Reducing Pagerank Communication via Propagation Blocking

Scott Beamer*
Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, California
sbeamer@lbl.gov

Krste Asanović David Patterson
Electrical Engineering & Computer Sciences Department
University of California
Berkeley, California
{krste,pattrsn}@eecs.berkeley.edu

***Abstract*—Reducing communication is an important objective, as it can save energy or improve the performance of a communication-bound application. The graph algorithm PageRank computes the importance of vertices in a graph, and it serves as an important benchmark for graph algorithm performance. If the input graph to PageRank has poor locality, the execution will need to read many cache lines from memory, some of which may not be fully utilized. We present *propagation blocking*, an optimization to improve spatial locality, and we demonstrate its application to PageRank. In contrast to cache blocking which partitions the graph, we partition the data transfers between vertices (propagations).**

If the input graph has poor locality, our approach will reduce communication. Our approach reduces communication more than conventional cache blocking if the input graph is sufficiently sparse or if number of vertices is sufficiently large relative to the cache size. To evaluate our approach, we use both simple analytic models to gain insights and precise hardware performance counter measurements to compare implementations on a suite of 8 real-world and synthetic graphs. We demonstrate our parallel implementations substantially outperform prior work in execution time and communication volume. Although we present results for PageRank, propagation blocking could be generalized to SpMV (sparse matrix multiplying dense vector) or other graph programming models.

I. Introduction

The bounty of transistors provided by Moore's Law has enabled increased computational speed and throughput, but total communication bandwidth has failed to keep up. As a consequence, some low-arithmetic intensity workloads are often bottlenecked by communication to memory on today's platforms. For these communication-bound workloads, the only way to improve performance is to either increase their effective memory bandwidth or decrease the amount of communication. Reducing communication can also save energy, as moving data consumes more energy than the arithmetic operations that manipulate it [1].

The amount of memory communication needed to execute a graph processing workload is a function of many things including: the algorithm, the cache size, the graph size, the graph layout, and the software implementation. To reduce communication, prior work examined improving the graph layout or reordering the computation to increase locality, and these optimizations are often beneficial. However, there are some graphs that are less amenable to layout or reordering transformations. Low-diameter graphs, such as social networks, are often such stubborn graphs with low locality.

In this work, we present *propagation blocking*, an optimization that improves spatial locality of low-locality graph workloads. By improving the spatial locality of a workload, our optimization accelerates the workload by reducing the amount of memory communication. Performing propagation blocking adds additional computation, but for a communication-bound workload, the benefit from improving spatial locality makes this tradeoff beneficial.

We select PageRank to evaluate propagation blocking, since it is often communication-bound due to its low arithmetic intensity. The PageRank algorithm was initially used to determine the popularity of webpages in order to assist web search algorithms, but it has since proven to be a useful benchmark. PageRank is an application of the linear algebra Sparse Matrix Multiplying Dense Vector (SpMV) kernel, and our propagation-blocking optimization can also be applied to SpMV.

We evaluate propagation blocking applied to PageRank on a suite of eight real-world and synthetic graphs using hardware performance counter measurements. Compared to our baseline implementation, on the seven low-locality graphs, our new approach reduces communication by 1.5 – 2.9× (2.3 average) and improves performance by 1.1 – 2.7× (1.8 average). On the one high-locality graph, propagation blocking hurts performance and communication efficiency by less than 5%. We validate our performance results by demonstrating our baseline substantially outperforms prior work, and by implementing a high-performance cache blocking implementation for comparison. We evaluate propagation blocking applied to PageRank, but it could also be applied to other graph algorithms or even SpMV, and we discuss its applicability to graph programming models in Section IX. Propagation blocking is typically advantageous to conventional cache blocking for graphs that are sufficiently large and sparse.

* Research performed primarily while studying at UC Berkeley

II. Background on PageRank

PageRank [2] has emerged as a popular graph benchmark as it exposes many of the challenges of graph processing while still being simple enough to ease analysis and implementation [3]. PageRank determines the "popularity" of a vertex by summing the scaled popularities of the vertices that point to it. This recurrence often results in cyclic dependencies, but as long as the graph is aperiodic, the scores will converge [4]. PageRank typically iterates until the scores converge within a specified tolerance of a fixed point. PageRank scores can also be computed or approximated by other techniques including spectral methods, but in this work we focus on the iterative method.

The PageRank score for a vertex u is ($d = 0.85$):

$$PR(u) = \frac{1-d}{|V|} + d \sum_{v \in N^-(u)} \frac{PR(v)}{|N^+(v)|}$$

At the core of this computation is the propagation of a vertex's score $PR(v)$ scaled by its out degree $|N^+(v)|$ to the vertex u it points to. In other words, a vertex's score $PR(u)$ is the sum of the contributions (scaled scores) from its incoming neighbors $N^-(u)$. In our discussion of implementing PageRank, we focus on the propagation of scaled scores between vertices as the other multiplications or additions are on scalar or on often reused values.

We perform much of our analysis of PageRank from the graph algorithm perspective, but to ease some explanations, we sometimes take the sparse linear algebra perspective, where the input graph can be viewed as a sparse adjacency matrix A such that each non-zero element A_{ij} represents an edge from vertex i to vertex j. The propagation and reduction of contributions in PageRank is an application of SpMV to A^T. There are two restrictions on SpMV within PageRank not present in generalized SpMV. First, the matrix A must be square since the rows and columns represent the same set of vertices. Second, the matrix is binary (graph is unweighted), so the adjacencies themselves are the useful information as there are no associated values to read with them. Additionally, the binary matrix results in each output element being a sum reduction instead of a dot product. Our proposed technique does not rely on these restrictions, and it can be extended to handle generalized SpMV.

There are two directions to perform the computation: pull (akin to row-major SpMV) and push (akin to column-major SpMV). In the pull direction, each vertex reads the contributions of its incoming neighbors and computes its own score (Algorithm 1). In the push direction, each vertex adds its contribution to the sums of its outgoing neighbors (Algorithm 2). After propagating the contributions in the push direction, a later pass will use those sums to compute the new scores. The pull direction is often more efficient since it only reads each neighbors' contributions rather than doing an atomic add to each neighbor's sum. Implementing the pull direction requires the transpose graph if the graph is directed, as the computation will need to know the incoming edges for each vertex. Both directions compute the same result, and the distinction is whether the sum (pull) or the contribution (push) is the vertex value immediately reused.

