GraSP: Distributed Streaming Graph Partitioning

Casey Battaglino
Georgia Institute of Technology
cjbattagl@gatech.edu

Robert Pienta
Georgia Institute of Technology
pientars@gatech.edu

Richard Vuduc
Georgia Institute of Technology
richie@gatech.edu

ABSTRACT

This paper presents a *distributed, streaming* graph partitioner, Graph Streaming Partitioner (GraSP), which makes partition decisions as each vertex is read from memory, simulating an online algorithm that must process nodes as they arrive. GraSP is a lightweight high-performance computing (HPC) library implemented in MPI, designed to be easily substituted for existing HPC partitioners such as ParMETIS. It is the first MPI implementation for streaming partitioning of which we are aware, and is empirically orders-of-magnitude faster than existing partitioners while providing comparable partitioning quality. We demonstrate the scalability of GraSP on up to 1024 compute nodes of NERSC's Edison supercomputer. Given a minute of run-time, GraSP can partition a graph three orders of magnitude larger than ParMETIS can.

Categories and Subject Descriptors

G.2.2 [**Mathematics of Computing**]: Discrete Mathematics—*Graph Algorithms*

General Terms

Theory

Keywords

graph partitioning, streaming graph partitioning

1. INTRODUCTION

We consider the problem of partitioning a power-law graph on a distributed memory system. Power-law graphs are ubiquitous in the real world, and arise particularly in social networks where data sizes are growing at enormous rates. As we will discuss, partitioning is a key step for algorithms that arise in applications such as fraud detection, bioinformatics, and social and information network analysis, among numerous others.

KDD '15 Sydney, Australia

The speed of data-mining algorithms on power-law graphs, at scale, is often limited by bottlenecks in network communication and load imbalance [16]. Partitioning is the common preprocessing step to find a mapping of the data to processors of the system that alleviates these two issues; in distributed computing the desired objective is generally the minimization of inter-partition edges (to minimize communication) subject to balanced partition size (to favor load balance).

Formally, we wish to partition the nodes of a graph into k balanced components with capacity $(1+\epsilon)\frac{N}{k}$, such that the number of edges crossing partition boundaries is minimized. Partitioning with these two requirements can be reduced to the minimum-bisection problem [9] and is therefore NP-Complete. Thus, computing an optimal mapping is generally computationally infeasible, and heuristic approaches are taken.

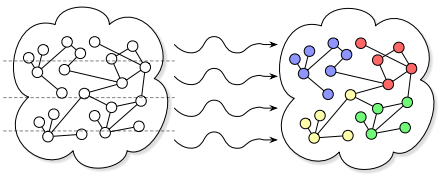

Figure 1: Parallel streaming partitioning.

To illustrate the role of partitioning on performance, consider a parallel Breadth-First Search (BFS), a central primitive for graph analysis where vertices are partitioned between two machines in a '1D' distribution [6]. During each BFS step, each process must communicate all newly explored target vertices to process that owns them. In Figure 2, if we have 4 processes, all 10 nonzeros in the non-diagonal blocks must be communicated at some point. A good partitioner concentrates nonzeros in the diagonal blocks, thereby reducing communication.[1]

Offline graph partitioning algorithms have existed for decades. They work by storing the graph in memory with complete information about the edges. Many variants of these algorithms exist [7] and range from spatial methods [10] to spectral methods [4]. Some of the most effective offline graph partitioners are multi-level partitioners, which recursively contract the graph to a small number of vertices, and then heuristically optimize the partitioning while expanding back

[1]Computing exact communication volume requires a hypergraph partitioner [8].

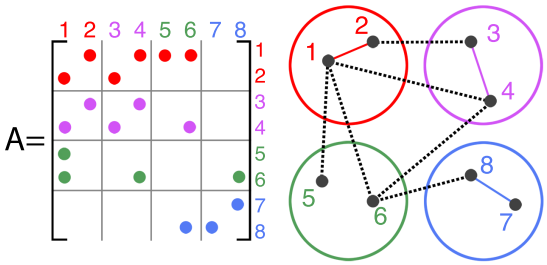

Figure 2: Graph 4-partition shown with corresponding adjacency matrix. The intra-partition edges are shown in their partition color, while inter-partition edges are shown as dotted black lines. Inter-partition edges or *cut-edges* result in additional network communication and lowered performance.

to the original graph [11]. These methods are especially effective on geometric graphs, that is, graphs that arise from some physical geometry, like the discretized finite element mesh of a physical object. Parallel multi-level partitioners will serve as the baseline comparison for our implementation.

