A Parallel Hill-Climbing Refinement Algorithm for Graph Partitioning

Dominique LaSalle and George Karypis
Department of Computer Science & Engineering,
University of Minnesota, Minneapolis, MN 55455, USA
{lasalle,karypis}@cs.umn.edu

***Abstract*—Graph partitioning is an important step in distributing workloads on parallel compute systems, sparse matrix reordering, and VLSI circuit design. Refinement algorithms are used to improve existing partitionings, and are traditionally difficult to parallelize. Existing parallel refinement algorithms make concessions in terms of quality in order to extract concurrency. In this work we present a new shared-memory parallel k-way method of refining an existing partitioning that can break out of local minima. This allows our method, unlike previous approaches, to both scale well in terms of the number of threads and produce clusterings of quality equal to serial methods. Our method achieves speedups of $5.7 - 16.7\times$ using 24 threads, while exhibiting only 0.52% higher edgecuts than when run serially. This is $6.3\times$ faster and 1.9% better quality than other parallel refinement methods which can break out of local minima.**

I. Introduction

Graph partitioning is used in a broad range of applications for decomposing sparse data into components with minimal interdependencies. Graph partitioning algorithms balance the vertices among partitions, and attempts to minimize the number of partition spanning edges. The quality of the partitioning plays a crucial role in the performance of applications. Increases in computer processing power have recently been achieved by increases in the number of processing cores per chip. This makes it imperative for graph partitioning algorithms to express a high degree of parallelism so as not to become bottlenecks in the applications which use them.

Modern solutions to the graph partitioning problem rely on the multilevel paradigm, which follows a simplify and conquer methodology. The graph is coarsened for several levels, a partitioning is found on the coarsest graph, and then is iteratively improved as the graph is uncoarsened. One of the major factors that determines the effectiveness of the multilevel paradigm is the refinement technique used to make local improvements to the partitioning as it is applied from the coarse levels to the fine levels.

Refinement algorithms that can break out of local minima in terms of the number of edges cut by a partitioning can lead to substantially better solutions than those algorithms that can not. The KL [18] and FM [9] refinement algorithms rely on a *move-and-revert* strategy in order to break out of these local minima. Vertices are speculatively moved when their movement increase the number of edges cut. If a partitioning with a lower edgecut is found after several moves it is saved. Otherwise, the partitioning is reverted back to the observed state with the lowest number of edges cut. However, if these speculative moves are not localized to the same part of the graph they become ineffective. Later methods such as CLIP/CDIP [7] and Multi-Try FM [23] addressed this issue by constraining these speculative moves to be adjacent. These move-and-revert refinement algorithms are inherently serial due to their need to evaluate and possibly save the partitioning's state after each move. Algorithms like k-way Pairwise FM [10] attempt to parallelize these schemes by using two-way refinement between pairs of partitions concurrently. This however can be time intensive and has limited parallelism.

In this paper we propose the Hill-Scanning algorithm, a new shared memory parallel refinement algorithm capable of breaking out of local minima. Unlike its predecessors, this new method exhibits a high degree of paralellism and as a result has a substantially lower runtime. We show that our method runs in $O(kn/p + (m/p)\log n)$ time, where k is the number of partitions, n is the number of vertices, m is the number of edges and p is the number of threads. In the serial environment, we show that Hill-Scanning produces solutions of equal quality to Multi-Try FM. We show that Hill-Scanning is well suited for multicore architectures, and is up to $6.3\times$ faster than other parallel refinement methods with hill-climbing capabilities when using 24 threads. Hill-Scanning produces solutions with 6.3% lower edgecut than parallel Greedy refinement and 1.9% lower edgecut than parallel Pairwise FM. We present strong scaling results with up to 24 threads and show that Hill-Scanning achieves speedups of $5.7 - 16.7\times$, while exhibiting only a 0.52% increase in edgecuts.

The rest of this paper is organized as follows. In Section II we define the graph partitioning problem and the notation used throughout this paper. In Section III provides an overview of the multilevel paradigm and graph partitioning concepts. Section IV reviews the relevant prior work. Section V presents the new Hill-Scanning algorithm. The conditions of the experiments are detailed Section VI. This is followed by the results of the experiments in Section VII. Finally, the paper is concluded in Section VIII.

II. Definitions & Notation

The graph partitioning problem takes as input a simple undirected graph $G = (V, E)$, consisting of a set of vertices V, and a set of edges E. Each edge is composed of an unordered

pair of vertices (i.e., $v, u \in V$). We use $n = |V|$ to denote the number of vertices, and $m = |E|$ to refer to the number of edges.

The objective of graph partitioning is to create k disjoint subsets of vertices (partitions), $V = V_1 \cup \ldots \cup V_k$, that minimize the number of edges between vertex sets. The total weight of the these partition spanning edges is referred to as the *edgecut*:

$$edgecut = \sum_{i=1}^{k} \sum_{v \in V_i} \sum_{u \in \Gamma(v), u \notin V_i} \theta\{v, u\}.$$

In this work, we are concerned with the balanced graph partitioning problem, which means the size (weight) of each partition is bounded by a balance constraint ϵ. That is,

$$k \frac{\max_i |V_i|}{|V|} \leq 1 + \epsilon.$$

In order to facilitate discussions about the effects of moving vertices, let $d_{int}(v)$ denote the sum of the weight of edges connecting v to the partition in which it resides (internal degree). Let $d_{ext}(v)$ denote the sum of the weight of edges connecting v to partitions other than the one in which it resides (external degree). Let $d_A(v)$ denote the sum of the weight of edges connecting the vertex v to the partition A. Finally, let $\Delta(v)$ denote the number of external partitions to which v is connected.

III. Background

The graph partition problem is known to be NP-Hard [2], and subsequently many advanced heuristics have been developed.

A. Multilevel Graph Partitioning

Based on multigrid solvers [1], multilevel methods for graph partitioning have become the de facto standard for developing high performance and high quality methods [11], [16], [21], [23]. The multilevel paradigm for graph partitioning consists of three phases. In the coarsening phase a series of coarser (smaller) approximations of the original graph G_0 are found, $G_1, \ldots, G_s$. Then, in the initial partitioning phase, a partitioning is found for the coarsest graph G_s. In the uncoarsening phase, the partitioning is projected back down through the series graphs, and is improved on each graph as the degrees of freedom for improving the solution increase with the number of vertices. This process is shown in Figure 1. In this solution, the best solution on the coarsest graph has an edgecut of six. During uncoarsening it can be improved to five, and then to three on the original graph. Buluç et al. [3] recently provided an overview of modern approaches to the graph partitioning problem.