Algorithm 1 PageRank in pull direction

Input: $G(V, E)$, number of iterations I
Output: PageRank scores for all vertices $PR[:]$

```
PR[:] ← 1/|V|
for each iteration i ∈ 1...I do
    for each vertex u ∈ V do
        contributions[u] ← PR[u]/OutDegree(u)
    for each vertex u ∈ V do
        sum ← 0
        for each incoming neighbor v of vertex u do
            sum ← sum + contributions[v]
        PR[u] ← (1 − d)/|V| + d × sum
```

Algorithm 2 PageRank in push direction

Input: $G(V, E)$, number of iterations I
Output: PageRank scores for all vertices $PR[:]$

```
PR[:] ← 1/|V|
for each iteration i ∈ 1...I do
    sums[:] ← 0
    for each vertex u ∈ V do
        contribution ← PR[u]/OutDegree(u)
        for each outgoing neighbor v of vertex u do
            sums[v] ← sums[v] + contribution
    for each vertex u ∈ V do
        PR[u] ← (1 − d)/|V| + d × sums[u]
```

III. Locality Challenges for PageRank

Locality is extremely important to PageRank performance, as improved locality can reduce communication which in turn will improve performance. Although a given algorithm may have a predictable number of reads and writes based on its input size, those reads and writes may translate into a variable number of memory requests to DRAM based on cache locality. Although modern processors have multiple levels of caches, in this work we consider only a single cache level representative of the last-level cache (LLC). This is reasonable, as the bandwidth between the LLC and DRAM is typically the communication bottleneck [5].

The propagation of a contribution to a sum is the core of PageRank's communication. To perform a propagation, the computation must read the graph adjacencies to identify which vertices are connected and subsequently access the contribution for the source vertex and the sum for the destination vertex. We categorize the memory traffic for this PageRank communication into "edge" traffic, which

accesses the graph adjacency information, and "vertex" traffic, which accesses the contributions or sums associated with vertices. The edge traffic typically enjoys good spatial locality, as most implementations process the neighbors of a vertex consecutively and many graph layouts (e.g., CSR) store neighbor identifiers continuously. The vertex traffic has the potential to have much lower locality, since a vertex could potentially be connected to any other vertex, and so the vertex value accesses are unlikely to be consecutive. For graphs with many vertices, the corresponding arrays that associate a value with each vertex are much larger than the cache, so non-consecutive accesses to these arrays are likely to have poor cache locality.

Vertex traffic is composed of accesses for the contributions as well as accesses for the sums, and it is challenging to improve locality for both access streams simultaneously. One of the vertex value accesses will have high temporal locality (probably even register allocated), but the other will have potentially low locality as there is no restriction on a vertex's location relative to its neighbor (Figure 1). For example, in the pull direction, the sum of incoming contributions will have high locality, but reading the neighbor's contributions could have low locality. For the push direction, the outgoing contribution will have high locality, but reading (and writing) its neighbors' sums could have low locality. The low locality vertex value accesses not only increase the number of memory requests (low temporal locality), but the low locality can also result in unused words within transferred cache lines (poor spatial locality). These unused words are problematic, as they waste bandwidth and energy.

Since the vertex values are stored contiguously in arrays, the graph labelling (or "layout") has a tremendous impact on the locality of the vertex value accesses. For accesses that will potentially have low locality, whether they actually experience low locality is determined by the graph's layout. An ideal high-locality graph layout when viewed by its adjacency matrix has all of its non-zeros in a narrow diagonal. Not only does this improve the spatial locality for processing one vertex since all of its neighbors' vertex values will be adjacent, but it also improves the temporal locality between vertices because there will be large overlaps between their neighbors. Unfortunately, some graphs' topologies make it difficult to find such an ideal layout. These graphs of interest are often low-diameter and are often social networks [6]. Alternatively, for some situations the time to compute and transform the graph into such a layout is not warranted.

Cache blocking [7] is a technique to reduce the negative impact of low-locality vertex value accesses. By partitioning the graph into blocks, the potential range for vertex values is reduced sufficiently such that the corresponding slices of the arrays for vertex values are small enough to reside in cache, thus improving the locality of that access stream (Figure 1). Unfortunately, this gain for the low-locality access stream comes at the expense of the high-locality access stream.

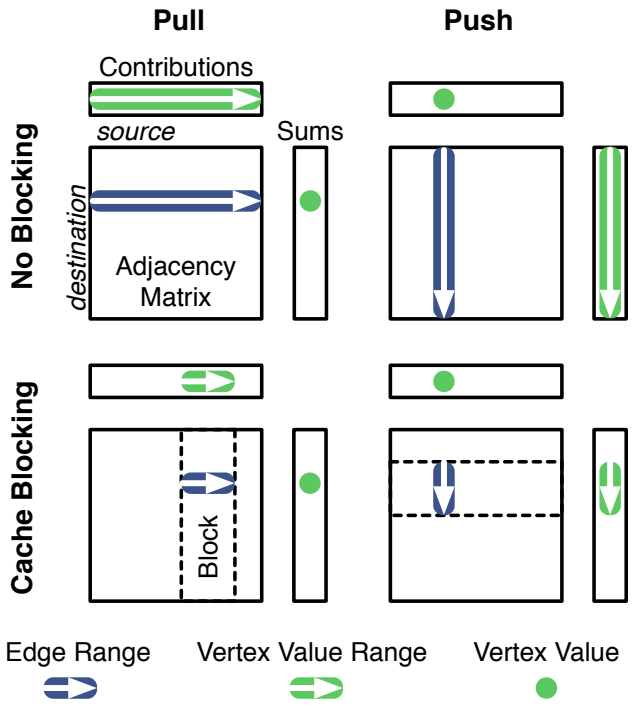

Figure 1. Accesses needed to process one vertex and its neighbors for PageRank, with or without 1D cache blocking for both pull (row-major) and push (column-major) directions. All approaches enjoy high locality when accessing the adjacency matrix (edge traffic). The vertex value arrays (contributions or sums) are much larger than the cache size, and thus accessing a sparse range within them could have low locality. Without blocking (top), one vertex value access obtains high locality at the expense of the other. Pull has high temporal locality on the sums, but low locality reading the contributions. Push has high locality reading the contributions, but low locality for the sums. Cache blocking (bottom) reduces the range of vertex values to the current block in order to improve locality.

For example, blocking in the pull direction will improve the locality of reading the neighbors' contributions, but worsen the locality of accessing the sum, as the sum must be re-read and written for each block. The more blocks the graph is partitioned into, the more times the sums must be re-accessed. The block size is a tradeoff between the localities of the two access streams, with the optimum size resulting in moderate locality for both streams. Cache blocking can be done in one or two dimensions and can be applied to either direction (push or pull). It is also worth distinguishing cache blocking for sparse matrices from register blocking for sparse matrices, which is often done to reduce the amount the index array is read [7].

