Improving Graph Partitioning for Modern Graphs and Architectures

Dominique LaSalle[†]
lasalle@cs.umn.edu

Md Mostofa Ali Patwary[*]
mostofa.ali.patwary@intel.com

Nadathur Satish[*]
nadathur.rajagopalan.satish@intel.com

Narayanan Sundaram[*]
narayanan.sundaram@intel.com

George Karypis[†]
karypis@cs.umn.edu

Pradeep Dubey[*]
pradeep.dubey@intel.com

[†]Department of Computer Science & Engineering
University of Minnesota
Minneapolis, Minnesota, 55455, USA

[*]Parallel Computing Lab
Intel Corporation
Santa Clara, California, 95054, USA

ABSTRACT

Graph partitioning is an important preprocessing step in applications dealing with sparse-irregular data. As such, the ability to efficiently partition a graph in parallel is crucial to the performance of these applications. The number of compute cores in a compute node continues to increase, demanding ever more scalability from shared-memory graph partitioners. Furthermore, datasets whose graphical representations have skewed degree distributions have gained in importance and exhibit very different characteristics than the graphs previously used in scientific computing. In this paper we present algorithmic improvements to the multithreaded graph partitioner mt-Metis. We address issues related to these new network-style graphs and explore techniques to improve performance and parallel scaling. We experimentally evaluate our methods on a 36 core machine, using 20 different graphs from a variety of domains. Our improvements decrease the runtime by $1.5 - 11.7\times$ and improve strong scaling by 82%.

1. INTRODUCTION

As the parallelism of modern processors increases, getting performance out of applications with irregular data access patterns is increasingly challenging. Graph partitioning is an important pre-processing step for irregular applications to achieve performance. On shared-memory platforms, graph partitioning can be used to reduce inter-core communication and cache misses.

Due to its importance, graph partitioning has received significant attention for work distribution in parallel applications [21] and locality maximization [18]. Modern methods rely on the multilevel paradigm [9] to find high quality solutions extremely fast [13, 19, 20]. While distributed-memory parallel partitioners [12, 5, 11] have been in use for almost two decades, methods [22, 4, 14] that exploit the shared-memory property of modern multicore processors have only recently been explored. These shared-memory methods offer improved performance and partition quality for partitioning within a compute node.

The number of cores per compute node has recently increased dramatically, and continues to do so. This demands that graph partitioners exhibit increased parallelism and efficiently make use of the cache hierarchy. High-performance single-node graph partitioners are an important stepping stone for future highly-scalable multi-node partitioners. Fully exploiting node-internal parallelism can decrease the degree of communication operations by one to two orders of magnitude.

The range of graphs needing to be partitioned has also increased. Where the focus used to solely be on mesh-style graphs with relatively uniform degree distributions, it now includes a variety of graphs from social and web sciences. Graphs from these domains tend to have skewed degree distributions and small diameters. These properties invalidate many previous assumptions in parallel multilevel graph partitioning, and it has been shown that new methods for the coarsening phase are needed [1, 17].

In this work, we present algorithmic improvements to the mt-Metis multithreaded graph partitioning framework and experimentally evaluate their effectiveness. We show that these modifications significantly improve performance of mt-Metis on modern architectures and graphs. Specifically, our contributions are:

- An efficient two-hop matching scheme which works well on graphs with highly skewed degree distributions without sacrificing performance or quality on graphs with more uniform degree distributions ($2.0\times$ geometric mean improvement for graphs with skewed degree distributions).
- Implementation level coarsening optimizations ($1.6\times$ geometric mean improvement for coarsening).
- An improved initial partitioning parallelization formulation ($1.8\times$ geometric mean improvement for initial partitioning).
- A method of performing parallel refinement that greatly reduces inter-core communication ($2.5\times$ geometric mean improvement for uncoarsening).

These improvements cumulatively result in speedups of $1.5-11.7\times$ and a geometric mean improvement of strong scaling by 82%, while preserving partition quality on 20 graphs from a variety of domains.

2. DEFINITIONS & NOTATION

Let $G = (V, E)$ denote a simple undirected graph, consisting of a vertex set V and an edge set V. A pair of unordered vertices makes up an edge (i.e., $e = \{v, u\}$ where $v, u \in V$). The number of vertices in the graph is denoted as $n = |V|$ and the number of edges is denoted by $m = |E|$. Vertices and edges can have weights associated with them denoted as $\eta(v)$ and $\theta(e)$ respectively. If no weights are specified, they are assumed to be one.

The balanced graph partitioning problem is defined as creating k disjoint sets of vertices (partitions), $V = V_1 \cup \ldots \cup V_k$, with the constraint that the sum of the weights in any given set not exceed some threshold ϵ greater than the average weight of a set:

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

The objective of the partitioning problem is to minimize the weight of inter-partition edges while not exceeding the balance constraint:

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

When discussing parallel graph partitioning, we refer to the number of threads as p.