Streaming Partitioning.

Streaming partitioning is the process of partitioning a graph in a single sweep, reading vertices and edges only once. Thus we incur $O(|V| + |E|)$ memory access, storage, and run time, with minimal overhead. Offline graph partitioners require the entire graph to be represented in memory, whereas streaming graph partitioning may process vertices as they arrive. This fits a model where input data arrive sequentially from a generating source (such as a web-crawler).

In an initial study, partitioning a 26 GB Twitter graph has been shown to take 8 hours using the fastest offline algorithms, and only 40 minutes with the FENNEL streaming partitioner, with similar partition quality [21]. This also suggests that we could do multiple, iterative passes of a streaming partitioner, all in a fraction of the time that an offline partitioner would take to terminate. This technique and its convergence properties have been explored by Nishimura and Ugander [18]. In this paper we demonstrate empirically that efficiently distributing this streaming partitioning process can reduce the run-time for problem of this magnitude to a matter of *seconds*.

Contributions.

We have developed GraSP, a fast, iterative, distributed streaming graph partitioner. It works by restreaming the distributed graph with tempered partition parameters to achieve a fast, parallel k-partitioning. When applied to scale-free graphs, GraSP attains an edgecut competitive with more sophisticated algorithms, but can operate on graphs multiple orders of magnitude larger within the same run-time.

For instance, ParMETIS takes at least 1 min to partition a Scale-21 RMAT graph (see § 3) on any number of compute nodes in our experiment, with run-time ballooning for larger scale graphs. GraSP performs a partitioning stream of a Scale-31 RMAT graph (with 1024 as many vertices and edges) on the same setup in under 20 seconds, with comparable edge-cut after 5-10 restreams.

GraSP operates on a distributed CSR graph representation, the same data structure used by ParMETIS, and can therefore be easily substituted in high-performance codes.

2. METHODOLOGY

While there are many possible heuristics for streaming partitioning [20], the most effective by far have been *weighted, greedy* approaches. We keep track of the partition assignments of vertices streamed so far (P_i^t for each process i at time t). As each vertex v is streamed, we count the edges from that vertex to each partition $|P_i^t \cap N(v)|$. This intuitively maximizes *modularity*, the ratio of intra-partition edges to inter-partition edges. However, using this value on its own would result in all vertices being assigned to a single, large partition. Thus, we exponentially *weight* the edge counts by the size of partitions $|P_i^t|$, relatively dampening the scores for partitions that are too large (but penalizing only lightly for small differences in size). This gives us two parameters: the linear importance of partition size to the score, α, and the exponential rate at which increasing partition size incurs a greater penalty, γ. This yields the basic 'FENNEL' algorithm [21] shown in Algorithm 1.

Set all P_i to $\emptyset$;
foreach $v \in V(G)$ *as it arrives at time* t **do**
  $j \leftarrow \underset{i \in \{1,\ldots,p\}}{\operatorname{argmax}} |P_i^t \cap N(u)| - \alpha\frac{\gamma}{2}|P_i^t|^{\gamma-1}$;
  Add v to set P_j^{t+1};
end

Algorithm 1: Serial streaming FENNEL partitioner

Exact computation of this algorithm as described is not possible in parallel, because P_i^{t-1} must be known to compute P_i^t. A multi-threaded approximation of this algorithm is easily performed by relaxing this requirement and using P_i^{t-p} to compute P_i^t, where p is the number of threads. This resulted in only a small drop in partition quality in our experiments.

To compute this algorithm in distributed memory, a naive approach is to constantly broadcast partition assignments as they are computed. Unless we use synchronization (which involves a massive performance reduction), this results in a drastic drop in partition quality, because the latency across a network is high enough that partition assignments are perpetually out of date.

Our implementation follows the methodology of 'restreaming partitioning' [18], which shows the single-pass algorithms of FENNEL and WDG [20, 21] can be repeated over the same data in the same order, yielding a convergent improvement in quality. This approach has other benefits that we utilize:

- Partition data is only communicated between streams, yielding high parallelism.
- Parameters (α, γ) can be 'tempered' to achieve higher-quality, balanced results that avoid immediate global minima.

