always hold due to the load balancing constraint since the original hypergraph $\mathcal{H}$ is more relaxed in terms of vertex moves, and some partitioning configurations for $\mathcal{H}$ may not be feasible for $\mathcal{H}'$ due to its larger vertex weights. However, removing identical vertices also reduces the search space for the refinement heuristics employed in the uncoarsening phase. As the experiments will show, this may lead the partitioner to a slightly better final partition.

We use the same approach in InrMem to detect the identical vertices in $\mathcal{H}$ and to create $\mathcal{H}'$. Hence, the implementations of the identical-net and -vertex removal heuristics are very similar except that the former one uses $xpids$-$pids$ whereas the latter uses $xnids$-$nids$ for efficient checksum computations. Although identical-vertex removal can be used for any hypergraph, for partitioning, we only use it on the original, finer one, since the coarser hypergraphs are generated by a vertex matching process which is expected to match the identical vertices with high probability.

C. Similar-Net Removal

The sparsification heuristics in the previous sections are solely based on the concept of identicality and they only aim to remove the redundancy from a large hypergraph. That is, they are only effective when there exists identical nets and/or vertices in the hypergraph. Here, we will describe the *similar-net removal* (SNR) heuristic, which can be used even when there is no redundancy. The heuristic can be considered as a lossy compression technique since it discards some information while sparsifying the hypergraph. Although discarding information usually worsen the quality of the final analysis/partition, the partitioning process will be faster since the hypergraph will be smaller. Such a quality/time tradeoff will be very useful in practice, especially when the performance of the application is not very sensitive against small changes in partitioning quality. The effectiveness of such a tradeoff depends on the discovery of a large amount of information which can be more or less compensated by a relatively smaller representative. Considering each net is a set of pins, to attack the information discovery problem, we use the well-known Jaccard similarity metric [27]:

Definition 3: The *similarity* between two nets n_i and n_j is defined as

$$J(n_i, n_j) = \frac{|\mathsf{pins}[n_i] \cap \mathsf{pins}[n_j]|}{|\mathsf{pins}[n_i] \cup \mathsf{pins}[n_j]|}.$$

Since the number of nets is large, it is infeasible to compute the similarity for each net pair. Instead, in this work, we propose computing a footprint of each net by using the minimum hash values, which is an efficient way to approximate $J(n_i, n_j)$ [28]. Let σ be a random permutation of the integers from 1 to $|\mathcal{V}|$, and $min_\sigma(n)$ is the first vertex id of a net $n \in \mathcal{N}$ under the permutation σ. Then,

$$\Pr[min_\sigma(n_i) = min_\sigma(n_j)] = J(n_i, n_j).$$

In other words, if x is a random variable s.t.

$$|x| = \begin{cases} 1, & \text{if } min_\sigma(n_i) = min_\sigma(n_j), \\ 0, & \text{otherwise.} \end{cases}$$

then x is an unbiased estimator of $J(n_i, n_j)$. However, its variance is too high. In practice, one can use multiple independent permutations and get the average of the x values to reduce the variance [28], [29]. However, this alone is not sufficient for efficient hypergraph sparsification since it is still based on pairwise similarity. In this work, to obtain a more efficient solution, we use t permutations σ_1 to σ_t and first generate a *minwise footprint* of each net. The similarity definition is then modified as follows:

Definition 4: Two nets n_i and n_j are similar if their minwise footprints are the same, where the footprint of a net $n \in \mathcal{N}$ is defined as

$$mf(n) = (min_{\sigma_1}(n), \ldots, min_{\sigma_t}(n)).$$

Following the definition, given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$, the similar-net heuristic constructs a partition $\{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_\kappa\}$ of the net set $\mathcal{N}$ where two nets n_i and n_j are in the same subset $\mathcal{N}_\ell$ if and only if their footprints are identical. It then generates a smaller hypergraph $\mathcal{H}' = (\mathcal{V}, \mathcal{N}')$ where $\mathcal{N}' = \{n'_1, n'_2, \ldots, n'_\kappa\}$ and each net n'_ℓ corresponds to $\mathcal{N}_\ell \subseteq \mathcal{N}$ for $1 \leq \ell \leq \kappa$. The net n'_ℓ is also called a *representative* net for nets in $\mathcal{N}_\ell$ and its cost is defined as

$$\mathsf{c}'[n'_\ell] \leftarrow \sum_{n \in \mathcal{N}_\ell} \mathsf{c}[n].$$

Since the connectivity of the nets in a subset $\mathcal{N}_\ell$ can be different, the representation of the connectivity information removed from $\mathcal{H}$ will be lossy, and the amount of discarded information depends on the assignment of $\mathsf{pins}'[n'_\ell]$. We investigated three options:

- *Large* (LRG): $\mathsf{pins}'[n'_\ell]$ is set to the pin set of the net with the largest pin set in $\mathcal{N}_\ell$.
- *Important* (IMP): $\mathsf{pins}'[n'_\ell]$ is set to the pin set of the net n which maximizes

$$\sum_{v \in \mathsf{pins}[n]} \left(\sum_{n_i \in \mathsf{pins}[v] \cap N_\ell} \mathsf{c}[n_i] \right).$$

  This metric prioritizes the pins which are connected to heavy nets with large $\mathsf{c}[.]$ values.
- *Union* (UNI): $\mathsf{pins}'[n'_\ell]$ is set to $\bigcup_{n \in \mathcal{N}_\ell} \mathsf{pins}[n]$.

After computing the footprints, we use the same approach in InrMem to detect similar nets in $\mathcal{H}$ and to create $\mathcal{H}'$.

We changed line 2 since we are now checking footprints instead of pin sets. Among other minor modifications, we also changed line 6 w.r.t. the option we use while generating the pin set of the representatives.

D. Parallelization of Sparsification Techniques

Although they are much cheaper than partitioning, the efficiency of the proposed heuristics may need to be improved by implementing them in parallel for other applications using hypergraphs. If this is the case, the first task, computation of the checksum values is pleasingly parallel since each thread can process a vertex/net without any concurrency problem. As the experiments show, the sorting phase of INRSRT dominates its execution time. Hence, it should be the first target for parallelization. After that, the false positive detection can be handled by multiple threads where each thread analyzes a different set of nets/vertices sharing the same hash value.

The proposed approach given in INRMEM is also amenable for parallelization. Similar to INRSRT, the checksum computations can be naively parallelized. After that one can do only a single pass on the nets to build a variant of $first$/$next$ data structure by simply storing the id of every net in it without doing an equality check. Figure 3 shows this variant after adding all the elements in the toy hypergraph of Figure 3 when CS1 is used. Then starting from the non-

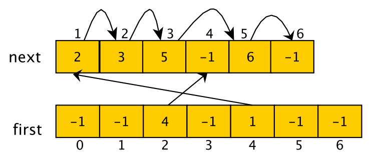


Figure 3. The $first$/$next$ structure when all the elements are added.

negative entries in the $first$ array, each thread can check the false positives throughout the path by following the $next$ pointers till a *null* pointer (-1) is reached. Starting from the one in Figure 3, after the false-positive checks, the same structure in Figure 2 will be obtained.

For some applications with huge and dynamic data, the heuristics may need to be executed several times and their efficiency can be of interest. Hence, we believe that the algorithmic details are important. However, in this work, we do not experiment with parallel implementations of the heuristics, since in the partitioning context, they are already cheap and their effectiveness in practice is much more important.