3. BACKGROUND

3.1 Multilevel Graph Partitioning

The most prevalent strategy for developing graph partitioning heuristics has been the multilevel paradigm [9]. The multilevel paradigm works by aggregating vertices together in the input graph, G_0, to form a coarser (smaller) graph G_1. This process repeats until a sufficiently coarse graph G_s is formed. This is known as the *coarsening* phase. Then, in the *initial partitioning* phase, a partitioning is found of the coarsest graph. This partitioning is then applied to the next finer (larger) graph, G_{s-1}, and then the partitioning is refined via a local-improvement technique. This process is repeated until the partitioning is applied and refined on the original graph. This is known as the *uncoarsening* phase. Buluç et al. [3] provide an overview of state-of-the-art graph partitioning techniques.

3.2 Multithreaded Graph Partitioning

In this work, we improve the performance of the mt-Metis [14] multithreaded graph partitioner. The design of mt-Metis focuses on reducing data movement between processing cores and memory banks. Each thread is assigned a contiguous set of vertices and their incident edges a priori, such that the number of edges per thread is roughly equal. Each thread is then responsible for operations on their portion of the graph and subsequently their portions of the graphs generated throughout the multilevel paradigm ($G_1, \ldots, G_s$). This maximizes data re-use per thread, and reduces the number of synchronization primitives required to ensure correct execution.

Each level of coarsening is made up of two parts: aggregation and contraction. In aggregation, each thread selects pairs of vertices connected by an edge to be merged together. This is an $O((m + n)/p)$ operation, as each vertex is checked to see if it has been matched, and if it has not been matched, it scans its edges to find an eligible neighbor to match with. A matching vector M is used to keep track of which vertices are unmatched, and which vertices have been matched together. An unmatched vertex will have an entry of pointing to itself $M(v) = v$, and a matched vertex with have an entry point to the vertex it matched with, $M(v) = u$ and $M(u) = v$. Vertices are also constrained in their maximum weight such that $\eta(v) + \eta(u)$ is small enough so as to allow sufficient freedom in finding balanced partitioning for the coarse graph.

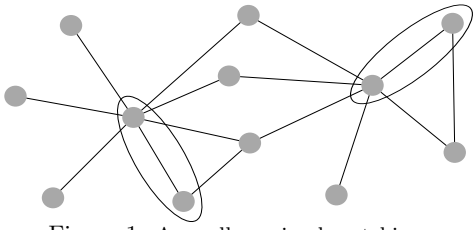


Figure 1: A small maximal matching.

At the start of contraction, a fine-coarse vertex mapping vector C is generated, such that for two matched vertices v and u, $C(v) = C(u)$. The entries in C range from zero to the number of coarse vertices. The adjacency lists of vertices are mapped through C. Then, for two merged vertices u and v, their mapped adjacency lists are merged to form the new adjacency list of their coarse vertex.

Coarsening continues until either the size of the coarsest graph G_s is below a threshold determined by the number of partitions k, or its size failed to reduce by a desired amount $|G_s| > \alpha|G_{s-1}|$, where $0 < \alpha < 1$.

Then, a k-way partitioning is induced on G_s, using mutlilevel recursive bisection. This multilevel process differs than the one used before as the maximum vertex weight constraint is relaxed, the new target size of coarsest graph is small (20 vertices), and serial FM [8] refinement used. Because the size of the graph is small, using a serial and more expensive technique at this point increases quality while having a minimal impact on runtime.

Again, under the assumption that the coarsest graph given as input for the initial partitioning is small, multiple partitionings of it are generated concurrently, and the best one is selected. A single thread is used to generate each partitioning, so as to avoid the cost of synchronization during these much smaller levels of the multilevel paradigm.

As coarsening has two parts, so does uncoarsening. In projection, the partition assigned to a coarse vertex is projected to its fine vertices, using the fine-coarse mapping C. In refinement, each thread is responsible for selecting which of its vertices to move between partitions. As vertices are moved, updates to their neighboring vertices are communicated between threads asynchronously, so as use as up-to-date information as possible when deciding to move a vertex.

3.3 Skewed Degree Distributions

A graph with a skewed vertex degree distribution has many vertices of low degree and only a few vertices of high degree. These graphs often have small maximal matchings, as shown in Figure 1. Other properties of these graphs include small diameters (the distance of the maximum length shortest path between any two vertices in the graph), as shortest

paths will often go through these high degree vertices. The previous assumptions that we could reduce the number of vertices in the graph by close to half by matching vertices along edges, and that the coarsest graph will be small, no longer hold.

If we only find small maximal matching during aggregation, the size of the next coarser graph will reduce by only a small faction, and G_{i+1} is nearly the same size as G_i. Unless coarsening is terminated early, this could cause a runtime of $O(n^2 + nm)$ as we would need $O(n)$ levels of coarsening, each of which takes $O(n + m)$ time. Furthermore, this would also like lead to very uneven vertex weights in the coarsest graph, as high degree vertices would be aggregate at each level and be of large weight, whereas low degree vertices would be unlikely to have been aggregated and be of low weight. This greatly restricts creating a balanced initial partitioning, and often leads to very poor quality and/or unbalanced partitions.