To implement cache blocking, the graph data structure needs to be modified to accomodate easy access to each block. One technique is to store the graph blocks each as their own graph (in CSR). If the graph is sufficiently sparse (e.g. expected number of nonzeros per row per block < 1), it may be advantageous to store each block as an edge list instead to cut down on the index traffic.

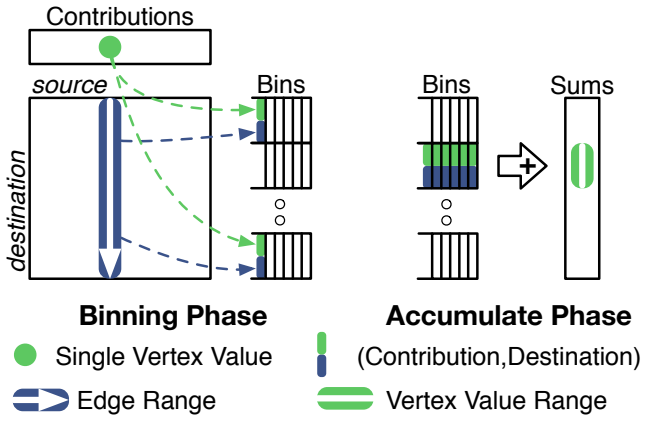

Figure 2. *Propagation blocking*. In the binning phase, vertices pair their contributions with their destination indices and insert them into the appropriate bins. In the accumulate phase, the contributions are reduced into the correct sums.

Algorithm 3 PageRank by propagation blocking

Input: $G(V, E)$, number of iterations I, number of bins B
Output: PageRank scores for all vertices $PR[:]$

```
PR[:] ← 1/|V|
for each iteration i ∈ 1...I do
    bins[:] ← {}
    for each vertex u ∈ V do                    ▷ Binning Phase
        contribution ← PR[u]/OutDegree(u)
        for each outgoing neighbor v of vertex u do
            insert (contribution, v) into bins[v/B]
    sums[:] ← 0
    for each index b ∈ 1...B do                 ▷ Accumulate Phase
        for each (contribution, v) ∈ bins[b] do
            sums[v] ← sums[v] + contribution
    for each vertex u ∈ V do
        PR[u] ← (1 − d)/|V| + d × sums[u]
```

The challenge for PageRank communication is that it is difficult for both vertex value memory access streams to obtain high locality simultaneously. With the push technique or the pull technique, one stream does well at the expense of the other. Cache blocking allows for a more continuous locality tradeoff between the two streams, but it does not allow for them both to have high locality simultaneously.

IV. Propagation Blocking

To improve the low locality vertex value accesses, we propose *propagation blocking*. Unlike cache blocking that blocks the graph itself, we block the propagations (transfers of vertex values). Propagation blocking stores the propagations to memory in a semi-sorted manner, so when the propagations are read later, the accesses will have better spatial locality. We split the propagation of contributions for the push technique into two phases: *binning* and *accumulate* (Figure 2 & Algorithm 3).

In the binning phase, all of the contributions are read, but instead of adding them directly to their sums, we insert them into bins corresponding to their destination vertices. Each bin corresponds to a distinct contiguous range of vertices. When inserting a contribution into a bin, we append the destination vertex identifier to create a (*contribution*, *destination*) pair. This binning phase is analogous to a radix partitioning done for a database join or a bucket sort. The binning phase enjoys good spatial locality, as inserting into a bin stores to consecutive addresses, and the number of bins is small enough (e.g., 64) that the insertion points of these bins can fit in cache simultaneously.

In the accumulate phase, one bin is processed at a time and the contributions are added to their appropriate sums guided by their paired destination identifiers. Because a bin corresponds to a reduced range of vertices, all of its corresponding sums can fit in cache at once, improving temporal locality. Reading the (contribution, destination) pairs from the bin also enjoys high spatial locality since they are contiguous.

Propagation blocking reduces communication through the use of additional memory space to store the propagations. The amount of additional memory required can be substantial, since each directed edge in the graph will need space for two words (contribution and destination). The additional memory is split into bins, and the number of bins is the number of vertices in the graph divided by the bin width. We select the bin "width" so the number of vertices associated with each bin is small enough that its corresponding slice of the sums array can fit in cache.

To further reduce communication, propagation blocking can leverage a deterministic layout for the bins. If the locations that contributions are written to within a bin are the same for multiple iterations on the same graph, the destination identifiers can be stored in a separate data structure. This halves the number of writes during the binning phase, as only the contributions and not their destinations need to be written. During the accumulate phase, the bin is read in lockstep with the destination indices to determine the destinations for each contribution. The separate arrays containing the destination indices can be reused after the first iteration of PageRank or even computed in advance.

In addition to consuming more memory capacity, propagation blocking performs additional load and store instructions in order to insert propagations into the bins and read propagations back from the bins. Since propagation blocking makes use of every word in each transferred cache line, it can transfer fewer cache lines to achieve a net memory communication reduction. Whether propagation blocking is advantageous depends on how well the baseline makes use of every word in each cache line it transfers.

V. Communication Model

We present simple analytic models for the amount of communication each PageRank implementation strategy performs in order to gain qualitative insights into their tradeoffs. Our models assume a uniform random input graph due to its simplicity, but we acknowledge a random graph is the worst case for many of these implementations due to its lack of locality. For communication volume, we use units of cache lines since that is the unit of transfer with main memory. Within cache lines we use the unit of words (e.g., 32-bit words). We model only a single global iteration of PageRank, since each implementation strategy we model communicates the same amount each iteration. We use the following parameters:

n	number of vertices ($\|V\|$)
k	average directed degree ($kn = \|E\|$)
b	number of words per cache line
c	number of words in cache
r	number of graph blocks for cache blocking

