Multithreaded clustering for multi-level hypergraph partitioning

Umit Catalyurek, Mehmet Deveci, Kamer Kaya, Bora Uçar

Multithreaded Clustering for Multi-level Hypergraph Partitioning

Ümit V. Çatalyürek, Mehmet Deveci, Kamer Kaya
The Ohio State University
Dept. of Biomedical Informatics
{*umit,mdeveci,kamer*}*@bmi.osu.edu*

Bora Uçar
CNRS and LIP, ENS Lyon
Lyon 69364, France
bora.ucar@ens-lyon.fr

***Abstract*—Requirements for efficient parallelization of many complex and irregular applications can be cast as a hypergraph partitioning problem. The current-state-of-the art software libraries that provide tool support for the hypergraph partitioning problem are designed and implemented before the game-changing advancements in multi-core computing. Hence, analyzing the structure of those tools for designing multithreaded versions of the algorithms is a crucial tasks. The most successful partitioning tools are based on the multi-level approach. In this approach, a given hypergraph is coarsened to a much smaller one, a partition is obtained on the the smallest hypergraph, and that partition is projected to the original hypergraph while refining it on the intermediate hypergraphs. The coarsening operation corresponds to clustering the vertices of a hypergraph and is the most time consuming task in a multi-level partitioning tool. We present three efficient multithreaded clustering algorithms which are very suited for multi-level partitioners. We compare their performance with that of the ones currently used in today's hypergraph partitioners. We show on a large number of real life hypergraphs that our implementations, integrated into a commonly used partitioning library PaToH, achieve good speedups without reducing the clustering quality.**

***Keywords*-Multi-level hypergraph partitioning; coarsening; multithreaded clustering algorithms; multicore programming**

I. Introduction

Hypergraph partitioning is an important problem widely encountered in parallelization of complex and irregular applications from various domains including VLSI design [1], parallel scientific computing [2], [3], sparse matrix reordering [4], static and dynamic load balancing [5], software engineering [6], cryptosystem analysis [7], and database design [8], [9], [10]. Being such an important problem, considerable effort has been put in providing tool support, see hMeTiS [11], MLpart [12], Mondriaan [13], Parkway [14], PaToH [15], and Zoltan [16].

All the tools above follow the multi-level approach. This approach consists of three phases: coarsening, initial partitioning, and uncoarsening. In the coarsening phase, the original hypergraph is reduced to a much smaller hypergraph after a series of coarsening levels. At each level, vertices that are deemed to be similar are grouped to form vertex clusters, and a new hypergraph is formed by unifying a cluster as a single vertex. That is, the clusters become the vertices for the next level. In the initial partitioning phase, the coarsest hypergraph is partitioned. In the uncoarsening phase, the partition found in the second phase is projected back to the original hypergraph where the partition is locally refined on the hypergraphs associated with each coarsening level.

The coarsening phase is the most important phase of the multi-level approach. This is for the following three reasons. First, the worst-case running time complexity of this phase is higher than the other two phases (initial partitioning and uncoarsening phases have, in most common implementations, linear worst-case running time complexity). Second, as the uncoarsening level performs local improvements, the quality of a partition is highly affected by the quality of the coarsening phase. For example, given a hypergraph, a coarsening algorithm, a conventional initial partitioning algorithm and a refinement algorithm based on the most common ones, very slight variations on vertex similarity metrics can effect the performance quite significantly (see for example the start of Section 5.1 of [17]). Third, it is usually the case that the better the coarsening, the faster the uncoarsening phase is. Therefore, the coarsening phase also affects the running time of the other phases.

Our aim in this paper is to efficiently parallelize the coarsening phase of PaToH, a well-known and commonly used hypergraph partitioning tool. The algorithmic kernel of this phase is a *clustering* algorithm that marks similar vertices to be coalesced. There are two classes of clustering algorithms in PaToH. The algorithms in the first class allow at most two vertices in a cluster. These algorithms are called *matching-based* or *matching* algorithms in short. The algorithms in the second class, called *agglomerative* algorithms, allow any number of vertices to become together to form a cluster. The most effective clustering algorithms in PaToH are agglomerative ones whereas the fastest ones are matching based. We propose efficient parallelization of these two classes of algorithms (Section III). We report practical experiments with PaToH (and its coarsening phase alone) on a recent multicore architecture (Section IV). Our techniques are easily applicable to some other sequential hypergraph partitioners, since they use the same multilevel approach and have similar data structures.

II. Background, problem formulation and related work

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 *size* of a net is the number of its pins, and the *degree* of a vertex is equal to the number of nets that contain it. Graph is a special instance of hypergraph such that each net has size two. We use $\mathsf{pins}[n]$ and $\mathsf{nets}[v]$ to represent the pins of a net n, and the set of nets that contain vertex v, respectively. Vertices can be associated with weights, denoted with $\mathsf{w}[\cdot]$, and nets can be associated with costs, denoted with $\mathsf{c}[\cdot]$.

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 $1 \leq k < \ell \leq K$,
- 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 $1 \leq k \leq K$,
- union of K parts is equal to $\mathcal{V}$, i.e., $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$.

In a 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., $\lambda_n > 1$).

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*

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

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

The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. Let $\chi(\Pi)$ denote the cost, i.e., *cutsize*, of a partition Π. There are various cutsize definitions [1] such as:

$$\chi(\Pi) = \sum_{n \in \mathcal{N}_E} \mathsf{c}[n] \tag{2}$$

$$\chi(\Pi) = \sum_{n \in \mathcal{N}} \mathsf{c}[n](\lambda_n - 1)\ . \tag{3}$$

In (2) and (3), each cut net n contributes $\mathsf{c}[n]$ and $\mathsf{c}[n](\lambda_n - 1)$ to the cutsize, respectively. The cutsize metric given in (2) will be referred to here as *cut-net* metric and the one in (3) will be referred as *connectivity* metric. Given ε and an integer $K > 1$, the hypergraph partitioning problem can be defined as the task of finding a balanced partition Π with K parts such that $\chi(\Pi)$ is minimized. The hypergraph partitioning problem is NP-hard [1].

B. Clustering algorithms for hypergraph partitioning

As said before, there are two classes of clustering algorithms: matching-based ones and agglomerative ones. The matching-based ones put at most two similar vertices in a cluster whereas the agglomerative ones allow any number of similar vertices. There are various similarity metrics—see for example [2], [18], [19]. All these metrics are defined on two adjacent vertices (one of them can be a vertex cluster). Two vertices are adjacent if they share a net, i.e., the vertices u and v are matchable if $\mathcal{N}_{uv} = \mathsf{nets}[u] \cap \mathsf{nets}[v] \neq \emptyset$. In order to find a given vertex u's adjacent vertices, one needs to visit each net $n \in \mathsf{nets}[u]$ and then visit each vertex $v \in \mathsf{pins}[n]$. Therefore, the computational complexity of the clustering algorithms is at least in the order of $\sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|^2$. As mentioned in the introduction, the other two phases of the multi-level approach have a linear time worst case time complexity. As $\sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|^2$ is most likely to be the bottleneck, an effective clustering algorithm's worst case running time should not pass this limit for the algorithm to be efficient as well.