For graphs with higher diameter, the edge density of the input graph G_0 tends to be of greater than or equal density of the coarsest graph G_s. The clusters of vertices (groups of highly inter-connected vertices) are sparsely connected in these graphs, which is what causes them to have high diameters. As a result, once the clusters get contracted, only the few inter-cluster edges are left exposed. However, for graphs with small diameters, the density of G_s increases. For the diameter to be small, the longest-shortest path can only pass through a small number of clusters. Thus, the interconnection of the clusters must be relatively dense, and the number of exposed edges in G_s must be high. This means that the amount of computation associated with creating a partitioning for G_s is dramatically higher, and can exceed that of coarsening and uncoarsening.

Abou-Rjeili and Karypis [1] studied several techniques for aggregating more than two vertices together per level to overcome the reduced size of maximal matchings in graphs with skewed degree distributions. Meyerhenke et al. [17] used sized-constrained label propagation to select small clusters of vertices to aggregate together per level. They showed that this could lead to greatly reduced runtime as well as improved partition quality.

4. ALGORITHMIC IMPROVEMENTS

4.1 Two-Hop Matching

Traditionally, vertices are aggregated together by finding maximal independent sets of edges to contract. This works well because it reduces the number of exposed edges on the graph (and subsequently exposed edge weight), and keeps the size of any coarse vertex from growing much faster than others. However, graphs with highly skewed degree distributions often contain only small maximal independent sets of edges. This causes the next coarser graph to be of similar size, and can cause many vertices to not grow in size at all between successive graphs.

To address this issue, we relax the constraint that two vertices being aggregated together must be connected via an edge. Instead, we allow two vertices to be aggregated together if they have a common neighbor. That is, if they are *two-hops* away on the graph. This has been investigated before in the context of finding vertex separators [7, 10] (to preserve sparsity in direct sparse methods) and graph clustering [2, 15].

To ensure we do not disrupt the quality achieved by traditional matching methods, we use two-hop matching as a secondary pass over the vertices after a maximal matching has been found. Our method assumes that each vertex in the graph has been visited, and that for each unmatched vertex, there exists no neighbor of that vertex for with which it is eligible to match. We group these unmatched vertices that are two-hops from each other into three classes: *leaves*, *twins*, *relatives*.

Leaf vertices are of degree one, and if they share the same parent, they are desirable to aggregate together. They are a subclass of twin vertices, but due to their prevalence in social networks and web graphs, using a special method to detect and match them is beneficial. Twin vertices are vertices which have identical neighbor lists. Relative vertices are vertices which are two hops away but do not have identical sets of neighbors. Relative vertices are the least desirable class to collapse, as doing so can hide good cuts in the coarser graphs. For these reasons we conditionally find and match each of these classes in the same order. If we have successfully matched over 75% of the vertices in the, we perform no two-hop matching. If after matching leaf vertices we still have not matched over 75%, we then perform twin matching. Finally, if this still does not yield a sufficiently large matching, we the match relatives. Below we show that finding all three classes takes at most $O(n \log n)$, but is often linear in the number of unmatched vertices.

4.1.1 Finding Leaves

To find leaves to aggregate together in linear time, we iterate over our set L of unmatched vertices of degree one. For each vertex in L, we add its neighbor to the set of root vertices R. For each root vertex r, we keep track of all the unmatched leaf vertices $L_r \subset L$ we have processed that are incident to r (i.e., for each root vertex, L_r is the set of leaf vertices attached to it). Then for each root vertex r, we can match pairs of leaves in the list L_r, as we know they are two-hops from each other. As the sum the size of the set of unmatched leaf vertices L plus the size of the set of root vertices R cannot exceed the number of vertices in the graph, matching leaf vertices takes at most $O(n)$ time.

4.1.2 Finding Twins

Twin vertices are the most expensive vertices to aggregate together. To minimize this cost, we limit the maximum degree of vertices we consider for twins to 64 (though different values may be more desirable depending on graph characteristics and computational resources). We first sort all of our prospective twin vertices into buckets by degree. As we know this degree is of a bounded range (2 through 64), this sorting can be done in linear time via radix sort. We then sort each bucket using the vertices' neighborhoods as keys. As we have bounded the size degree of these vertices, we can compare two adjacency lists in linear time, giving us a $O(n \log n)$ complexity. During this sorting process we remove and match two vertices when their adjacency lists are equal. To further speedup this process, we first generate a hash of each vertex's adjacency list, and only perform the comparison based sort on vertices with equal hashes.

4.1.3 Finding Relatives

Finding pairs of relative vertices to match can be done using the same process as finding leaf vertices. However, because these vertices can be of degree larger than one, the size of the set of the root vertices is no longer bounded by the number of candidates, but by the number of edges incident to our candidates and also still bounded by the total number of vertices in the graph. This makes the complexity of finding relative vertices of $O(n)$.

4.2 Coarsening Optimizations

During contraction we must translate adjacency lists to point at the new coarse vertices, and merge adjacency lists of vertices that have been aggregated together. From a matrix standpoint, this involves merging columns and rows of the adjacency matrix together. In our previous work [14], the approach here was to use a hash table to accumulate values for each coarse adjacency list. This ensured that when perform random accesses into the hash table, it resided in cache and reduced latency. For graphs with large maximum degree, a dense vector was used instead to avoid collisions in the hash table, but incurring the cost of latency associated with DRAM accesses.

