usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Accelerating PageRank using Partition-Centric Processing

Kartik Lakhotia, *University of Southern California;* Rajgopal Kannan, *US Army Research Lab;* Viktor Prasanna, *University of Southern California*

https://www.usenix.org/conference/atc18/presentation/lakhotia

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC ’18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

Open access to the Proceedings of the
2018 USENIX Annual Technical Conference
is sponsored by USENIX.

Accelerating PageRank using Partition-Centric Processing

Kartik Lakhotia[1], Rajgopal Kannan[2], Viktor Prasanna[1]
[1]Ming Hsieh Department of Electrical Engineering, University of Southern California
[2]US Army Research Lab
[1]{klakhoti, prasanna}@usc.edu, [2]rajgopal.kannan.civ@mail.Mil

Abstract

PageRank is a fundamental link analysis algorithm that also functions as a key representative of the performance of Sparse Matrix-Vector (SpMV) multiplication. The traditional PageRank implementation generates fine granularity random memory accesses resulting in large amount of wasteful DRAM traffic and poor bandwidth utilization. In this paper, we present a novel Partition-Centric Processing Methodology (PCPM) to compute PageRank, that drastically reduces the amount of DRAM communication while achieving high sustained memory bandwidth. PCPM uses a Partition-centric abstraction coupled with the Gather-Apply-Scatter (GAS) programming model. By carefully examining how a PCPM based implementation impacts communication characteristics of the algorithm, we propose several system optimizations that improve the execution time substantially. More specifically, we develop (1) a new data layout that significantly reduces communication and random DRAM accesses, and (2) branch avoidance mechanisms to get rid of unpredictable data-dependent branches.

We perform detailed analytical and experimental evaluation of our approach using 6 large graphs and demonstrate an average 2.7× speedup in execution time and 1.7× reduction in communication volume, compared to the state-of-the-art. We also show that unlike other GAS based implementations, PCPM is able to further reduce main memory traffic by taking advantage of intelligent node labeling that enhances locality. Although we use PageRank as the target application in this paper, our approach can be applied to generic SpMV computation.

1 Introduction

Graphs are the preferred choice of data representation in many fields such as web and social network analysis [9, 3, 29, 10], biology [17], transportation [15, 4] etc. The growing scale of problems in these areas has generated substantial research interest in high performance graph analytics. A large fraction of this research is focused on shared memory platforms because of their low communication overhead compared to distributed systems [26]. High DRAM capacity in modern systems further allows in-memory processing of large graphs on a single server [35, 33, 37]. However, efficient utilization of compute power is challenging even on a single node because of the (1) low computation-to-communication ratio and, (2) irregular memory access patterns of graph algorithms. The growing disparity between CPU speed and DRAM bandwidth, termed memory wall [42], has become a key issue in high performance graph analytics.

PageRank is a quintessential algorithm that exemplifies the performance challenges posed by graph computations. It iteratively performs Sparse Matrix-Vector (SpMV) multiplication over the adjacency matrix of the target graph and the current PageRank vector $\overrightarrow{PR}$ to generate new PageRank values. The irregularity in adjacency matrices leads to random accesses to $\overrightarrow{PR}$ with poor *spatial* and *temporal* locality. The resulting cache misses and communication volume become the performance bottleneck for PageRank computation. Since many graph algorithms can be similarly modeled as a series of SpMV operations [37], optimizations on PageRank can be easily generalized to other algorithms.

Recent works have proposed the use of Gather-Apply-Scatter (GAS) model to improve locality and reduce communication for SpMV and PageRank [43, 11, 5]. This model splits computation into two phases: *scatter* current source node values on edges and *gather* propagated values on edges to compute new values for destination nodes. The 2-phased approach restricts access to either the current $\overrightarrow{PR}$ or new $\overrightarrow{PR}$ at a time. This provides opportunities for cache-efficient and lock-free parallelization of the algorithm.

We observe that although this approach exhibits several attractive features, it also has some drawbacks leading to inefficient memory accesses, both quantitative as well as qualitative. First, we note that while scattering, a vertex *repeatedly* writes its value on all outgoing edges, resulting in large number of reads and writes. We also observe that the Vertex-centric graph traversal in [11, 5] results in *random* DRAM accesses and the Edge-centric traversal in [34, 43] scans edge list in coordinate format which increases the number of reads.

Our premise is that by changing the focus of computation from a single vertex or edge to a *cacheable* group of vertices (partition), we can effectively identify and reduce redundant edge traversals as well as avoid random accesses to DRAM, while still retaining the benefits of GAS model. Based on these insights, we develop a new

Partition-Centric approach to compute PageRank. The major contributions of our work are:

1. We propose a Partition-Centric Processing Methodology (PCPM) that propagates updates from nodes to partitions and reduces the redundancy associated with GAS model.
2. By carefully evaluating how a PCPM based implementation impacts algorithm behavior, we develop several system optimizations that substantially accelerate the computation, namely, (a) a new data layout that drastically reduces communication and random memory accesses, (b) branch avoidance mechanisms to remove unpredictable branches.
3. We demonstrate that PCPM can take advantage of intelligent node labeling to further reduce the communication volume. Thus, PCPM is suitable even for high locality graphs.
4. We conduct extensive analytical and experimental evaluation of our approach using 6 large datasets. On a 16-core shared memory system, PCPM achieves 2.1×−3.8× speedup in execution time and 1.3×−2.5× reduction in main memory communication over state-of-the-art.
5. We show that PCPM can be easily extended to weighted graphs and generic SpMV computation (section 3.5) even though it is described in the context of PageRank algorithm in this paper.

2 Background and Related Work

2.1 PageRank Computation

In this section, we describe how PageRank is calculated and what makes it challenging for the conventional implementation to achieve high performance. Table 1 lists a set of notations that we use to mathematically represent the algorithm.

Table 1: List of graph notations

$G(V,E)$	Input directed graph
A	adjacency matrix of $G(V,E)$
$N_i(v)$	in-neighbors of vertex v
$N_o(v)$	out-neighbors of vertex v
$\overrightarrow{PR_i}$	PageRank value vector after i^{th} iteration
$\overrightarrow{SPR}$	scaled PageRank vector $\left(SPR(v) = \frac{PR_i(v)}{\lvert N_o(v)\rvert}\right)$
d	damping factor in PageRank algorithm

PageRank is computed iteratively. In each iteration, all vertex values are updated by the new weighted sum of their in-neighbors' PageRank, as shown in equation 1.

$$PR_{i+1}(v) = \frac{1-d}{|V|} + d \sum_{u \in N_i(v)} \frac{PR_i(u)}{\lvert N_o(u)\rvert} \tag{1}$$

PageRank is typically computed in pull direction [35, 38, 37, 30] where each vertex pulls the value of its in-neighbors and accumulates into its own value, as shown in algorithm 1. This corresponds to traversing A in a column-major order and computing the dot product of each column with the scaled PageRank vector $\overrightarrow{SPR}$.

Algorithm 1 Pull Direction PageRank (PDPR) Iteration

```
for $v \in V$ do
    $temp = 0$
    for all $u \in N_i(v)$ do
        $temp += PR[u]$
    $PR_{next}[v] = \frac{(1-d) \times |V|^{-1} + d \times temp}{\lvert N_o(v)\rvert}$
swap$(PR, PR_{next})$
```

In the pull direction implementation, each column completely owns the computation of the corresponding element in the output vector. This enables all columns of A to be traversed asynchronously in parallel without the need to store partial sums in memory. On the contrary, in the push direction, each node updates its out-neighbors by adding its own value to them. This requires a row-major traversal of A and storage for partial sums since each row contributes partially to multiple elements in the output vector. Further, synchronization is needed to ensure conflict-free processing of multiple rows that update the same output element.