The sequential implementations of the clustering algorithms in PaToH proceed in the following way to have a running time proportional to $\sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|^2$. The vertices are visited in a given (possibly random) order and each vertex u, if not clustered yet, is tried to be clustered with the most similar vertex or cluster. In the matching-based ones, the current vertex u if not matched yet, chooses one of its unmatched adjacent vertices according to a criterion. If such a vertex v exists, the matched pair u and v are marked as a cluster of size two. If there is no unmatched adjacent vertex of u, then vertex u remains as a singleton cluster. In the agglomerative ones, the current vertex u, if not marked to be in a cluster yet, can choose a cluster to join (thus forming a cluster of size at least three), or can create another cluster with one of its unmatched adjacent vertices (thus forming a cluster of size two). Hence in agglomerative clustering, vertex u never remains as a singleton, as long as it is not isolated (i.e., not connected to any net).

For the clustering algorithms in this paper, there exists a representative vertex for each cluster. When a vertex $u \in \mathcal{V}$ is put into a cluster, we set $\mathsf{rep}[u]$ to the representative of this group. When a singleton vertex u chooses another one v, we choose one as the representative and set $\mathsf{rep}[u]$ and $\mathsf{rep}[v]$ accordingly. For all the algorithms, we assume that $\mathsf{rep}[u]$ is initially **null** for all $u \in \mathcal{V}$. This will also be true if u remains singleton at the end of the algorithm.

Algorithm 1 presents one of the matching-based clustering algorithms that are available in PaToH. In this algorithm, the vertex u (if not matched yet) is matched with currently unmatched neighbor v with the maximum connectivity, where the connectivity refers to the sum of the costs of the common nets. This matching algorithm is called as Heavy Connectivity Matching (HCM) in PaToH [2], and Inner

Algorithm 1: Sequential greedy matching (HCM)

```
   Data: H = (V, N), rep
1  for each vertex u ∈ V in a given order do
       if rep[u] = null then
2          adj_u ← {}
           for each net n ∈ nets[u] do
               for each vertex v ∈ pins[n] do
                   if rep[v] ≠ null then
3                      pins[n] ← pins[n] \{v}
                   else
4                      if v ∉ adj_u then
5                          adj_u ← adj_u ∪ {v}
6                      conn[v] ← conn[v] + c[n]
           v* ← u
           conn* ← 0
           for each vertex v ∈ adj_u do
               if conn[v] > conn* and v ≠ u then
                   conn* ← conn[v]
                   v* ← v
               conn[v] ← 0
           if u ≠ v* then
               rep[v* ] ← u
               rep[u] ← u
```

Algorithm 2: Sequential agglomer. clustering (HCC)

```
   Data: H = (V, N), maxW, rep
   for each vertex u ∈ V in a given order do
       if rep[u] = null then
           adj_u ← {}
           for each net n ∈ nets[u] do
               for each vertex v ∈ pins[n] do
                   if v ∉ adj_u then
                       adj_u ← adj_u ∪ {v}
               conn[v] ← conn[v] + c[n]
           v* ← u
           conn* ← 0
           for each vertex v ∈ adj_u do
               if u = v then
                   continue
               v^r ← rep[v]
               if v^r = null then
                   v^r ← v
               if v^r ≠ v then
                   conn[v^r] ← conn[v^r] + conn[v]
                   conn[v] ← 0
                   adj_u ← adj_u ∪ {v^r} \ {v}
1              totW ← w[u] + w[v^r]
               if conn[v^r] > conn* then
                   if totW < maxW then
                       conn* ← conn[v^r]
                       v* ← v^r
           for each vertex v ∈ adj_u do
               conn[v] ← 0
           if u ≠ v* then
               rep[v* ] ← v*
               rep[u] ← v*
               w[v* ] ← w[v* ] + w[u]
```

Product Matching (IPM) in Zoltan [20] and Mondriaan [13]. One can have different variations of this algorithm by changing the vertex visit order (line 1) and/or using different scaling schemes while computing the contribution of each net to its pins (line 6). The array $\mathsf{conn}[\cdot]$ of size $|\mathcal{V}|$ is necessary to compute the connectivity of the vertex u and all its adjacent vertices in time linearly proportional to the number of adjacent vertices. The operation at line 3 is again for efficiency. It removes the matched vertices from $\mathsf{pins}[n]$, hence the next searches on $\mathsf{pins}[n]$ will take less time.

Algorithm 2 presents one of the agglomerative clustering algorithms that are available in PaToH. Similar to the sequential HCM algorithm, vertices are again visited in a given order. If a vertex u has already been clustered, it is skipped. However, an unclustered vertex u can choose to join an existing cluster, can start a new cluster with a vertex, or stay as a singleton cluster. Therefore, compared to the previous algorithm, all adjacent vertices of the current vertex u are considered for selection. In order to avoid building an extremely large cluster (which would cause load balance problem in initial partitioning and refinement phases), we also enforce that weight of a cluster must be smaller than a given value $maxW$. Our experience shows that such restriction is not needed in matching based clustering, since at each level only at most two vertices can be clustered together.

In Algorithm 2, we use the total shared net cost (heavy connectivity clustering) as the similarity metric. In practice (and in our experiments), we use the absorption clustering metric (implemented in PaToH) which divides the contribution of each net to the number of its pins. That is, a net n contributes $\mathsf{c}[n]/|\mathsf{pins}[n]|$ to the similarity value instead of $\mathsf{c}[n]$. This metric favors clustering vertices connected via nets of small sizes. The sequential code in PaToH also divides the overall similarity score between two vertices by the weight of the cluster which will contain u (the value $totW$ at line 1). Hence, to compare the performance of our multithreaded clustering algorithms with PaToH, we also use this modified similarity metric in our implementations. However, for simplicity, we will continue to use the heavy connectivity clustering metric in the paper.

C. Metrics

We define the metrics of *cardinality* and *quality* to compare different clustering methods. The cardinality is defined as the number of clustering decisions taken by an algorithm, i.e.,

$$\text{cardinality} = \sum_{u \in \mathcal{V}, \mathsf{rep}[u] \neq \mathbf{null}} (|\{v \in \mathcal{V} : \mathsf{rep}[v] = u\}| - 1) .$$

In the multi-level framework, this represents the reduction on the number of vertices between two consecutive coarsening levels. The quality of a clustering is defined as the sum of the similarities between each vertex pair which resides in the same cluster, i.e.,

$$\text{quality} = \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{C}_u} \sum_{n \in \mathcal{N}_{uv}} \frac{\mathsf{c}[n]}{2} ,$$

where $\mathcal{C}_u = \{v \in \mathcal{V} \setminus \{u\} : \mathsf{rep}[v] = \mathsf{rep}[u]$ and $\mathsf{rep}[v] \neq \mathbf{null}\}$ is the set of vertices which are in the same cluster with u, and $\mathcal{N}_{uv}$ is the set of nets shared by u and v.