For graphs with skewed vertex degree distributions, this is undesirable as the majority of the vertices have adjacency lists which can be merged in a hash table with few collisions. We can determine how many coarse vertices we will generate during aggregation. We then do a pass over the coarse vertices to be generated and calculate an upper bound on the degree of each coarse vertex (the sum of the degrees of fine vertices). We assign low degree vertices numbers increasing from zero, and high degree vertices numbers decreasing from the number of coarse vertices. This ensures then during contraction, that we can use a hash table for the set of low degree vertices, and a dense vector the set of high degree vertices where it is actually necessary.

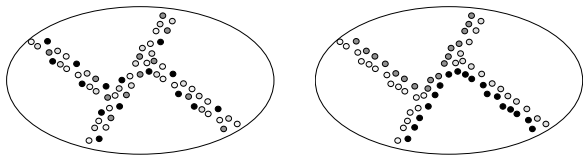
During both aggregation and contraction, most of the memory accesses are through indirection arrays. In order to reduce the effects of latency, we use software prefetching. In aggregation, this consists of prefetching the locations of the match vector for neighbor vertices. During contraction, we prefetch the location of the coarse vertex mapping for the vertices in the adjacency lists.

4.3 Cache Oriented Initial Partitioning

The past approach for creating the initial partitioning relied on the fact that the coarsest graph was relatively small, and thus the amount of work required to create a partitioning was small. In this case, it is better to let several threads create initial partitionings via recursive bisection independently, avoiding synchronization overheads. However, this limited parallelism in the initial partitioning phase to the number of partitionings we wished to create.

Our new method instead conditionally chooses to split the threads into independent groups to reduce inter-core communication. If the coarsest graph is large enough with respect to the number of threads, the threads will cooperatively work together to create the initial bisection. The threads will then split into two groups and recursively partition each half of the graph.

However, if the size of the coarsest graph is small enough with respect to the number of threads, the threads then break up into several groups, and each group independently generates a partitioning of the graph. We create our groups based on thread IDs which we bind to CPU cores, with the goal of creating groups that do not cross processor boundaries, thus making good use of shared caches and minimizing communication distance.



(a) (b)

Figure 2: The different shades vertices are assigned to different threads. The original assignment is shown in (a), where vertices in the boundary of the same partition may be assigned to many different threads. The migrated assignment is shown in (b), where boundary vertices of each partition have been assigned to a single thread.

4.4 Boundary Migration

Vertices are statically assigned to threads during refinement, despite the load imbalance this can cause, for two reasons. First, is that we do multiple iterations of refinement, making it beneficial for threads to operate on the same set of boundary vertices in each iterations to promote data re-use. Second, is that the task size is exceedingly small, just a single vertex, and the overhead of task scheduling would dominate the runtime (using a larger task size would not guarantee that more than one vertex in a task would be on the boundary and require work).

In our previous work [14], threads performed refinement on the vertices they were assigned at graph generation (or input for the first level). This resulted in significant core-to-core communication. As seen in Figure 2a, boundary vertices can be scattered among threads. Any time a vertex is moved, vertices owned by other threads must be updated. Handling these updates asynchronously means a lot of time is wasted processing small messages from other threads during refinement and handling these updates in large batches or synchronously at the end of each iteration can result in extra work being performed in the form of suboptimal or discarded moves.

To address this issue, we introduce the notion of *boundary migration*. During the projection step of uncoarsening, we change the thread assignment of boundary vertices, so that rather than each thread owning vertices scattered throughout the boundary, each thread owns a relatively continuous chunk of boundary vertices as seen in Figure 2b. We change the assignment of only boundary vertices so as to minimize the cost of this migration. Partitions are assigned to threads via hashing, and the boundary vertices are migrated to the threads to which their partitions were assigned.

To perform this migration, we create t buckets to place vertices in, where each bucket corresponds to the partitions assigned to each of the t threads. Each thread counts the number of boundary vertices that it owns at the start of projection destined for each bucket. A global prefixsum is computed such that each thread knows the starting index at which to insert its boundary vertices into the buckets. This is then followed by the threads copying their boundary vertices into the buckets. Once all threads have finished copying their boundary vertices into the buckets, each thread then retrieves the boundary vertices in the bucket corresponding

Table 1: Graphs

Graph	Vertices	Edges	Max Deg
333SP	3,712,815	11,108,633	28
AS365	3,799,275	11,368,076	14
NLR	4,163,763	12,487,976	20
asia.osm	1,1950,757	12,711,603	9
hugetrace-00020	16,002,413	23,998,813	3
road_usa	23,947,347	28,854,312	9
Serena	1,391,349	31,570,176	248
audikw1	943,695	38,354,076	344
dielFilterV3real	1,102,824	44,101,598	269
delaunay_n24	16,777,216	50,331,601	26
europe.osm	50,912,018	54,054,660	13
Flan_1565	1,564,794	57,920,625	80
nlpkkt240	27,993,600	373,239,376	27
flickr	820,878	6,625,280	10,891
eu-2005	862,664	16,138,468	68,963
soc-pokec	1,632,803	22,301,964	14,854
wikipedia-2007.	3,566,908	42,375,912	187,671
soc-LiveJournal1	4,847,571	42,851,237	20,333
com-orkut	3,072,441	117,185,083	33,313
uk-2002	18,520,486	261,787,258	194,955

to the partitions it was assigned.