Performance Challenges: Sparse matrix layouts like Compressed Sparse Column (CSC) store all non-zero elements of a column sequentially in memory allowing fast column-major traversal of A [36]. However, the neighbors of a node can be scattered anywhere in the graph and reading their values results in random accesses (single or double word) to $\overrightarrow{SPR}$ in pull direction computation. Similarly, the push direction implementation uses a Compressed Sparse Row (CSR) format for fast row-major traversal of A but suffers from random accesses to the partial sums vector. These low locality and fine granularity accesses incur high cache miss ratio and contribute a large fraction to the overall memory traffic as shown in fig. 1.

2.2 Related Work

The performance of PageRank depends heavily on the locality in memory access patterns of the graph (which we refer to as *graph locality*). Since node labeling has significant impact on graph locality, many prior works have investigated the use of node reordering or clustering [7, 22, 6, 2] to improve the performance of graph algorithms. Reordering based on spatial and temporal locality aware placement of neighbors [39, 20] has

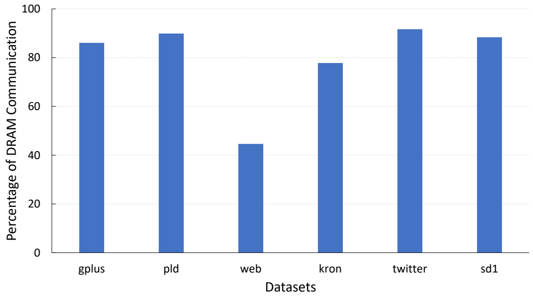


Figure 1: Percentage contribution of vertex value accesses to the total DRAM traffic in a PageRank iteration.

been shown to further outperform the well known clustering and tree-based techniques. However, such sophisticated algorithms also introduce substantial preprocessing overhead which limits their practicability. In addition, scale-free graphs like social networks are less tractable by reordering transformations because of their skewed degree distribution.

Cache Blocking (CB) is another technique used to accelerate graph processing [41, 32, 45]. CB induces locality by restricting the range of randomly accessed nodes and has been shown to reduce cache misses [24]. CB partitions A along rows, columns or both into multiple block matrices. However, SpMV computation with CB requires the partial sums to be re-read for each block. The extremely sparse nature of these block matrices also reduces the reuse of cached vertex data [31].

Gather-Apply-Scatter (GAS) is another popular model incorporated in many graph analytics frameworks [23, 34, 13]. It splits the analytic computation into *scatter* and *gather* phases. In the *scatter* phase, source vertices transmit updates on all of their outgoing edges and in the *gather* phase, these updates are processed to compute new values for corresponding destination vertices. The *updates* for PageRank algorithm correspond to *scaled PageRank values* defined earlier in section 2.1.

Binning exploits the 2-phased computation model by storing the updates in a semi-sorted manner. This induces *spatio-temporal* locality in access patterns of the algorithm. Binning can be used in conjunction with both Vertex-centric or Edge-centric paradigms. Zhou et al. [43, 44] use a custom sorted edge list with Edge-centric processing to reduce DRAM row activations and improve memory performance. However, their sorting mechanism introduces a non-trivial pre-processing cost and imposes the use of COO format. This results in larger communication volume and execution time than the CSR based Vertex-centric implementations [5, 11].

GAS model is also *inherently sub-optimal* when used with either Vertex-centric or Edge-centric abstractions. This is because it traverses the entire graph twice in each iteration. Nevertheless, Binning with Vertex-centric GAS (BVGAS) is the state-of-the-art methodology on shared memory platforms [5, 11] and we use it as baseline for comparison in this paper.

3 Partition-Centric Processing

We propose a new Partition-Centric Processing Methodology (PCPM) that significantly improves the efficiency of processor-memory communication over that achievable with current Vertex-centric or Edge-centric methods. We define *partitions* as disjoint sets of contiguously labeled nodes. The Partition-Centric abstraction then perceives the graph as a set of links from each node to the partitions corresponding to the neighbors of the node. We use this abstraction in conjunction with the 2-phased Gather-Apply-Scatter (GAS) model.

During the PCPM scatter phase, each thread processes one partition at a time. Processing a partition p means propagating messages from nodes in p to the neighboring partitions. A message to a partition p' comprises of the update value of source node ($PR[v]$) and the list of out-neighbors of v that lie in p'. PCPM caches the vertex data of p and streams the messages to the main memory. The messages from p are generated in a Partition-centric manner i.e. messages from all nodes in p to a neighboring partition p' are generated consecutively and are not interleaved with messages to any other partition.

During the gather phase, each thread scans all messages destined to one partition p at a time. A message scan applies the update value to all nodes in the neighbor list of that message. Partial sums of nodes in p are cached and messages are streamed from the main memory. After all messages to p are scanned, the partial sums (new PageRank values) are written back to DRAM.

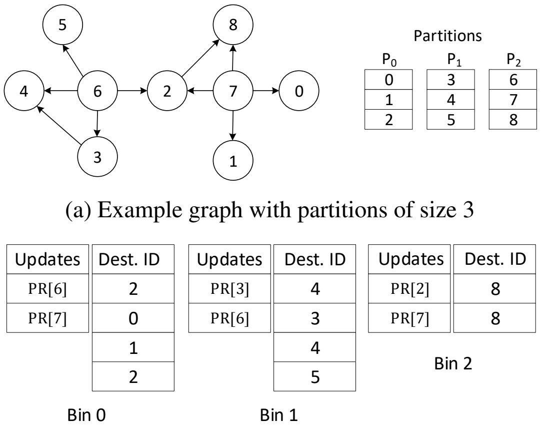
With static pre-allocation of distinct memory spaces for each partition to write messages, PCPM can asynchronously scatter or gather multiple partitions in parallel. In this section, we provide a detailed discussion on PCPM based computation and the required data layout.

3.1 Graph Partitioning

We employ a simple approach to divide the vertex set V into partitions. We create equisized partitions of size q where partition P_i owns all the vertices with index $\in [i*q, (i+1)*q)$ as shown in fig. 2a. As discussed later, the PCPM abstraction is built to easily take advantage of more sophisticated partitioning schemes and deliver further performance improvements (the trade-off is time complexity of partitioning versus performance gains). As we show in the results section, even the simple partitioning approach described above delivers significant performance gains over state-of-the-art methods.

Each partition is also allocated a contiguous memory space called *bin* to store updates (*update_bins*) and corresponding list of destination nodes (*destID_bins*) in the incoming messages. Since each thread in PCPM scatters or gathers only one partition at a time, the random accesses to vertex values or partial sums are limited to address range equal to the partition size. This improves temporal locality in access pattern and in turn, overall cache performance of the algorithm.

Before beginning PageRank computation, each partition calculates the offsets (address in bins where it must start writing from) into all *update_bins* and *destID_bins*. Our scattering strategy dictates that the partitions write to bins in the order of their IDs. Therefore, the offset for a partition P_i into any given bin is the sum of the number of values that all partitions with ID $< i$ are writing into that bin. For instance, in fig. 2, the offset of partition P_2 into *update_bins*[0] is 0 (since partitions P_0 and P_1 do not write to bin 0). Similarly, its offset into *update_bins*[1] and *update_bins*[2] is 1 (since P_1 writes one update to bin 1 and P_0 writes one update to bin 2). Offset computation provides each partition fixed and disjoint locations to write messages. This allows PCPM to parallelize partition processing without the need of locks or atomics.