Although the definitions are generic to be used for both matching-based and agglomerative clustering, we do not use these criteria to compare a matching-based clustering heuristic with an agglomerative one, since the latter has an obvious advantage.

D. Related work

For a given hypergraph $\mathcal{H}=(\mathcal{V},\mathcal{N})$, let A be the vertex-net incidence matrix, i.e., the rows of A correspond to the vertices of $\mathcal{H}$, and the columns of A correspond to the nets of $\mathcal{H}$ such that $a_{vn} = 1$ iff $v \in \mathsf{pins}[n]$. Consider now the symmetric matrix $M = AA^T - diag(AA^T)$. The matrix M can be effectively represented by an undirected graph $\mathcal{G}(M)$ with $|\mathcal{V}|$ vertices and having an edge of weight m_{uv} between two vertices u and v if $m_{uv} \neq 0$. That is, there is a one-to-one correspondence between the vertices of $\mathcal{H}$ and $\mathcal{G}(M)$. As $m_{uv} \neq 0$ iff the vertices u and v of $\mathcal{H}$ are adjacent, the correspondence implies that a matching among the vertices of $\mathcal{H}$ corresponds to a matching on the vertices of $\mathcal{G}(M)$. Therefore, various matching algorithms and heuristics for graphs can be used on $\mathcal{G}(M)$ to find a matching among the vertices of $\mathcal{H}$.

Bisseling and Manne [21] propose a distributed memory, 1/2-approximate algorithm to find weighted matchings in graphs. Building on this work, Çatalyürek et al. [22] present efficient distributed-memory parallel algorithms and scalable implementations. Halappanavar et al. [23] present an efficient shared-memory implementations for computing 1/2-approximate weighted matchings. For maximum cardinality matching problem, in a recent work, Patwary et al. [24] propose a distributed memory, sub-optimal algorithm.

There are a number of reasons why we cannot use aforementioned algorithms. First and foremost, storing the graph $\mathcal{G}(M)$ requires a large memory. The time required to compute this graph is about as costly as computing a matching in $\mathcal{H}$ in a sequential execution. Second, it is our experience (with the coarsening algorithms within the multi-level partitioner PaToH) that one does not need a maximum weighted matching, nor a maximum cardinality one, nor an approximation guarantee to find helpful coarsening schemes. Third, while matching the vertices of a hypergraph, we sometimes need to avoid matched vertices become too big, or favor vertex clusters with smaller weights (due to multi-level nature of the partitioning algorithm), and vertices that are mostly related via nets of smaller size. These modifications can be incorporated into the graph $\mathcal{G}(M)$ by adjusting the edge weights (or by leaving some edges out). This will help reduce the memory requirements of the graph matching based algorithms. However, the computational cost of constructing the graph remains the same. Almost all of the most effective sequential clustering algorithms implemented in PaToH for coarsening purposes has the same worst case time complexity but are much faster in practice. We, therefore, cannot afford building the graph $\mathcal{G}(M)$ or its modified versions and call existing graph matching algorithms. Furthermore, agglomerative clustering algorithms cannot be accomplished by using the aforementioned algorithms.

We highlight that the matching-based clustering algorithms considered in this paper can be perceived as a graph matching algorithm adjusted to work on an implicit representation of the graph $\mathcal{G}(M)$ or its modified versions. However, to the best of our knowledge, there is no immediate parallel graph-matching based algorithm that is analogous to the agglomerative clustering algorithms considered in this work (although variants of agglomerative coarsening for graphs exist, see for example [25] and [26]).

As a sanity check, we implemented a modified version of the sequential 1/2-approximation algorithm of Halappanavar et al. [23] which works directly on hypergraphs. In other words, instead of explicitly constructing the graph $\mathcal{G}(M)$, adjacencies of vertices are constructed on the fly, like Algorithm 1. We compared the quality and cardinality of this algorithm with that of the greedy sequential matching HCM. The approximation algorithm obtained matchings with better quality by 14% while the cardinalities were the same. However, this good performance comes with a significant execution time overhead, yielding 6.5 times slower execution. When we integrated the 1/2-approximation algorithm into the coarsening phase of PaToH, we observed that better matching quality helps the partitioner to obtain better cutsize. For example, when the partitioner is executed 10 times with random seeds, the average cutsize of 1/2-approximation algorithm is 8% better than the one obtained by using HCM. However, when we compare the minimum of these cutsizes, HCM outperforms the approximation algorithm by 2%. Moreover, the difference between the minimum cut obtained by using HCM with the average cut obtained by using the approximation algorithm is 14% in favor of HCM. After these preliminary experiments, we decided to parallelize HCM and HCC, since they are much faster, and one can obtain better cutsize by using them.

III. Multithreaded Clustering for Coarsening

In this section, we will present three novel parallel greedy clustering algorithms. The first two are matching-based and the third one is a greedy agglomerative clustering method.

A. Multithreaded matching algorithms

To adapt the greedy sequential matching algorithm for multithreaded architectures, we use two different approaches. In the first one, we employ a locking mechanism which prevents inconsistent matching decisions between the threads. In the second approach, we let the algorithm run almost without any modifications and then use a conflict resolution mechanism to create a consistent matching.

The lock-based algorithm is given in Algorithm 3. The structure of the algorithm is similar to the sequential one except the lines 2 and 5, where we use the atomic CHECKANDLOCK operation. To lock a vertex u, this operation first checks if u is already locked or not. If not, it locks it and returns **true**. Otherwise, it returns **false**. Its atomicity guarantees that a locked vertex is never considered for a matching. That is, both the visited vertex u (at line 1) and the adjacent vertex v must not be locked to consider the matching of u and v. If they are, and if the similarity of v is bigger than the current best (at line 3), the algorithm keeps v as the best candidate v^*. When a better candidate is found, the old one is UNLOCKed to make it available for other threads (line 6), and to construct better matchings in terms of cardinality and quality.

