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

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

George Karypis[†]
karypis@cs.umn.edu

[†]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. In this paper we present algorithmic improvements to the multithreaded graph partitioner mt-Metis. 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 [18] and locality maximization [15]. Modern methods rely on the multilevel paradigm [7] to find high quality solutions extremely fast [11, 16, 17]. While distributed-memory parallel partitioners [10, 4, 9] have been in use for almost two decades, methods [19, 3, 12] 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.

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 the performance of mt-Metis on modern architectures and graphs. Specifically, our contributions are: 1) An efficient two-hop matching scheme which works well on graphs with and without highly skewed degree distributions. 2) Implementation level coarsening optimizations. 3) An improved initial partitioning parallel formulation. 4) A method of performing parallel refinement that greatly reduces inter-core communication. 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.

IA^3 2015, November 15-20 2015, Austin, TX, USA

ACM ACM 978-1-4503-4001-4/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2833179.2833188

2. BACKGROUND

The balanced graph partitioning problem is defined as creating k disjoint sets of vertices (partitions) which minimizes the number of partition-spanning edges, with the constraint that partitions be of near equal weight.

2.1 Multilevel Graph Partitioning

The most prevalent strategy for developing graph partitioning heuristics has been the multilevel paradigm [7]. 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. [2] provide an overview of state-of-the-art graph partitioning techniques.

2.2 Multithreaded Graph Partitioning

In this work, we improve the performance of the mt-Metis [12] 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. In contraction, each thread constructs the portion of the graph it will own in the next level. Coarsening continues until either the size of the coarsest graph is within a multiple of the number of partitions, or the rate of coarsening slows below some threshold.

Once coarsening stops, several initial partitionings are created of the coarsest graph in parallel, and the best partitioning is selected.

Uncoarsening is made up of projection and refinement steps. In projection, the partition assigned to a coarse vertex is projected to its fine vertices. 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. ALGORITHMIC IMPROVEMENTS

Our algorithmic modifications encompassed all phases of the multilevel paradigm, including aggregation, contraction, initial partitioning, and refinement.

3.1 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 [6, 8] (to preserve sparsity in direct sparse methods) and graph clustering [1, 13].

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. We group 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. 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. 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 graph, no two-hop matching is performed. Otherwise, leaf vertices are matched together. If after matching leaf vertices, less than 75% of the vertices are matched, we then perform twin matching. Finally, if this still does not yield a sufficiently large matching (75%), we then match relatives. Finding all three classes can be done in $O(n \log n)$ time, but is often linear in the number of unmatched vertices.

3.2 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 [12], we used a hash table to accumulate values for each coarse vertex's adjacency list if the maximum degree of the graph was small, and used a dense vector when the maximum degree was large. 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.

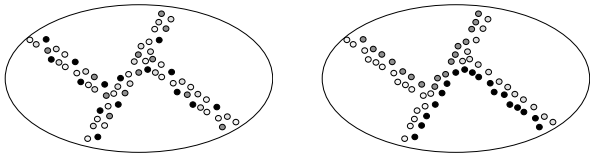
Instead, we do a pass over the coarse vertices to be generated and calculate an upper bound for the degree of each coarse vertex. We then contract all low degree vertices first using a hash table, followed by contracting high degree vertices using a dense vector.

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.

3.3 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 the past approach, several threads would create initial partitionings via recursive bisection independently, avoiding synchronization overheads. However, parallelism in the initial partitioning phase is then limited to the number of partitionings to be created.

Our new method instead conditionally splits the threads into independent groups to reduce inter-core communication, if the coarsest graph is small enough with respect to the number of threads. 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. 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.



(a) (b)

Figure 1: 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 new assignment is shown in (b), where boundary vertices of each partition have been assigned to a single thread.

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	11,950,757	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

3.4 Boundary Re-assignment

In our previous work [12], threads performed refinement on the vertices they were assigned at graph generation (or input for the first level) and are on the partition boundary. This can result in boundary vertices being scattered among threads as illustrated in Figure 1a. 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. Handling these updates in large batches 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 re-assignment*. 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 1b. We change the assignment of only boundary vertices so as to minimize the cost of this re-assignment. Partitions are assigned to threads via hashing, and the boundary vertices are re-assigned to the threads to which their partitions were 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 thread that owns the partition in which it resides.