The first phase, coarsening, captures much of the connectivity information of the graph by collapsing well connected vertices together. As vertices and edges are combined during coarsening, their weights are updated accordingly such that a balanced partitioning of the coarsest graph is a balanced partitioning of the finest graph with the same edgecut. The fastest methods for generating the coarser graphs are often based on Heavy Edge Matching [16], which uses only the edge weight for prioritizing vertices to collapse together. Advanced techniques have been proposed that use more information when deciding which vertices to collapse [22], and offer improved quality at the cost of runtime. As a result, relatively simple methods can be used during the initial partitioning phase. The most common approach is to induce a bisection via a truncated breadth first search and then apply refinement to the bisection [15]. This technique is then used recursively to derive the k-way partitioning. The last phase, uncoarsening, is made up of two alternating steps: projection and refinement. In projection the partition label of each coarse vertex is applied to the fine vertices that compose it. As no vertices are moved between partition during projection, the edgecut and the partition weights remain unchanged. It is during refinement that vertices are moved, resulting in changes to the edgecut and the partition weights. This can have a dramatic effect on the edgecut of a partitioning. Section III-C describes current state of the art refinement methods.

B. Recursive Bisection

In order to use techniques that create two-way partitionings (bisections) to generate k-way partitionings, a process known as *recursive bisection* is used. In this process, the graph is first partitioned into two halves, G^1 and G^2. If k is equal to two, then partitioning stops and the two partitions are returned. If k is greater than two, then the process recurses, generating a $\lceil k/2 \rceil$-way partitioning of G^1 and $\lfloor k/2 \rfloor$-way partitioning of G^2. In all, $k - 1$ bisections are made. In order to still achieve balance among all k partitions, the target partition weights at each bisection must be based on $\eta(V)\lceil k/2 \rceil$ and $\eta(V)\lfloor k/2 \rfloor$ respectively. When comparing near-linear time partitioning algorithms, methods using recursive bisection require roughly $\log(k)$ times more work to create a k-way partitioning than methods that directly create a k-way partitioning.

C. Refinement

Refinement techniques range from light weight greedy approachesto more intensive flow-based techniques [24]. While the multilevel paradigm has been shown to produce solutions of good quality as result of the contracted edge weight [14], refinement techniques capable of breaking out of local minima offer a means to explore a wider range of solutions and improve quality.

The Greedy [16] algorithm, is an extremely fast method for converging on a local minima in the edgecut of a k-way partitioning. The Greedy algorithm works by making several iterations over the vertices located on partition boundaries until no improvement is made in an iteration, or a maximum number of iterations has been performed. In each iteration, vertices are moved individually in descending order of gain (reduction in edgecut) until, with the restriction that each vertex can move only once per iteration, there are no more positive gain moves to be made. This method of converging on a local minima of the edgecut has been successfully parallelized, both on shared-memory [19] and distributed-memory [17] architectures.

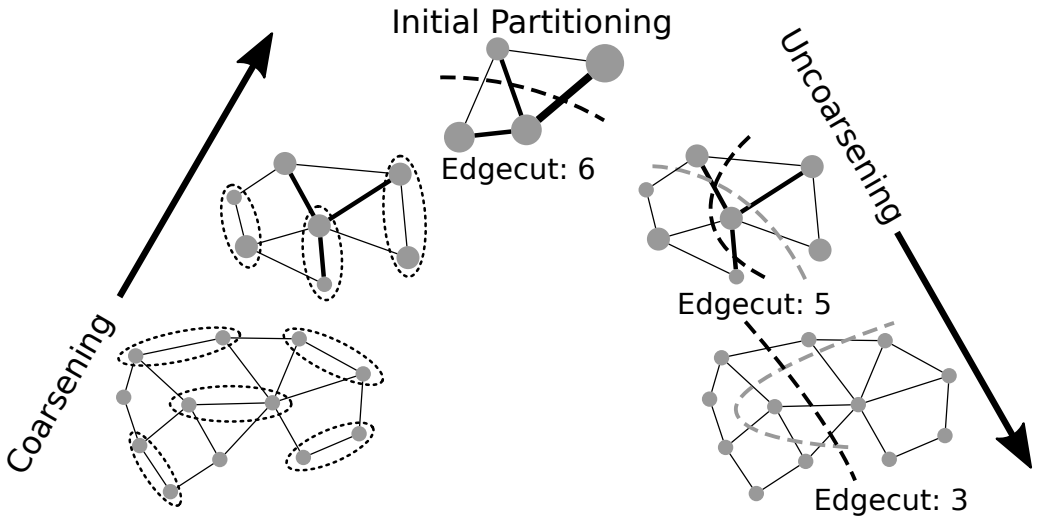

Fig. 1: The multilevel paradigm as used for graph partitioning. The coarsening phase is shown on the left where pairs of vertices are matched and then contracted together to form the next coarser graph. The initial partition phase is at the top, where a partitioning for the coarsest graph is found. The uncoarsening phase is on the right were we project our initial solution to the coarser graphs, and use refinement to improve it for each graph.

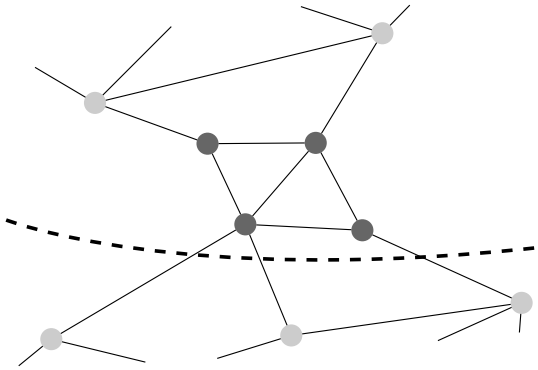

Fig. 2: A group of size four (the darker shaded vertices) corresponding to a hill in the objective state. Moving all four vertices across the partition boundary (dashed line) would result in a net decrease in edgecut, but moving only a subset of the group would result in a net increase.

Further decreasing the edgecut beyond a local minima, requires moving more than one vertex. These groups of vertices whose movement presents a net decrease in the edgecut form *hills* in the objective state. Moving this group of vertices results in an upwards slope in the edgecut objective followed by a downward slope. These groups of vertices which correspond to hills, have a large number of edges interconnecting them. The sum of the weights of the edges connecting a vertex in this group to other vertices in the group is higher than the sum of the weights of the edges connecting the vertex to vertices outside the group in any one partition. Thus moving any single member of the group will result in an increase in edgecut, but moving the group as a whole may result in a net decrease in edgecut. Figure 2 shows a hill composed of the four darker shaded vertices, whose movement across the partition boundary would result in a net edgecut decrease. The process of speculatively moving individual vertices so as to effectively move these groups of is referred to as *hill-climbing*. The capability to hill-climb defines a class of high quality refinement techniques, and is the focus of this work.