Throughout all iterations of the current level of refinement a thread is responsible for moving and updating the vertices which it was received in this process. When a vertex is pulled into the boundary, it is assigned to the that owns the partition in which it resides.

For the case where the number of threads is significantly less than the number of partitions, we assign multiple threads to a partition. We can then assign a vertex on the boundary in this partition to one of the partition's threads based on the opposing partition to which the vertex is most connected. When a vertex internal to a partition is pulled into the boundary (e.g., one of its neighbors was moved to another partition) it is assigned to one of the partition's threads via hashing. This hashing is done rather than assigning the vertex to the thread that pulled it into the boundary, as two or more threads may concurrently pull the same vertex into the boundary by moving its neighbors.

5. EXPERIMENTAL METHODOLOGY

The graphs used in our experiments are listed in Table 1. They are divided into two groups: those with normal degree distributions, and those with skewed degree distributions. The group of graphs with normal degree distributions are from the University of Florida Sparse Matrix Collection (UFSMC) [6]. These graphs are a combination of scientific meshes, road networks, and non-linear programming matrices.

The group of graphs with skewed degree distribution are from UFSMC and the Stanford Large Network Dataset Collection [16]. These are a combination of web and social networks. For networks that were originally directed, we converted them to undirected weighted graphs via $A + A^T$ (where A is the directed version of the adjacency matrix). These graphs with skewed degree distributions have maximum degrees several orders of magnitude larger than those with normal degree distributions.

The runtimes presented in the following sections are the mean of ten runs of the partitioners using different random seeds.

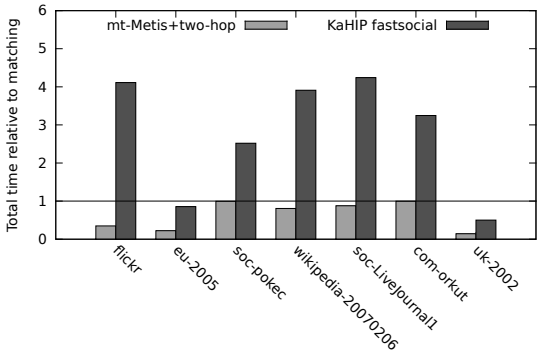


Figure 3: Two-hop matching and KaHIP's LP-based aggregation compared to mt-Metis, run serially and $k = 64$.

We used an Intel® Xeon®[1] E5-2699 v3 processor based system for the experiments. The system consists of two processors, each with 18-cores running at 2.3 GHz (a total of 36 cores) with 45 MB L3 cache and 64GB memory. The system is based on the Haswell microarchitecture and runs Redhat Linux (version 6.5). All our code is developed using C and is compiled using the GNU GCC version 4.8.3, using the -O3 optimization flag. For comparison we used KaHIP version 0.71*c* from `http://algo2.iti.kit.edu/documents/kahip/index.html`, PT-Scotch version 6.0.4 from `http://gforge.inria.fr/projects/scotch/`, and ParMetis version 4.0.3 from `http://cs.umn.edu/~metis`.

6. RESULTS

In this section we first evaluate our algorithmic improvements individually. We then evaluate net effect of our algorithmic improvements. We will refer to mt-Metis with these algorithmic improvements from Section 4 as mt-Metis-opt in the following experiments. Finally, we compare mt-Metis-opt to other parallel partitioners.

6.1 Coarsening

6.1.1 Aggregation

Figure 3 shows results of running mt-Metis serially with two-hop matching and KaHIP using the `fastsocial` configuration which uses size-constrained label propagation based aggregation. The runtimes are normalized to that of mt-Metis without two-hop matching running serially. As can be seen, allowing two-hop matching significantly reduces runtime, up to 7.0× for uk-2002, and a geometric mean for these seven graphs of 2.0×, as it allows the number of vertices in the graph to reduce by almost half at each level. This impacts not only the amount of work done in coarsening, but also the amount of work done in uncoarsening as well. The amount of time spent in initial partitioning is also reduced, as the size of the coarsest graph, G_s is smaller due to coarsening not exiting early.

The speedup from two-hop matching also brought with it an improvement in quality, decreasing the geometric mean of the number of cut edges by 3.2%. KaHIP's label propagation based aggregation allows it to detect a larger structures while coarsening, and does a better job leaving low-cut areas

[1] Intel, Xeon, and Intel Xeon Phi are trademarks of Intel Corporation in the U.S. and/or other countries.

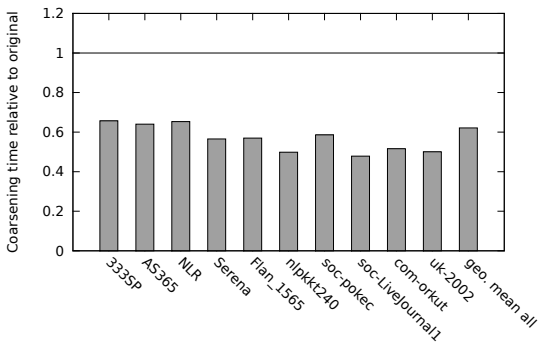


Figure 4: Coarsening runtime reduction due to optimizations (geometric mean for all 20 graphs), using 36 threads and $k = 64$.