(a) Example graph with partitions of size 3

(b) Bins store update value and list of destination nodes

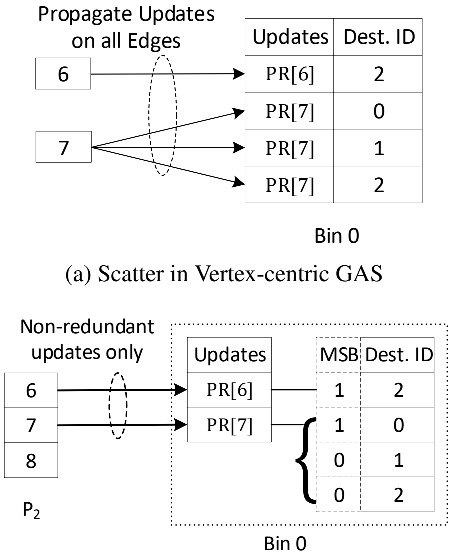
Figure 2: Graph Partitioning and messages inserted in bins during scatter phase

Note that since the destination node IDs written in the first iteration remain unchanged over the course of algorithm, they are written only once and reused in subsequent iterations. The reuse of destination node IDs along with the specific system optimizations discussed in section 3.2 and 3.3 enables PCPM to traverse only a fraction of the graph during scatter phase. This dramatically reduces the number of DRAM accesses and *gets rid of the inherent sub-optimality of GAS model.*

3.2 Partition-Centric Update Propagation

The unique abstraction of PCPM naturally leads to transmitting a single update from a node to a neighboring partition. In other words, even if a node has multiple neighbors in a partition, it inserts only one update value in the corresponding *update_bins* during scatter phase (algorithm 2). Fig. 3 illustrates the difference between Partition-Centric and Vertex-centric scatter for the example graph shown in fig. 2a.

PCPM manipulates the Most Significant Bit (MSB) of destination node IDs to indicate the range of nodes in a partition that use the same update value. In the *destID_bins*, it consecutively writes IDs of all nodes in the neighborhood of same source vertex and sets the MSB of first ID in this range to 1 for demarcation (fig. 3b). Since MSB is reserved for this functionality, PCPM supports graphs with upto 2 billion nodes instead of 4 billion for 4 Byte node IDs. However, to the best of our knowledge, this is enough to process most of the large publicly available datasets.



(a) Scatter in Vertex-centric GAS

(b) Scatter in PCPM

Figure 3: PCPM decouples *update_bins* and *destID_bins* to avoid redundant update value propagation

The gather phase starts only after all partitions are processed in the scatter phase. PCPM gather function sequentially reads updates and node IDs from the bins of the partition being processed. When gathering partition P_i, an update value $PR[v]$ should be applied to all out-neighbors of v that lie in P_i. This is done by checking the MSB of node IDs to determine whether to apply the previously read update or to read the next update, as shown in algorithm 2. The MSB is then masked to generate the true ID of destination node whose partial sum is updated.

Algorithm 2 describes PCPM based PageRank computation using a row-wise partitioned CSR format for adjacency matrix A. Note that PCPM only writes updates for *some* edges in a node's adjacency list, specifically the first outgoing edge to a partition. The remaining edges to that partition are *unused*. Since CSR stores adjacencies of a node contiguously, the set of first edges to neighboring partitions is interleaved with other edges. Therefore, we have to scan all outgoing edges of each vertex during scatter phase to access this set, which decreases efficiency. Moreover, the algorithm can potentially switch bins for each update insertion, leading to random writes to DRAM. Finally, the manipulation of MSB in node indices introduces additional data dependent branches which hurts the performance. Clearly, CSR adjacency matrix is not an efficient data layout for graph processing using PCPM. In the next section, we propose a PCPM-specific data layout.

Algorithm 2 PageRank iteration in PCPM using CSR format. Writing of *destID_bins* is not shown here.

```
$q \to$ partition size, $P \to$ set of partitions
for all $p \in P$ do in parallel                          ▷ Scatter
    for all $v \in p$ do
        $prev\_bin \leftarrow \infty$
        for all $u \in N_o(v)$ do
            if $\lfloor u/q \rfloor \neq prev\_bin$ then
                insert $PR[v]$ in $update\_bins[\lfloor u/q \rfloor]$
                $prev\_bin \leftarrow \lfloor u/q \rfloor$
$PR[:] \leftarrow 0$
for all $p \in P$ do in parallel                          ▷ Gather
    while $destID\_bins[p] \neq \emptyset$ do
        pop $id$ from $destID\_bins[p]$
        if $MSB(id) \neq 0$ then
            pop $update$ from $update\_bins[p]$
        $PR[id \,\&\, bitmask] \mathrel{+}= update$
for all $v \in V$ do in parallel                          ▷ Apply
    $PR[v] \leftarrow \frac{(1-d)/|V| + d \times PR[v]}{|N_o(v)|}$
```

3.3 Data Layout Optimization

In this subsection, we describe a new bipartite Partition-Node Graph (PNG) data layout that brings out the true Partition-Centric nature of PCPM. During the scatter phase, PNG prevents unused edge reads and ensures that all updates to a bin are streamed together before switching to another bin.

We exploit the fact that once *destID_bins* are written, the only required information in PCPM is the connectivity between nodes and partitions. Therefore, edges going from a source to all destination nodes in a single partition can be *compressed* into one edge whose new destination is the corresponding partition number. This gives rise to a bipartite graph G' with disjoint vertex sets V and P (where $P = \{P_0, \ldots, P_{k-1}\}$ represents the set of partitions in the original graph), and a set of directed edges E' going from V to P. Such a transformation has the following effects:

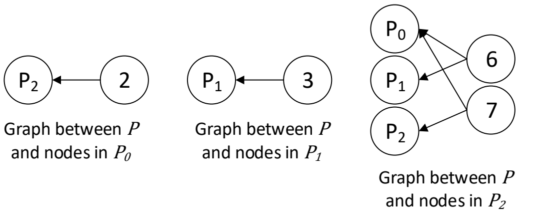

P2
2
Graph between P and nodes in P0
P1
3
Graph between P and nodes in P1
P0
P1
P2
6
7
Graph between P and nodes in P2

Figure 4: Partition-wise construction of PNG $G'(P,V,E')$ for graph $G(V,E)$ (fig. 2a). $|E'|$ is much smaller than $|E|$.

1. $Eff_1 \rightarrow$ the unused edges in original graph are removed
2. $Eff_2 \rightarrow$ the range of destination IDs reduces from $|V|$ to $|P|$.

The advantages of Eff_1 are obvious but those of Eff_2 will become clear when we discuss the storage format and construction of PNG.

The compression step reduces memory traffic by eliminating unused edge traversal. However note that scatters to a bin from source vertices in a partition are still interleaved with scatters to other bins. This can lead to random DRAM accesses during the scatter phase processing of a (source) partition. We resolve this problem by *transposing* the adjacency matrix of bipartite graph G'. The rows of the transposed matrix represent edges grouped by destination partitions which enables streaming updates to one bin at a time. This advantage comes at the cost of random accesses to source node values during the scatter phase. To prevent these random accesses from going to DRAM, we construct PNG on a per-partition basis i.e. we create a separate bipartite graph for each partition P_i with edges between P and the nodes in P_i (fig. 4). By carefully choosing q to make partitions *cacheable*, we ensure that all requests to source nodes are served by the cache resulting in *zero random DRAM accesses*.

