Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 15-019

A Parallel Hill-Climbing Refinement Algorithm for Graph Partitioning

Dominique LaSalle, George Karypis

November 25, 2015

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 re-ordering, and VLSI circuit design. Producing high quality graph partitionings while effectively utilizing available CPU power is becoming increasingly challenging due to the rising number of cores per processor. This not only increases the amount of parallelism required of the partitioner, but also the degree partitionings it is to generate. 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. Our method matches the quality of the current high-quality serial refinement methods, and 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 than other parallel refinement methods.

1 Introduction

Unstructured data is becoming increasingly common. Decomposing this data is an important preprocessing step in many computations. Graph partitioning is a technique for decomposing this data such that the dependencies between components are minimized. To model the data as a graph for partitioning, the data elements are represented as vertices in the graph, and the relationships between the elements are modeled as edges. A decomposition of the data will balance the data elements among partitions, and minimize the number of partition spanning edges.

Modern solutions to the graph partitioning problem rely on the multilevel paradigm. One of the major factors that determines the effectiveness of the multilevel paradigm, is the refinement technique used to improve the partitioning as it is applied from the coarse levels to the fine levels. Variations of the Kernighan-Lin [19] refinement scheme are used for improving graph bisections serially, and Greedy [17] refinement is commonly used for k-way and/or parallel refinement [18, 20].

The Kernighan-Lin scheme allows for breaking out of local minima by moving vertices that increase the edgecut on their own, but decrease it as a whole. This capability, known as *hill-climbing*, has been and integral part of high quality refinement schemes. Current methods [8, 6, 24] use the same move-and-revert strategy, where many vertices are moved across partition boundaries, and then the partitioning state is reverted back to the best observed state. While this is an effective strategy, it is inherently serial. Attempts have been made to parallelize these schemes by using two-way refinement between pairs of partitions concurrently [9, 12]. This however can be time intensive and has limited parallelism.

In this paper we examine existing refinement schemes for multilevel graph partitioning and present a new shared memory parallel method for directly refining a k-way partitioning that incorporates hill-climbing. Our new method, Hill-Scanning, produces solutions of equal quality to

Pairwise FM [9] and Multi-Try FM [24]. 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. We present strong scaling results with up to 24 threads and show that it achieves speedups of $5.7 - 16.7\times$, while exhibiting only 0.52% increase in edgecuts.

This paper is organized as follows. In Section 2 we define the graph partitioning problem and the notation used throughout this paper. In Section 3 we review the relevant prior work. We present our new Hill-Scanning algorithm in Section 4. We detail the conditions of our experiments in Section 5. This is followed by the results of our experiments in Section 6. Finally, we conclude the paper in Section 7.

2 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*:

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

In this work, we are concerned with the balanced graph partitioning problem, which means the size of each partition set 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 internal degree of v, that is, the sum of the weight of edges connecting v to the partition in which it resides. Let $d_{ext}(v)$ denote the external degree of v, that is, the sum of the weight of edges connecting v to partitions other than the one in which it resides. 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.

3 Related Work

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

3.1 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 [10, 17, 22, 24]. These methods work by generating a series of coarser (smaller) approximations of the original graph, finding a partitioning for the coarsest of these graphs, and applying this solution back through the series

of graphs. For each graph this solution is applied to, a refinement method is used to improve the partitioning the finer (larger) graph, making use of the increased degrees of freedom.

Buluç et al. [3] recently provided an overview of the modern approaches to the graph partitioning problem. The fastest methods for generating the coarser graphs are often based on Heavy Edge Matching [17], which uses only the edge weight for prioritizing vertices to collapse together. More advance techniques have been proposed that use more information when deciding which vertices to collapse [23], and offer improved quality at the cost of runtime.

3.2 Refinement

Refinement can have a large impact on the quality of a solution generated via the multilevel paradigm. Refinement techniques range from light weight greedy approaches [17], to more intensive flow-based techniques [25]. While the multilevel paradigm has been shown to produce solutions of good quality as result of the contracted edge weight [15], refinement techniques capable of breaking out of local minima offer a means to explore a wider range of solutions and improve quality.

The Greedy [17], 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 boundary of the partitioning 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 greedy order to reduce the edgecut until, with the restriction that each vertex can move only once per iteration. This method of converging on local minima of the edgecut has been successfully parallelized on shared-memory parallel architectures [20].

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 are referred to as *hills*. The process of moving these groups of vertices to decrease the edgecut 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.

