POSTER: GPOP: A cache and memory-efficient framework for Graph Processing Over Partitions

Kartik Lakhotia[†], Rajgopal Kannan[‡], Sourav Pati[†], Viktor Prasanna[†]
{[†]University of Southern California, [‡]Army Research Lab-West }, Los Angeles, CA, USA
{klakhoti,rajgopak,spati,prasanna}@usc.edu

Abstract

Graph analytics frameworks, typically based on Vertex-centric or Edge-centric paradigms suffer from poor cache utilization, irregular memory accesses, heavy use of synchronization primitives or theoretical inefficiency, that deteriorate overall performance and scalability. In this paper, we generalize the partition-centric PageRank computation approach [1] to develop a novel Graph Processing Over Partitions (GPOP) framework that enables cache-efficient, work-efficient and scalable implementations of several graph algorithms. For large graphs, we observe that GPOP is upto 19× and 6.1× faster than Ligra and GraphMat, respectively.

1 Introduction

Shared-memory platforms are a popular choice for graph analysis as they offer significantly lower communication overhead compared to distributed systems. However, Graph computations are characterized by large communication volume and irregular access patterns that make it challenging to efficiently utilize the resources even on a single machine. The conventional push-pull vertex-centric processing [3] generates fine grained random accesses that decrease the utility of wide memory buses and deterministic caching features of new architectures. Contrarily, approaches targeting locality and streaming accesses require traversing all vertices/edges of the graph and are not work-efficient for algorithms with dynamic active vertex sets, such as BFS and Nibble algorithm.

To this purpose, we develop the Graph Processing over Partitions (GPOP) framework that comprehensively targets the issues of cache, memory communication and work-efficiency. The major contributions of our work are as follows:

1. We propose the GPOP framework with novel optimizations that (a) improves cache performance, (b) achieves high DRAM bandwidth, (c) minimizes the use of synchronization primitives and, (d) *guarantees* work efficiency of a given algorithm.
2. GPOP provides an easy to program set of APIs allowing selective continuity in frontiers across iterations. This functionality is essential for many algorithms such as Nibble, Heat Kernel PageRank etc., but is not supported intrinsically by the current frameworks.

This work is supported by DARPA under Contract Number FA8750-17-C-0086, NSF under Contract Numbers CNS-1643351 and ACI-1339756 and AFRL under Grant Number FA8750-18-2-0034.

PPoPP '19, February 16–20, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6225-2/19/02.
https://doi.org/10.1145/3293883.3299108

2 Graph Processing Over Partitions

GPOP divides the vertex set into *cacheable* disjoint partitions and implements each iteration of an algorithm in 2 phases:

1. Inter-partition communication (*Scatter*) → send vertex data to other partitions in the form of messages. A message also contains adjacency information enlisting the destination vertices who would consume the vertex's data.
2. Intra-partition updates (*Gather*) → process received messages to update state or value of vertices in the partition.

GPOP ensures high *temporal locality* by reusing partition's vertex data which is designed to fit in private cache of a core. At the same time, it also enjoys high *spatial locality* of message reads and writes by storing them in consecutive locations in per-partition memory spaces called *bins*. Since GPOP parallelizes computation over partitions, we create at least $4t$ partitions (where t is the number of threads) to ensure good load balance with dynamic scheduling.

Apart from generality, GPOP also ensures theoretical work-efficiency of processing an iteration. This is unlike PageRank computation in [1] which requires traversing all edges of the graph in every iteration. GPOP chooses between the following two scatter modes based on an analytical model that evaluates the tradeoff in work done per active edge versus maximizing main memory performance:

Source-centric (SC) mode: When the number of active edges are small, messages are only generated from active vertices. 1) Messages from a given vertex are generated before processing the next vertex. 2) Successive messages to any partition q from active vertices in partition p are written to contiguous addresses in the bin corresponding to $p \rightarrow q$. This enables efficient data packing in the cache lines.

SC mode is optimal in terms of work done. However, a thread will switch partitions (bins) being written into hurting the sustained memory bandwidth and overall performance.

Partition-centric (PC) mode: All vertices in the partition are scattered. To ensure communication-efficiency, all messages destined to a given partition are generated consecutively and written to contiguous memory locations, without being interleaved with messages to any other partition. In this mode, the order of message generation always stays the same and adjacency information written once can be *reused* across multiple iterations to reduce communication volume.