Eff_2 is crucial for *transposition* of bipartite graphs in all partitions. The number of offsets required to store a transposed matrix in CSR format is equal to the range of destination node IDs. By reducing this range, Eff_2 reduces the storage requirement for offsets of each matrix from $O(|V|)$ to $O(|P|)$. Since there are $|P|$ partitions, each having one bipartite graph, the total storage requirement for edge offsets in PNG is $O(|P|^2)$ instead of $O(|V| \times |P|)$.

Although PNG construction looks like a 2-step approach, we actually merge *compression* and *transposition* into a single step. We first scan the outgoing edges of all nodes in a partition and individually compute the in-degree of all the destination partitions while discarding unused edges. A prefix sum of these degrees is carried out to compute the offsets array for CSR matrix. The same offsets can also be used to allocate disjoint writing locations into the bins of destination partitions. In the next scan, the edge array in CSR is filled with source node IDs completing both *compression* and *transposition*. PNG construction can be easily parallelized over all partitions to accelerate the pre-processing effectively.

Algorithm 3 shows the pseudocode for PCPM scatter phase using PNG layout. Unlike algorithm 2, the scatter function in algorithm 3 does not contain data dependent branches to check and discard unused edges. Using PNG provides drastic performance gains in PCPM scatter phase with little pre-processing overhead.

Algorithm 3 PCPM scatter phase using PNG layout

```
$G'(P,V,E') \rightarrow$ PNG, $N_i^p(p') \rightarrow$ in-neighbors of partition $p'$ in bipartite graph of partition $p$
for all $p \in P$ do in parallel                ▷ Scatter
    for all $p' \in P$ do
        for all $u \in N_i^p(p')$ do
            insert $PR[u]$ into $update\_bins[p']$
```

3.4 Branch Avoidance

Data dependent branches have been shown to have significant impact on performance of graph algorithms [14] and PNG removes such branches in PCPM scatter phase. In this subsection, we propose a branch avoidance mechanism for the PCPM gather phase. Branch avoidance enhances the sustained memory bandwidth but does not impact the amount of DRAM communication.

Note that the **pop** operations shown in algorithm 2 are implemented using pointers that increment after reading an entry from the respective bin. Let $destID\_ptr$ and $update\_ptr$ be the pointers to $destID\_bins[p]$ and $update\_bins[p]$, respectively. Note that the $destID\_ptr$ is incremented in every iteration whereas the $update\_ptr$ is only incremented if $MSB[id] \neq 0$.

To implement the branch avoiding gather function, instead of using a condition check over $MSB(id)$, we add it directly to $update\_ptr$. When $MSB(id)$ is 0, the pointer is not incremented and the same update value is read from cache in the next iteration; when $MSB(id)$ is 1, the pointer is incremented executing the **pop** operation on $update\_bins[p]$. The modified pseudocode for gather phase is shown in algorithm 4.

3.5 Weighted Graphs and SpMV

PCPM can be easily extended for computation on weighted graphs by storing the edge weights along with destination IDs in *destID_bins*. These weights can be read in the gather phase and applied to the source node value before updating the destination node. PCPM can also be extended to generic SpMV with non-square matrices by partitioning the rows and columns separately. In this case, the outermost loops in scatter phase (algorithm 3) and gather phase (algorithm 4) will iterate over row partitions and column partitions of A, respectively.

Algorithm 4 Branch Avoiding gather function in PCPM

```
$PR[:] = 0$
for all $p \in P$ do in parallel                ▷ Gather
    $\{destID\_ptr,\ update\_ptr\} \leftarrow 0$
    while $destID\_ptr < size(destID\_bins[p])$ do
        $id \leftarrow destID\_bins[p][destID\_ptr++]$
        $update\_ptr += MSB(id)$
        $id \leftarrow id\ \&\ bitmask$
        $PR[id] += update\_bins[p][update\_ptr]$
```

4 Comparison with Vertex-centric GAS

The Binning with Vertex-centric GAS (BVGAS) method allocates multiple bins to store incoming messages ($(update, destID)$ pairs). If bin width is q, then all messages destined to $v \in [i*q,(i+1)*q)$ are written in bin i. The scatter phase traverses the graph in a Vertex-centric fashion and inserts the messages in respective bins of the destination vertices. Number of bins is kept small to allow insertion points for all bins to fit in cache, providing good spatial locality. The gather phase processes one bin at a time as shown in algorithm 5, and thus, enjoys good temporal locality if bin width is small.

Algorithm 5 PageRank Iteration using BVGAS

```
$q \rightarrow$ bin width, $B \rightarrow$ no. of bins
for $v \in V$ do                                ▷ Scatter
    $PR[v] = PR[v]/|N_o(v)|$
    for all $u \in N_o(v)$ do
        insert $(PR[v], u)$ into $bins[\lfloor u/q \rfloor]$
$PR[:] = 0$
for $b = 0$ to $B-1$ do                         ▷ Gather
    for all $(update, dest)$ in $bins[b]$ do
        $PR[dest] = PR[dest] + update$
for all $v \in V$ do                            ▷ Apply
    $PR[v] = \frac{(1-d)}{|V|} + d \times PR[v]$
```

Unlike algorithm 5, in our BVGAS implementation, we write the destination IDs only in the first iteration. We also use small cached buffers to store updates before writing to DRAM. This ensures full cache line utilization and reduces communication during scatter phase [5].

Irrespective of all the locality advantages and optimizations, BVGAS inherently suffers from redundant reads and writes of a vertex value on all of its outgoing

Table 2: List of model parameters

Original Graph $G(V,E)$		**PNG layout** $G'(P,V,E')$	
n	no. of vertices ($\lvert V\rvert$)	k	no. of partitions ($\lvert P\rvert$)
m	no. of edges ($\lvert E\rvert$)	r	compression ratio ($\lvert E\rvert / \lvert E'\rvert$)
Architecture		**Software**	
c_{mr}	cache miss ratio for source value reads in PDPR	d_v	sizeof (updates/PageRank value)
l	sizeof (cache line)	d_i	sizeof (node or edge index)

edges. This redundancy manifests itself in the form of BVGAS' inability to utilize *high locality* in graphs with optimized node labeling. PCPM on the other hand, uses graph locality to reduce the fraction of graph traversed in scatter phase. Unlike PCPM, the Vertex-centric traversal in BVGAS can also insert consecutive updates into different bins. This leads to random DRAM accesses and poor bandwidth utilization. We provide a quantitative analysis of these differences in the next section.

5 Analytical Evaluation

We derive performance models to compare PCPM against conventional Pull Direction PageRank (PDPR) and BVGAS. Our models provide valuable insights into the behavior of different methodologies with respect to varying graph structure and locality. Table 2 defines the parameters used in the analysis. We use a synthetic kronecker graph [28] of scale 25 *(kron)* as an example for illustration purposes.

5.1 DRAM Communication

We analyze the amount of data exchanged with main memory per iteration of PageRank. We assume that data is accessed in quantum of one cache line and BVGAS exhibits full cache line utilization. Since destination indices are written only in the first iteration for PCPM and BVGAS, they are not accounted for in this model.