2.1 GraSP

GraSP operates on a distributed graph G in distributed CSR format. We take as input the parameters α, γ, the number of partitions p (assumed to be equal to the number

of MPI processes), the number of re-streams n_s, and the 'tempering' parameter t_α. GRASP then performs n_s iterative passes over the graph (in identical random order), multiplicatively increasing the balance parameter by t_α with each pass. This promotes a high-quality, but less-balanced partition early on, while further promoting balance with each subsequent pass [18].

Between each pass, the partition information is communicated across all processors using the MPI ALLGATHER primitive, which is often optimized for a given network architecture. The pseudocode for GRASP is shown in Algorithm 2.

for *each process* p **do in parallel**
  $vorder \leftarrow rand\_perm(\{0, \ldots, |V(G_{local})|\})$;
  Randomly assign local vertices to partitions $P^0_{i,p}$;
end
for $run \leftarrow \{1 \ldots n_s\}$ **do**
  for *each process* p **do in parallel**
    foreach $v \in vorder$ **do**
      $j \leftarrow \underset{i \in \{1,\ldots,p\}}{\text{argmax}} |P^t_{i,p} \cap N(u)| - \alpha \frac{\gamma}{2} |P^t_{i,p}|^{\gamma-1}$;
      Add v to set $P^{t+1}_{j,p}$;
    end
  end
  MPI_ALLGATHER global partition assignments;
  $\alpha \leftarrow t_\alpha \alpha$
end

Algorithm 2: Parallel Restreaming performed by GRASP.

This method is illustrated graphically in Figure 2.1. In practice, we store the partitioning in a single compressed array, updating partition assignments in-place while storing a running count of the partition sizes. To increase accuracy we occasionally recompute global partition sizes using the MPI_ALLREDUCE primitive.

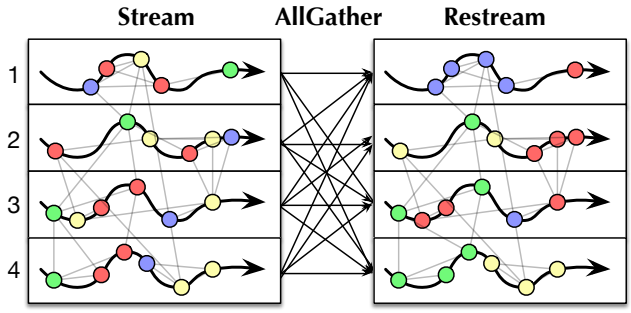

Figure 3: Two parallel restreaming steps on four processes.

Each process computes $\mathcal{O}\left(n_s \cdot \frac{|E|+|V|}{p}\right)$ work, and the network incurs a time of $n_s \cdot T_{allgather}(|V(G)|)$.

We can determine n_s by restreaming until some criteria is satisfied (either that we have encountered a local minimum, or we have achieved a good tradeoff between balance and edge-cut), or by choosing a number of restreamings and setting the tempering parameter t_α so that we achieve perfect balance within that number. In our experiments, we generally see good partitions within 10 restreams.

Table 1: Basic properties of graphs in SNAP data set [14], and λ for one pass. $\lambda_{r,2} = 0.5, \lambda_{r,8} = 0.87$

Data Set	N	nnz	$\lambda_{p=2}$	$\lambda_{p=8}$
soc-LiveJournal	4,847,571	68,993,773	0.234	0.463
as-Skitter	1,696,415	22,190,596	0.166	0.324
cit-Patents	3,774,768	16,518,948	0.402	0.726
roadNet-CA	1,971,281	5,533,214	0.186	0.360
web-Google	916,428	5,105,039	0.189	0.336
wiki-Talk	2,394,385	5,021,410	0.411	0.752
amazon0302	262,111	1,234,877	0.202	0.370
soc-Slashdot0902	82,168	948,464	0.236	0.382
ca-AstroPh	18,772	396,160	0.232	0.413
cit-HepPh	34,546	421,578	0.343	0.646
email-EuAll	265,214	420,045	0.280	0.538
Oregon-1	11,492	46,818	0.224	0.406
p2p-Gnutella04	10,879	39,994	0.415	0.747

3. EVALUATION

We ran our distributed experiments on a subset of the Edison machine at NERSC, featuring 5576 compute nodes with two 12-core Intel "Ivy Bridge" processors per node and a Cray Aries interconnect.