Algorithm 3: Parallel lock-based matching

```
   Data: H = (V, N), rep
1  for each vertex u ∈ V in parallel do
2      if CHECKANDLOCK(u) then
           adj_u ← {}
           for each net n ∈ nets[u] do
               for each vertex v ∈ pins[n] do
                   if v ∉ adj_u then
                       adj_u ← adj_u ∪ {v}
                   conn[v] ← conn[v] + c[n]
           v* ← u
           conn* ← 0
3          for each vertex v ∈ adj_u do
4              if conn[v] > conn* then
5                  if CHECKANDLOCK(v) then
                       if u ≠ v* then
6                          UNLOCK(v*)
                       conn* ← conn[v]
                       v* ← v
               conn[v] ← 0
           if u ≠ v* then
               rep[u] ← u
               rep[v* ] ← u
```

As a different approach without a lock mechanism, we modify the sequential code slightly and execute it in a multithreaded environment. If the **for** loop at line 1 of Algorithm 1 is executed in parallel, different threads may set $\mathsf{rep}[u]$ to different values for a vertex u. Hence, the rep array will contain inconsistent decisions. To solve this issue, one can make each thread use a private rep array and store all of its matching decisions locally. Then, a consistent matching can be devised from this information in another phase. Another idea is keeping the sequential code (almost) as is, letting the threads create conflicts, and resolving the conflicts later. Our preliminary experiments show that there is not much difference between the performances of these two approaches in terms of the cardinality and the quality. However, the first one requires more memory: one rep array per thread compared to a shared one. Hence, we followed the second idea and use a conflict resolution scheme with $\mathcal{O}(|\mathcal{V}|)$ complexity. Algorithm 4 shows the pseudocode of our parallel resolution-based algorithm.

Algorithm 4: Parallel resolution-based matching

```
   Data: H = (V, N), rep
   for each vertex u ∈ V in parallel do
       if rep[u] = null then
           adj_u ← {}
           for each net n ∈ nets[u] do
               for each vertex v ∈ pins[n] do
                   if v ∉ adj_u then
                       adj_u ← adj_u ∪ {v}
                   conn[v] ← conn[v] + c[n]
           v* ← u
           conn* ← 0
           for each vertex v ∈ adj_u do
1              if rep[v] = null then
                   if conn[v] > conn* then
                       if u ≠ v then
                           conn* ← conn[v]
                           v* ← v
               conn[v] ← 0
2          if rep[u] = rep[v] = null then
3              rep[u] ← v*
4              rep[v* ] ← u
5  for each vertex u ∈ V in parallel do
       v ← rep[u]
       if v ≠ null and u ≠ rep[v] then
           rep[u] ← null
   for each vertex u ∈ V in parallel do
       v ← rep[u]
       if v ≠ null and u < v then
           rep[u] ← u
```

Our conflict resolution scheme starts at line 5 of Algorithm 4. Note that instead of setting a fixed representative

for the matched vertices u and v^*, we set their rep values to each other. This bidirectional information is then used in our resolution scheme to check if the rep array contains inconsistent information. That is if $u \neq \mathsf{rep}[\mathsf{rep}[u]]$ for a vertex u, we know that at least two threads matched either u or v with different vertices. If this is the case, the resolution scheme acts greedily and aggressively sets $\mathsf{rep}[u]$ to **null** indicating u will be unmatched. After the first loop at line 5, which is executed in parallel, the rep array will contain the matching decisions consistent with each other. Then, with the last parallel loop, we set the representatives for each matched pair.

The proposed resolution scheme is sufficient to obtain a valid matching in the multithreaded setting. However, we slightly modify Algorithm 1 to obtain better matchings. Since each conflict will probably cost a pair, and losing a pair reduces matching cardinality and hence, quality, we desire lesser conflicts. To avoid them, at line 1 of Algorithm 4, we check if a vertex v adjacent to u is already matched. If we detect an already matched candidate, we do not consider it as a possible mate for u. Furthermore, at line 2, we again verify if u and v are already matched right before matching them. This check is necessary since either of them could have been matched by another thread after the current one starts considering them.

B. Multithreaded agglomerative clustering

To adapt the sequential agglomerative clustering algorithm to multithreaded setting, we use the same lock-based approach integrated to Algorithm 3. The pseudocode of the parallel agglomerative clustering algorithm is given in Algorithm 5. The algorithm visits the vertices in parallel, and when a thread visits a vertex u, it tries to lock u. If u is already locked the thread skips u and visits the next vertex. If u is not locked but is already a member of a cluster, the thread unlocks u. Since a cluster cannot be the source of a new cluster, this is necessary. On the other hand, if u is a singleton vertex, the thread continues by computing the similarity values for each adjacent vertex and then traverses the adjacency list adj_u along the same lines as the sequential algorithm. The main difference here is the locking request for v^r (line 1) which is either set to v if v is singleton, or to the representative of the cluster that v resides in. Before considering v^r as a matching candidate, this lock is necessary. However, if v^r is already the best candidate, it is not so (since the thread has already grabbed v^r). When the lock is granted, the thread checks if the adjacent vertex v, which was singleton before, has been assigned to a cluster by another thread. If this is the case, the thread unlocks the representative and continues with the next adjacent vertex. Otherwise, it recomputes the total weight of u and v^r (line 3) since new vertices might have been inserted to v^r's cluster by other threads. Since insertions can only increase $\mathsf{w}[v^r]$ and $\mathsf{conn}[v^r]$, we do not need to compare $\mathsf{conn}[v^r]$ with $conn^*$ again. On the other hand, since we cannot construct clusters with large weights, we need to check if $totW$ is still smaller than $maxW$ (line 4). When the best candidate v^* is found, we put u in the cluster v^* represents and update the rep and w arrays accordingly. Unlike the matching based algorithms, a cluster is allowed to be a candidate more than once throughout the execution. Hence, at the end of the iteration (lines 5 and 6) we unlock all the vertices that are locked during this iteration.

C. Implementation Details

To obtain lock functionality for the multithreaded clustering algorithms described in the previous section, we use the compare and exchange CPU instruction which exists in x86 and Itanium architectures. We first allocate a lock array of length $|\mathcal{V}|$ and initialize all entries to 0. For each call of the corresponding function in C, `__sync_bool_compare_and_swap`, the entry related with the lock request is compared with 0. In case of equality, it is set to 1, and the function returns **true**. On the other hand, if the entry is not 0 then it returns **false**. To unlock a vertex, we simply set the related entry in the lock array to 0.

Although this function provides great support and flexibility for concurrency, our preliminary experiments show that it can also reduce the efficiency of a multithreaded algorithm. To alleviate this, we try reduce the number of calls on this function by adding an if statement before each lock request which helps us to see if the lock is really necessary. We observe significant improvements on the execution times due to these additional **if** statements. For example, the parallel lock-based matching algorithm described in the previous section should be implemented as in Algorithm 6 to make it much faster. We stress that the **if** statements at lines 1 and 3 do not change anything in the execution flow. That is if a vertex is not locked, it cannot be also matched since a matched vertex always stays locked. Hence everything that can pass the lock requests (lines 2 and 4) can also pass the previous if statements. However, the opposite is not true.