IV. Experimental Results

For the experiments, we used a machine with a 2.27GHz dual quad-core Intel Xeon (Bloomfield) CPU and 48GB main memory. All the codes is implemented in C++ and compiled with g++ version 4.5.2. To create our test instances, we used a set of 28 matrices from the dataset of $10th$ DIMACS implementation challenge on graph partitioning and graph clustering [30]. We have used the row-net hypergraph representation of the matrices [1]. For all the matrices in our set, the number of vertices (and nets) is between 5×10^5 and 5×10^6, and the number of pins is roughly between 3×10^6 and 10×10^7. The properties of the hypergraphs we use are given in Table I.

Table I
HYPERGRAPHS USED IN THE EXPERIMENTS

Hypergraph	Class	#Vertices/ #Nets	#Identical Vertices	#Pins
coPapersDBLP	Citation	540,486	285,113	30,491,458
eu-2005	Cluster.	862,664	29,979	32,276,936
in-2004	Cluster.	1,382,908	122,984	27,182,946
delaunay_n19	Delan.	524,288	0	3,145,646
delaunay_n20	Delan.	1,048,576	0	6,291,372
delaunay_n21	Delan.	2,097,152	0	12,582,816
hugetrace-00000	Frames	4,588,484	0	13,758,266
packing-500x100x100	Numer.	2,145,852	9	34,976,486
venturiLevel3	Numer.	4,026,819	0	16,108,474
channel-500x100x100	Numer.	4,802,000	0	85,362,744
rgg_n_2_19_s0	Random	524,288	24,433	6,539,532
rgg_n_2_20_s0	Random	1,048,576	46,228	13,783,240
rgg_n_2_21_s0	Random	2,097,152	88,681	28,975,990
rgg_n_2_22_s0	Random	4,194,304	169,311	60,718,396
ca2010	Redistr.	710,145	68,325	3,489,366
tx2010	Redistr.	914,231	116,811	4,456,272
af_shell9	Sparse	504,855	403,884	17,084,020
audikw1	Sparse	943,695	629,360	76,708,152
ldoor	Sparse	952,203	814,652	45,570,272
ecology2	Sparse	999,999	0	3,995,992
ecology1	Sparse	1,000,000	0	3,996,000
thermal2	Sparse	1,227,087	0	7,352,268
af_shell10	Sparse	1,508,065	1,206,452	51,164,260
G3_circuit	Sparse	1,585,478	0	6,075,348
kkt_power	Sparse	2,063,494	195,078	12,964,640
nlpkkt120	Sparse	3,542,400	0	93,303,392
belgium.osm	Street	1,441,295	103	3,099,940
netherlands.osm	Street	2,2166,88	61	4,882,476

In our first experiment, we compare the performance of the algorithms and checksum functions described in Section III. Figure 4 shows their execution times, false positive costs, and occupancies normalized w.r.t. those of INRSRT, when equipped with CS1. As expected, the checksum functions are performing slightly better in terms of occupancy when used with INRSRT since the size of hash range is not limited, i.e., the same with the range of a 32-bit `int`. For INRMEM, this number is q, the first prime number after $|\mathcal{N}|$ which is much smaller than the range of an `int` in practice. We observed that except CS1, all other functions have an occupancy value close to one which is the optimal occupancy. Hence, an $\mathcal{O}(|\mathcal{N}|)$ hash range can be considered as good for practical performance. Yet, it is not enough and a good checksum function is necessary. As the figure shows, even a small increase on the occupancy leads to a high false positive cost. This is somehow expected because when there is more than one representative sharing the same checksum value, all the nets corresponding to a representative need to be compared with other representatives. And when the cardinalities of identical/similar-net sets are large, there can be a large false-positive overhead. The experiments show that this overhead is usually much smaller than the overhead of the sorting phase of INR-SORT. Even for INRMEM equipped with CS1, which has the highest false positive cost, the execution time is much smaller than all the INRSRT variants. But INRMEM equipped with CS2 is the best identical-net removal variant according to our experiments, since the checksum function CS2 is as good as CS3 and MurmurHash, and at the same time, computationally cheaper. Hence, we will continue to use it with INR-SORT for all our removal experiments in the rest of this section.