PDPR: The pull technique scans all edges in the graph once (algorithm 1). For a CSR format, this requires reading n edge offsets and m source node indices. PDPR also reads m source node values that incur cache misses generating $mc_{mr}l$ Bytes of DRAM traffic. Outputting new PageRank values generates nd_v Bytes of writes to DRAM. The total communication volume for PDPR is:

$$PDPR_{comm} = m(d_i + c_{mr}l) + n(d_i + d_v) \tag{2}$$

BVGAS: The scatter phase (algorithm 5) scans the graph and writes updates on all outgoing edges of the source node, thus communicating $(n+m)d_i + (n+m)d_v$ Bytes. The gather phase loads updates and destination node IDs on all the edges generating $m(d_i + d_v)$ Bytes of read traffic. At the end of gather phase, nd_v Bytes of new PageRank values are written in the main memory. Total communication volume for BVGAS is therefore, given by:

$$BVGAS_{comm} = 2m(d_i + d_v) + n(d_i + 2d_v) \tag{3}$$

PCPM with PNG: Number of edge offsets in bipartite graph of each partition is k. Thus, in the scatter phase (algorithm 3), a scan of PNG reads $(k \times k + m/r)d_i$ Bytes. The scatter phase further reads n PageRank values and writes updates on m/r edges. The gather phase (algorithm 4) reads m destination IDs and m/r updates followed by n new PageRank value writes. Net communication volume in PCPM is given by:

$$PCPM_{comm} = m\left(d_i\left(1 + \frac{1}{r}\right) + \frac{2d_v}{r}\right) + k^2 d_i + 2nd_v \tag{4}$$

Comparison: Performance of pull technique depends heavily on c_{mr}. In the worst case, all accesses are cache misses i.e. $c_{mr} = 1$ and in best case, only cold misses are encountered to load the PageRank values in cache i.e. $c_{mr} = nd_v/ml$. Assuming $k^2 \ll n \ll m$, we get $PDPR_{comm} \in [md_i, m(d_i + l)]$. On the other hand, communication for BVGAS stays constant. With $\theta(m)$ additional loads and stores, $BVGAS_{comm}$ can never reach the lower bound of $PDPR_{comm}$. Comparatively, $PCPM_{comm}$ achieves optimality when for every vertex, all outgoing edges can be compressed into a single edge i.e. $r = m/n$. In the worst case when $r = 1$, PCPM is still as good as BVGAS and we get $PCPM_{comm} \in [md_i, m(2d_i + 2d_v)]$. Unlike BVGAS, $PCPM_{comm}$ achieves the same lower bound as $PDPR_{comm}$.

Analyzing equations 2 and 3, we see that BVGAS is profitable compared to PDPR when:

$$c_{mr} > \frac{d_i + 2d_v}{l} \tag{5}$$

In comparison, PCPM offers a more relaxed constraint on c_{mr} (by a factor of $1/r$) becoming advantageous when:

$$c_{mr} > \frac{d_i + 2d_v}{rl} \tag{6}$$

The RHS in eq. 5 is constant indicating that BVGAS is advantageous for low locality graphs. With optimized node ordering, we can reduce c_{mr} and outperform BVGAS. On the contrary, $r \in [1, m/n]$ in the RHS of eq. 6 is a function of locality. With an optimized node labeling, r also increases and enhances the performance of PCPM. Fig. 5 shows the effect of r on predicted DRAM communication for the *kron* graph. Obtaining an optimal nodel labeling that makes $r = m/n$ might be very difficult or even impossible for some graphs. However, as can be observed from fig. 5, DRAM traffic decreases rapidly for $r \leq 5$ and converges slowly for $r > 5$. Therefore, a node reordering that can achieve $r \approx 5$ is good enough to optimize communication volume in PCPM.

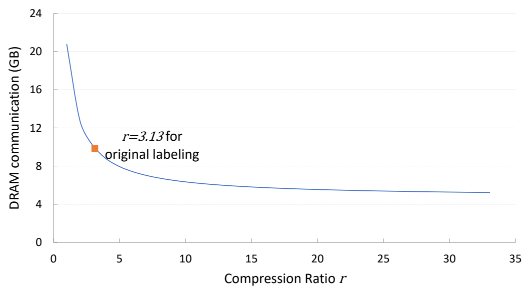


Figure 5: Predicted DRAM traffic for *kron* graph with $n = 33.5$ M, $m = 1070$ M, $k = 512$ and $d_i = d_v = 4$ Bytes.

5.2 Random Memory Accesses

We define a random access as a non-sequential jump in the address of memory location being read from or written to DRAM. Random accesses can incur latency penalties and negatively impact the sustained memory bandwidth. In this subsection, we model the amount of random accesses performed by different methodologies in a single PageRank iteration.

PDPR: Reading edge offsets and source node IDs in pull technique is completely sequential because of the CSR format. However, all accesses to source node PageRank values served by DRAM contribute to potential random accesses resulting in:

$$PDPR_{ra} = O(mc_{mr}) \tag{7}$$

BVGAS: In scatter phase of algorithm 5, updates can potentially be inserted at random memory locations. Assuming full cache line utilization for BVGAS, for every l Bytes written, there is at most 1 random DRAM access. In gather phase, all DRAM accesses are sequential if we assume that bin width is smaller than the cache. Total random accesses for BVGAS are then given by:

$$BVGAS_{ra} = O\left(\frac{md_v}{l}\right) \tag{8}$$

PCPM: With the PNG layout (algorithm 3), there are at most k bin switches when scattering updates from a partition. Since there are k such partitions, total number of random accesses in PCPM is bound by:

$$PCPM_{ra} = O(k*k) = O(k^2) \tag{9}$$

Comparison: BVGAS exhibits less random accesses than PDPR. However, $PCPM_{ra}$ is much smaller than both $BVGAS_{ra}$ and $PDPR_{ra}$. For instance, in the *kron* dataset with $d_v = 4$ Bytes, $l = 64$ Bytes and $k = 512$, $BVGAS_{ra} \approx 66.9$ M whereas $PCPM_{ra} \approx 0.26$ M.

Although it is not indicated in algorithm 5, the number of data dependent unpredictable branches in cache bypassing BVGAS implementation is also $O(m)$. For every update insertion, the BVGAS scatter function has to check if the corresponding cached buffer is full (section 4). In contrast, the number of branch mispredictions for PCPM (using branch avoidance) is $O(k^2)$ with 1 misprediction for every destination partition (p') switch in algorithm 3. The derivations are similar to random access model and for the sake of brevity, we do not provide a detailed deduction.

6 Experimental Evaluation

6.1 Experimental Setup and Datasets

We conduct experiments on a dual-socket Ivy Bridge server equipped with two 8-core Intel Xeon E5-2650 v2 processors@2.6 GHz running Ubuntu 14.04 OS. Table 3 lists important characteristics of our machine. Memory bandwidth is measured using STREAM benchmark [25]. All codes are written in C++ and compiled using G++ 4.7.1 with the highest optimization -O3 flag. The memory statistics are collected using Intel Performance Counter Monitor [40]. All data types used for indices and PageRank values are 4 Bytes.

Table 3: System Characteristics

Socket	no. of cores	8
	shared L3 cache	25MB
Core	L1d cache	32 KB
	L2 cache	256 KB
Memory	size	128 GB
	Read BW	59.6 GB/s
	Write BW	32.9 GB/s

We use 6 large real world and synthetic graph datasets coming from different applications, for performance evaluation. Table 4 summarizes the size and sparsity characteristics of these graphs. *Gplus* and *twitter* are follower graphs on social networks; *pld*, *web* and *sd1* are hyperlink graphs obtained by web crawlers; and *kron* is a scale 25 graph generated using Graph500 Kronecker generator. The *web* is a very sparse graph but has high locality obtained by a very expensive pre-processing of node labels [6]. The *kron* graph has higher edge density as compared to other datasets.

