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 important in distributing workloads on parallel compute systems, computing sparse matrix re-orderings, and designing VLSI circuits. Refinement algorithms are used to improve existing partitionings, and are essential for obtaining high-quality partitionings. Existing parallel refinement algorithms either extract concurrency by sacrificing in terms of quality, or preserve quality by restricting concurrency. In this work we present a new shared-memory parallel algorithm for refining an existing k-way partitioning that can break out of local minima and produce high-quality partitionings. This allows our algorithm to scale well in terms of the number of processing cores and produce clusterings of quality equal to serial algorithms. Our algorithm achieves speedups of $5.7-16.7\times$ using 24 cores, 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 algorithms 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 attempt to minimize the number of partition spanning edges. The quality of the partitioning plays a crucial role in the performance of applications. The size of graphs that need to be partitioned in these applications has dramatically increased. However, the majority of increases in computer processing power have been achieved by increases in the number of processing cores per chip and not by increases in the throughput of the cores themselves. This makes it imperative for graph partitioning algorithms to express a high degree of parallelism so as to take advantage of modern processors.

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 being cut by a partitioning can lead to substantially better solutions than those algorithms that can not. The FM [1] refinement algorithm relies 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 being 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 [2] and Multi-Try FM [3] 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 [4] attempt to parallelize these schemes by using two-way refinement between pairs of independent partitions concurrently. This however can be time intensive and has limited parallelism.

In this paper we propose the Hill-Scanning algorithm, a shared memory parallel refinement algorithm capable of breaking out of local minima. Unlike its predecessors, this new method exhibits a high degree of parallelism and as a result exhibits significant speedup over other methods. In the serial environment, we show that Hill-Scanning produces solutions of equal quality to Multi-Try FM. We show that Hill-Scanning is up to $6.3\times$ faster than other parallel refinement methods with hill-climbing capabilities when running on 24 cores. Hill-Scanning produces solutions with 6.3% lower edgecut than parallel Greedy refinement and 1.9% lower edgecut than parallel k-way Pairwise FM. We present strong scaling results on up to 24 cores and show that Hill-Scanning achieves speedups of $5.7-16.7\times$, while exhibiting only a 0.52% increase in edgecuts over its serial counterpart.

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*:

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

where $\Gamma(v)$ is the set of vertices connected by edges to v.

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

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

III. Background

Because the graph partitioning problem is NP-Hard [5], many advanced heuristics have been developed. The multilevel paradigm for graph partitioning has become the de facto standard for developing high-performance and high quality algorithms [6], [7], [8], [3]. The multilevel paradigm computes the partitioning of a graph in 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 of the solution increase with the number of vertices.

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. Because coarsening contracts most of the edges in the graph, relatively simple methods can be used during the initial partitioning phase. The most common approach is to induce a partitioning via a truncated breadth first search and then apply refinement to the partitioning [9]. 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 partitions 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.

While the multilevel paradigm has been shown to produce solutions of good quality as result of the contracted edge weight [10], refinement techniques capable of breaking out of local minima offer a means to explore a wider range of solutions, and finding partitionings of higher quality.

The Greedy [7] algorithm, is an extremely fast method for converging on a local minima. 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 there are no more positive gain moves to be made.

Further decreasing the edgecut beyond a local minima, requires moving more than one vertex. A *hill* is a set of vertices whose collective movement results in a reduction of edgecut, but the movement of any subset does not. Each member of a hill will have more edge weight connecting it to the group than to any one partition (excluding the other members of the group). The process of speculatively moving individual vertices so as to find a new local minima is referred to as *hill-climbing*. Moving individual vertices speculatively allows for moving enough members of these groups such that each of the remaining unmoved members will result in edgecut reductions. The capability to hill-climb defines a class of high-quality refinement techniques.

IV. Related Work

One of the most widely used refinement methods is the Fiduccia-Matteyses (FM) [1] algorithm. At each step in the algorithm, 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 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. All possible moves are made before the algorithm reverts back to the best observed state.

In order to refine a k-way partitioning, Gong and Lim [4] 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.

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 decreases the edgecut, for speculative vertex moves (which increase the edgecut), it interferes with finding new local minima.

To address this issue, Dutt and Deng [2] 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 the 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 hill will be moved.