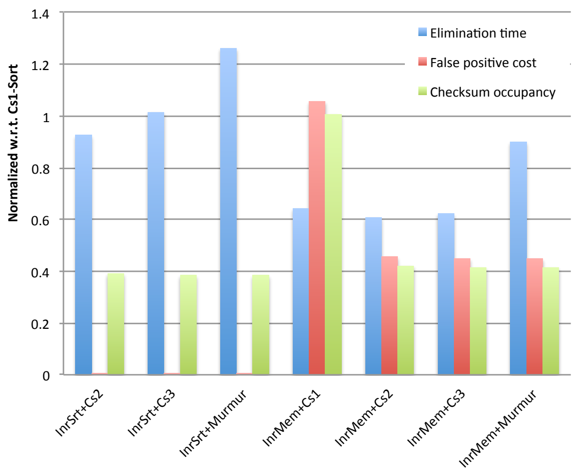


Figure 4. Performance of INRSRT and INRMEM with various checksum functions. The results are normalized with that of INRSRT equipped with CS1.

Table II
AVERAGE PARTITIONING TIMES (IN SECONDS) FOR BASE UMPA AND ITS VARIANTS WITH IDENTICAL-NET (INR) AND -VERTEX REMOVAL (IVR) HEURISTICS. THE LAST COLUMN SHOWS THE SPEEDUP ON THE PARTITIONING TIME WHEN COMPARED WITH THE BASE

K	Base UMPa	INR	INR+IVR	Speedup
2	7.08	6.87	5.98	1.18
8	8.20	7.28	6.43	1.27
32	12.21	8.88	7.95	1.53
128	29.04	13.73	12.73	2.28
512	143.57	44.13	44.19	3.25
1,024	382.89	119.47	115.98	3.30

Table II shows the execution times of base UMPa and its variants equipped with identical-net and -vertex removal heuristics. Compared with the base, we obtain between 1.18–3.30 speedups while partitioning our hypergraphs to a different number of parts. The speedup values are increasing with K and this makes the heuristics more promising and beneficial since the overhead of the partitioning problem is usually an issue for large K values.

We observe that most of the improvement is obtained after removing identical nets. As Table I shows, not all of our hypergraphs contain identical vertices. But some have a lot: for the graph *ldoor*, 85% of the vertices have some number of identical copies in the graph. When this redundancy is removed, we obtain an additional speedup of more than 2, for $K = 512$, compared to UMPa equipped with INR. On the other hand, $14/28$ of the matrices in the test set have less than 103 identical vertices hence, the overhead of executing a vertex removal heuristic will only be a burden. Fortunately, the heuristic is efficient and on average it does not increase the overall time. When K is small, it even helps to reduce the partitioning time.

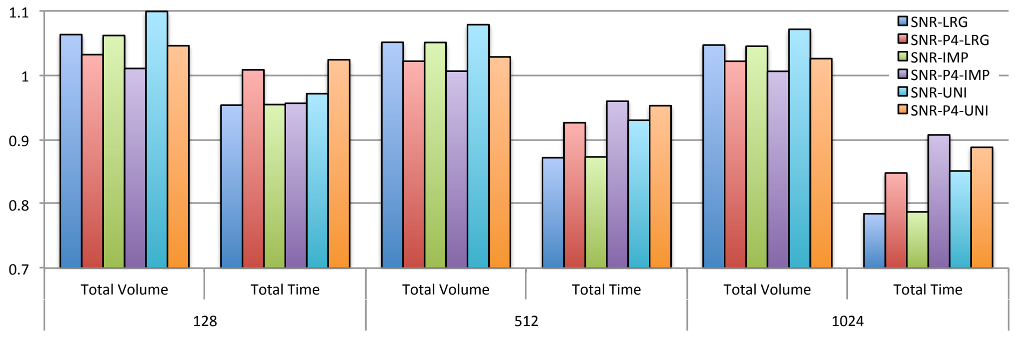
Table III
AVERAGE PARTITIONING QUALITY (CUTSIZE) FOR BASE UMPA AND ITS VARIANTS WITH IDENTICAL-NET (INR) AND -VERTEX REMOVAL (IVR) HEURISTICS. THE LAST COLUMN SHOWS THE IMPROVEMENT ON THE QUALITY METRIC WHEN COMPARED TO THE BASE