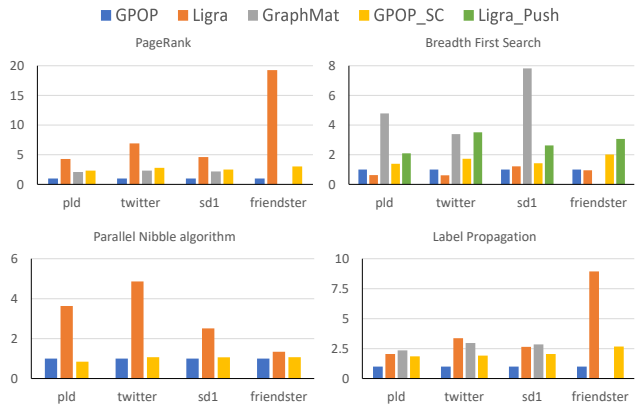

Figure 1. Comparison of Execution time (normalized with GPOP runtime) for various Graph algorithms

The Gather phase enjoys high temporal locality by caching vertex data to be updated and sequential memory bandwidth by reading messages in a bin consecutively. Moreover, during both the phases, any given partition is exclusively processed by one thread which enables *atomic* and *lock-free* communication and computation.

In particular, GPOP achieves sequential access bandwidth for processing partitions with large number of active edges while slightly increasing the work. However, the analytical models and internal data structures of GPOP ensure that in all cases, work done is always within a *constant* (predetermined) factor of the number of active edges of a partition.

Finally, GPOP is designed with programmability in mind. It abstracts away underlying parallelism and programming model details from the user. GPOP's unique and easy to program APIs enable selective continuity of active frontier across iterations, which is required by many graph algorithms (such as Nibble algorithm for personalized PageRank). For a detailed description of GPOP's system optimizations and programming interface, we refer the readers to [2].

3 Evaluation

We conduct experiments on a dual-socket Broadwell server with two 18-core processors and 256 KB L2 cache per core. We evaluate the performance of GPOP using 4 algorithms on large graphs (upto $2.6B$ edges): PageRank, BFS, Label Propagation and parallel Nibble algorithm.

Figure 1 compares the execution time of GPOP against two most popular frameworks - Ligra [3] and GraphMat [4]. To explore the overall benefit of dual communication modes in GPOP (section 2), we also measure the runtime with only SC mode scatter (GPOP_SC).

GPOP consistently outperforms GraphMat for all the algorithms by $2\times - 6.1\times$ speedup. Except BFS, GPOP executes all other algorithms faster than Ligra achieving upto $19\times$ speedup for PageRank. The speedup is more substantial for large graphs such as *Friendster* where the cache and memory bandwidth optimizations of GPOP become extremely crucial for performance. For PageRank and Label Propagation, GPOP is also $1.8\times - 3.4\times$ faster than GPOP_SC.

In case of BFS, direction optimization in Ligra enables early termination of adjacency list iterators, reducing the number of edges traversed. GPOP and GraphMat do not support pull direction processing and traverse all active edges. However, GPOP is still $0.61\times - 0.95\times$ as fast as Ligra. For comparison, we also show that BFS in Ligra without direction optimization (Ligra_Push) is upto $3.1\times$ slower than GPOP.

In parallel Nibble, the algorithm explores very few vertices in local neighborhood of the seed set. The frontiers are also small and GPOP is unable to utilize the PC mode for high memory performance. Consequently, both GPOP and GPOP_SC provide similar performance for Nibble algorithm.

Scalability: We evaluate the scalability of GPOP using PageRank and BFS on synthetic graphs with size ranging from $\{rmat22, |E| = 64M\}$ to $\{rmat27, |E| = 2048M\}$. GPOP demonstrates good scalability for BFS, achieving upto $17.9\times$ speedup over single thread. In case of PageRank, GPOP achieves upto $10.5\times$ speedup with 36 threads and scales poorly after 16 threads. This is because PageRank always uses PC mode scatter and nearly saturates the bandwidth with ≈ 20 threads.

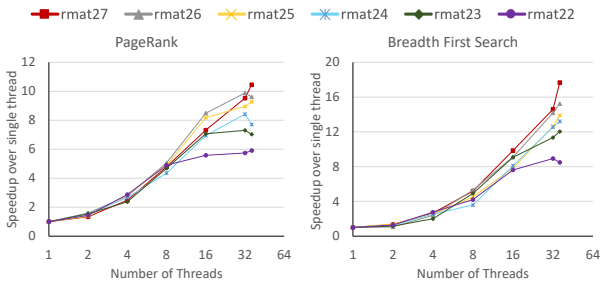

Figure 2. Strong scaling of GPOP for PageRank and BFS

4 Conclusion

In this paper, we presented the GPOP framework for cache and memory efficient graph processing. We experimentally demonstrated the scalability of GPOP and its performance benefits over Ligra and GraphMat frameworks.

References

[1] Kartik Lakhotia, Rajgopal Kannan, and Viktor Prasanna. 2018. Accelerating PageRank using Partition-Centric Processing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.

[2] Kartik Lakhotia, Sourav Pati, Rajgopal Kannan, and Viktor Prasanna. 2018. GPOP: A cache-and work-efficient framework for Graph Processing Over Partitions. *arXiv preprint arXiv:1806.08092* (2018).

[3] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, Vol. 48. ACM, 135–146.

[4] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1214–1225.