To read the graph from a CSR layout requires kn/b cache lines to read the adjacencies and $2n/b$ cache lines to read the indices. Our index uses 64-bit pointers to support greater than 4 billion edges, so we count each of these pointers as two words. To read the vertex values, looking up the destination sums requires reading n/b cache lines. The contributions could potentially be in cache, and we approximate the miss hit rate as $1 - c/n$ (assuming $n > c$). Since there are kn directed edges, reading the contributions requires $(1 - c/n)kn$ cache lines. To output the final scores requires writing n/b cache lines. Altogether, the number of cache lines the pull technique reads is:

$$\left(\left(1 - \frac{c}{n}\right) + \frac{1}{b} + \frac{3}{kb}\right) kn$$

A. Cache Blocking

We model the communication for 1D cache blocking in the push direction using a CSR data structure for each block. Reading the adjacencies still requires reading kn/b cache lines, however, with r blocks we will also read r index arrays, so reading in all of the graph requires a total of $(k + 2r)n/b$ cache line reads. Presumably, the blocks are small enough such that $n/r < c$, so the only vertex value traffic needed is compulsory, but the blocked vertices will be re-read, yielding $(r + 1)n/b$ cache line reads. To output the final scores requires writing n/b cache lines. Altogether, the number of cache lines 1D cache blocking on CSR requires reading is:

$$(k + 3r + 1)\frac{n}{b}$$

If the graph is sufficiently sparse ($k < 2r$), using an edge list to hold each block instead of CSR reduces the index communication, yielding a cache line read total of:

$$(2k + r + 1)\frac{n}{b}$$

We do not model 2D cache blocking since in our context, 2D cache blocking will not communicate significantly less than 1D cache blocking. As 2D cache blocks are processed temporally, they will effectively merge into a 1D cache block along the dimension they are being processed along. This effective 1D cache block will communicate the same as if it were truly a 1D cache block.

B. Propagation Blocking

We assume there are an adequate number of bins such that $n/r < c$. Like the pull case, reading the CSR graph requires reading $(k+2)n/b$ cache lines. Reading the source vertex values requires n/b cache line reads and outputting the final scores requires n/b cache line writes. Propagating contributions writes $2kn/b$ cache lines in the binning phase and reads $2kn/b$ cache lines in the accumulate phase. Reusing the destination indices saves kn/b writes from the binning phase. Altogether, the number of cache lines propagation blocking reads is:

$$\left(3 + \frac{3}{k}\right)\frac{kn}{b}$$

and writes (when reusing destination indices):

$$\left(1 + \frac{1}{k}\right)\frac{kn}{b}$$

C. Commentary

For the pull technique, the miss rate $(1 - c/n)$ strongly impacts the amount of traffic. Comparing the total communication of propagation blocking to the pull technique, we see propagation blocking will be advantageous when:

$$b \geq \frac{3}{1 - c/n}$$

This tradeoff is intuitive, as the opportunity for propagation blocking is if the pull technique frequently misses the cache and does not utilize every word in each cache line transferred.

Propagation blocking is advantageous to cache blocking using an edge list when:

$$r \geq 2k + 2$$

To first order, the amount of traffic per vertex for cache blocking is proportional to r, while for propagation blocking it is proportional to k. These dominant factors correspond to the attribute each technique is blocking. Cache blocking breaks up the graph, and r is proportional to n/c. For larger graphs, cache blocking will use more blocks and reload the vertex values more times, which will decrease its communication efficiency. Propagation blocking breaks up the propagations, which are proportional to k. Thus, propagation blocking will not have a change in communication efficiency for larger graphs. From our simple models, we see propagation blocking will communicate less than cache blocking when the graph is sparse enough and has sufficiently more vertices than can fit in the cache.

Short Name	Description	# Vertices (M)	# Edges (M)	Degree	Symmetric	References
urand	Uniform Random Graph	134.2	2,147.5	16.0	Y	[8]
kron	Kronecker Synthetic Graph	134.2	2,125.7	16.0	Y	[9], [10]
twitter	Twitter Follow Links	61.6	1,468.4	23.8	N	[11]
friend	Friendster	124.8	3,612.1	28.9	Y	[12]
cite	Academic Citations	49.8	949.6	19.0	N	[13]
coath	Academic Coauthorships	119.9	1,293.8	10.8	Y	[13]
web	webbase-2001	118.1	632.1	5.4	N	[14]
webrnd	webbase-2001 Randomized	118.1	632.1	5.4	N	[14]

Table I
GRAPHS USED FOR EVALUATION. ALL GRAPHS COME FROM REAL-WORLD DATA EXCEPT URAND AND KRON.

VI. EVALUATION

To perform our evaluation, we use a suite of sparse, low-diameter graphs that come from a diverse range of applications (Table I). All of the graphs except urand and kron come from real-world data sources. Each graph also uses the vertex labelling provided by its original data source, which is often chosen intelligently. The uniform random graph (urand) represents the worst case for locality. The kron graph is generated akin to Graph500's input graphs, and when compared to urand which is the same size, it demonstrates the impact of a strong power-law degree distribution on locality. The twitter and friend graphs come from crawls of social networks. Using the data on academic publications provided by the Microsoft Academic Graph [13], we generate a graph of all coauthorships (with duplicate edges removed) and a graph of all paper citations. The graph web has a great vertex labelling, and to show the locality benefit of that labelling, we randomize web's labelling to produce webrnd. Since PageRank computation is proportional to the number of directed edges, in this work, we use the directed degree since we find it to be a more instructive metric. The directed degree of an undirected (symmetric) graph is twice its average degree.

For consistency, we start all of our implementations from the same codebase, and we use the GAP Benchmark Suite's implementation of PageRank [3], [15]. We use the following:

- **Baseline** is the reference implementation and it computes PageRank in the pull direction.
- **Cache Blocking (CB)** improves our baseline by performing 1D cache blocking. It computes in the push direction, and it uses an edge list for each block.
- **Propagation Blocking (PB)** implements our propagation-blocking technique in the push direction.
- **Deterministic Propagation Blocking (DPB)** improves upon PB by using the optimization of storing the destination indices separately so that during the binning phase, only the contributions need to be written.

For our blocking implementations, we first tune the block width and then compute the number of blocks based on the width. After testing many block widths, we determined our implementations perform best when the corresponding vertex value array segments are 512 KB. We explore the bin width tradeoffs for propagation blocking later in this section. Each result in this section is the average of multiple trials of a single iteration of PageRank. We do not include the time to block the graph for CB or to allocate the bins for PB, as these can be done in advance or reused for other algorithms.

Codebase	Time (s)	Memory Reads (M)	Reads / second (M)	Instructions Executed (B)
Baseline	2.49	2,269	911.7	16.2
CSB (SpMV)	4.12	2,504	608.0	58.4
Galois	5.06	2,535	501.3	44.9
GraphMat	3.75	2,338	623.1	88.8
Ligra	4.54	3,983	877.8	36.1

Table II
PERFORMANCE OF A SINGLE PAGERANK ITERATION ON URAND GRAPH. BASELINE SIGNIFICANTLY OUTPERFORMS PRIOR WORK.