IV. Related Work

One of the earliest methods for refining a two-way partitioning is that of the Kernighan-Lin refinement algorithm [18]. It works by going through the vertices and identifies the most beneficial pairs of vertices to swap between partitions. It does this continually until all vertices have been swapped. It then reverts back to the best state of the partitioning that was observed while performing these swaps. This process repeats until no improved states are found. In its original form this method has an $O(n^2 \log n)$ runtime.

Fiduccia and Mattheyses [9] improved upon this method by relaxing the balance constraint and moving vertices one at a time instead of swapping pairs. At each step, the vertex whose movement between partitions would result in the largest decrease in edgecut and still result in a balanced partitioning is found. This is accomplished by using of two priority queues to identify the vertex of maximum gain (largest decrease in edgecut) in each partition. If a vertex from either partition can be moved without violating the balance constraint, then the priority queue with the highest gain vertex is selected. Otherwise, the priority queue for moving vertices to the lower weight partition is selected. Again, all possible moves are made before the algorithm reverts back to the best observed state. For graphs with non-uniform edge weights, this algorithm runs in $O(m \log n)$ time ($O(m)$ time for uniform edge weights using a special data structure for tracking the move of the most gain).

In order to refine a k-way partitioning, Gong and Lim [10] introduced k-way Pairwise FM (KPM) refinement, which identifies independent pairs of partitions and performs FM on these pairs. Then, a new set of independent pairs is selected and refined. This repeats until all partition boundaries have had refinement applied. Because this method directly refines a k-way partitioning, it can lead to partitionings with lower edgecut than using FM via recursive bisection. Parallelism can be extracted by refining these independent pairs of partitions concurrently. However, there are $k(k-1)/2$ possible partition boundaries to refine, which can make this a costly method for large numbers of partitions.

One undesirable side effect of selecting the vertex of maximum gain at each step is that for two vertices moved in sequence when refining large graphs it is unlikely that those two vertices will have common neighbors. While this still works for vertices whose movement results in decreases of edgecut, for speculative vertex moves (which increase the edgecut), it interferes with finding new local minima in terms of edgecut. This is because if a group of tightly connected vertices is moved across the partition boundary and vertices outside of this group have been moved speculatively, the net edgecut decreasing resulting from the group being moved across the boundary may be hidden by edgecut increase of the other speculative moves, causing the algorithm to not a find a state of lower edgecut.

To address this issue, Dutt and Deng [7] proposed the CLIP/CDIP variants of FM. After all vertices have been inserted into the priority queue, the vertices have their priority set to zero while preserving the order in the queue. Then, the top vertex v is extracted and moved, and all of its neighbors have their priorities updated. This results in the priorities of all the neighbors of v being set equal to change in gain resulting from v's movement. The neighbors of v in v's new partition receive negative priorities as they now have one less edge connecting them to the opposing partition. The neighbors of v in v's original partition receive positive priorities as they now have one more edge connecting them to the opposing partition. The vertices remaining in the priority queue are then moved as in FM. This ensures that the subsequent moves are neighbors of v from the same partition, and makes it more likely that a group of vertices corresponding to a hill will be moved.

To use to FM to refine k-way partitionings and enforce the localized search pattern of CLIP/CDIP, Sanders and Schulz [23] introduced a variant of FM named Multi-Try FM. It uses multiple small trials of FM per iteration. A trial consists of inserting a random boundary vertex, the seed vertex, into a priority queue. Once this vertex is extracted and moved, its neighbors from its original partition are added to the priority queue. The trial then continues to extract and move vertices from the priority queue to the same partition to which the seed vertex was moved. Vertices are added to the priority queue when their neighbors are moved. The trial stops when the priority queue is empty or the balance constraint would be violated by further moves. The partitioning is then reverted back to the best state observed during the trial. Once the trial terminates, a new seed vertex is selected and this process continues until all boundary vertices in the graph have been visited. The complexity of this algorithm is kept to $O(m \log n)$ by marking vertices as visited when they enter the priority queue, and preventing them from re-entering it until the next iteration. To further reduce the runtime, an early exit can be taken from each trial if after a certain number of speculative moves no new minima is found.

V. Hill-Scanning Refinement

In this section we present our new algorithm for refining k-way partitions in parallel. Our algorithm, named Hill-Scanning, fills the gap left between serial refinement algorithms capable of breaking out of local minima, and parallel greedy algorithms capable of fully utilizing multicore systems.

The Hill-Scanning algorithm is based on the observation that the move-and-revert strategy used by FM and its related algorithms break out of local minima by pulling the tightly connected groups of vertices which correspond to hills in the objective state across partition boundaries. However, this approach has two failings. First, it is inherently serial, as a linear ordering of moves is needed in order accurately track edgecut and revert partitioning states. Second, each group of vertices is only considered for moving to a single partition rather than to the partition of a largest net gain. That is, it may be the case that while the first vertex in the group is connected to partition i with the majority of its edge weight, the group as a whole may be connected to partition j with a larger total edge weight.

The Hill-Scanning algorithm, at a very high level, works by identifying these groups of vertices which correspond to hills in the objective state prior to moving them. Once a group is identified, it is moved to the partition of maximum gain for the group as a whole. On a shared memory system where all processing elements have access to all of the data, we can efficiently identify these groups of vertices in parallel.

A. Two-Way Hill-Scanning

The Hill-Scanning (HS) algorithm for use in refining a two-way partitioning is outlined in Algorithm 1. It takes three input arguments, the graph G, the current partitioning P, and the maximum size of a group of vertices ϕ to identify as corresponding to a hill in the objective state.

Each iteration works as follows. First, all of the boundary vertices, those with edges connecting them to the opposing partition, are inserted into a priority queue. The gain associated with moving a vertex is used as the priority. This gain for the vertex v is the sum of the weight of the edges connecting it to the opposing partition minus the sum of the weight of the edges connecting it to the partition in which it resides:

$$gain = d_{ext}(v) - d_{int}(v).$$

Vertices are extracted from this queue and are considered for moving. If the gain associated with moving a vertex v is positive, and moving v would not violate the balance constraint, v is moved to the opposing partition, and its

Algorithm 1 Hill-Scanning Refinement

```
1: function HILLSCAN(G,P,φ)
2:     q ← priority queue
3:     repeat
4:         insert boundary vertices into q
5:         while |q| > 0 do
6:             v ← pop(q)
7:             if positive gain for v then
8:                 move v
9:             else
10:                h ←FindHill(v,G,P,φ)
11:                if h ≠ ∅ then
12:                    move h
13:                end if
14:            end if
15:        end while
16:    until no vertices are moved
17:    return P
18: end function
```

Algorithm 2 Hill Finding

```
1: function FINDHILL(v,G,P,φ)
2:     h ← ∅
3:     q ← priority queue
4:     insert v into q
5:     while |q| > 0 and |h| < φ do
6:         u ← pop(q)
7:         h ← h ∪ {u}
8:         if moving h is beneficial then
9:             break
10:        end if
11:        Add all u ∈ Γ(u), ∈ A to q
12:    end while
13:    if moving h is beneficial then
14:        return h
15:    else
16:        return ∅
17:    end if
18: end function
```

neighbors are updated in the priority queue. Vertices with zero gain will still be moved if it improves the balance of the partitioning.