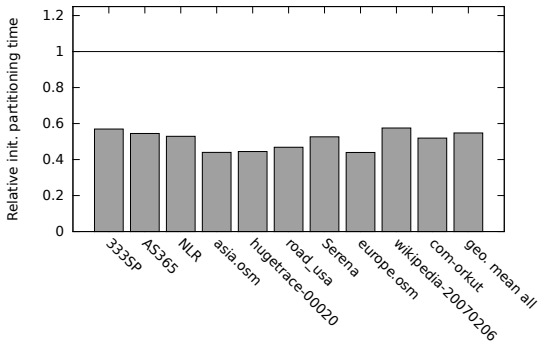


Figure 5: Cache oriented initial partitioning compared with independent initial partitioning (geometric mean for all 20 graphs), using 36 threads and $k = 64$.

of the graph uncontracted. This resulted in a 14.4% lower geometric mean edgecut for the seven graphs than two-hop matching. This is largely a result of KaHIP finding good partitionings of uk-2002 with half the edgecut of mt-Metis. The graph uk-2002 has a strong cluster structure (few inter-cluster edges), and being able to detect those edges was crucial to finding small cuts on this graph. However, the amount of work associated with just a few iterations of label propagation far exceeds that of matching (and two-hop matching), causing KaHIP to have a geometric mean runtime 4.4× higher than mt-Metis with two-hop matching.

6.1.2 *Implementation Level Optimizations*

The results of our coarsening optimizations are shown in Figure 4, for the ten graphs on which they had the largest impact. The geometric mean for all 20 graphs is shown on the right. Our optimizations resulted in a geometric mean speedup of 1.6× for all 20 graphs. Software prefetching resulted in large gains for the denser mesh-style graphs where we had a sufficient number of edges per vertex with which to look ahead. For the larger network style graphs, our two part contraction using both a hash table and a dense vector, played a large role in achieving near 2× speedups.

6.2 Initial Partitioning

The runtime of the new parallel formulation of initial partitioning for ten graphs is shown in Figure 5, and the geometric mean for all 20 graphs is shown on the right. The mean reduction in runtime was 45%, or a 1.8× speedup. While the semi-cooperative creation of initial partitionings means an increase in overhead, the increased parallelism more than made up for it. While the largest decrease in initial partitioning time was achieved on the sparse graphs as using all 36 threads did not result in cache conflicts, the largest impact on total running time was for the case where the coarsest graph was relatively dense. For example, for com-orkut which had a relatively dense coarsest graph, the total runtime decreased by 21%, and for Flan_1565, which had a relatively sparse coarsest graph, the total runtime decreased by less than 4%.

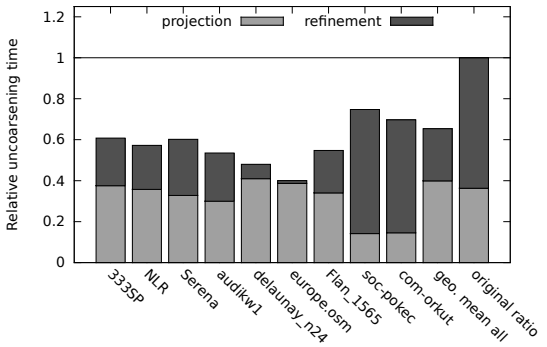


Figure 6: Uncoarsening runtime using boundary migration compared to static assignment (geometric mean is for all 20 graphs), using 36 threads and $k = 64$.

6.3 Uncoarsening

In Figure 6, we show the effects on runtime of migrating boundary vertices on ten of the graphs. The geometric mean for all 20 graphs is shown on the right. The time spent in refinement dramatically decreases when vertices are migrated, the geometric mean decreased by 2.5×. This is because the amount of updates that need to be communicated between threads dramatically decreases, and decisions regarding vertex movement are more likely to up to date and not be undone in a later iteration.

Projection however, because it now includes the time it takes to migrate the boundary vertices, had its geometric mean runtime increase by 10%. This changed the percentage of time spent in projection from making up 36% of uncoarsening, to 61%. The net effect of boundary migration reduced the geometric mean runtime of the uncoarsening phase when using 36 threads by 35%, and by up to 60% for the road network, europe.osm. Because road networks tend to have very sparse cuts, the cost of communication between threads plays a significant role in the runtime of refinement where there is little useful work done. By migrating boundary vertices (of which there are very few), we minimize this communication, which has a large impact on the runtime.

6.4 Overall Improvements

We present the net effects of our improvements in Figure 7, where we compare the runtime of our algorithmic improvements in mt-Metis-opt with mt-Metis. The geometric mean reduction in runtime was 49%, or a speedup of 1.96×.