To perform our evaluation, we use a dual-socket Intel Ivy Bridge server with E5-2667 v2 processors, similar to what one would find in a datacenter. Each socket contains eight 3.3 GHz cores and 25 MB of last-level cache (LLC). The server has 256 GB of DDR3-1600 DRAM provided by 16 DIMMS. To access hardware performance counters, we use Intel PCM [16]. We compile all code with gcc-4.8, except the external baselines that use Cilk from icc 14. To ensure consistency across runs, we disable Turbo Boost (dynamic voltage and frequency scaling) and use only one thread per core (no hyperthreads). All results in this work use all 16 cores except Figure 3.

A. Baseline Validation

To ground our work, we validate the performance of our baseline implementation by comparing it to four established codebases. We use the PageRank implementations from Galois [17], GraphMat [18], and Ligra [19]. We also use the Compressed Sparse Blocks (CSB) SpMV implementation, but since it does not perform the additional computations necessary for PageRank, our measurements overestimate its performance for PageRank [20].

Table II presents the performances of our baseline and the four established codebases computing a single iteration of PageRank. Although our baseline implementation is simple, its litheness allows it to communicate the least and execute

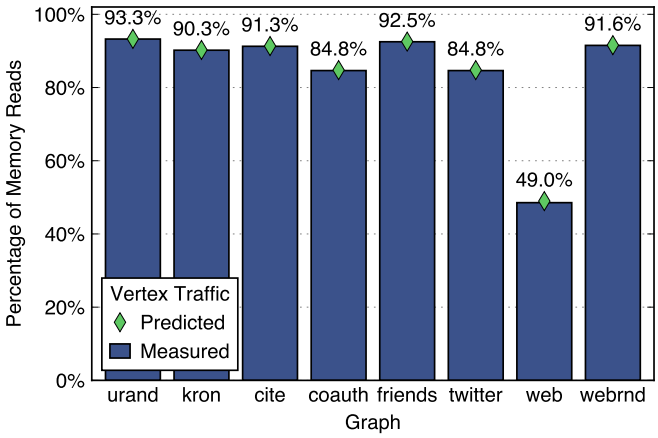

Figure 3. Vertex value traffic consumes the majority of memory requests for the low-locality graphs. The web graph has high locality.

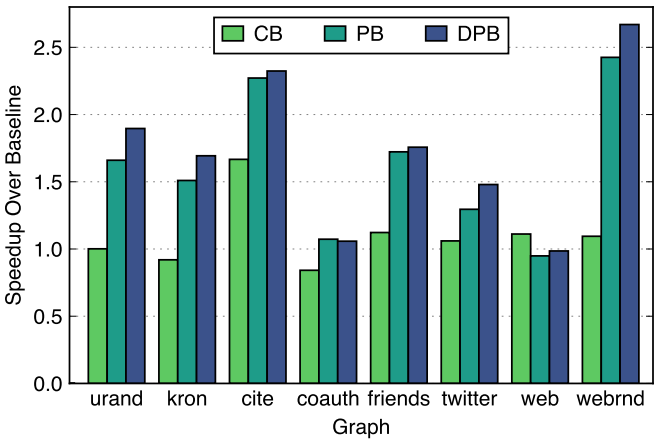

Figure 4. Execution time improvement over baseline. Blocking approaches improve performance unless the graph already has high locality (web).

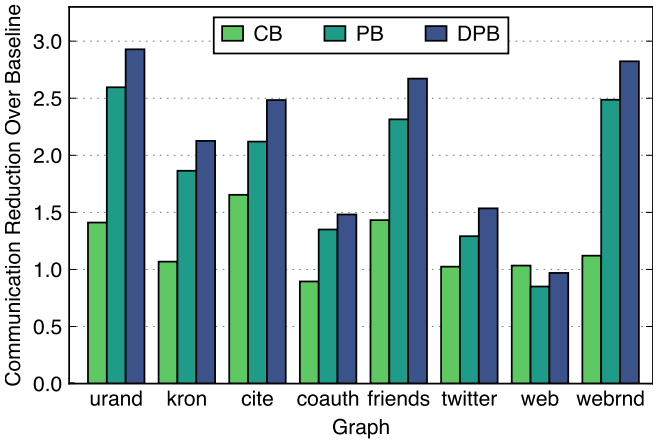

Figure 5. Communication volume reduction over baseline. Blocking reduces communication unless the graph already has high locality (web).

the fewest instructions while still utilizing the most memory bandwidth. Both our baseline and Ligra obtain high memory bandwidth utilization (with synthetic microbenchmarks we achieve a maximum of 1,191 M memory requests/second), but Ligra is slower because it performs additional communication. CSB, Galois, and GraphMat all execute so many additional instructions that their memory bandwidth utilization is bottlenecked by the instruction window size [5]. Overall, our baseline implementation is substantially faster than prior work ($> 1.5\times$), so by transitivity, any performance improvements over our baseline represent substantial improvements over prior work.

B. Quantifying the Opportunity for Propagation Blocking

To gauge the opportunity for how much our blocking techniques can improve locality, we first measure how much locality there is in our benchmark graphs. In theory, the amount of memory communication for the vertex traffic and the edge traffic (reading the graph) should be equal, but if the vertex value accesses have low locality, the vertex traffic can consume much more than half of the memory traffic. Figure 3 shows that most of our input graphs have low-locality layouts, as the vertex traffic consumes far more than the expected 50%. To measure the fraction of edge memory traffic, we process each graph twice: once with our baseline implementation, and once only reading the graph. The traffic we measure for reading only the graph is also in close agreement with our model from Section V.

The web and webrnd graphs have exactly the same topology, but web's optimized layout enjoys many more cache hits for its vertex accesses, which in turn reduces the total number of memory requests. This impact is visible in Figure 3, as the same number of accesses to read the vertex values for the web graph are reduced to near the expected 50%. Although the kron graph is the same size as the urand graph, its power-law degree distribution improves the temporal locality of vertex value accesses, and so it too enjoys more cache hits, reducing its vertex traffic. Overall, all of our input graphs except the web graph have low locality and could thus be amenable to blocking.

C. Comparing Blocking Approaches

All three blocking implementations deliver substantial performance (Figure 4) and communication (Figure 5) improvements relative to our baseline. The reductions in communication are greater than the reductions in execution time because our baseline implementation utilizes more memory bandwidth. The web graph defies these trends, because it is the only graph with a high-locality layout.