We evaluate GRASP by its runtime as well as the quality of the partition that it produces, which we measure with *fraction of cut edges* λ.

$$\lambda = \frac{\text{Number of edges cut by partition}}{\text{Total number of edges}} \quad (1)$$

where lower numbers represent a higher degree of locality. We can compare this to our baseline, the expected quality of a random $k-$partition, $\lambda_r = \frac{k-1}{k}$. Any partitioner that produces partitions with $\lambda < \lambda_r$ has improved the parallel locality of the partitions.

Balance is also an important metric in partitioning. Our basic metric for balance is the number of vertices in the largest partition divided by the number of vertices in the smallest partition, and we design our restreaming framework to perform a tempered restream until balance is within a decent tolerance (≈ 1.2).

3.1 Test Graphs

We measure our approach with both synthetic and real-world graphs. While synthetic graphs make for excellent scalability experiments, demonstration on real-world networks is important to verify that the partitioner works well in practice.

3.1.1 *Real-world Graphs*

The SNAP dataset is a collection of real-world networks collected by Leskovec and collaborators [2, 14]. Many networks in this collection are power-law and scale-free representatives of social networks (such as collaboration networks, citation networks, email networks, and web graphs). We consider these to be excellent representative networks for a variety of domains. It is these types of networks that will continue to increase in size in the years to come. We ran GRASP on a representative selection of these graphs, and outline the results in Table 1 and in § 3.3.

3.1.2 *Synthetic Graphs*

For scalability experiments we generated random undirected power-law Kronecker (RMAT) graphs of varying scale in parallel using the Graph500 Reference implementation [1].

Table 2: Edge and vertex counts for generated RMAT graphs of each scale.

Scale	26	27	28	29	30	31
|V(G)|	67M	134M	268M	537M	1.07B	2.15B
|E(G)|	1.07B	2.14B	4.29B	8.58B	17.1B	34.3B

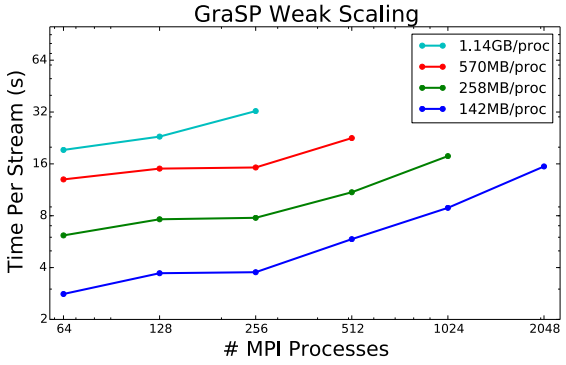

Figure 4: Per-stream times of GraSP in a weak-scaling experiment. This demonstrates that we can scale to very large problem sizes without network overhead dominating the runtime.

Kronecker graphs are commonly used in HPC graph benchmarks and testing, and arbitrarily large instances can be efficiently generated in parallel. The *scale* of an RMAT graph is equal to $\log |V(G)|$, and the edge-factor is the average number of edges per node, which we hold constant at 16. Vertex and edge counts for the scales we experiment on are shown in Table 2.

3.2 Scalability

3.2.1 *Weak Scaling*

Weak-scaling holds the amount of data per process constant as we increase the number of processes. In our experimental setup we achieve this by doubling the number of MPI processes every time we increase the scale of the RMAT generator. This yields the per-stream timing experiments in Figure 4, where each line is labeled with the size of data per process:

This demonstrates that, for a reasonable number of MPI processes, we can scale up our problem sizes without encountering wasteful overhead from the network.

3.2.2 *Strong Scaling*

In strong-scaling, the size of the data is fixed while the number of processes inreases. Strong-scaling is heavily penalized by serial portions of code (as dictated by Amdahl's law) and growing network overhead. GraSP exhibits a high degree of parallelism, illustrated in Figure 5.

While ParMETIS can't execute in a reasonable time on the problem sizes we demonstrate for GraSP, we show a small strong-scaling experiment in Table 4.

Performance inevitably plateaus for GraSP as local problem sizes become small in the face of increasing network overhead. However, for smaller degrees of parallelism we demonstrate near-linear scaling.

Table 3: Weak scaling results for ParMETIS on RMAT graphs, with 2^{18} vertices per compute node.

#procs	8	16	32	64	128
Time (s)	5.01	10.2	25.0	64.0	167.0

Table 4: Comparison of run-time and partition quality between ParMETIS and GraSP for a Scale-22 RMAT graph.