In order to use to FM to refine k-way partitionings and enforce the localized search pattern of CLIP/CDIP, Sanders and Schulz [3] introduced 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 and FM proceeds as normal. Once the trial terminates, a new unvisited seed vertex is selected and this process continues until all boundary vertices in the graph have been visited.

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 between serial refinement algorithms capable of breaking out of local minima, and parallel greedy algorithms capable of utilizing multicore systems.

The Hill-Scanning algorithm is based on the observation that the move-and-revert strategy used by FM and its variants, breaks out of local minima by pulling hills across partition boundaries. However, this approach has two shortcommings. First, it is inherently serial, as a linear ordering of moves is needed in order accurately track edgecut and revert partitioning states. Second, each hill 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 hill is connected to partition i with the majority of its edge weight, the hill as a whole may be connected to partition j with a larger total edge weight.

The Hill-Scanning algorithm, at a high level, works by identifying hills and treating them as a single entity when considering which vertices to move. On a shared memory system where all processing elements have access to all of the data, hills can be efficiently identified and moved in parallel.

Each iteration works as follows. First, all of the boundary vertices, those with edges connecting them to other partitions, are inserted into a priority queue.

The gain associated with moving a vertex to partition i is:

$$gain = d_{P_i}(v) - d_{int}(v),$$

where $d_{P_i}(v)$ is the sum of the edgeweight connecting v to the partition i and $d_{int}(v)$ is the sum of the edgeweight connecting v to the partition in which it resides. We use an approximation of this gain for all incident partitions as the priority:

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

where $d_{ext}(v)$ is the sum of the edgeweights connecting v to partitions other than the one in which it resides and $\Delta(v)$ is the number of external partitions to which v is connected.

Vertices are extracted from this queue and are considered for moving. If the actual gain associated with moving a vertex v is positive, and moving v would not violate the balance constraint, v is moved, 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, an attempt is made to find a hill rooted at v. If a hill is identified rooted v whose movement would result in a positive gain, the hill is moved.

A hill is found by first initializing an empty hill, and inserting the root vertex v into the priority queue. The hill is then grown by extracting vertices from the priority queue and adding them 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 in the same partition are added to the priority queue and the search continues. If the hill reaches the maximum allowable size and would not result in positive gain if moved, it is discarded.

To keep the runtime down, each time an edge is traversed when searching for 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, an iteration is ended early 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).

The move-and-revert strategy of FM like algorithms is difficult to parallelize due to the need of a strict ordering of moves. While methods have been proposed for running FM on independent subgraphs [11], 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 [12] for parallelizing Greedy refinement on shared-memory architectures.

Each refinement iteration is split into two phases: upstream and downstream. 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. Refinement stops when no moves are made during an iteration, or the maximum number

of iterations has been reached.

VI. Experimental Setup

Table I shows various information about the graphs used in the experiments presented in Section VII. These are undirected graphs. The first set of graphs (wing through auto) are all graphs with greater than $100,000$ edges from the Graph Partitioning Archive [13]. The second set of graphs are the non-zero patterns of some of the largest matrices from the University of Florida Sparse Matrix Collection [16].

These experiments were performed 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. We implemented HS, and the other refinement algorithms 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* [9]. Each refinement scheme terminates when an iteration completes without any moves, or a maximum of 8 iterations have been performed.

VII. Results

The serial runtime and edgecut of the partitionings produced by the different refinement schemes are shown in Table II. The results presented are the geometric mean of 25 runs. Runtime includes the entire multilevel process: coarsening, initial partitioning, and uncoarsening. The entire time is shown rather than just the time for refinement in order to include the overhead associated with the recursive bisection required by RB-FM [1] in our comparison, as well as to place the runtime cost of the refinement schemes in perspective.

Greedy [7] 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. 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 [3], can be efficiently parallelized.

KPM [4] 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 for partitionings with relatively dense partition connectivity can be exceedingly expensive.