The detailed results in Table III show that propagation blocking greatly reduces the amount of read traffic, but its overall communication reduction is reduced by the additional write traffic to store propagations to bins. To perform propagation blocking, PB and DPB perform on average four times as many instructions as the baseline,

	Pull Baseline				Propagation Blocking (PB)				Deterministic Propagation Blocking (DPB)			
	Time	Memory (M)		Instructions	Time	Memory (M)		Instructions	Time	Memory (M)		Instructions
Graph	(s)	Reads	Writes	Exec. (B)	(s)	Reads	Writes	Exec. (B)	(s)	Reads	Writes	Exec. (B)
urand	2.50	2,269.1	162.9	16.2	1.50	467.0	469.8	76.8	1.32	481.0	349.5	74.1
kron	2.03	1,570.3	158.9	17.3	1.34	463.7	463.7	76.2	1.20	472.5	340.7	73.2
cite	1.30	777.5	77.4	6.9	0.57	202.8	200.4	33.7	0.56	203.3	140.9	32.4
coauth	0.99	673.8	123.1	10.9	0.92	297.6	292.7	47.9	0.93	308.4	229.5	47.0
friends	3.72	3,285.2	219.7	23.4	2.16	753.5	760.4	125.5	2.12	769.9	541.9	120.6
twitter	1.02	686.0	103.9	9.7	0.79	307.8	304.0	51.7	0.69	305.3	209.2	49.0
web	0.44	161.8	127.3	7.6	0.46	173.8	166.2	25.9	0.45	172.7	125.6	24.9
webrnd	1.22	697.1	139.3	7.7	0.50	169.0	167.4	25.9	0.46	168.7	127.5	24.9

Table III
DETAILED PERFORMANCE RESULTS FOR BASELINE AND BOTH PROPAGATION BLOCKING IMPLEMENTATIONS

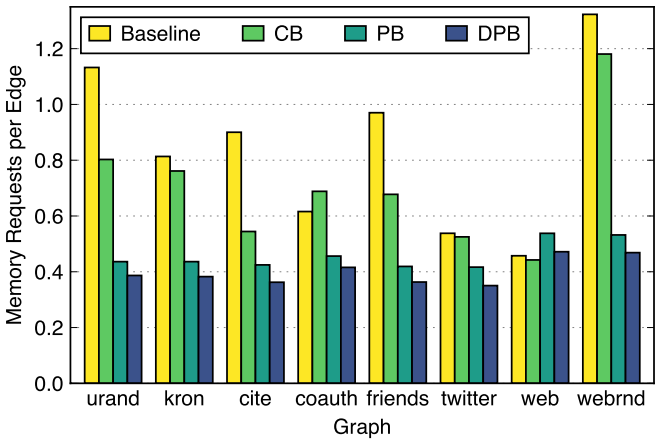

Figure 6. Memory requests per edge. As model predicts, propagation blocking (PB & DPB) performs nearly constant communication per edge.

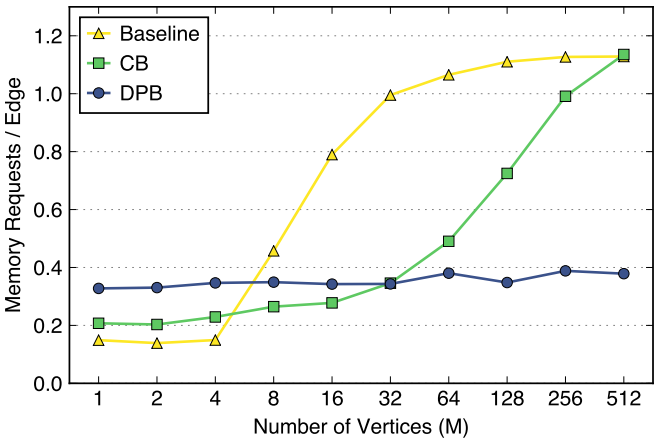

Figure 7. Communication efficiency for uniform random graphs of degree=16 and varied number of vertices. DPB is best for large graphs.

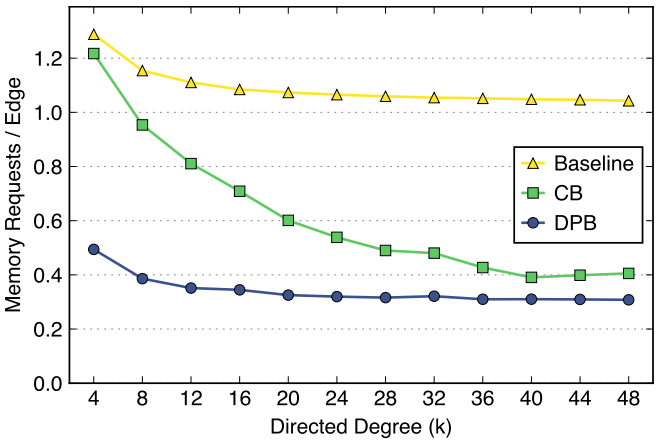

Figure 8. Communication efficiency for uniform random graphs of 128 M vertices and varied degree. DPB is best for sparse graphs.

and this reduces propagation blocking's memory bandwidth utilization. DPB's optimization of reusing destination indices substantially reduces the amount of write traffic, which in turn boosts performance.

Figure 6 shows the amount of communication normalized by the number of directed edges in the graph. Since all implementations process the same number of edges, this ratio from the GAIL metrics allows us to concisely compare communication efficiencies [21]. In general, memory communication efficiency for propagation blocking is consistent, so whether propagation blocking is more efficient that the baseline is determined by how efficient the baseline is. Figure 6 shows that the blocking implementations fail to obtain a large improvement over the baseline on the web graph because the web graph's high locality naturally reduces communication for the baseline, which gives the baseline much of the benefit of blocking.

To quantify the topological properties necessary for blocking to be advantageous, we artificially control locality by generating graphs of the same degree but with varying numbers of vertices (Figure 7). With fewer vertices, the vertex values are more likely to remain in cache. For our baseline, once the graph becomes sufficiently large, it overflows the cache and moves more data. For CB, the number of blocks increases along with the number of vertices, which causes the vertex values to be re-read more times. The constant number of memory requests per edge for DPB indicates that the number of edges is the primary determinant of memory traffic, and it is not the expected hit rate (c/n) like the

baseline, or the number of blocks r (proportional to n/c) for CB. Thus, which approach communicates the least depends on the number of vertices relative to the cache size. For the smallest graphs, blocking is unmerited and our baseline is the most communication efficient. For mid-size graphs, cache blocking is the most efficient, but as the graph gets too large, the overhead of reloading the vertex values makes it less efficient. For the largest graphs, propagation blocking provides the most scalable communication.

In Figure 8 we vary the degree of the input graph to find the sparsity for which DPB is advantageous to CB. Since all of the graphs have the same number of vertices, CB will use the same number of blocks. For denser graphs, CB's communication efficiency improves because it has more useful work to amortize the compulsory traffic of reloading the vertex values for each block. In this experiment, DPB communicates substantially less than CB if the directed degree is 36 or less. For graphs with more vertices, that degree cutoff will be higher, since CB will have more compulsory traffic to amortize. These experiments confirm our prediction from Section V that propagation blocking will be more communication efficient for graphs that are larger and sparser.