#procs	λ_{metis}	λ_{grasp}	t_{metis}(s)	$t_{grasp}(s)$
8	0.36	0.29	307.8	0.72
16	0.38	0.41	221.9	0.45
32	0.40	0.54	194.9	0.31

3.3 Quality

In Table 1 we show some properties of our real test-graphs, as well as the performance of our streaming partitioner on them, for $p = 2$ and $p = 8$ partitions..

We confirm the validity of the restreaming approach on the SNAP data sets for the two values of p in Figs. 6 and 7, respectively. The tradeoff between vertex balance and partition quality for a large scale GraSP computation is demonstrated in § 3.4.

In a direct comparison to ParMETIS, Figure 4, demonstrates that GraSP finds comparable partition quality in a small fraction of the time, although it computes a worse edge-cut than ParMETIS when partitioning a small graph into a large number of partitions.

3.4 Analysis

Our scalability tests have demonstrated that GraSP is highly parallel and performs quality partitions far faster than more sophisticated algorithms. A single stream over a 34 billion edge, 2.1 billion node network can be done in just 15 seconds. Performing a constant number of restreams while tempering the balance parameter allows us to find a good tradeoff between partition balance and partition quality.

Partitions of power-law graphs are known to involve such a tradeoff [13]. Continuously better cuts can be found as

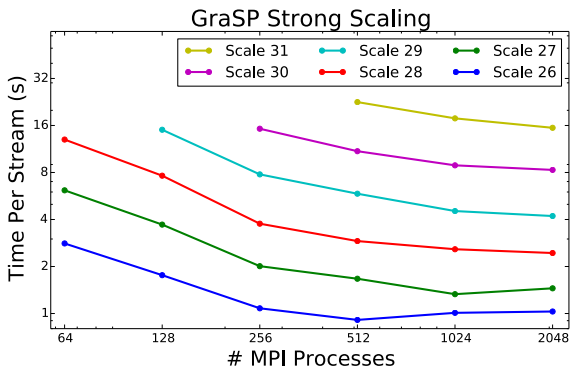

Figure 5: Per-stream times of GraSP for various strong-scaling data sizes. For instance, we can perform a single partitioning pass over a 34 billion edge, 2.1 billion node network in just 15 seconds.

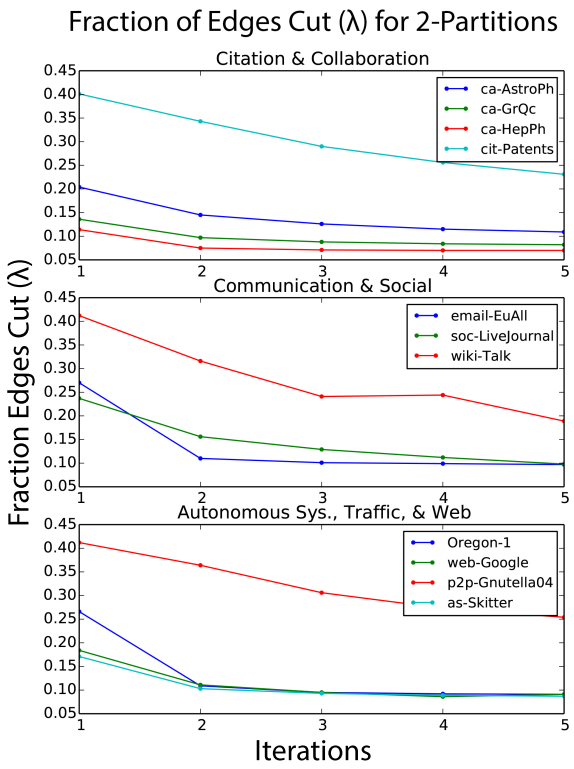

Figure 6: Improvement in the edges cut (λ) over 5 passes for bi-partitions of each graph. Because there are only two partitions, the algorithm is able to quickly fix mistakes it made in the initial partitioning. Many of the errors made in the first pass are fixed in the second iteration, with diminishing improvement thereafter.