For the 20 graphs a range of speedups of $1.5 - 11.7\times$ was observed. The top of this range was achieved on uk-2002. This is largely due to the improved coarsening from two-hop matching, but was also influenced by large gains from our coarsening optimizations and restructured initial partitioning. The geometric mean cut for the twenty graphs remained relatively unchanged with our algorithmic improvements (0.7% higher for mt-Metis-opt, due to higher edgecuts

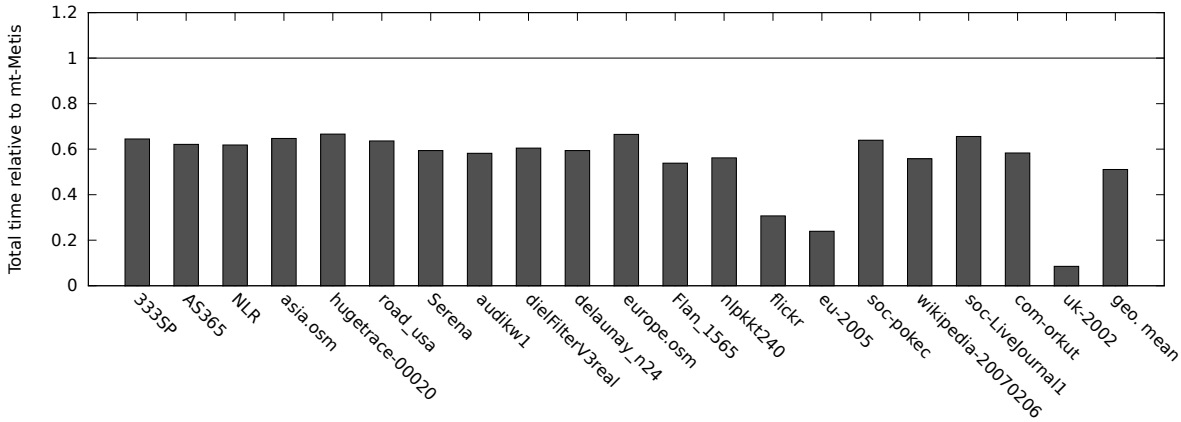


Figure 7: Comparison of runtime of mt-Metis-opt with mt-Metis, using 36 threads and $k = 64$.

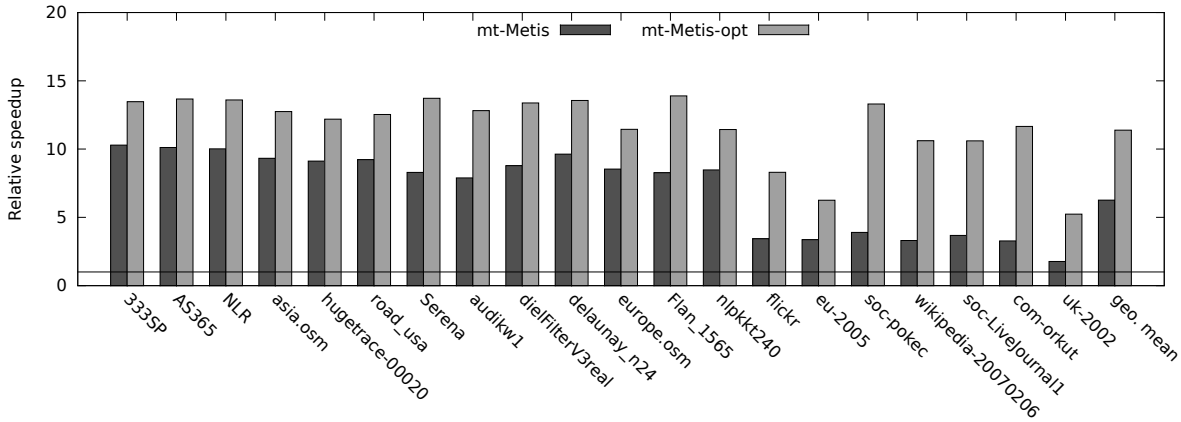


Figure 8: Comparison of strong scaling of mt-Metis-opt with mt-Metis, using 36 threads and $k = 64$.

on the road networks).

Our improvements not only made a significant difference in runtime, but also in terms of strong scaling, as shown in Figure 8. Where previously mt-Metis achieved a geometric mean speedup of 6.3× using 36 threads, held back in part by poor scaling on skewed degree distribution graphs, with our changes mt-Metis-opt scales to 11.4×. This is an improvement of 82%. For the skewed degree distribution graphs, two-hop matching help shift much of the runtime into the coarsening phase, which tends to scale the best as large amount of work per thread with little synchronization required. Furthermore, our changes to initial partitioning and uncoarsening, increased the scalability of the remaining time. This is evident when looking at the still substantial speedups for the graphs with non-skewed degree distributions.

In Figure 9, we compare the mt-Metis-opt with ParMetis and Pt-Scotch. The graphs graphs wikipedia-20070206 and uk-2002 are not included in this experiment as ParMetis and Pt-Scotch ran out of memory while attempting to partition them. A parallel version of the KaHIP partitioner has not been released, and as such we do not compare against it here.

The geometric mean runtime of mt-Metis-opt was 2.9× lower than ParMetis for the 18 graphs. Pt-Scotch's geometric mean runtime was 7.6× higher than that of ParMetis, largely due to its use of recursive bisection, which requires roughly $\log k$ iterations through the multilevel paradigm (which is six for $k = 64$).