We used the same hypergraph data structure with PaToH. We store the ids of the vertices of each net n, that is its pins, consecutively in an array ids of size $\sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|$. We also keep another array xids of size $|\mathcal{N}| + 1$, which stores the start index of each net's pins. Hence, in our implementation, the pins of a net n, denoted by $\mathsf{pins}[n]$ in the pseudocodes, are stored in $\mathsf{ids}[\mathsf{xids}[n]]$ through $\mathsf{ids}[\mathsf{xids}[n+1]-1]$.

With the data structures above, the computational complexity of the clustering algorithms in this paper are in the order of $\sum_{n \in \mathcal{N}} |\mathsf{pins}[n]|^2$, since all non-loop lines in their pseudocodes have $\mathcal{O}(1)$ complexity. For example, in Algorithm 1, to remove matched vertices from $\mathsf{pins}[n]$ (line 3), we keep a pointer array netend of size $\mathcal{N}$ where $\mathsf{netend}[n]$ initially points to the last vertex in $\mathsf{pins}[n]$ for all $n \in \mathcal{N}$. Then, to execute $\mathsf{pins}[n] \leftarrow \mathsf{pins}[n] \setminus \{v\}$, we only decrease $\mathsf{netend}[n]$ and swap v with the vertex in the

Algorithm 5: Parallel agglomerative clustering

```
   Data: H = (V, N), maxW, rep
   for each vertex u ∈ V in parallel do
       if CHECKANDLOCK(u) then
           if rep[u] ≠ null then
               UNLOCK(u)
               continue
           adj_u ← {}
           for each net n ∈ nets[u] do
               for each vertex v ∈ pins[n] do
                   if v ∉ adj_u then
                       adj_u ← adj_u ∪ {v}
               conn[v] ← conn[v] + c[n]
           v* ← u
           conn* ← 0
           for each vertex v ∈ adj_u do
               if u = v then
                   continue
               v^r ←rep[v]
               if v^r = null then
                   v^r ← v
               if v^r ≠ v then
                   conn[v^r] ← conn[v^r] + conn[v]
                   #replace v with v^r
                   conn[v] ← 0
                   adj_u ← adj_u ∪ {v^r} \ {v}
               totW ← w[u] + w[v^r]
               if conn[v^r] > conn* and totW < maxW then
1                  if v^r = v* or CHECKANDLOCK(v^r) then
2                      if rep[v] ≠ v^r and rep[v] ≠ null then
                           UNLOCK(v^r)
                           continue
3                      totW ← w[u] + w[v^r]
4                      if totW < maxW then
                           conn* ← conn[v^r]
                           v* ← v^r
                           if u ≠ v* then
                               UNLOCK(v*)
                       else
                           UNLOCK(v^r)
           for each vertex v ∈ adj_u do
               conn[v] ← 0
           if u ≠ v* then
               rep[v* ] ← v*
               rep[u] ← v*
               w[v* ] ← w[v* ] + w[u]
5              UNLOCK(v*)
6          UNLOCK(u)
```

Algorithm 6: Parallel lock-based matching: modified

```
   for each vertex u ∈ V in parallel do
1      if rep[v] = null then
2          if CHECKANDLOCK(u) then
               ...
               for ··· do
                   ...
                   if ··· then
3                      if rep[v] = null then
4                          if CHECKANDLOCK(v) then
                               ...
           ...
           ...
```

new location. In this way, we also keep the list of vertices connected to each net unchanged since we only reorder them.

In the actual implementation of Algorithm 1, the set adj_u corresponds to an array of maximum size $|\mathcal{V}|$, and an integer which keeps the number of adjacent vertices in the array. With this pair, the vertex addition (line 5) and reset (line 2) operations take constant time. Furthermore, to find if a vertex v is a member of adj_u (line 4), we use $\mathsf{conn}[v]$ since the edge costs are positive, and $\mathsf{conn}[v] > 0$ if and only if $v \in \mathsf{adj}_u$. The implementation of these lines are the same for other algorithms.

IV. Experimental Results

The algorithms are tested on a computer with 2.27GHz dual quad-core Intel Xeon CPUs with 2-way hyper-threading enabled, and 48GB main memory. All of the algorithms are implemented in C and OpenMP. The compiler is icc version 11.1 and -O3 optimization flag is used.

To generate our hypergraphs, we used real life matrices from the University of Florida (UFL) Sparse Matrix Collection (http://www.cise.ufl.edu/research/sparse/matrices). We randomly choose 70 large, square matrices from the library and create corresponding hypergraphs using the column-net hypergraph model [2]. An overall summary of the properties of these hypergraphs is given in Table I. The complete list of matrices is at http://bmi.osu.edu/~kamer/multi_coarse_matrices.txt.

Table I
Properties of the hypergraphs used in the experiments

	min	max	average
# vertices	256,000	9,845,725	1,089,073
# pins	786,396	57,156,537	6,175,717
$\frac{\#pins}{\#vertices}$	1.91	39.53	6.61

A. Individual performance of the algorithms

We first compare the performance of the multithreaded clustering algorithms with respect to the cardinality, quality and speedup as standalone clustering algorithms.

1) Performance on cardinality and quality: For matching based clustering, a matching with the best quality can be found by first constructing the matrix $M = AA^T - diag(AA^T)$ where A is the vertex-net incidence matrix. Then a maximum weighted matching in $\mathcal{G}(M)$, the associated weighted graph of M, is also the maximum quality matching.

We use Gabow's maximum weighted matching algorithm [27]—implemented by Rothberg and available as a part of The First DIMACS Implementation Challenge (available at ftp://dimacs.rutgers.edu/pub/netflow). This algorithm has a time complexity of $\mathcal{O}(|\mathcal{V}|^3)$ for a graph on $|\mathcal{V}|$ vertices and finds a maximum weighted matching in the graph, not necessarily among the maximum cardinality ones.

Due to the running time complexity of the maximum quality matching algorithm, it is impractical to obtain the relative performance of the clustering algorithms on our original dataset. We therefore use an additional data set containing considerably small matrices. The new dataset contains all 289 square matrices in the UFL sparse matrix collection with at least $3,000$ and at most $10,000$ rows. We construct the hypergraphs for each of these matrices and find the maximum quality matching on them.

Table II
RELATIVE PERFORMANCE OF THE SEQUENTIAL AND PARALLEL MATCHING BASED CLUSTERING ALGORITHMS W.R.T. TO THE MAXIMUM QUALITY MATCHINGS (# OF THREADS = 8).