Table 4: Graph Datasets

Dataset	Description	# Nodes (M)	# Edges (M)	Degree
gplus [12]	Google Plus	28.94	462.99	16
pld [27]	Pay-Level-Domain	42.89	623.06	14.53
web [6]	Webbase-2001	118.14	992.84	8.4
kron [28]	Synthetic graph	33.5	1047.93	31.28
twitter [19]	Follower network	61.58	1468.36	23.84
sd1 [27]	Subdomain graph	94.95	1937.49	20.4

6.2 Implementation Details

We use a simple hand coded implementation of algorithm 1 for PDPR and parallelize it over vertices with static load balancing on the number of edges traversed. Our baseline does not incur overheads associated with similar implementations in frameworks [35, 30, 37] and hence, is faster than framework based programs [5].

To parallelize BVGAS scatter phase (algorithm 5), we give each thread a fixed range of nodes to scatter. Work per thread is statically balanced in terms of the number of edges processed. We also give each thread distinct memory spaces corresponding to all bins to avoid atomicity concerns in scatter phase. We use the Intel AVX non-temporal store instructions [1] to bypass the cache while writing updates and use 128 Bytes cache line aligned buffers to accumulate the updates for streaming stores [5]. BVGAS gather phase is parallelized over bins with load balanced using OpenMP dynamic scheduling. The optimal bin width is empirically determined and set to 256 KB (64K nodes). As bin width is a power of 2, we use bit shift instructions instead of integer division to compute the destination bin from node ID.

The PCPM scatter and gather phases are parallelized over partitions and load balancing in both the phases is done dynamically using OpenMP. Partition size is empirically determined and set to 256 KB. A detailed design space exploration of PCPM is discussed in section 6.3.2.

All the implementations mentioned in this section execute 20 PageRank iterations on 16 cores. For accuracy of the collected information, we repeat these algorithms 5 times and report the average values.

6.3 Results

6.3.1 Comparison with Baselines

Execution Time: Fig. 6 gives a comparison of the GTEPS (computed as the ratio of giga edges in the graph to the runtime of single PageRank iteration) achieved by different implementations. We observe that PCPM is $2.1-3.8\times$ faster than the state-of-the-art BVGAS implementation and upto $4.1\times$ faster than PDPR. BVGAS achieves constant throughput irrespective of the graph structure and is able to accelerate computation on low locality graphs. However, it is worse than PDPR for high locality (*web*) and dense (*kron*) graphs. PCPM is able to outperform PDPR and BVGAS on all datasets, though the speedup on *web* graph is minute because of high performance of PDPR. Detailed results for execution time of BVGAS and PCPM during different phases of computation are given in table 5. PCPM scatter phase benefits from a multitude of optimizations to achieve a dramatic $5\times$ speedup over BVGAS scatter phase.

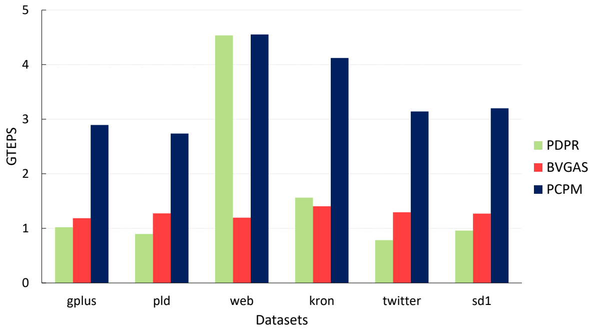


Figure 6: Performance in GTEPS. PCPM provides substantial speedup over BVGAS and PDPR.

Table 5: Execution time per iteration of PageRank for PDPR, BVGAS and PCPM

	PDPR	BVGAS			PCPM		
Dataset	Total Time(s)	Scatter Time(s)	Gather Time(s)	Total Time(s)	Scatter Time(s)	Gather Time(s)	Total Time(s)
gplus	0.44	0.26	0.12	0.38	0.06	0.1	0.16
pld	0.68	0.33	0.15	0.48	0.09	0.13	0.22
web	0.21	0.58	0.23	0.81	0.04	0.17	0.21
kron	0.65	0.5	0.22	0.72	0.07	0.18	0.25
twitter	1.83	0.79	0.32	1.11	0.18	0.27	0.45
sd1	1.97	1.07	0.42	1.49	0.24	0.35	0.59

Communication and Bandwidth: Fig. 7 shows the amount of data communicated with main memory normalized by the number of edges in the graph. Average communication in PCPM is $1.7\times$ and $2.2\times$ less than BVGAS and PDPR, respectively. Further, PCPM memory traffic per edge for *web* and *kron* is lower than other graphs because of their high compression ratio (table 6). The normalized communication for BVGAS is almost constant and therefore, its utility depends on the efficiency of pull direction baseline.

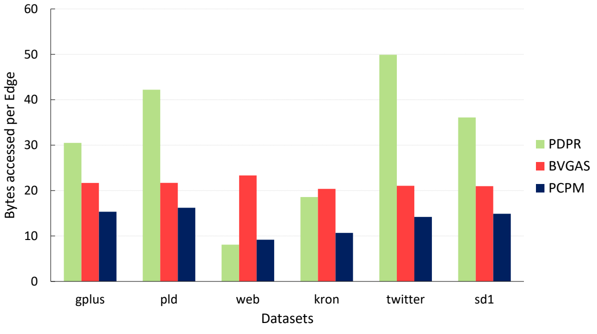


Figure 7: Main memory traffic per edge. PCPM communicates the least for all datasets except the *web* graph.

Note that the speedup obtained by PCPM is larger than the reduction in communication volume. This is because by avoiding random DRAM accesses and unpredictable branches, PCPM is able to efficiently utilize the available DRAM bandwidth. As shown in fig. 8, PCPM can sustain an average 42.4 GB/s bandwidth compared to 33.1 GB/s and 26 GB/s of PDPR and BVGAS, respectively. For large graphs like *sd1*, PCPM achieves $\approx 77\%$ of the peak read bandwidth (table 3) of our system. Although both PDPR and BVGAS suffer from random memory accesses, the former executes very few instructions and therefore, has better bandwidth utilization.

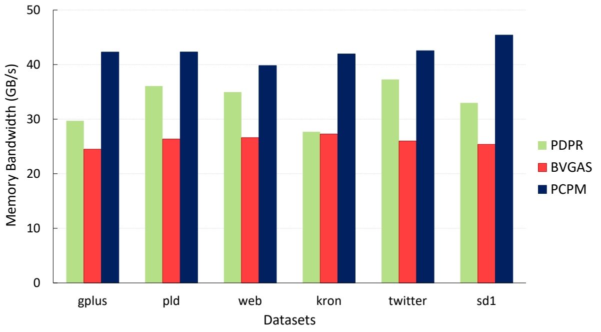


Figure 8: Sustained Memory Bandwidth for different methods. PCPM achieves highest bandwidth utilization.

Table 6: Locality vs compression ratio r. GOrder improves locality in neighbors and increases compression

		Original Labeling		**GOrder Labeling**	
Dataset	#Edges in Graph (M)	#Edges in PNG (M)	r	#Edges in PNG (M)	r
gplus	463	243.8	1.9	157.4	2.94
pld	623.1	347.7	1.79	166.7	3.73
web	992.8	118.1	8.4	126.8	7.83
kron	104.8	342.7	3.06	169.7	6.17
twitter	1468.4	722.4	2.03	386.2	3.8
sd1	1937.5	976.9	1.98	366.2	5.29

The reduced communication and streaming access patterns in PCPM also enhance its energy efficiency resulting in lower μJ/edge consumption as compared to BVGAS and PDPR, as shown in fig. 9. Energy efficiency is important from an eco-friendly computing perspective as highlighted by the Green Graph500 benchmark [16].