K	Base UMPa	INR	INR+IVR	Impr.
2	2,826.78	2,762.04	2,760.49	2.4%
8	15,183.28	15,026.15	14,915.84	1.8%
32	41,656.66	41,833.82	41,541.03	0.3%
128	102,209.71	101,865.58	101,690.31	0.5%
512	223,187.20	223,126.15	222,582.41	0.3%
1,024	321,706.01	321,728.75	319,865.46	0.6%

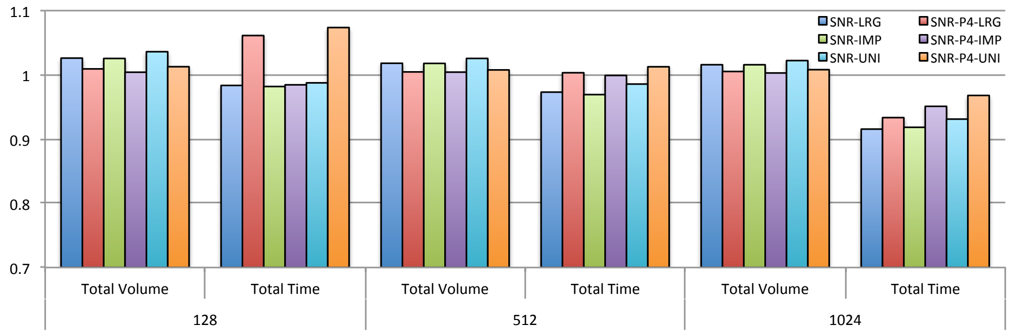
As Table III shows, the quality of the final partition does not reduce when the proposed INR and IVR heuristics are used. On the contrary, as described in Section III-B, removing identical vertices reduces the search space and hence, helps the refinement heuristics while minimizing the partitioning objectives. As the experiments show, when both heuristics are used, the partitioning quality is increased within a small margin, 0.3%–2.4%, on average.

To measure the performance of the similar-net removal heuristic, we compared the performance of INR+IVR with and without SNR. Figure 5 shows the average partitioning times and qualities of the version with SNR normalized w.r.t. INR+IVR for $K = \{128, 512, 1024\}$. To analyze various tradeoff configurations, we tried 4 different variants of SNR. In Figures 5.(a) and Figures 5.(b), we used $t = 4$ and 8 permutations to generate the footprints. Since the footprints are larger in the latter, the heuristic is more restricted, and a smaller number of similar nets are removed. For each t value, in addition to SNR, we used another variant SNR-P4 which restricts the removal process to only the nets with 4 or more pins. In the figures, SNR-X and SNR-P4-X denote the similar-net removal heuristics used with the representative selection option X $\in$ {LRG,IMP,UNI}.

When $t = 8$, we obtain around 10% additional reduction on partitioning time with only 1%–2% reduction on quality. On the other hand, when $t = 4$, since the heuristic is less restricted, the improvements on the partitioning time are larger: 22% and 15%, respectively, for the variants SNR-LRG and SNR-P4-LRG. For these variants the reductions on the partitioning quality are only 5% and 2%, respectively.



(a) $t = 4$, i.e., 4 permutations are used to generate minwise footprints.



(b) $t = 8$, i.e., 8 permutations are used to generate minwise footprints.