	Quality			Cardinality		
	min	max	gmean	min	max	gmean
Sequential	0.24	1.00	0.81	0.85	1.68	1.02
Lock-based	0.32	1.00	0.83	0.74	1.65	1.02
Resolution-based	0.25	0.99	0.74	0.77	1.78	0.99

The relative performance of an algorithm is computed by dividing its cardinality and quality scores to those of Gabow's quality matching algorithm. Table II shows the minimum, the maximum, and the geometric mean of all 289 relative performance for each algorithm. As the table shows, the sequential algorithm and its parallel lock-based variant are only 17–19% far from the optimal in terms of quality and almost equal in terms of cardinality. Considering the difference in computational complexities, we can argue that their relative performance is reasonably good. The lock-based algorithm performs slightly better than the sequential one. This demonstrates that while reducing the execution time, the proposed lock-based parallelization does not hamper the performance of the sequential matching algorithm in terms of both cardinality and quality. For this experiment, the parallel algorithms were executed with 8 threads.

Figure 1 shows the performance profiles generated to analyze the results in more detail. A point (x, y) in the profile graph means that with y probability, the quality of the matching found by an algorithm is more than max/x where max is the maximum quality for that instance. The figure shows that for this data set, the resolution-based algorithm performs worse than the other algorithms. The lock-based algorithm obtains matchings which are at most 1.25 times less than max with 74% probability. However, the resolution-based matching has this performance only for 45% probability. While obtaining matchings having at most 15% worse quality than the optimum, the probabilities are 56% and 28% for the lock- and resolution-based algorithms, respectively. Hence, the former performs two times better than the latter.

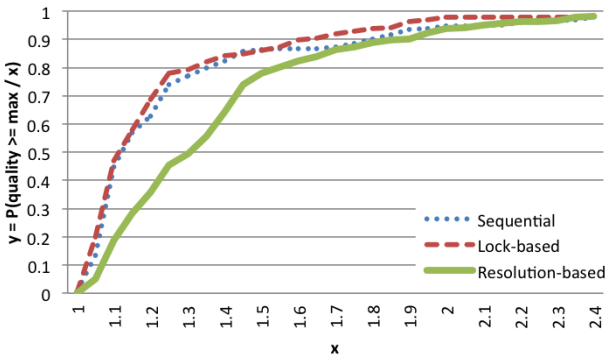


Figure 1. Performance profiles for sequential and multithreaded matching algorithms with respect to the maximum quality matchings. A point (x, y) in the profile graph means that with y probability, the quality of the matching found by an algorithm is larger than max/x where max is the maximum quality for that instance.

For the original data set with large hypergraphs, the relative performance of the multithreaded algorithms are given with respect to that of their sequential versions. Table III shows the statistics for this experiment.

Table III
RELATIVE PERFORMANCE OF THE PARALLEL MATCHING-BASED AND AGGLOMERATIVE CLUSTERING W.R.T. TO THEIR SEQUENTIAL VERSIONS (# OF THREADS = 8).

	Quality			Cardinality		
	min	max	gmean	min	max	gmean
Lock-based	0.79	1.1	0.99	0.99	1.01	1.01
Resolution-based	0.63	1.1	0.91	0.78	1.01	0.97
Agglomerative	0.56	1.3	1.01	0.94	1.03	0.99

Table III shows that the multithreaded versions of lock-based and agglomerative algorithms perform as good as their sequential versions in terms of cardinality and quality. Hence, once again, we can conclude that parallelization via locks does not diminish the performance of the multithreaded algorithms. On the other hand, the resolution-based algorithm is outperformed by other algorithms in terms of quality.

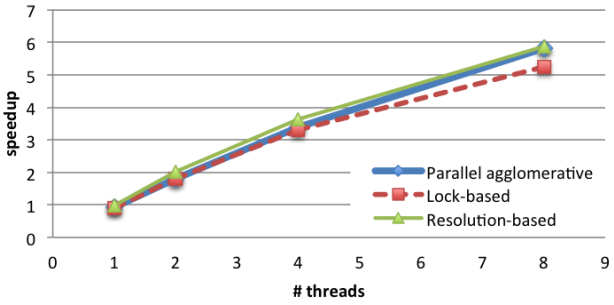


Figure 2. Speedups for matching and clustering: For the lock- and resolution-based algorithms, speedup is computed by using the execution times of sequential greedy matching algorithm. For the parallel agglomerative one, its sequential version is used.

Table IV shows the average numbers of matched vertices and conflicts for the proposed resolution-based algorithm with respect to the number of threads. To compute the averages, we execute the algorithm on each hypergraph ten times and report the geometric mean of these results. As expected, the number of the conflicts increases with the number of threads. However, when compared with the cardinality of the matching, the conflicts are at most 0.7% of the total match count for a single graph instance. This shows that the probability of a conflict is very low even with 8 threads.

Table IV
THE AVERAGE MATCHING CARDINALITY AND THE NUMBER OF CONFLICTS FOR THE PROPOSED RESOLUTION-BASED ALGORITHM WITH RESPECT TO THE NUMBER OF THREADS.

Thread #	$\#match$	$\#conflict$	$\frac{\#conflict}{\#match}$
1	290206.9	0.0	0.000000
2	290103.6	17.8	0.000061
4	290052.9	18.9	0.000065
8	289965.9	24.1	0.000083

2) Speedup: Figure 2 shows the speedups achieved by the multithreaded algorithms. On the average, the algorithms obtain decent speedups compared to their sequential versions up to 8 threads. In 8-thread experiments, the resolution-based algorithm has the highest speedup of 5.87, which is followed by the parallel agglomerative algorithm with a speedup of 5.82. The lock-based algorithm has the least speedup of 5.23 in this category. However, this is still decent especially when we consider the 5–7% overhead due to OpenMP and atomic operations.

To analyze the scalability of the algorithms from a closer point of view, we draw the speedup profiles in Figure 3. The resolution-based algorithm has better scalability in general. For example, with 4 threads, the probability that the resolution-based algorithm obtains a speedup of at least 3.2 is 83%. The same probabilities for the lock-based and parallel agglomerative algorithms are 54% and 65%, respectively. With 8 threads, the resolution-based version achieves at least 6.6 speedup for 33% the hypergraphs. However, the lock-based and parallel agglomerative algorithms achieve the same speedup only in for 16% and 20% of them. Hence, the resolution-based algorithm is the best among the ones proposed in this paper in terms of scalability.

B. Multi-level performance of the algorithms

As mentioned in the introduction, we integrated our algorithms into the coarsening phase of PaToH [15]. In this section, we first investigate how our algorithms scale for the clustering operations inside PaToH. The overall performance of a clustering algorithm in such a setting can be different from the standalone performance of the same algorithm, since in the multi-level framework, the hypergraphs are coarsened until the coarsest hypergraph is considerably small (for example, until the number vertices reduces below 100).

Figure 4 shows that the speedups on the multi-level clustering part are slightly worse than that of the standalone clustering. For example, the average speedups for the $8(4)$ threads case, are $5.25(3.22)$, $5.56(3.40)$, and $5.47(3.23)$ for the lock-based, resolution-based, and parallel agglomerative algorithms, respectively.