Effect of Locality: To assess the impact of locality on different methodologies, we relabel the nodes in our graph datasets using the GOrder [39] algorithm. We refer to the original node labeling in graph as *Orig* and GOrder labeling as simply *GOrder*. *GOrder* increases spatial locality by placing nodes with common in-neighbors closer in the memory. As a result, outgoing edges of the nodes tend to be concentrated in few partitions which increases the compression ratio r as shown in table 6. However, the *web* graph exhibits near optimal compression ($r = 8.4$) with *Orig* and does not show improvement with *GOrder*.

Table 7 shows the impact of *GOrder* on DRAM communication. As expected, BVGAS communicates a constant amount of data for a given graph irrespective of the labeling scheme used. On the contrary, memory traffic generated by PDPR and PCPM decreases because of reduced c_{mr} and increased r, respectively. These observations are in complete accordance with the performance models discussed in section 5.1. The effect on PCPM is not as drastic as PDPR because after r becomes greater than a threshold, PCPM communication decreases slowly as shown in fig. 5. Nevertheless, for almost all of the datasets, the net data transferred in PCPM is remarkably lesser than both PDPR and BVGAS for either of the vertex labelings.

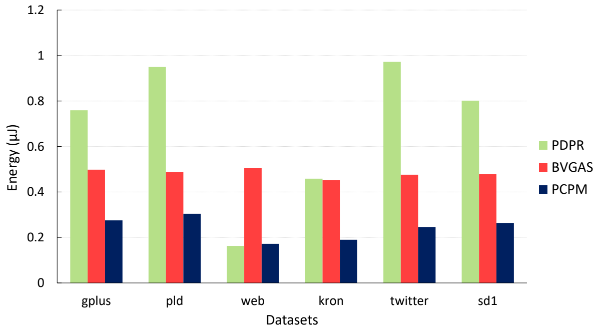


Figure 9: DRAM energy consumption per edge. PCPM benefits from reduced communication and random memory accesses.

Table 7: DRAM data transfer per iteration (in GB). PDPR and PCPM benefit from optimized node labeling

	PDPR		**BVGAS**		**PCPM**	
Dataset	*Orig*	*GOrder*	*Orig*	*GOrder*	*Orig*	*GOrder*
gplus	13.1	7.4	9.3	9.3	6.6	5.1
pld	24.5	10.7	12.6	12.5	9.4	6.1
web	7.5	7.6	21.6	21.3	8.5	8.4
kron	18.1	10.8	19.9	19.5	10.4	7.5
twitter	68.2	31.6	28.8	28.2	19.4	13.4
sd1	65.1	23.8	37.8	37.8	26.9	15.6

6.3.2 PCPM Design Space Exploration

Partition size represents an important tradeoff in PCPM. Large partitions force neighbors of each node to fit in fewer partitions resulting in better compression but poor locality. Small partitions on the other hand ensure high locality random accesses within partitions but reduce compression. We evaluate the impact of partition size on the performance of PCPM by varying it from 32 KB (8K nodes) to 8 MB (2M nodes). We observe a reduction in DRAM communication volume with increasing partition size (fig. 10). However, increases partition size beyond what cache can accommodate results in cache misses and a drastic increase in the DRAM traffic. As an exception, the performance on *web* graph is not heavily affected by partition size because of its high locality.

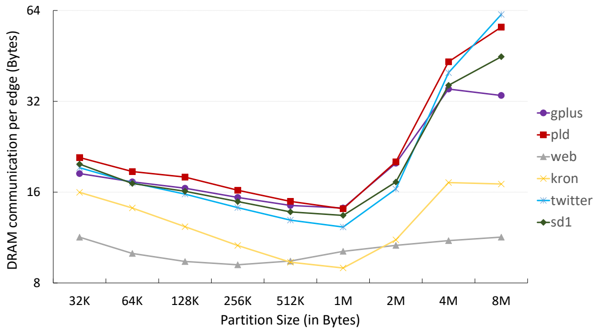


Figure 10: Impact of partition size on communication volume. Very large partitions result in cache misses and increased DRAM traffic.

The execution time (fig. 11) also benefits from communication reduction and is penalized by cache misses for large partitions. Note that for partition sizes > 256 KB and <= 1 MB, communication volume decreases but execution time increases. This is because in this range, many requests are served from the larger shared L3 cache which is slower than the private L1 and L2 caches. This phenomenon decelerates the computation but does not add to DRAM traffic.

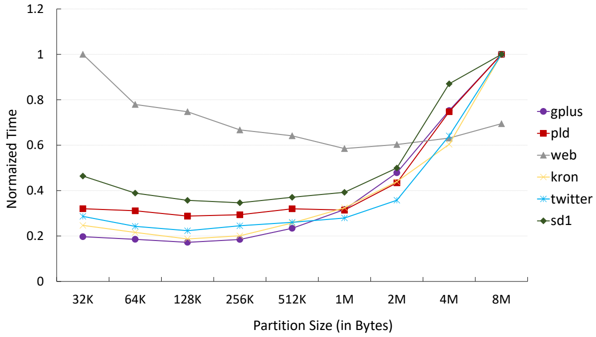


Figure 11: Impact of partition size on execution time.

Table 8: Pre-processing time of different methodologies. PNG construction increases the overhead of PCPM

Dataset	PCPM	BVGAS	PDPR
gplus	0.25s	0.1s	0s
pld	0.32s	0.15s	0s
web	0.26s	0.18s	0s
kron	0.43s	0.22s	0s
twitter	0.7s	0.27s	0s
sd1	0.95s	0.32s	0s

6.3.3 Pre-processing Time

We assume that adjacency matrix in CSR and CSC format is available and hence, PDPR does not need any pre-processing. Both BVGAS and PCPM however, require a beforehand computation of bin size and write offsets incurring non-zero pre-processing time as shown in table 8. In addition, PCPM also constructs the PNG layout. Fortunately, the computation of write offsets can be easily merged with PNG construction (section 3.3) to reduce the overhead. The pre-processing time also gets amortized over multiple iterations of PageRank.

7 Conclusion and Future Work

In this paper, we formulated a Partition-Centric Processing Methodology (PCPM) that perceives a graph as a set of links between nodes and partitions instead of nodes and their individual neighbors. We presented several features of this abstraction and developed data layout and system level optimizations to exploit them.

We conducted extensive analytical and experimental evaluation of our approach. Using a simple index based partitioning, we observed an average 2.7× speedup in execution time and 1.7× reduction in DRAM communication volume over state-of-the-art. In the future, we will explore edge partitioning models [21, 8] to further reduce communication and improve load balancing for PCPM.

Although we demonstrate the advantages of PCPM on PageRank, we show that it can be easily extended to generic SpMV computation. We believe that PCPM can be an efficient programming model for other graph algorithms or graph analytics frameworks. In this context, there are many promising directions for further exploration. For instance, the streaming memory access patterns of PNG enabled PCPM are highly suitable for High Bandwidth Memory (HBM) and disk-based systems. Exploring PCPM as a programming model for heterogenous memory or processor architectures is an interesting avenue for future work.

PCPM accesses nodes from only one graph partition at a time. Hence, G-Store's smallest number of bits representation [18] can be used to reduce the memory footprint and DRAM communication even further. Devising novel methods for enhanced compression can also make PCPM amenable to be used for large-scale graph processing on commodity PCs.

Acknowledgements: This material is based on work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract Number FA8750-17-C-0086, National Science Foundation (NSF) under Contract Numbers CNS-1643351 and ACI-1339756 and Air Force Research Laboratory under Grant Number FA8750-15-1-0185. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA, NSF or AFRL. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

References

[1] Intel c++ compiler 17.0 developer guide and reference, 2016. Available at `https://software.intel.com/en-us/intel-cplusplus-compiler-17.0-user-and-reference-guide`.