Figure 5. Partitioning times and qualities when the similar-net removal heuristic is used in additino to UMPa equipped with the identical-net and -vertex removal heuristics (INR+IVR). The results are normalized w.r.t. INR+IVR. In both figures, SNR-X is the proposed similar-net removal heuristic with the representative selection option X. The SNR-P4-X variant processes only the nets with 4 or more pins.

Overall, by using all the proposed heuristics, we obtained 4.2 speedup w.r.t. base UMPa with 4% reduction on the quality or a 3.9 speedup with only 2% quality reduction.

For the representative selection options, the experiments do not reveal any significant evidence to differentiate LRG and IMP. However, the results show that UNI is slightly worse than the other two for exploiting the time/quality tradeoff. This result is somehow expected since unifying pin sets creates a representative net with a large number of pins which can create a burden for the upcoming matching and refinement heuristics during the multilevel scheme.

V. Conclusion and Future Work

We investigated a set of lossless and lossy hypergraph sparsification heuristics in the context of hypergraph partitioning, which is widely used in parallel computing for load balancing and ordering. As the experiments show, the heuristics are highly effective and efficient, and can be easily integrated to the tools used in practice. The effectiveness of the heuristics increases with the number of parts. This makes them more promising and effective since the partitioning overhead becomes an issue while parallelizing a computation with today's high performance machines which have a large number of processors.

In addition to the proposed heuristics, there exist some simple techniques which have already been employed in UMPa base version such as trivially handling the vertices connected to only one net and vice versa. Since these techniques are relatively straightforward, we do not report on them in this work. Vertices with degree 1 have been studied also for the *betweenness centrality* computations [31] where the practical implications of the existence of graph theoretical structures such as articulation points and bridges were also investigated. We believe that an extension of these techniques to hypergraphs is an interesting study. As a future work, we are planning to analyze how to exploit such structures in the graph and hypergraph partitioning context.

References

[1] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 7, pp. 673–693, 1999.

[2] Ü. V. Çatalyürek, C. Aykanat, and B. Uçar, "On two-dimensional sparse matrix partitioning: Models, methods, and a recipe," *SIAM Journal on Scientific Computing*, vol. 32, no. 2, pp. 656–683, 2010.

[3] Ü. V. Çatalyürek, C. Aykanat, and E. Kayaaslan, "Hypergraph partitioning-based fill-reducing ordering for symmetric matrices," *SIAM Journal on Scientific Computing*, vol. 33, no. 4, pp. 1996–2023, 2011.

[4] L. Grigori, E. G. Boman, S. Donfack, and T. A. Davis, "Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization," *SIAM J. Sci. Comput.*, vol. 32, no. 6, pp. 3426–3446, Nov. 2010.

[5] Ü. V. Çatalyürek, E. Boman, K. Devine, D. Bozdağ, R. Heaphy, and L. Riesen, "A repartitioning hypergraph model for dynamic load balancing," *Journal of Parallel and Distributed Computing*, vol. 69, no. 8, pp. 711–724, Aug 2009.

[6] S. Tan, J. Bu, C. Chen, B. Xu, C. Wang, and X. He, "Using rich social media information for music recommendation via hypergraph model," *ACM Trans. Multimedia Comput. Commun. Appl.*, vol. 7S, no. 1, pp. 22:1–22:22, 2011.

[7] A. Strehl and J. Ghosh, "Cluster ensembles — a knowledge reuse framework for combining multiple partitions," *J. Mach. Learn. Res.*, vol. 3, pp. 583–617, 2003.

[8] S. Shekhar, C.-T. Lu, S. Chawla, and S. Ravada, "Efficient join-index-based spatial-join processing: A clustering approach," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 6, pp. 1400–1421, nov/dec 2002.

[9] D.-R. Liu and M.-Y. Wu, "A hypergraph based approach to declustering problems," *Distributed and Parallel Databases*, vol. 10, pp. 269–288, 2001.