D. Selecting Bin Size

We vary the bin width to determine its impact on propagation blocking's performance. Once the vertex value array slices that correspond to each bin are small enough to fit in the cache, there is not much change in communication volume (Figure 9). The web graph is insensitive to bin width since its high-locality layout obviates blocking. Once memory communication is minimized, there is additional execution time benefit to using slightly smaller bins (Figure 10). However, making the bins too small makes them too numerous, which causes more L1 cache misses for bin insertions during the binning phase. These L1 misses reduce performance, but they do not greatly increase memory traffic because they result in mostly L3 hits. For our platform, we select a bin width of 512 KB, as it is typically the fastest while communicating little.

The number of bins is a tradeoff between locality in the two phases of propagation blocking. Increasing the number of bins reduces the likelihood the bin insertion points will remain in cache during the binning phase, but it decreases the bin width which in turn increases the likelihood of the sums remaining in cache during the accumulate phase. Figure 11 shows the time spent in each phase while processing the urand graph, and it shows the tradeoff between cache misses in the binning phase or the accumulate phase.

The results indicate our DPB implementation can reduce communication substantially, but there are cases when our other implementations communicate less. The locality of the graph determines whether one should use the pull baseline (high locality) or either CB or DPB (low locality).

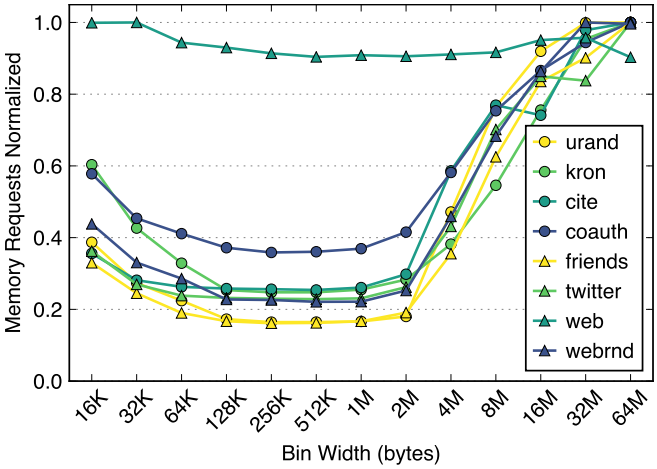

Figure 9. Impact of bin width on communication volume of propagation blocking. Once bins are sufficiently small, communication is reduced.

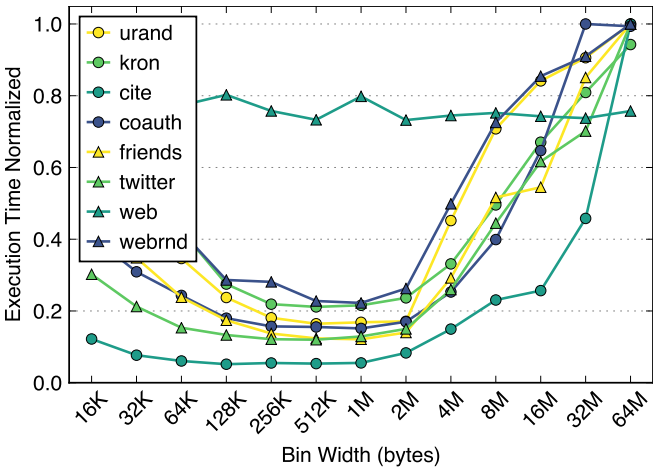

Figure 10. Impact of bin width on execution time of propagation blocking. We select 512 KB due to consistently good performance.

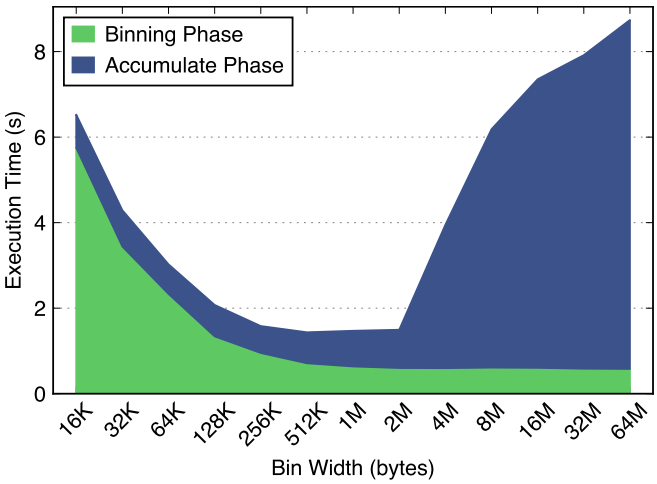

Figure 11. Execution time breakdown for DPB on urand. Our selected width of 512 KB balances time between the two phases.

Unfortunately, a graph's locality is not easy to measure quickly, but hopefully from the context of the application there should be hints as to its locality. The degree and the number of vertices determine whether one should use DPB (lower degree and more vertices) or CB (higher degree and fewer vertices). Fortunately, those topological parameters are easy to access and the decision to use DPB or CB could be made dynamically at runtime.

VII. Implementation Details

To improve the performance of our propagation-blocking implementation, we use a number of optimizations. We restrict our bin widths to powers of two so we can use a shift instructions to quickly compute a destination bin (instead of an integer divide), and this improves performance during the binning phase. To reduce the amount of communication during the binning phase, we use non-temporal stores and other optimizations from prior work on radix partitioning [22], [23], [24]. By using Intel's non-temporal (streaming) store instructions, we instruct the processor to bypass the cache when performing the writes to the bins, which obviates the read from memory for the write allocate [25]. We use small cache line-aligned fixed-size buffers to coalesce our writes for the streaming stores. For copying from our buffers in cache to the bins in memory, we obtain the best performance using the AVX non-temporal stores, but we also experimented with the SSE2 streaming stores and the ERMSB idiom [25]. During the accumulate phase, we use software prefetch instructions to improve memory bandwidth utilization for reading the currently processed bin.

To parallelize propagation blocking, we make different design decisions for each phase. For the binning phase, we give each thread its own set of bins to avoid atomicity concerns, and we use static scheduling so we can anticipate how large to allocate the bins in advance. To balance our static work allocation, we assign work based on the number of edges rather than vertices since degrees can vary substantially. For the accumulation phase, we dynamically assign vertex ranges to threads, and each thread processes its range's constituent bins. Since only one thread processes a vertex range, there are no need for atomics. When increasing the number of active threads on a given hardware platform, we find it is often best to decrease the bin width since the additional threads contend for the same cache capacity.