[2] ABOU-RJEILI, A., AND KARYPIS, G. Multilevel algorithms for partitioning power-law graphs. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing* (2006), IPDPS'06, IEEE Computer Society, pp. 124–124.

[3] ALBERT, R., JEONG, H., AND BARABÁSI, A.-L. Internet: Diameter of the world-wide web. *nature 401*, 6749 (1999), 130.

[4] ALDOUS, J. M., AND WILSON, R. J. *Graphs and applications: an introductory approach*, vol. 1. Springer Science & Business Media, 2003.

[5] BEAMER, S., ASANOVIĆ, K., AND PATTERSON, D. Reducing pagerank communication via propagation blocking. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International* (2017), IEEE, pp. 820–831.

[6] BOLDI, P., ROSA, M., SANTINI, M., AND VIGNA, S. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web* (2011), ACM, pp. 587–596.

[7] BOLDI, P., SANTINI, M., AND VIGNA, S. Permuting web and social graphs. *Internet Mathematics 6*, 3 (2009), 257–283.

[8] BOURSE, F., LELARGE, M., AND VOJNOVIC, M. Balanced graph edge partition. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2014), KDD '14, ACM, pp. 1456–1465.

[9] BRODER, A., KUMAR, R., MAGHOUL, F., RAGHAVAN, P., RAJAGOPALAN, S., STATA, R., TOMKINS, A., AND WIENER, J. Graph structure in the web. *Computer networks 33*, 1-6 (2000), 309–320.

[10] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H. C., ET AL. Tao: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference* (2013), pp. 49–60.

[11] BUONO, D., PETRINI, F., CHECCONI, F., LIU, X., QUE, X., LONG, C., AND TUAN, T.-C. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *Proceedings of the 2016 International Conference on Supercomputing* (2016), ACM, p. 37.

[12] GONG, N. Z., XU, W., HUANG, L., MITTAL, P., STEFANOV, E., SEKAR, V., AND SONG, D. Evolution of social-attribute networks: measurements, modeling, and implications using google+. In *Proceedings of the 2012 Internet Measurement Conference* (2012), ACM, pp. 131–144.

[13] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), USENIX, pp. 17–30.

[14] GREEN, O., DUKHAN, M., AND VUDUC, R. Branch-avoiding graph algorithms. In *Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures* (2015), ACM, pp. 212–223.

[15] HAKLAY, M., AND WEBER, P. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing 7*, 4 (2008), 12–18.

[16] HOEFLER, T. Green graph500. Available at `http://green.graph500.org/`.

[17] HUBER, W., CAREY, V. J., LONG, L., FALCON, S., AND GENTLEMAN, R. Graphs in molecular biology. *BMC bioinformatics 8*, 6 (2007), S8.

[18] KUMAR, P., AND HUANG, H. H. G-store: high-performance graph store for trillion-edge processing. In *High Performance Computing, Networking, Storage and Analysis, SC16: International Conference for* (2016), IEEE, pp. 830–841.

[19] KWAK, H., LEE, C., PARK, H., AND MOON, S. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web* (2010), ACM, pp. 591–600.

[20] LAKHOTIA, K., SINGAPURA, S., KANNAN, R., AND PRASANNA, V. Recall: Reordered cache aware locality based graph processing. In *High Performance Computing (HiPC), 2017 IEEE 24th International Conference on* (2017), IEEE, pp. 273–282.

[21] Li, L., Geda, R., Hayes, A. B., Chen, Y., Chaudhari, P., Zhang, E. Z., and Szegedy, M. A simple yet effective balanced edge partition model for parallel computing. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems* (2017), SIGMETRICS '17 Abstracts, ACM, pp. 6–6.

[22] Liu, W.-H., and Sherman, A. H. Comparative analysis of the cuthill–mckee and the reverse cuthill–mckee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis 13*, 2 (1976), 198–213.

[23] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 135–146.

[24] Malicevic, J., Lepers, B., and Zwaenepoel, W. Everything you always wanted to know about multicore graph processing but were afraid to ask. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (2017), USENIX, pp. 631–643.

[25] McCalpin, J. D. Stream benchmark. *Link: www. cs. virginia. edu/stream/ref. html# what 22* (1995).

[26] McSherry, F., Isard, M., and Murray, D. G. Scalability! but at what cost? In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (2015), HOTOS'15, USENIX Association, pp. 14–14.

[27] Meusel, R., Vigna, S., Lehmberg, O., and Bizer, C. The graph structure in the web: Analyzed on different aggregation levels. *The Journal of Web Science 1*, 1 (2015), 33–47.

[28] Murphy, R. C., Wheeler, K. B., Barrett, B. W., and Ang, J. A. Introducing the graph 500. *Cray Users Group (CUG) 19* (2010), 45–74.

[29] Newman, M. E., Watts, D. J., and Strogatz, S. H. Random graph models of social networks. *Proceedings of the National Academy of Sciences 99*, suppl 1 (2002), 2566–2572.

[30] Nguyen, D., Lenharth, A., and Pingali, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 456–471.

[31] Nishtala, R., Vuduc, R. W., Demmel, J. W., and Yelick, K. A. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing 18*, 3 (2007), 297–311.

[32] Penner, M., and Prasanna, V. K. Cache-friendly implementations of transitive closure. *Journal of Experimental Algorithmics (JEA) 11* (2007), 1–3.

[33] Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., and Haradasan, M. Managing large graphs on multi-cores with graph awareness. 41–52.

[34] Roy, A., Mihailovic, I., and Zwaenepoel, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[35] Shun, J., and Blelloch, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices* (2013), vol. 48, ACM, pp. 135–146.

[36] Siek, J. G., Lee, L.-Q., and Lumsdaine, A. *The Boost Graph Library: User Guide and Reference Manual, Portable Documents*. Pearson Education, 2001.

[37] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Anderson, M. J., Vadlamudi, S. G., Das, D., and Dubey, P. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment 8*, 11 (2015), 1214–1225.

[38] Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. Gunrock: A high-performance graph processing library on the gpu. In *ACM SIGPLAN Notices* (2016), vol. 51, ACM, p. 11.

[39] Wei, H., Yu, J. X., Lu, C., and Lin, X. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data* (2016), ACM, pp. 1813–1828.

[40] Willhalm, T., Dementiev, R., and Fay, P. Intel performance counter monitor-a better way to measure cpu utilization. 2012. *URL: http://software.intel.com/en-us/articles/intel-performance-counter-monitor-a-better-way-to-measure-cpuutilization* (2016).

[41] WILLIAMS, S., OLIKER, L., VUDUC, R., SHALF, J., YELICK, K., AND DEMMEL, J. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. *Parallel Computing 35*, 3 (2009), 178–194.

[42] WULF, W. A., AND MCKEE, S. A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news 23*, 1 (1995), 20–24.

[43] ZHOU, S., CHELMIS, C., AND PRASANNA, V. K. Optimizing memory performance for fpga implementation of pagerank. In *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on* (2015), IEEE, pp. 1–6.

[44] ZHOU, S., LAKHOTIA, K., SINGAPURA, S. G., ZENG, H., KANNAN, R., PRASANNA, V. K., FOX, J., KIM, E., GREEN, O., AND BADER, D. A. Design and implementation of parallel pagerank on multicore platforms. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE* (2017), IEEE, pp. 1–6.

[45] ZHU, X., HAN, W., AND CHEN, W. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), USENIX Association, pp. 375–386.