If the gain associated with moving a vertex v is not positive, we attempt to build a vertex group rooted at v. If we identify a group rooted v whose movement would result in a positive gain, we move the group to the opposing partition. If a vertex is moved by itself or as part of a group, it is locked in place and prevented from moving for the rest of the iteration.

The hill building function used by Algorithm 1 is shown in Algorithm 2. This function is what separates the Hill-Scanning algorithm from the Greedy algorithm. How far we explore looking for a group of vertices corresponding to a hill, the maximum group size ϕ, determines the trade-off between runtime and quality. Because our algorithm is for use in the multilevel setting, we can use a relatively small value for ϕ, based on the intuition that very large groups of vertices will likely have been moved during a coarser round of refinement. We have found $\phi = 16$ provides a good balance of runtime and quality, as shown in Section VII.

The function `FindHill` starts by initializing an empty group, and inserting the root vertex v into the priority queue. The group is then grown by extracting vertices from the priority queue and adding them to the group. If the gain associated with moving the entire group is positive, the loop exits and the group is returned. Otherwise, the neighbors of u are added to the priority queue and the search continues. If the group reaches the maximum allowable size and would not result in positive gain if moved, it is discarded and an empty set is returned.

To keep the runtime down, each time an edge is traversed when attempt to find group corresponding to a hill, it is marked as *traveled* for that direction. During each iteration, an edge will be traversed at most once in each direction. This prevents vertices from being repeatedly inserted into the priority queue as groups are discarded. Furthermore, we observed that groups built earlier in a refinement pass were far more likely to be moved than those later in the pass. As such, we add an early exit when $\sqrt{b(V)}$ groups have been built and discarded, where $b(V)$ is the number of vertices on the boundary (i.e., vertices with edges connecting them to vertices in the opposing partition).

In the two-way setting, our Hill-Scanning algorithm is functionally similar to CLIP/CDIP [7] speculatively moves vertices in a localized area. However, because Hill-Scanning identifies hills before moving them, we can extend it to the k-way and parallel settings.

B. k-Way Hill-Scanning

As with the two-way version of the algorithm, in the k-way version we insert all boundary vertices into a priority queue. To accurately order vertices in the priority queue based on their gain, we would need to track:

$$gain = \max_{P_i \in P'} d_{P_i}(v) - d_{int}(v),$$

where P' is the set of partitions that for which moving v to would not violate the balance constraint. This however would require updating vertex priorities frequently as partition weights and vertex connectivity change.

Instead we use the approximate gain associated with moving the vertex v out of its partition as the priority:

$$priority = \frac{d_{ext}(v)}{\sqrt{\Delta(v)}} - d_{int}(v),$$

where $\Delta(v)$ is the number of external partitions to which v is connected. For vertices that are only connected to a single external partition, this accurately models their priority. For vertices connected to more than one external partition, this favors vertices connected to fewer partitions while not over penalizing vertices connected to too many partitions.

In the k-way setting we need to build the hill before deciding to which partition it will be moved. This requires several algorithmic changes the `FindHill` function in Algorithm 2.

We can no longer model the gain associated with a hill as the sum of the external edge weights minus the sum of the internal edge weights, as the external edge weights can be split among multiple partitions. To accurately model the gain associated with a hill h in partition A as we build it, we keep a vector W of length k, which stores the connectivity of the hill to all partitions. Each time we add a vertex v to the hill, we scan its adjacency list, adding the weight of the edges to the corresponding entries in W. For edges connecting v to the hill, instead of adding the entry for A in W, we subtract the weight from it. Thus, after adding each vertex the gain associated with moving the hill to partition B is $W_B - W_A$.

While Multi-Try FM [23] is applicable to k-way partitionings, once it moves its seed vertex for a trial, it moves all subsequent vertices in that trial to the same partition. This can lead to hills being moved to a partition of lesser gain, or not moved at all. This can cause Multi-Try FM to require more iterations to migrate the hill to its most desirable partition. Because HS identifies a hill before it decides where to move it, it ensures the hill will be moved to the partition of maximum gain.

C. Parallel k-Way Hill-Scanning

The move-and-revert strategy of KL/FM like algorithms is difficult to parallelize due to the need of a serialized order of moves. While methods have been proposed for running FM on independent subgraphs [20], these require some degree of pre-partitioning.

Because Hill-Scanning does not use a move-and-revert strategy, we can use coarse grained parallelism. The movement of vertices in Hill-Scanning is similar to that in Greedy refinement, so we model the parallelization of Hill-Scanning after the method [19] for parallelizing Greedy refinement on shared-memory architectures, which operates as follows.

Each refinement iteration is split into two phases: upstream and downstream. At the start of each iteration, we assign each partition a random unique integer label. These labels are then used to induce an acyclic flow on the partition-graph (a graph in which each partition is represented by a single vertex). During the upstream phase, vertices are only considered for moving to partitions with higher labels than the label of the partition in which they currently reside. In the downstream phase, vertices are only considered for moving to partitions with lower labels. Threads insert the vertices they own into local priority queues. They then proceed to extract and move vertices from their priority queues in the same manner as the serial version of the algorithm. When building a hill, threads may select vertices owned by other threads. Updated information regarding the state of the partitioning and vertex locations are communicated asynchronously between threads via message queues. Once all threads have emptied their priority queues, they synchronize and begin the next phase of the refinement iteration.

We do not use any thread synchronization primitives when building hills, and as a result it is possible for two or more threads to concurrently build overlapping hills. This may cause the overlapping hills to be moved to separate partitions, and possibly increase the edgecut. However, this race condition occurs rarely, and when it does, the misplaced hill segment will be moved to its correct location in the next iteration.

D. Complexity

In this section analyze the complexity of Hill-Scanning refinement. We use k as the number of partitions, p as the number of threads, n as the number of vertices, and m as the number of edges. We start by establishing the complexity of two-way Hill-Scanning, and work our way up to showing that the full parallel k-way version of our algorithm runs in $O(kn/p + (m/p)\log n)$ time. We use a priority queue model which has $O(\log q)$ complexity for insert, update, and extract operations [5], where q is the number of elements in the priority queue.