One of the earliest methods for refining a two-way partitioning is that of the Kernighan-Lin algorithm [19]. Originally proposed as a direct means of inducing a partition, it works by first randomly assigning vertices to each partition. It then goes 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. It is shown that this bisection (two-way partitioning) method can be used to create k-way partitionings via recursion. This process of recursively splitting a graph to achieve a k-way partitioning is known as *recursive bisection*.

Fiduccia and Mattheyses [8] improved upon this method by relaxing the balance constraint and moving vertices one at a time instead of swapping. Priority queues are used to identify the order in which to move vertices. Again, all possible moves are made, before the algorithm reverts back to the best observed state. For arbitrarily weighted graphs, this algorithm runs in $O(m \log n)$ time ($O(m)$ time for uniform edge weights using a special data structure).

Gong and Lim [9] proposed k-way Pairwise FM (KPM) refinement, which identifies independent pairs of partitions and performs FM on these pairs. A new set of independent pairs is selected and is refined. This repeats until all partition boundaries have had refinement applied. Unlike using FM for recursive bisection, this directly optimizes the k-way edgecut. However, there are $k(k-1)/2$ possible partition boundaries to refine, which can make this a costly method for large numbers of partitions.

Dutt and Deng [6] proposed the CLIP/CDIP variants of FM for hypergraph partitioning. After

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 ←BuildHill(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
```

all vertices have been inserted into the priority queue, they have their priority all set to zero while preserving the ordering. Then, the top vertex v is extracted and moved, and all of its neighbors have their priorities updated. This has the effect of restricting the search space of FM to the moved vertex v.

Sanders and Schulz [24] introduced a variant of FM which also focuses on localized vertex moves, but in a k-way setting. Their variant, named Multi-Try FM, uses multiple small trials of FM per iteration. A trial starts by inserting a random vertex into the priority queue, the seed vertex. Once this vertex is extracted and moved, its neighbors are added to the priority queue. One sided FM then continues with the restriction that vertices only move from the same source partition to the same destination partition as seed vertex for the trial. A new seed vertex is selected, and this process continues until all 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 the enter the priority queue, and preventing them from re-entering it until the next iteration.

Hill climbing has also effectively been accomplished by re-coarsening a graph and then applying refinement at coarser levels. This approach, although computationally expensive, has been shown to result in high quality partitionings [14, 28, 25].

4 Hill-Scanning Refinement

In this section, we present our Hill-Scanning refinement algorithm. We first describe a simplified version of this algorithm in Section 4.1, for refining two-way partitions serially. We then show how to extend this algorithm to the two-way setting in Section 4.2. We then present the full version of our algorithm, for refining k-way partitionings on shared-memory parallel architectures in Section 4.3.

Algorithm 2 Hill Building