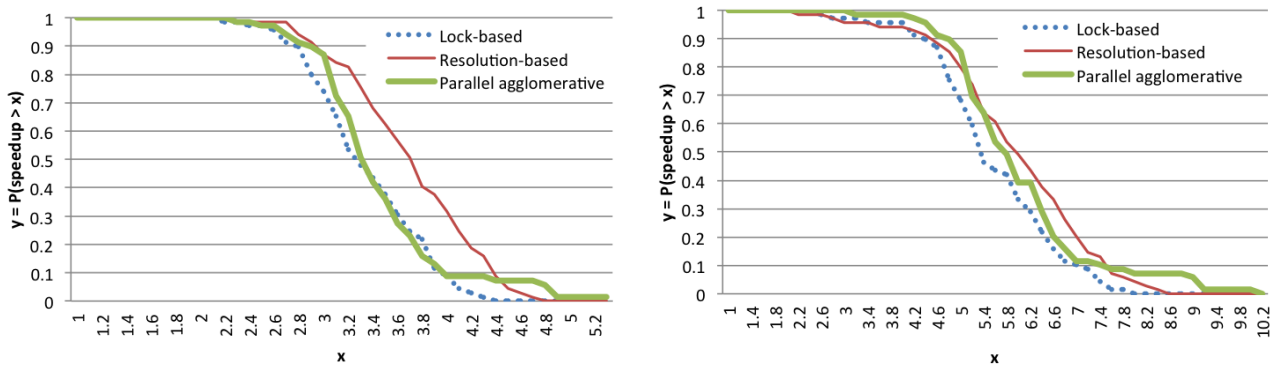
Since we only parallelize the clustering operations inside the partitioner, the speedups we obtain on the overall execution time cannot be equal to the number of threads, even in the ideal case. To find the ideal speedups, we use Amdahl's law [28]:

$$speedup_{ideal} = \frac{1}{(1-r) + \frac{r}{\#threads}} \tag{4}$$

where r is the ratio of the total clustering time to the total time of a sequential execution. To find the ideal speedup on the average, we compute (4) for each hypergraph and then take the geometric mean since we do the same for actual speedups.

Figure 5 shows the ideal and actual speedups of the multi-threaded algorithms. Since the ideal speedup lines (in dashed style) are drawn by using different sequential algorithms for the matching-based and agglomerative clustering, we separated these two cases and draw two different charts. On the average, all the algorithms obtain speedups close to the ideal. Among the matching-based algorithms, the lock-based one is more efficient since its speedup is closer to the ideal. This is interesting since according to Figure 4, it has less speedup on the total clustering time. At first sight, this looks like an anomaly because this is the only part that has been parallelized. However, the since lock-based algorithm's quality is better (Table III), there is probably less work remaining for the refinement heuristics in the uncoarsening phase. Hence, in total, one can achieve better speedup by using the lock-based algorithm rather than the resolution-based one.

We also obtain good speedups with the parallel agglomerative algorithm. For 2, 4, and 8 threads, the algorithm makes



(a) $\#threads = 4$ (b) $\#threads = 8$

Figure 3. Speedup profiles for the multithreaded algorithms: A point (x, y) in the profile graph means that with y probability, the speedup obtained by the parallel algorithm will be more than x.

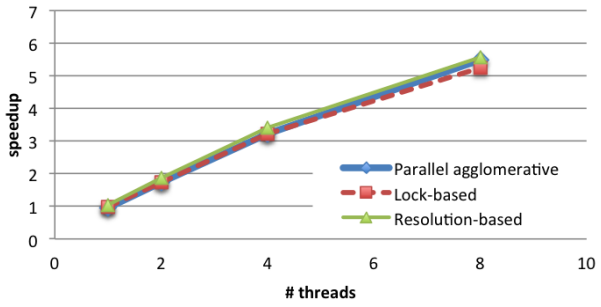


Figure 4. Speedups on the time spent by the clustering algorithms in the multi-level approach.

PaToH only 6%, 6% and 10% slower, respectively, than the best possible parallel execution time.

Parallelization of both matching-based and agglomerative clustering algorithm reduces the total execution time significantly. As mentioned before, matching-based algorithms are faster than the agglomerative ones. However, Figure 6 shows that PaToH is 20–30% faster when an agglomerative algorithm is used in the coarsening phase rather than a matching based one. According to our experiments, the coarsening phase is indeed 25% slower with the agglomerative clustering algorithm. However, the total execution time is 13% less. The difference comes from the reduction on the time of initial partitioning and uncoarsening/refinement phases. The initial partitioning takes less time because the coarsest hypergraph has fewer number of vertices with an agglomerative algorithm. In addition, the agglomerative clustering results in 25% less cutsize compared to the matching-based clustering. Hence, we can claim that it is more suitable for the cutsize definition given in (3).

When equipped with the multithreaded clustering algorithms, the cutsize of the partition found by PaToH is almost equal to the original cutsize obtained by using the sequential versions. For the agglomerative case, the cutsize changes only up to 1%. This is also true with the lock-based matching algorithm when compared with sequential greedy matching. For the resolution-based matching algorithm, there is at most 3% percent increase in the cutsize on average. On the other hand, as shown above the algorithms scale reasonably well.

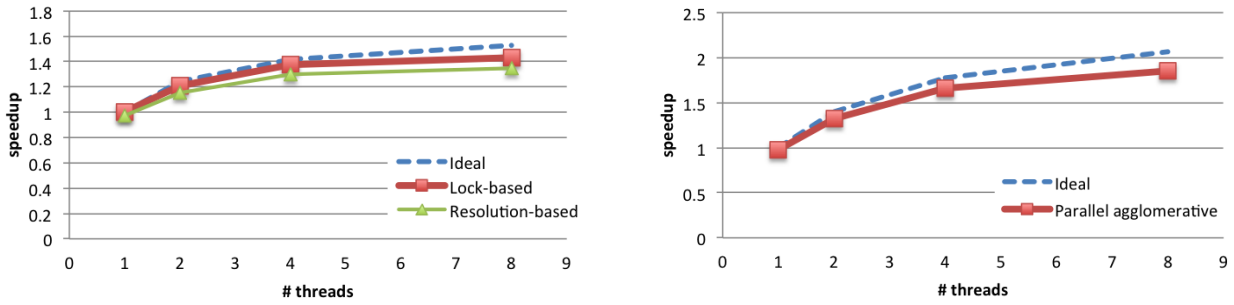
V. Conclusion

Clustering algorithms are the most time consuming part of the current-state-of-the-art hypergraph partitioning tools that follow the multi-level framework. We have investigated the matching-based and agglomerative clustering algorithms. We have argued that the matching-based clustering algorithms can be perceived as a matching algorithm on an implicitly represented undirected, edge weighted graph, whereas there is no immediate equivalent algorithm for the agglomerative ones.