4. 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) [5]. 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 [14]. These are a combination of web and social networks.

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

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. For comparing the effectiveness of the two-hop aggregation, we used KaHIP version 0.71*c* from `http://algo2.iti.kit.edu/documents/kahip/index.html`.

5. RESULTS

Two-hop matching significantly reduces runtime, up to 7.0× for uk-2002, and a geometric mean for the seven skewed-degree graphs of 2.0×, as it allows the number of vertices in the graph to reduce by almost half at each level. 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%. We also compared the effects of two-hop matching against KaHIP's label propagation based aggregation. While KaHIP finds partitionings with lower edgecuts on the graph with strong community structure, and had a lower geometric mean edgecut for the seven graphs by 14.4%, it had a geometric mean runtime for the seven graphs 4.4× higher than mt-Metis with two-hop matching.

Our coarsening optimizations resulted in a geometric mean speedup for the coarsening phase 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.

Our improvements to parallel refinement significantly reduced the runtime of the uncoarsening phase, a geometric mean decrease of 35%. The largest reduction in runtime was 60%, for the road network, europe.osm.

We present the net effects of our improvements in Figure 2, where we compare the parallel speedup of our algorithmic improvements in mt-Metis-opt with mt-Metis using 36 threads, relative to that of mt-Metis run serially. The geometric mean reduction in runtime was 49%, or a speedup of 1.96×, for out algorithmic improvements.

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 improve-

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

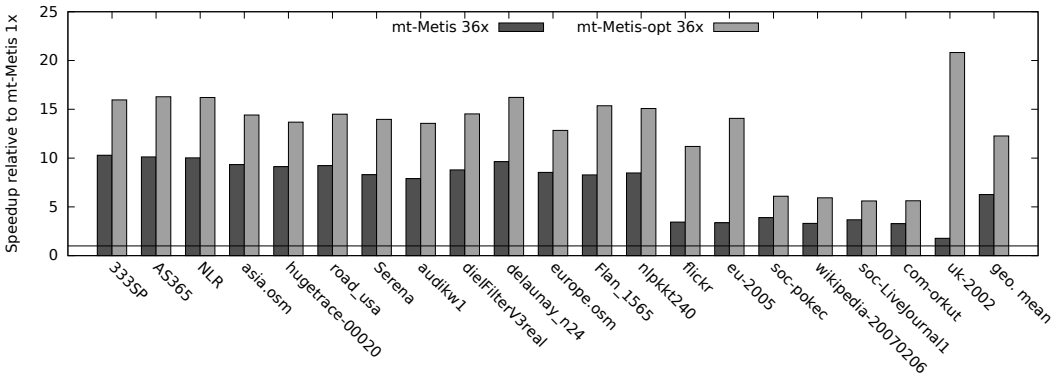


Figure 2: Comparison of runtime of mt-Metis-opt with mt-Metis, using 36 threads and $k = 64$.

ments (0.7% higher for mt-Metis-opt, due to higher edgecuts on the road networks).

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× (12.3× compared to mt-Metis). This is an improvement of 82%. For the skewed degree distribution graphs, two-hop matching shifts much of the runtime into the coarsening phase, which tends to scale the best as it has a large amount of work per thread with little synchronization required. Furthermore, our changes to initial partitioning and uncoarsening, increase the scalability of the remaining time. This is evident when looking at the still substantial speedups for the graphs with non-skewed degree distributions.

6. CONCLUSION

In this paper we presented several modifications to the shared-memory parallel graph partitioner mt-Metis. These modifications resulted in performance increases of $1.5 - 11.7\times$, 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.

7. REFERENCES

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

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

[3] 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.

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

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

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

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

[8] 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.

[9] 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.

[10] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. Supercomputing '96, Washington, DC, USA, 1996. IEEE Computer Society.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. HPCN, pages 493–498, London, UK, 1996. Springer-Verlag.

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

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

[19] 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.