```
1: function BuildHill(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
```

4.1 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 hill size ϕ.

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 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 hill rooted at v. If we identify a hill rooted v with a positive gain, we move the hill to the opposing partition. If a vertex is moved by itself or as part of a hill, it is locked in place and prevented from moving for the rest of the iteration.

The intuition behind this algorithm is that we want to avoid the *move-and-revert* process used in KL/FM like algorithms, and instead improve the partitioning at each successive state. By attempting to move each vertex individually before searching for a hill, we are able to overlap fine-grain and coarse-grain partitioning improvements.

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 hill, the maximum hill 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 hills will likely have been moved during a coarser round of refinement. We have found $\phi = 16$ provides a good balance of runtime and quality.

The function `BuildHill` starts by initializing an empty hill, and inserting the root vertex v into the hill priority queue. The hill is then grown by extracting vertices from the priority queue. When a vertex u is extracted from the top of the priority queue, it is added to the hill. If the gain associated with moving the entire hill is positive, the loop exits and the hill is returned. Otherwise, the neighbors of u are added to the priority queue. If the hill reaches the maximum allowable size and would not result in a 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 building 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 hills are discarded. Furthermore, we observed that hills 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 $\sqrt{b(V)}$ hills 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 [6] searching for hills in a localized area at a time. However, because it identifies hills hills before moving them, we can extend it to the k-way and parallel settings.

4.2 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 `BuildHill` 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 [24] is also works on k-way partitions, 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.

4.3 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 [21], 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 [20] 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.

4.4 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 in Section 4.4.1, 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.

4.4.1 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 in 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.

4.4.2 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 per edge, adding an addition $O(m)$ term to the complexity. This is hidden by the larger terms we derived in Section 4.4.1 for operations on the priority queues. Thus, combining the cost of considering what partition to a vertex/hill to that of the cost of the priority queues, the cost of k-way Hill-Scanning becomes $O(kn + m \log n)$,

4.4.3 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 inserted into a given thread's priority queue is n/p, making the cost of performing an update as $(\log(n/p))$. 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 4.4.1. 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 $O(kn/p + (m/p)\log n)$.

5 Experimental Setup

5.1 Data

Table 1 shows the 30 graphs and their sizes used Section 6. 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 [27]. The second set of graphs are the non-zero patterns of some of the largest matrices from the University of Florida Sparse Matrix Collection [5].

5.2 System Configuration

These experiments were run on a machine with 2× 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.

Table 1: Graphs used for experiments

Graph [27]	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 [11]	6,815,744	13,624,320	ldoor [5]	952,203	22,785,136
Serena [13]	1,391,349	31,570,176	audikw1 [5]	943,695	38,354,076
channel-500x. [29]	4,802,000	42,681,372	dielFilterV3. [7]	1,102,824	44,101,598
Flan_1565 [13]	1,564,794	57,920,625	nlpkkt240 [26]	27,994,600	373,239,376

5.3 Implementation

We implemented HS, and the other refinement algorithms detailed below in the mt-Metis multi-threaded graph partitioning framework. The version used for these experiments is mt-Metis 4.4. The matching scheme used is *Heavy Edge Matching* [16]. Each refinement scheme terminates when an iteration completes without any moves, or a maximum of 8 iterations have been performed.

5.3.1 Greedy

We used the existing implementation of parallel Greedy [20] refinement in mt-Metis for these experiments.

5.3.2 FM

We implemented a boundary-only variant of the Fiduccia-Mattheyses [8] (FM) algorithm for use in recursive bisection. 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.

5.3.3 KPM