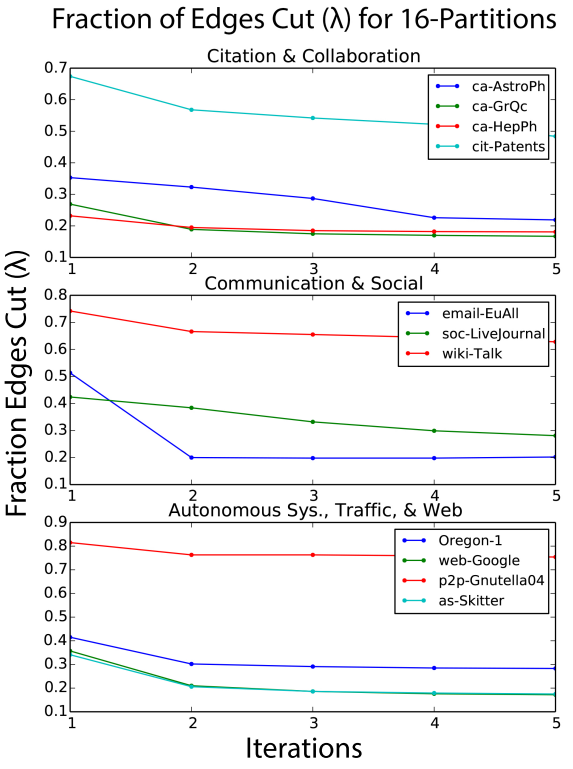

Figure 7: Improvement in edges cut (λ) over 5 passes for 16-partitions of each graph. Dividing the graph into 16 partitions makes the minimum edge cut problem much more challenging. Similar to the bi-partition results, we experience the best gain in the second pass and less in subsequent passes.

we relax our requirements for vertex balance. To illustrate this, we show the tempering process of GraSP computing on a Scale-28 RMAT graph on 64 processes. In Fig. 8 we show how partition balance and λ change as we continue to restream the graph. We begin with a random ordering (which tends towards perfect balance and worst-case quality λ_r). Balance immediately worsens, albeit with excellent partition quality, and then the tempering process increases balance at the expense of higher edge-cut. Eventually we reach a point within the balance tolerance and terminate.

In Figure 9 we illustrate the tradeoff curve inherent in this process.

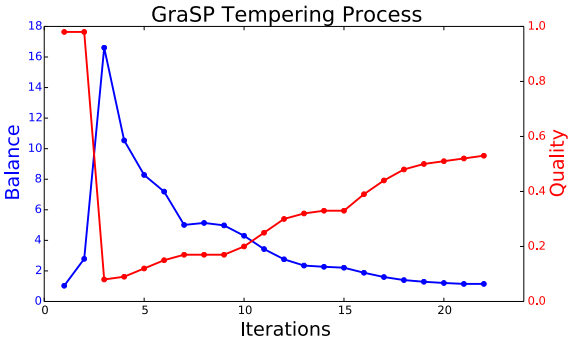

Figure 8: Time-series of tempering process on a Scale-28 RMAT graph on 64 MPI processes, beginning from a random partition. Lower quality is better, while the optimal balance is 1.

4. RELATED WORK

Partitioning is an important step in many algorithms. In HPC applications ranging from simulation to web analytics, the quality of partitions can strongly affect the parallel performance of many algorithms. Partitioning can also be used to identify community structure. We mention here a small sample of contemporary work in graph partitioning.

Streaming partitioning for a variety of heuristics was first presented by Stanton and Kliot [20], the Weighted Deterministic Greedy approach generalized by Tsourakakis, et. al [21], and the benefits of restreaming for convergence and parallelism determined by Nishimura and Ugander [18], although large-scale parallel experiments and benchmarks were not performed. Our implementation is the first parallel HPC-oriented study that we are aware of.

Streaming partitioning has been successfully adapted for edge-centric partitioning schemes like X-Stream [19]. X-Stream uses edge partitioning, to streams edges rather than vertices, which takes advantage of increased sequential memory access bandwidth.

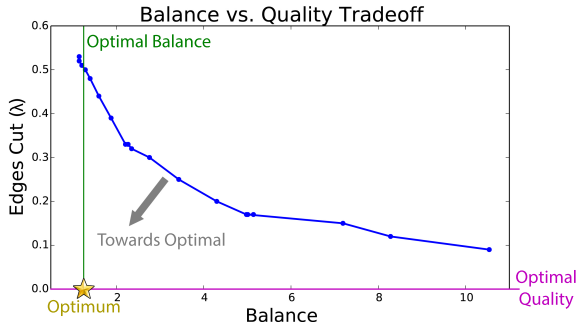

Figure 9: Tradeoff between node balance and edge-cut (of a 64-partition) encountered during tempering process.