1) Complexity of 2-Way Hill-Scanning: In two-way hill-scanning for a graph with n vertices and m edges, we will insert and extract up to $O(n)$ vertices into/from the priority queue. Because we lock each vertex after moving it, we move at most $O(n)$ vertices and perform at most $O(m)$ neighbor updates from these moves. We can then say the cost of moving vertices and selecting single vertices to move is $O(m \log n)$.

When we build a hill, we mark each edge as traveled, for each direction. This means that each edge may be traversed at most twice during an iteration, and thus the number of insertions and updates to the priority queue is bounded by the number of edges. As the priority queue can have at most all of the vertices in the graph in it, the cost building hills is $O(m \log n)$. Combining this with the cost of selecting and moving vertices, we see that the total complexity of HS is $O(m \log n)$ per iteration.

2) Complexity of k-Way Hill-Scanning: In terms of complexity, k-way Hill-Scanning differs from two-way hill-scanning in two places: determining the best partition to which move a vertex, and building hills. When determining which partition to move a vertex (or a hill), we can consider at most k partitions. This add an additional $O(kn)$ term to the complexity.

When building hills in k-way Hill-Scanning, can at most perform an operation on the vector W once per edge, adding an addition $O(m)$ term to the complexity. This is hidden by the larger terms we derived in Section V-D1 for operations on the priority queues. Thus, combining the cost of considering what partition a vertex/hill should be moved, to that of the cost of the priority queues, the cost of k-way Hill-Scanning becomes $O(kn + m \log n)$ per iteration.

3) Complexity of Parallel k-Way Hill-Scanning: We assign n/p vertices and their incident edges to each thread, and for the rest of this analysis we assume the assigned incident edges per thread is m/p. Thus, the maximum number of vertices in a given thread's priority queue at one time is n/p, making the cost of performing an update as $O(\log(n/p))$. As each thread

has m/p edges associated with its n/p vertices, each thread will need to update its vertices at most m/p times. This gives a complexity of $O(kn/p + (m/p)\log(n/p))$ for selecting and moving vertices.

We know the total cost of building hills is bounded by $O(m \log n)$, due to the marking of edges as traveled, as derived in Section V-D1. While threads can traverse edges and visit vertices other than their own while building hills, two threads cannot traverse the same edge in the same direction. The means we have at most m edge traversals split among p threads. This gives hill building a complexity of $O((m/p)\log n)$, which is larger than the $O((m/p)\log(n/p))$ term for selecting and moving vertices. Finally, parallel k-way Hill-Scanning has a complexity of $O(kn/p+(m/p)\log n)$ per iteration.

VI. Experimental Setup

A. Data

Table I shows the 30 graphs and their sizes used Section VII. These are undirected and unweighted graphs. The first set of graphs (wing through auto) are all graphs with greater than $100,000$ edges from the Graph Partitioning Archive [26]. The second set of graphs are the non-zero patterns of some of the largest matrices from the University of Florida Sparse Matrix Collection [6].

B. System Configuration

These experiments were run on a machine with $2\times$ 12 core Xeon E5-2680v3 @ 2.5GHz processors and 64GB of memory. The operating system was CentOS 6.6, running the Linux kernel version 2.6.32. The code was compiled using GCC 4.9.2.

C. Implementation

We implemented HS, and the other refinement algorithms detailed below in the mt-Metis multithreaded graph partitioning framework. The version used for these experiments is mt-Metis 0.4. The matching scheme used is *Heavy Edge Matching* [15]. Each refinement scheme terminates when an iteration completes without any moves, or a maximum of 8 iterations have been performed. Each of the five refinement schemes evaluated in the follow experiments are implemented as detailed below.

1) Greedy: We used the existing implementation of parallel Greedy [19] refinement in mt-Metis for these experiments.

2) RB-FM: We implemented a boundary-only variant of the Fiduccia-Mattheyses [9] algorithm for use in recursive bisection (RB-FM). Due to FM's serial nature, only a single thread is used for refining each bisection. After each bisection, the threads split into two groups, to perform the remaining bisections on each half of the graph. We used a limit of 100 vertices being moved without gain before FM reverts back to the minimum bisection.

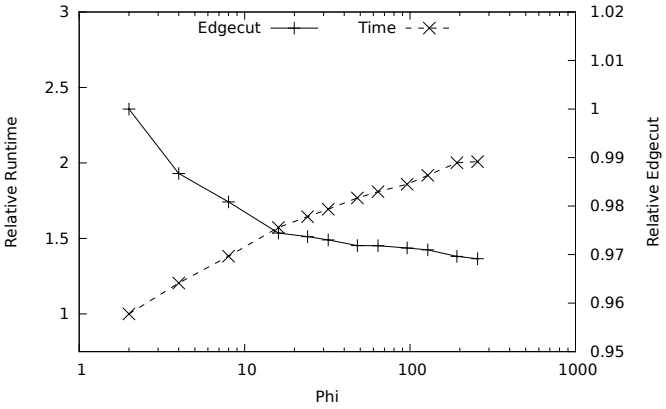

Fig. 3: Effects of varying ϕ on Flan_1565

3) KPM: For our implementation k-way pairwise FM (KPM), we split the 8 iterations into two local iterations and four global iterations. That is, each time a pair of partitions is visited, two iterations of FM are performed on the local bisection, and all partition pairs are cycled through four times. Parallelism is expressed by refining pairs of partition in parallel in each global iteration, which has a maximum concurrency of $k/2$. We used a limit of 32 vertices being moved without gain before FM reverts back to the minimum bisection between a pair of partitions.

4) MTFM: Because Multi-Try FM [23] (MTFM) is a serial algorithm, it is only included in our first set of experiments. To keep the runtime down, we used a limit of 16 vertices being moved without finding a new minima before the seeded FM reverts back to the best state, and the next seed vertex is selected.

5) HS: We implemented the parallel k-way version of the Hill-Scanning (HS) algorithm we detailed Section V-C. For the k-way and parallel experiments, we used a maximum group size, ϕ of 16.

VII. Results

First, we examine the effect of varying the different parameters of the Hill-Scanning (HS) algorithm on runtime and quality in Section VII-A. This is followed by a comparison of HS with Greedy, FM, and KPM refinement schemes in Section VII-B. Finally, in Section VII-C, we perform strong scaling experiments to examine the effects of parallelization on runtime and quality of HS.