For our implementation k-way pairwise FM (KPM), we split the 8 iterations into two local iterations and four global iterations. 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.

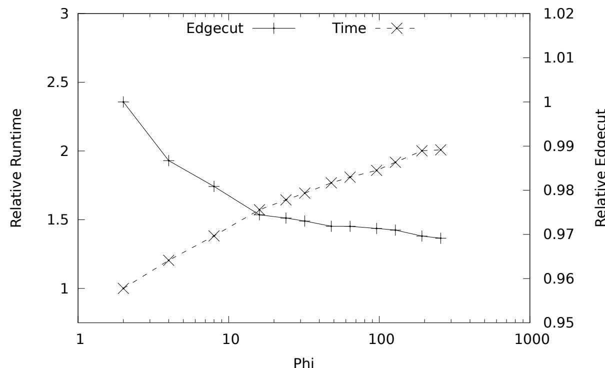

Figure 1: Effects of varying ϕ on Flan_1565

5.3.4 MTFM

For our serial experiments, we implemented Multi-Try FM [24] (MTFM), To keep the runtime down, we used a limit of 16 vertices being moved without gain before the seeded FM reverts back to the best state, and the next seed vertex is selected.

5.3.5 HS

We implemented the parallel k-way version of the Hill-Scanning (HS) algorithm we detailed Section 4.3. For the k-way and parallel experiments, we used a maximum hill size, ϕ of 16.

6 Results

First, we examine the effect of varying the different parameters of the Hill-Scanning algorithm on runtime and quality in Section 6.1. This is followed by a comparison of HS with Greedy, FM, and KPM refinement schemes in Section 6.2. Finally, in Section 6.3, we perform strong scaling experiments to examine the effects of parallelization on runtime and quality of HS.

6.1 Parameters

In Figure 1, we present the effects of varying ϕ, the maximum size of a hill, on the quality and runtime of refinement for the graph Flan_1565. As the hill size increases, the further inside of a partition is explored, and thus the runtime increases relatively steadily. However, at a hill size of 16, the increase in quality slows as the larger hills are less likely to be moved compared to 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.

6.2 k-way Refinement

The performance of the different refinement schemes, run serially, is compared in Table 2. The results presented are the geometric mean of 25 runs. Runtime includes the entire multilevel process: coarsening, initial partitioning, and uncoarsening. 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 moves fewer vertices. The Hill-Scanning algorithm was the second fastest method, and the multilevel process using HS took only 27.1% longer than using Greedy. However, HS resulted in the lowest

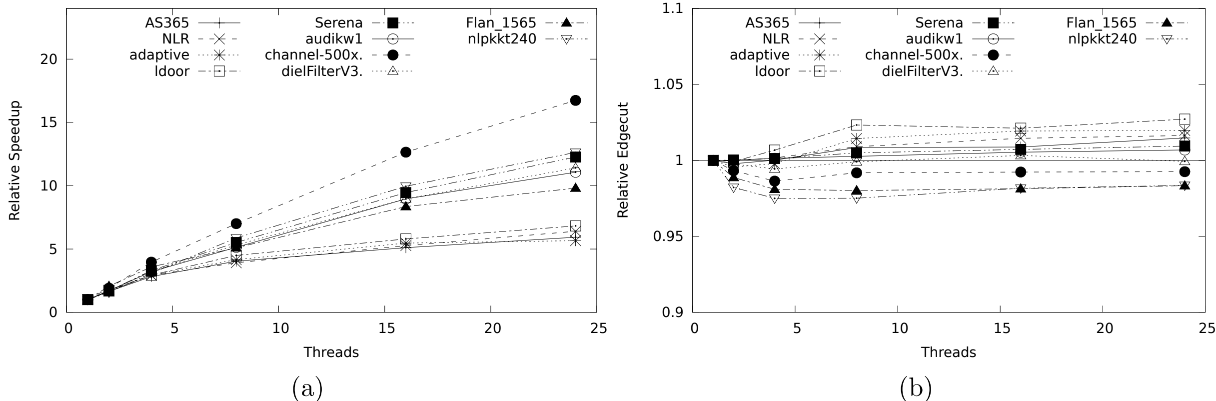

Figure 2: Strong scaling of Hill-Scanning: (a) relative speedup, (b) relative edgecut.

mean edgecut of the refinement schemes, and had the smallest mean edgecut on 14 of the 30 graphs. Multi-Try FM 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.

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 of 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.

6.3 Parallel k-way Refinement

Figure 2a shows the strong scaling of the HS algorithm. HS achieves speedups between 5.7× and 16.7×, with a geometric mean of 9.3× using 24 threads. This compares to a geometric mean speedup for the Greedy algorithm of only 2.7×. 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.

Figure 2b shows the relative edgecut as the number of threads increases. The resulting edgecut changed 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 multilevel process using the four parallel refinement schemes in Figure 3a. 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 closes the gap with Greedy refinement with this degree of parallelism, 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 does not fully express p parallelism until

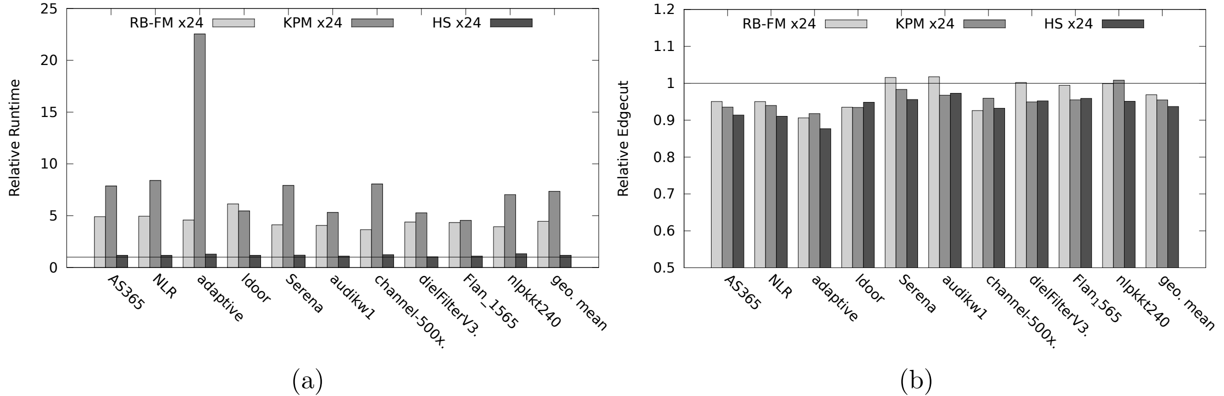

Figure 3: Comparison of parallel refinement schemes against Greedy refinement using 24 threads: (a) relative partitioning time, (b) relative edgecut.

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 3b 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, but it also produces the best quality among parallel refinement schemes.

7 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 hill climb, allowing it to find solutions of high quality. By identifying the groups of vertices that form hills before they are moved, we are able to move the group of vertices to the partition of maximum gain. We showed that our method when run serially is competitive with 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 is parallel. We showed that the Hill-Scanning algorithm 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. 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Manuel Holtgrewe, Peter Sanders, and Christian Schulz. Engineering a scalable high quality graph partitioner. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 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, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 7(1):69–79, 1999.

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

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

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

[18] 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.

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

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

[28] Chris Walshaw. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1-4):325–372, 2004.

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

Table 2: 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