[10] M. M. Özdal and C. Aykanat, "Hypergraph models and algorithms for data-pattern-based clustering," *Data Mining and Knowledge Discovery*, vol. 9, pp. 29–57, 2004.

[11] G. Karypis and V. Kumar, *hMeTiS: A hypergraph partitioning package*, Minneapolis, MN 55455, 1998.

[12] A. Caldwell, A. Kahng, and I. Markov, "Improved algorithms for hypergraph bipartitioning," in *Proc. Design Automation Conference*, june 2000, pp. 661–666.

[13] B. Vastenhouw and R. H. Bisseling, "A two-dimensional data distribution method for parallel sparse matrix-vector multiplication," *SIAM Review*, vol. 47, no. 1, pp. 67–95, 2005.

[14] A. Trifunovic and W. Knottenbelt, "Parkway 2.0: A parallel multilevel hypergraph partitioning tool," in *Proc. ISCIS*, ser. LNCS. Springer Berlin / Heidelberg, 2004, vol. 3280, pp. 789–800.

[15] Ü. 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.

[16] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Uçar, "UMPa: A multi-objective, multi-level partitioner for communication minimization, 10th DIMACS implementation challenge: Graph partitioning and graph clustering," 2012, http://www.cc.gatech.edu/dimacs10/.

[17] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, V. Leung, L. A. Riesen, C. Vaughan, Ü. V. Çatalyürek, D. Bozdağ, W. Mitchell, and J. Teresco, *Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; User's Guide*, Sandia National Laboratories, Albuquerque, NM, 2007, tech. Report SAND2007-4748W.

[18] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph-partitioning-based models and methods for exploiting cache locality in sparse-matrix vector multiplication," Bilkent University, Dept. of Computer Engineering, Tech. Rep., 2012.

[19] A. N. Yzelman and R. H. Bisseling, "Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods," *SIAM J. Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009.

[20] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek, "Parallel hypergraph partitioning for scientific computing," in *Proc. of 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006.

[21] C. Aykanat, B. B. Cambazoğlu, and B. Uçar, "Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices," *Journal of Parallel and Distributed Computing*, vol. 68, no. 5, pp. 609–625, May 2008.

[22] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*. Chichester, U.K.: Wiley–Teubner, 1990.

[23] T. N. Bui and C. Jones, "A heuristic for reducing fill-in sparse matrix factorization," in *Proc. 6th SIAM Conf. Parallel Processing for Scientific Computing*. SIAM, 1993, pp. 445–452.

[24] C. Ashcraft, "Compressed graphs and the minimum degree algorithm," *SIAM J. Sci. Comput.*, vol. 16, no. 6, pp. 1404–1411, 1995.

[25] B. Hendrickson and E. Rothberg, "Improving the run time and quality of nested dissection ordering," *SIAM J. Sci. Comput.*, vol. 20, no. 2, pp. 468–489, 1998.

[26] A. Appleby, "SMHasher & MurmurHash," 2012, http://code.google.com/p/smhasher/.

[27] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bulletin del la Société Vaudoise des Sciences Naturelles*, vol. 37, pp. 547–579, 1901.

[28] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations (extended abstract)," in *Proc. 13th ACM Symposium on Theory of Computing*, ser. STOC '98, 1998, pp. 327–336.

[29] V. Satuluri, S. Parthasarathy, and Y. Ruan, "Local graph sparsification for scalable clustering," in *Proc. 2011 ACM SIGMOD International Conference on Management of Data*, 2011, pp. 721–732.

[30] "10th DIMACS Implementation Challenge: Graph Partitioning and Graph Clustering," 2012, http://www.cc.gatech.edu/dimacs10/.

[31] A. E. Sariyüce, E. Saule, K. Kaya, and Ümit V. Çatalyürek, "Shattering and compressing networks for centrality analysis," Biomedical Informatics Department, The Ohio State University, Tech. Rep., 2012.