A. Maximum Hill Size

In Figure 3, we present the effects of varying ϕ, the maximum size of a group to move, on the quality and runtime of refinement for the graph Flan_1565. As the group size increases, more vertices are considered for moving at once, and thus the runtime increases steadily. Conversely, the larger group size allows the algorithm more groups of vertices to move, and a further reduction in the edgecut. However, at a group size of 16, the rate of reduction in edgecut decreases as the larger groups are less likely to be moved compared to

TABLE I: Graphs used for experiments

Graph [26]	Vertices	Edges	Graph	Vertices	Edges
t60k	60,005	89,440	wing	62,032	121,544
fe_pwt	36,519	144,794	fe_body	45,087	163,734
vibrobox	12,328	165,250	finan512	74,752	261,120
bcsstk33	8,738	291,583	bcsstk29	13,992	302,748
brack2	62,631	366,559	fe_ocean	143,437	409,593
fe_tooth	78,136	452,591	bcsstk31	35,588	572,914
fe_rotor	99,617	66,2431	598a	110,971	741,934
bcsstk32	44,609	985,046	bcsstk30	28,924	1,007,284
wave	156,317	1,059,331	144	144,649	1,074,393
m14b	214,765	1,679,018	auto	448,695	3,314,611
AS365 [4]	3,799,275	11,368,076	NLR [4]	4,163,763	12,487,976
adaptive [12]	6,815,744	13,624,320	ldoor [6]	952,203	22,785,136
Serena [13]	1,391,349	31,570,176	audikw1 [6]	943,695	38,354,076
channel-500x. [27]	4,802,000	42,681,372	dielFilterV3. [8]	1,102,824	44,101,598
Flan_1565 [13]	1,564,794	57,920,625	nlpkkt240 [25]	27,994,600	373,239,376

smaller ones. Due to this, we use a value of 16 for ϕ for the remainder of the experiments presented here as it gives a good balance of both quality and speed.

B. Serial Algorithm Performance

The performance of the different refinement schemes, run serially, is compared in Table II. The results presented are the geometric mean of 25 runs. Runtime includes the entire multilevel process: coarsening, initial partitioning, and uncoarsening. We present the entire time rather just the time for refinement in order to include the overhead associated with the recursive bisection required RB-FM in our comparison, as well as to place the runtime cost of the refinement schemes in perspective.

As expected, Greedy refinement is the fastest method, but also results in the worst quality (highest edgecuts). It is faster than the other methods not only because it does fewer calculation per vertex moved, but also because it tends to move fewer vertices. The HS algorithm was the second fastest method, and the multilevel process using HS took only 27.1% longer than using Greedy in this serial setting. However, among the refinement schemes, HS resulted in the lowest mean edgecut and had the smallest mean edgecut on 14 of the 30 graphs. MTFM had a geometric mean edgecut for the 30 graphs 1% higher than HS, and had the smallest mean edgecut on three of the graphs. Both HS and MTFM focus on making localized k-way moves, which is why their behavior when run serially is similar. However, HS, unlike MTFM, can be effectively parallelized. In Section VII-C we show that this dramatically reduces the runtime required and makes HS well suited for current architecture trends of increasing parallelism.

RB-FM did well on the smaller graphs, averaging the lowest edgecut on two of the graphs. The fact that optimal bisections applied recursively do not correspond to an optimal k-way cut played less of a role on these smaller graphs. KPM found solutions of similar quality to HS and MTFM, and had the lowest mean edgecut for eleven of the 30 graphs, most of which were the larger graphs. This is because on the larger graphs, more vertices could be moved between a pair of partitions without violating the balance constraint. KPM however, was also the slowest method, especially on the larger graphs. This high runtime is the result of running FM on each connected pair of partitions, which on for partitionings with relatively dense partition connectivity can be exceedingly expensive.

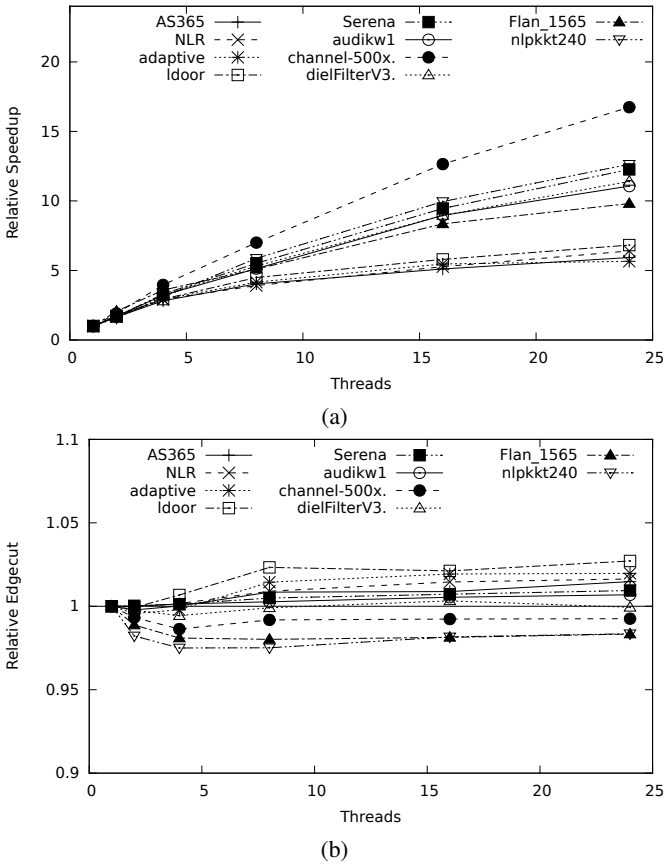

Fig. 4: Strong scaling of Hill-Scanning: (a) relative speedup, (b) relative edgecut.

C. Parallel Algorithm Performance

Figure 4a shows the scaling of the HS algorithm with respect to the number of threads (strong scaling). Here we are measuring only the time spent in refinement, and not the time

TABLE II: Edgecut and serial runtimes for 64-way partitionings with a 0.03 balance constraint.