The largest difference of runtime between mt-Metis-opt and the distributed partitioners was on the graph eu-2005, where mt-Metis-opt was 8.7× faster than ParMetis. This large difference in runtime was due to ParMetis' inability to coarsen the graph. ParMetis was forced to stop coarsening at 344, 515 vertices, where as mt-Metis-opt coarsened eu-2005 down to 6, 779 vertices before starting the recursive bisection in initial partitioning. The smallest difference was on the graph asia.osm, where mt-Metis-opt was 2.0× faster than ParMetis. This graph is very sparse with an average degree of just only slightly more than two, and has an extremely small boundary on 64-way partitions (only 0.01% of the vertices were on the boundary). As a result of these properties, 90% of the time was spent in coarsening and the projection step of uncoarsening, and our coarsening optimizations were targeted at graphs where the work associated with the edges was much greater than that of the work associated with the vertices. For the denser mesh-style graphs, dielFilterV3real and Flan_1565, mt-Metis-opt was 3.1× faster than ParMetis, largely due to our coarsening optimizations and the much smaller refinement time resulting from boundary migration.

7. CONCLUSION

Multilevel graph partitioning is a complex process, with several different sub-processes involving highly irregular access patterns. Achieving high performance on modern parallel architectures over a variety of inputs is a significant challenge. In this paper we presented several modifications

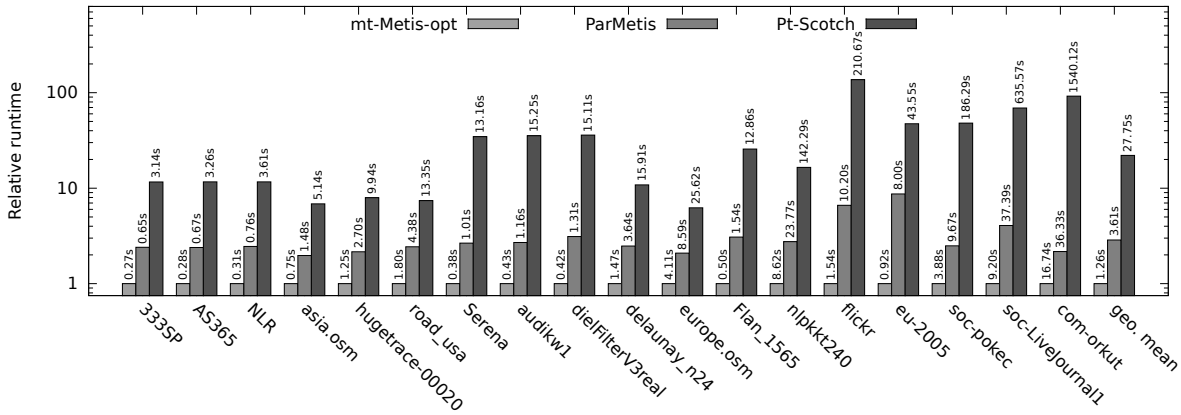


Figure 9: Comparison of modified mt-Metis with other partitioners, using 36 threads/processes and $k = 64$. Runtimes are relative with respect to the runtime of mt-Metis-opt. Absolute runtimes in seconds are shown above the corresponding bars.

to the shared-memory parallel graph partitioner mt-Metis. These modifications resulted in performance increases of 1.5–11.7×, and increased strong scaling by 82%, while preserving partition quality. Our modifications include an efficient method for performing two-hop matchings, a new parallel formulation of initial partitioning, a method for reducing communication during uncoarsening, and implementation level optimizations for coarsening.

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.

8. REFERENCES

[1] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 10–pp. IEEE, 2006.

[2] B. F. Auer and R. H. Bisseling. Graph coarsening and clustering on the gpu. *Graph Partitioning and Graph Clustering*, 588:223, 2012.

[3] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. *CoRR*, abs/1311.3144, 2013.

[4] Ü. V. Çatalyürek, M. Deveci, K. Kaya, and B. Ucar. Multithreaded clustering for multi-level hypergraph partitioning. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 848–859. IEEE, 2012.

[5] C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318–331, 2008.

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

[7] I. S. Duff and J. K. Reid. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 22(2):227–257, June 1996.

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

[9] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Labratories, 1993.

[10] B. Hendrickson and E. Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific Computing*, 20(2):468–489, 1998.

[11] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[12] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[13] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

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

[15] D. LaSalle and G. Karypis. Multi-threaded modularity based graph clustering using the multilevel paradigm. *Journal of Parallel and Distributed Computing*, 2014.

[16] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[17] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel graph partitioning for complex networks. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1055–1064, May 2015.

[18] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *Parallel and Distributed Systems, IEEE Transactions on*, 15(9):769–782, 2004.

[19] 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*, HPCN Europe 1996, pages 493–498, London, UK, UK, 1996. Springer-Verlag.

[20] P. Sanders and C. Schulz. Engineering multilevel graph partitioning algorithms. In C. Demetrescu and M. Halldórsson, editors, *Algorithms - ESA 2011*, volume 6942 of *Lecture Notes in Computer Science*, pages 469–480. Springer Berlin / Heidelberg, 2011.

[21] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135–148, 1991.

[22] X. Sui, D. Nguyen, M. Burtscher, and K. Pingali. Parallel graph partitioning on multicore architectures. In *Languages and Compilers for Parallel Computing*, pages 246–260. Springer, 2011.