A survey by Buluç, et. al [7] provides an excellent overview of conventional HPC graph partitioners, from spectral to spatial. Boman, et. al show how conventional graph partitioning can be used to optimize distributed SpmV [5]. However, recent approaches to scale conventional multi-level partitioners to billion-node graphs can still take hours [23]. Streaming partitioners on the other hand have attracted attention in the field of dynamic Graph Databases. For networks with dynamic structure, iterative approaches can dynamically adjust the partitions to suit changing graph structure. Vaquero et al. propose a method for iteratively adjusting graph partitions to cope with changes in the graph, using only local information [22]. This work demonstrated the power and scalability of leveraging local data to improve partition quality, especially to reduce the edges cut. "Sedge," or Self Evolving Distributed Graph Management Environment also takes advantage of dynamically managing and modifying partitions to reduce network communication and improve throughput [24].

Frameworks like Pregel [17], make use of hashing-based partition schemes. These allow constant-time lookup and prediction of partition location based on only the vertex ids. GraphLab [15] also uses a hashed, random partitioning method, which essentially produces a worst-case edgecut (λ_r), but which has the benefit that $H(v)$ can be called at any time to return the compute node that owns v. Khayyat et al. showed that a preprocessed partitioning of large-scale graphs is insufficient to truly minimize network communication [12]. They propose another dynamic partition approach that allows vertex migration during runtime to maintain balanced load.

5. CONCLUSION

In this work, we demonstrated GRASP, a *distributed, streaming* partitioner.

While Power-Law graphs are considered to be very difficult to partition [3], we have demonstrated that a very simple, fast algorithm is capable of significantly reducing communication in their parallel computation. Using the methodology outlined by Nishimura and Ugander [18] and applying an HPC framework we have scaled the partitioning process to graphs with billions of nodes in a matter of seconds, while more sophisticated graph partitioners struggle on graphs that are orders of magnitude smaller.

We have demonstrated our implementation on both real world and high-scale synthetic graphs on a leading supercomputer. GRASP is scalable and can partition a graph of 34.3 billion edges in 15 seconds, while maintaining partition quality comparable to what competing implementations achieve on smaller-scale graphs.

References

[1] Graph 500. http://www.graph500.org/. Accessed: 2015-03-30.

[2] Snap networks. http://snap.stanford.edu/data/index.html. Accessed: 2015-03-30.

[3] A. Abou-Rjeili and G. Karypis. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 124–124, Washington, DC, USA, 2006. IEEE Computer Society.

[4] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. *Journal of the ACM (JACM)*, 56(2):5, 2009.

[5] E. G. Boman, K. D. Devine, and S. Rajamanickam. Scalable matrix computations on large scale-free graphs using 2d graph partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 50:1–50:12, New York, NY, USA, 2013. ACM.

[6] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 65:1–65:12, New York, NY, USA, 2011. ACM.

[7] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Technical Report*. November 2013.

[8] U. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. Parallel Distrib. Syst.*, 10(7):673–693, July 1999.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[10] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. In *In Proceedings of International Parallel Processing Symposium*, pages 418–427, 1995.

[11] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[12] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 169–182, New York, NY, USA, 2013. ACM.

[13] K. Lang. Finding good nearly balanced cuts in power law graphs. Technical report, 2004.

[14] J. Leskovec. Stanford Large Network Dataset Collection.

[15] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[16] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[17] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, and G. Inc. Pregel: A system for large-scale graph processing. In *In SIGMOD*, pages 135–146, 2010.

[18] J. Nishimura and J. Ugander. Restreaming graph partitioning: Simple versatile algorithms for advanced balancing. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1106–1114, New York, NY, USA, 2013. ACM.

[19] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[20] I. Stanton and G. Kliot. Streaming graph partitioning for large distributed graphs. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '12, pages 1222–1230, New York, NY, USA, 2012. ACM.

[21] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. Fennel: Streaming graph partitioning for massive scale graphs. 2012.

[22] L. Vaquero, F. Cuadrado, D. Logothetis, and C. Martella. Adaptive partitioning for large-scale dynamic graphs. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 35:1–35:2, New York, NY, USA, 2013. ACM.

[23] L. Wang, Y. Xiao, B. Shao, and H. Wang. How to partition a billion-node graph. Technical Report MSR-TR-2013-102, February 2013.

[24] S. Yang, X. Yan, B. Zong, and A. Khan. Towards effective partition management for large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 517–528, New York, NY, USA, 2012. ACM.