	Greedy		RB-FM		KPM		MTFM		HS	
Graph	Edgecut	Time (s)	Edgecut	Time (s)	Edgecut	Time (s)	Edgecut	Time (s)	Edgecut	Time (s)
t60k	2,565	0.085	2,433	0.202	**2,378**	0.481	2,445	0.108	2,401	0.109
wing	9,727	0.146	9,074	0.309	8,783	1.351	8,772	0.248	**8,592**	0.220
fe_pwt	9,451	0.091	9,124	0.204	8,776	0.485	8,929	0.132	**8,775**	0.126
fe_body	5,710	0.079	**5,236**	0.221	5,289	0.429	5,458	0.097	5,352	0.096
vibrobox	54,046	0.205	54,799	0.293	53,405	0.994	53,028	0.232	**52,835**	0.247
finan512	11,500	0.148	**10,710**	0.395	11,388	0.803	11,632	0.297	11,350	0.183
bcsstk33	116,821	0.237	117,623	0.280	114,427	0.725	**114,168**	0.257	114,322	0.273
bcsstk29	63,929	0.127	62,348	0.266	**61,149**	0.485	63,432	0.138	62,413	0.148
brack2	29,805	0.150	28,721	0.408	**28,104**	1.113	28,555	0.219	28,414	0.217
fe_ocean	27,312	0.198	23,011	0.586	**22,826**	2.032	23,385	0.364	22,896	0.349
fe_tooth	39,987	0.169	39,009	0.484	38,219	1.365	38,261	0.273	**38,032**	0.265
bcsstk31	67,391	0.168	65,103	0.386	**63,852**	0.953	65,470	0.215	64,568	0.225
fe_rotor	52,616	0.199	51,553	0.667	50,279	1.953	50,815	0.336	**50,251**	0.326
598a	63,413	0.225	63,346	0.745	**60,299**	2.303	60,756	0.401	60,440	0.370
bcsstk32	107,911	0.170	102,943	0.535	**102,382**	1.021	104,614	0.215	103,550	0.220
bcsstk30	190,499	0.193	187,906	0.546	**183,650**	0.968	186,719	0.237	185,576	0.257
wave	95,460	0.274	95,142	0.952	91,778	2.859	90,803	0.522	**90,515**	0.485
144	88,039	0.264	87,836	0.964	84,254	2.827	84,245	0.490	**83,643**	0.455
m14b	109,570	0.335	108,677	1.411	103,996	3.484	104,563	0.625	**103,878**	0.576
auto	191,400	0.681	191,933	2.841	183,624	6.652	181,215	1.355	**180,367**	1.203
AS365	54,767	3.154	52,993	15.519	51,943	15.984	50,740	3.822	**50,356**	3.669
NLR	60,287	3.538	58,430	17.249	57,437	17.553	55,775	4.319	**55,378**	4.134
adaptive	48,634	4.452	44,364	20.789	44,149	24.593	42,651	5.911	**42,344**	5.330
ldoor	439,153	1.624	420,011	9.834	**414,884**	5.846	421,377	1.771	415,463	1.816
Serena	1,852,915	3.140	1,869,282	15.720	1,813,903	31.495	1,762,200	5.079	**1,760,964**	4.828
audikw1	2,945,167	3.269	2,976,115	17.174	2,838,997	23.877	**2,830,537**	4.941	2,835,015	4.951
channel-500x.	1,356,670	6.633	1,274,048	31.187	1,305,570	64.801	1,274,548	12.781	**1,260,250**	11.258
dielFilterV3.	2,442,877	3.652	2,432,023	20.173	**2,321,037**	26.516	2,335,146	5.272	2,321,886	5.054
Flan_1565	2,463,890	4.376	2,392,417	25.932	**2,317,798**	19.220	2,344,077	5.412	2,335,178	5.846
nlpkkt240	10,303,386	65.096	10,326,787	297.996	10,171,102	156.130	**9,751,898**	96.392	9,763,379	108.195
Geo. Mean	105760.6	0.540	102440.1	1.795	100294.6	3.457	100542.3	0.791	**99634.8**	0.764

The five refinement algorithms: Greedy, Parallel Recursive Bisection FM (RB-FM), k-way Pairwise FM (KPM), Multi-Try FM (MTFM), and Hill-Scanning (HS), run on the graphs from the Graph Partitioning Archive [26] in the top section, and the University of Florida Sparse Matrix Collection [6] in the bottom section. The lowest mean edgecut achieved per graph is highlighted in bold.

spent in the rest of the phases of the multilevel paradigm. HS achieves speedups between $5.7\times$ and $16.7\times$, with a geometric mean of $9.3\times$ using 24 threads. This compares to a geometric mean speedup for the parallel Greedy algorithm of only $2.7\times$. Because HS performs more work per iteration (more vertices visited and more vertices moved), the overheads associated with parallelizing refinement are a smaller fraction of the runtime. Furthermore, HS is able to achieve a degree of dynamic load balancing. This is because each edge can only be traversed at most twice, and as threads build groups they essentially steal work from each other by traversing edges connected to the vertices of other threads.

Figure 4b shows the relative edgecut as the number of threads increases. The resulting edgecut changed only slightly for most of the graphs as the degree of parallelism was increased. The edgecut increased the most for ldoor, going up by 2.7%, and decreased the most for Flan_1565, decreasing 1.7%. However, after eight threads, these changes largely plateau as we increase the number of threads to 24. The geometric mean increase across all ten graphs was only 0.52%, demonstrating the stability of parallel HS.

We compare the total runtime of the entire multilevel process using the four parallel refinement schemes in Figure 5a. The geometric mean runtimes for RB-FM, KPM, and HS using 24 threads are plotted relative to the runtime of Greedy using 24 threads to create 64-way partitions. HS greatly reduces the difference in runtime with the Greedy algorithm, averaging only 17% longer total partitioning time. Not only are RB-FM and KPM slower when run serially, they both have limited parallelism, resulting in substantially longer runtime using 24 threads compared to HS. RB-FM must operate serially when making the first bisection, and cannot express p way concurrency until after the first $\log p$ bisections. While KPM can express up to $k/2$ way concurrency via edge coloring the partition-graph G_P, many of the resulting colors will have less than $k/2$ edges when G_P is not a complete graph, further limiting the degree of parallelism.

Figure 5b shows the geometric mean edgecut of RB-FM, KPM, and HS relative to Greedy using 24 threads. HS had a geometric mean edgecut 6.3% lower than Greedy. This is 3.4% and 1.9% lower than RB-FM and KPM respectively. This shows that not only is HS extremely fast and scales well in terms of the number of threads used, but it also produces the best quality among parallel refinement schemes. While RB-FM and KPM have the ability to hill-climb, the movement of hills is restricted to the bisection currently being refined. For RB-FM, this is particularly restrictive as bisections are never revisited as the algorithm recurses. In KPM, we see this have less of an impact as the different pairs of partitions are cycled though multiple times during refinement. HS is able to achieve