We have proposed two different multithreaded implementations of the matching-based clustering algorithms. The first one uses atomic lock operations to prevent inconsistent matching decisions made by two different threads. The second one lets the threads perform matchings as they would do in a sequential setting and then later on resolves the conflicts that would arise. We have also proposed a multithreaded agglomerative clustering algorithm. This algorithm also uses locks to prevent conflicts.

We have presented different sets of experiments on a large number of hypergraphs. Our experiments have demonstrated that the multithreaded clustering algorithms perform almost as good as their sequential counter parts, sometimes even better in terms of clustering quality and cardinality. The experiments have also shown that our algorithms achieve decent speedups (the best was 5.87 with 8 threads).

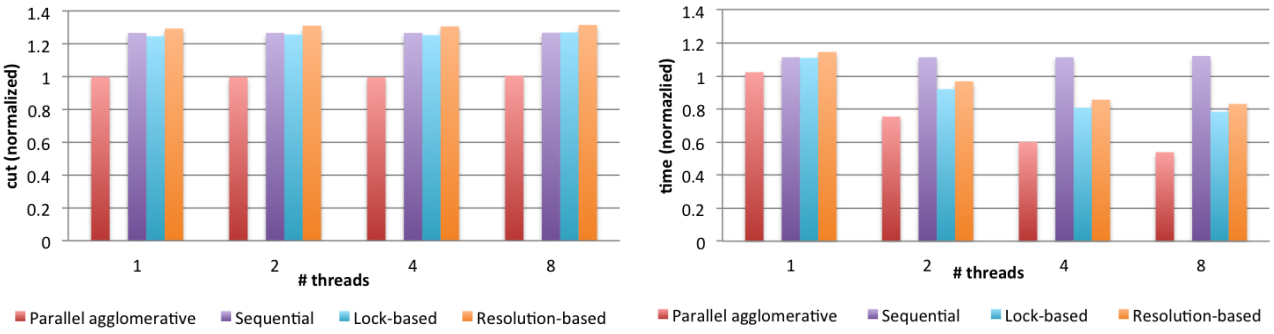
We integrated our algorithms to a well-known hypergraph partitioner PaToH. This integration makes PaToH 1.85 times faster where the ideal speedup is 2.07. In addition, it does not worsen the cutsizes obtained.



(a) Matching-based clustering

(b) Agglomerative clustering

Figure 5. Overall speedup on the total execution time of PaToH. The ideal speedup line is drawn by using Amdahl's law.



(a) Minimum cutsize found by PaToH

(b) Total execution time

Figure 6. The minimum cut and execution time of PaToH when equipped with the clustering algorithms in this paper. The numbers are normalized with respect to that of agglomerative clustering algorithm.

We observed that clusterings with better quality helps the partitioner to obtain better cuts. Fortunately, the multi-level framework may tolerate slower algorithms which generate better clusterings in terms of cardinality and quality. This is because of the fact that such clusterings will reduce the time required for the initial partitioning and uncoarsening phases. However, there is a limit with this tolerance. If the algorithm is too slow, one can execute the partitioner several times with a faster, parallelizable algorithm that generates clusterings with acceptable quality and achieve even better cutsizes.

Acknowledgment

This work was supported in parts by the DOE grant DE-FC02-06ER2775 and by the NSF grants CNS-0643969, OCI-0904809, and OCI-0904802.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] Ü. V. Çatalyürek, E. Boman, K. Devine, D. Bozdag, 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] R. H. Bisseling, J. Byrka, S. Cerav-Erbas, N. Gvozdenović, M. Lorenz, R. Pendavingh, C. Reeves, M. Röger, and A. Verhoeven, "Partitioning a call graph," in *Proceedings Study Group Mathematics with Industry 2005, Amsterdam*. CWI, Amsterdam, 2005.

[7] R. H. Bisseling and I. Flesch, "Mondriaan sparse matrix partitioning for attacking cryptosystems by a parallel block lanczos algorithm–A case study," *Parallel Computing*, vol. 32, no. 7–8, pp. 551–567, 2006.

[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. Ozdal 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 *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, 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 *Computer and Information Sciences - ISCIS 2004*, ser. Lecture Notes in Computer Science, C. Aykanat, T. Dayar, and I. Korpeoglu, Eds. 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] 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.

[17] B. Uçar, Ü. V. Çatalyürek, and C. Aykanat, "A matrix partitioning interface to PaToH in Matlab," *Parallel Computing*, vol. 36, no. 5–6, pp. 254–272, 2010.

[18] C. J. Alpert and A. B. Kahng, "Recent directions in netlist partitioning: A survey," *VLSI Journal*, vol. 19, no. 1–2, pp. 1–81, 1995.

[19] G. Karypis, "Multilevel hypergraph partitioning," University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, Tech. Rep. 02-25, 2002.

[20] K. Devine, E. Boman, R. Heaphy, R. Bisseling, and Ü. V. Çatalyürek, "Parallel hypergraph partitioning for scientific computing," in *Proceedings of 20th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2006.

[21] F. Manne and R. H. Bisseling, "A parallel approximation algorithm for the weighted maximum matching problem," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, K. Karczewski, J. Dongarra, and J. Wasniewski, Eds., vol. 4967. Springer-Verlag Berlin, 2008, pp. 708–717.

[22] Ü. V. Çatalyürek, F. Dobrian, A. Gebremedhin, M. Halappanavar, and A. Pothen, "Distributed-memory parallel algorithms for matching and coloring," in *2011 International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Parallel Computing and Optimization (PCO'11)*, 2011, pp. 1966–1975.

[23] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen, "Approximate weighted matching on emerging manycore and multithreaded architectures," *International Journal of High Performance Computing Applications*, 2011, under revision.

[24] M. M. A. Patwary, R. H. Bisseling, and F. Manne, "Parallel greedy graph matching using an edge partitioning approach," in *Proceedings of the fourth international workshop on High-level parallel programming and applications*, ser. HLPP '10. New York, NY, USA: ACM, 2010, pp. 45–54.

[25] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs", in *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006, pp. 10.

[26] C. Chevalier and I. Safro, "Comparison of Coarsening Schemes for Multilevel Graph Partitioning", in *Learning and Intelligent Optimization*, ser. Lecture Notes in Computer Science, T. Stützle, Ed., vol. 5851. Springer-Verlag Berlin, 2009, pp. 191–205.

[27] H. N. Gabow, "An efficient implementation of Edmonds' algorithm for maximum matching on graphs," *J. ACM*, vol. 23, pp. 221–234, April 1976.

[28] G. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in *Proceedings of AFIP'67*. ACM, 1967, pp. 483–485.