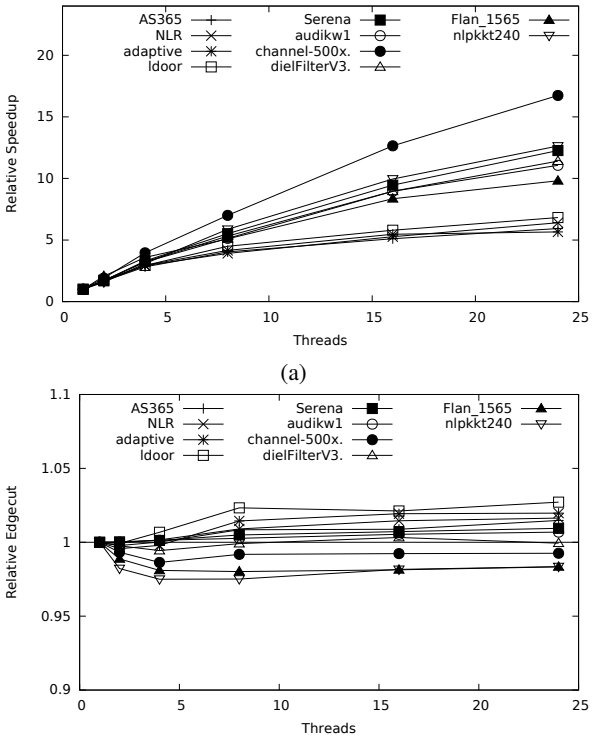

Fig. 1: Strong scaling of Hill-Scanning: (a) relative speedup, (b) relative edgecut.

Figure 1a 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 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 find hills they essentially steal work from each other by traversing edges connected to the vertices of other threads.

Figure 1b 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 2a. The geometric mean runtimes for RB-FM, KPM, and HS using

TABLE I: Graphs used for experiments

Graph [13]	Vertices	Edges	Graph	Vertices	Edges	Graph	Vertices	Edges
t60k	60,005	89,440	wing	62,032	121,544	AS365 [14]	3,799,275	11,368,076
fe_pwt	36,519	144,794	fe_body	45,087	163,734	NLR [14]	4,163,763	12,487,976
vibrobox	12,328	165,250	finan512	74,752	261,120	adaptive [15]	6,815,744	13,624,320
bcsstk33	8,738	291,583	bcsstk29	13,992	302,748	ldoor [16]	952,203	22,785,136
brack2	62,631	366,559	fe_ocean	143,437	409,593	Serena [17]	1,391,349	31,570,176
fe_tooth	78,136	452,591	bcsstk31	35,588	572,914	audikw1 [16]	943,695	38,354,076
fe_rotor	99,617	66,2431	598a	110,971	741,934	channel-500x. [18]	4,802,000	42,681,372
bcsstk32	44,609	985,046	bcsstk30	28,924	1,007,284	dielFilterV3. [19]	1,102,824	44,101,598
wave	156,317	1,059,331	144	144,649	1,074,393	Flan_1565 [17]	1,564,794	57,920,625
m14b	214,765	1,679,018	auto	448,695	3,314,611	nlpkkt240 [20]	27,994,600	373,239,376

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 [13] in the top section, and the University of Florida Sparse Matrix Collection [16] in the bottom section. The lowest mean edgecut achieved per graph is highlighted in bold.

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 2b 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 the lowest mean edgecut because each hill it identifies is free to move to any partition to which it is connected.