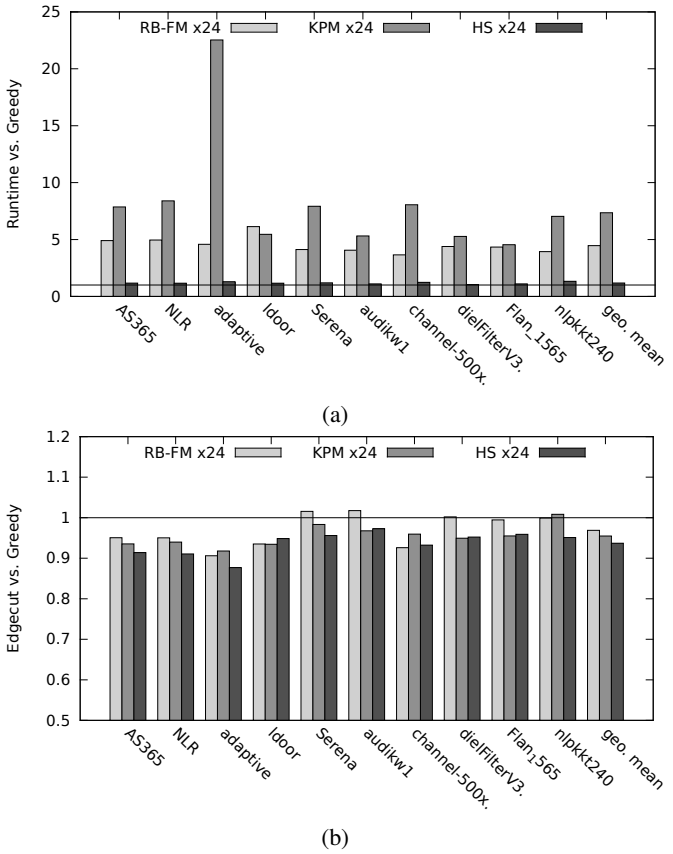

Fig. 5: Comparison of parallel refinement schemes against Greedy refinement using 24 threads: (a) the total partitioning time (for all phases of the multilevel paradigm), (b) the final edgecut.

the lowest mean edgecut because each group it identifies is free to move to any partition which is connected.

VIII. Conclusion

In this paper we presented a novel shared-memory parallel refinement algorithm for graph partitioning, Hill-Scanning. Our parallel algorithm has the ability to break out of local minima, allowing it to find solutions of high quality. By identifying the groups of vertices that correspond with hills in the objective state, we are able to move the group of vertices to the partition of maximum gain for the group as a whole. We showed that our method when run serially fairs well against other serial methods, a 3.5% lower geometric mean runtime and produces partitions of equal quality. Unlike other hill-climbing refinement algorithms, the Hill-Scanning algorithm can efficiently parallelized, which is quickly becoming a requirement to achieve even modest performance on modern processors. We showed that the Hill-Scanning algorithm has a parallel time complexity of $O(kn/p + (m/p)\log n)$, where k is the number of partitions, n is the number of vertices, m is the number of edges and p is the number of threads. Our strong scaling experiments showed that Hill-Scanning achieves $5.7 - 16.7\times$ speedup when run with 24 threads, while only producing 0.52% higher edgecuts than when run serially. Compared to parallel Greedy refinement, this is only a 17% increase in runtime while offering a 6.3% decrease in the edgecut. This is $6.3\times$ faster and 1.9% lower edgecut than parallel Pairwise FM. This new algorithm paves the way for generating high quality graph partitionings using the full capabilities of modern computer architectures.

Acknowledgment

This work was supported in part by NSF (IIS-0905220, OCI-1048018, CNS-1162405, IIS-1247632, IIP-1414153, IIS-1447788), Army Research Office (W911NF-14-1-0316), Intel Software and Services Group, and the Digital Technology Center at the University of Minnesota. Access to research and computing facilities was provided by the Digital Technology Center and the Minnesota Supercomputing Institute.

References

[1] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390, 1977.

[2] Thang Nguyen Bui and Curt Jones. Finding good approximate vertex and edge partitions is np-hard. *Information Processing Letters*, 42(3):153 – 159, 1992.

[3] Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[4] Siew Yin Chan, Teck Chaw Ling, and Eric Aubanel. The impact of heterogeneous multi-core clusters on graph partitioning: an empirical study. *Cluster Computing*, 15(3):281–302, 2012.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter Priority queues. The MIT Press, 3rd edition, 2009.

[6] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011.

[7] Shantanu Dutt and Wenyong Deng. Vlsi circuit partitioning by cluster-removal using iterative improvement techniques. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 194–200. IEEE Computer Society, 1997.

[8] Adam Dziekonski, Adam Lamecki, and Michal Mrozowski. Tuning a hybrid gpu-cpu v-cycle multilevel preconditioner for solving large real and complex systems of fem equations. *Antennas and Wireless Propagation Letters, IEEE*, 10:619–622, 2011.

[9] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Design Automation, 1982. 19th Conference on*, pages 175 –181, june 1982.

[10] Jianya Gong and Sung Kyu Lim. Multiway partitioning with pairwise movement. In *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*, pages 512–516. IEEE, 1998.

[11] Bruce Hendrickson and Robert Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[12] Vincent Heuveline. Hiflow 3: a flexible and hardware-aware parallel finite element package. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, page 4. ACM, 2010.

[13] Carlo Janna, Andrea Comerlati, and Giuseppe Gambolati. A comparison of projective and direct solvers for finite elements in elastostatics. *Advances in Engineering Software*, 40(8):675–685, 2009.

[14] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 29. ACM, 1995.

[15] George Karypis and Vipin Kumar. Multilevel graph partitioning schemes. In *ICPP (3)*, pages 113–122, 1995.

[16] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[17] George Karypis and Vipin Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[18] B. W. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell system technical journal*, 49(1):291–307, 1970.

[19] Dominique LaSalle and George Karypis. Multi-threaded graph partitioning. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 225–236. IEEE, 2013.

[20] Dominique LaSalle and George Karypis. Efficient nested dissection for multicore architectures. In *Euro-Par 2015, Parallel Processing, 21st International Euro-Par Conference*, Lecture Notes in Computer Science. IEEE, Springer, 2015.

[21] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proceedings of the International Conference and Exhibition on High-Performance Computing and Networking*, HPCN Europe 1996, pages 493–498, London, UK, UK, 1996. Springer-Verlag.

[22] Ilya Safro, Peter Sanders, and Christian Schulz. Advanced coarsening schemes for graph partitioning. *CoRR*, abs/1201.6488, 2012.

[23] Peter Sanders and Christian Schulz. Engineering multilevel graph partitioning algorithms. In Camil Demetrescu and Magns Halldrsson, editors, *Algorithms - ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer Berlin / Heidelberg, 2011.

[24] Peter Sanders and Christian Schulz. Distributed evolutionary graph partitioning. In David A. Bader and Petra Mutzel, editors, *ALENEX*, pages 16–29. SIAM / Omnipress, 2012.

[25] Olaf Schenk, Andreas Wächter, and Martin Weiser. Inertia-revealing preconditioning for large-scale nonconvex constrained optimization. *SIAM Journal on Scientific Computing*, 31(2):939–960, 2008.

[26] A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning. *J. Global Optimization*, 29(2):225–241, 2004.

[27] Markus Wittmann and Thomas Zeiser. Technical note: Data structures of ilbdc lattice boltzmann solver. 2011.