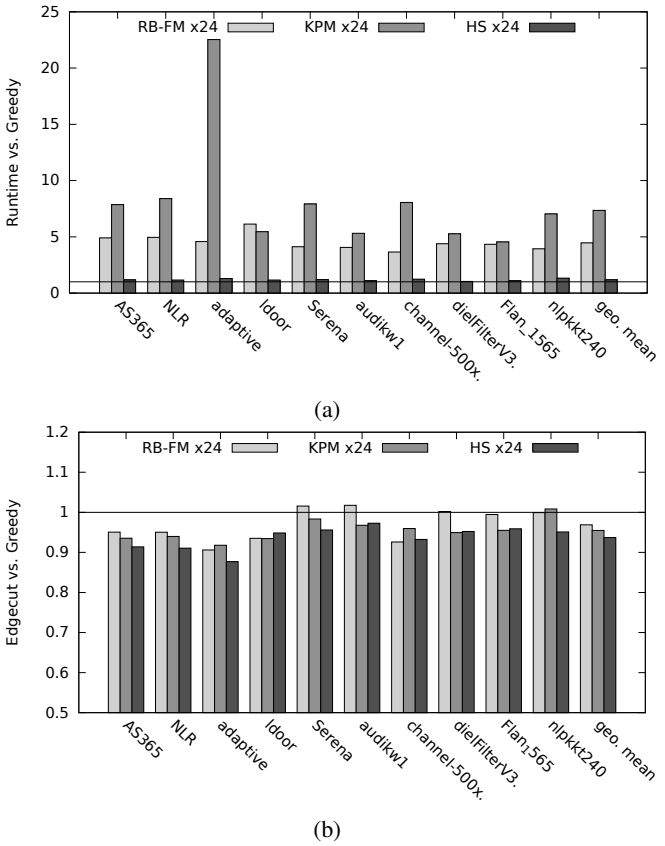

Fig. 2: Comparison of parallel refinement schemes against Greedy refinement using 24 threads: (a) the total partitioning time, (b) the final edgecut.

VIII. Conclusion

In this paper we presented the Hill-Scanning algorithm, a shared-memory parallel refinement algorithm for graph partitioning. 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. 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.

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] C. Fiduccia and R. Mattheyses, "A linear-time heuristic for improving network partitions," in *Design Automation, 1982. 19th Conference on*, june 1982, pp. 175 –181.

[2] S. Dutt and W. Deng, "Vlsi circuit partitioning by cluster-removal using iterative improvement techniques," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society, 1997, pp. 194–200.

[3] P. Sanders and C. Schulz, "Engineering multilevel graph partitioning algorithms," in *Algorithms - ESA 2011*, ser. Lecture Notes in Computer Science, C. Demetrescu and M. Halldrsson, Eds. Springer Berlin / Heidelberg, 2011, vol. 6942, pp. 469–480. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23719-5_40

[4] J. Gong and S. K. Lim, "Multiway partitioning with pairwise movement," in *Computer-Aided Design, 1998. ICCAD 98. Digest of Technical Papers. 1998 IEEE/ACM International Conference on*. IEEE, 1998, pp. 512–516.

[5] T. N. Bui and C. Jones, "Finding good approximate vertex and edge partitions is np-hard," *Information Processing Letters*, vol. 42, no. 3, pp. 153 – 159, 1992.

[6] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/224170.224228

[7] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, pp. 96–129, 1998.

[8] F. Pellegrini and J. 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*, ser. HPCN Europe 1996. London, UK, UK: Springer-Verlag, 1996, pp. 493–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=645560.658570

[9] G. Karypis and V. Kumar, "Multilevel graph partitioning schemes," in *ICPP (3)*, 1995, pp. 113–122.

[10] ——, "Analysis of multilevel graph partitioning," in *Proceedings of the 1995 ACM/IEEE conference on Supercomputing*. ACM, 1995, p. 29.

[11] D. LaSalle and G. Karypis, "Efficient nested dissection for multicore architectures," in *Euro-Par 2015, Parallel Processing, 21st International Euro-Par Conference*, ser. Lecture Notes in Computer Science, IEEE. Springer, 2015.

[12] ——, "Multi-threaded graph partitioning," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 225–236.

[13] A. J. Soper, C. Walshaw, and M. Cross, "A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph Partitioning," *J. Global Optimization*, vol. 29, no. 2, pp. 225–241, 2004.

[14] S. Y. Chan, T. C. Ling, and E. Aubanel, "The impact of heterogeneous multi-core clusters on graph partitioning: an empirical study," *Cluster Computing*, vol. 15, no. 3, pp. 281–302, 2012.

[15] V. 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*. ACM, 2010, p. 4.

[16] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[17] C. Janna, A. Comerlati, and G. Gambolati, "A comparison of projective and direct solvers for finite elements in elastostatics," *Advances in Engineering Software*, vol. 40, no. 8, pp. 675–685, 2009.

[18] M. Wittmann and T. Zeiser, "Technical note: Data structures of ilbdc lattice boltzmann solver," 2011.

[19] A. Dziekonski, A. Lamecki, and M. 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*, vol. 10, pp. 619–622, 2011.

[20] O. Schenk, A. Wächter, and M. Weiser, "Inertia-revealing preconditioning for large-scale nonconvex constrained optimization," *SIAM Journal on Scientific Computing*, vol. 31, no. 2, pp. 939–960, 2008.