VIII. Related Work

SpMV is communication-bound, and as a consequence, much of the prior work on optimizing for performance has naturally also optimized communication [26]. Bender et al. provide complexity bounds on the amount of communication for SpMV [27], and their I/O model matches our model of a cache and main memory. Nishtala et al. provide criteria for when cache blocking will be advantageous for SpMV, and it is consistent with our results [7]. For example, we find the number of vertices (x in their formulation) and the randomness of the matrix to be important requirements.

There has been extensive prior work on reordering graphs and sparse matrices in order to improve locality, but unfortunately, no reordering technique is beneficial for all input graphs [6]. The Cuthill-McKee [28] technique and its follow-on RCM [29] apply BFS as a heuristic to find a good ordering. Compressing graphs exposes many of the same locality challenges, and the WebGraph framework is specifically designed to compress web crawls [30].

Relabelling the graph transforms the graph spatially, but the graph can also be transformed temporally. Changing the order in which edges are read from the graph can improve locality. Cache-oblivious algorithms use space-filling curves to obtain reasonable locality without any knowledge of the size of the cache, and they have been successfully applied to SpMV [31] and PageRank [32]. Unfortunately, these temporal transformations can greatly complicate parallelization.

In addition to reordering graphs to improve locality, there is also prior work on blocking to improve locality for graph processing. Park et al. demonstrate locality benefits from tiling for the all pairs shortest paths problem [33]. Xie et al. accelerate PageRank by breaking the graph into blocks and iterating more frequently on more rapidly changing blocks [34]. The milk language extension automatically batches up indirect memory references in order to improve locality [35]. Zhang et al. demonstrate the locality benefits of relabelling a graph based on vertex degree, and they introduce CSR segmenting, a more efficient means of 1D cache blocking [36]. We recently learned of related work by Buono et al. that improves locality for SpMV by breaking the computation into two phases, and their approach is quite similar to propagation blocking [37]. Azad et al. extend that two-phase approach to handle SpMV with sparse vectors [38].

IX. Discussion

In this work we introduce propagation blocking by applying it to PageRank, but it is more widely applicable. More broadly, propagation blocking reduces communication for a sparse all-to-all transfer by using binning to bound the irregularity. We originally conceived of propagation blocking to improve the locality of inter-vertex message passing within GBSP, a bulk-synchronous parallel (BSP) domain-specific language (DSL) for graph processing [39], [40].

Propagation blocking can be easily extended to handle more general forms of SpMV, such as SpMV on non-square matrices and non-binary matrices. To support weighted graphs (non-binary matrices), the weights can be read in lockstep with the adjacencies and applied directly to the contributions during the binning phase. Propagation blocking can also be applied to SpMV-centric graph processing models as well as many vertex-centric programming models that operate in the push direction.

There are additional benefits to propagation blocking. Since the amount of communication for propagation blocking is proportional to the number of propagations, unlike cache blocking, propagation blocking experiences no loss in communication efficiency if only a subset of the vertices are active. Another benefit of propagation blocking is the predictability of its memory access patterns ease its implementation for systems with scratchpad memories. Since the access ranges are bounded, all of the necessary data can be transfered in bulk by software between the on-chip local store and off-chip memory.

In our evaluation, we compare the time it takes to perform PageRank once the graph has been loaded and optimized, but the time to optimize the graph is also worth considering. It may be worthwhile to optimize the graph less if the reduction in graph preprocessing time is greater than the increase in kernel execution time. Fortunately, preparation for propagation blocking is substantially faster than preparation for cache blocking or relabelling a graph, thus making propagation blocking advantageous in more usage scenarios.

As demonstrated by our benchmark graph web, when a high-locality graph layout is available, communication for PageRank is naturally reduced. Unfortunately, such high-locality graph layouts are not always available. Blocking is a great way to improve locality, and the amount of communication for cache blocking is proportional to the number of vertices while for propagation blocking it is proportional to the number of edges. Thus, for large sparse graphs, propagation blocking will communicate less, and our simple communication models and performance results demonstrate this.

ACKNOWLEDGEMENTS

Research partially funded by DARPA Award Number HR0011-12-2-0016, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, Nokia, NVIDIA, Oracle, and Samsung. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

REFERENCES

[1] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," *International Parallel & Distributed Processing Symposium (IPDPS)*, 2013.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999.

[3] S. Beamer, K. Asanović, and D. A. Patterson, "The GAP benchmark suite," arXiv:1508.03619, August 2015.

[4] P. Berkhin, "A survey on pagerank computing," *Internet Mathematics*, vol. 2, no. 1, pp. 73–120, 2005.

[5] S. Beamer, K. Asanović, and D. A. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," *International Symposium on Workload Characterization (IISWC)*, 2015.

[6] J. Shun, "Shared-memory parallelism can be simple, fast, and scalable," Ph.D. dissertation, Carnegie Mellon University, 2015.

[7] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007.

[8] P. Erdős and A. Réyni, "On random graphs. I," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[9] "Graph500 benchmark." `www.graph500.org`.

[10] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, "Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication," *European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2005.

[11] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" *International World Wide Web Conference (WWW)*, 2010.

[12] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012.

[13] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, and K. Wang, "An overview of Microsoft academic service (MAS) and applications," *World Wide Web Consortium (W3C)*, 2015.

[14] T. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, pp. 1:1 – 1:25, 2011.

[15] "GAP benchmark suite reference code v0.8." `https://github.com/sbeamer/gapbs`.

[16] "Intel performance counter monitor." `www.intel.com/software/pcm`.

[17] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," *Symposium on Operating Systems Principles (SOSP)*, 2013.

[18] N. Sundaram, N. R. Satish, M. M. A. Patwary, S. R. Dulloor, S. G. Vadlamudi, D. Das, and P. Dubey, "GraphMat: High performance graph analytics made productive," *Proceedings of the VLDB Endowment*, 2015.

[19] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013.

[20] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009, pp. 233–244.

[21] S. Beamer, K. Asanović, and D. A. Patterson, "GAIL: The graph algorithm iron law," *Workshop on Irregular Applications: Architectures and Algorithms (IA^3), at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2015.

[22] O. Polychroniou and K. A. Ross, "A comprehensive study of main-memory partitioning and its application to large-scale comparison-and radix-sort," in *SIGMOD International Conference on Management of Data*, 2014, pp. 755–766.

[23] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich, "On the surprising difficulty of simple things: the case of radix partitioning," *Proceedings of the VLDB Endowment*, vol. 8, no. 9, pp. 934–937, 2015.

[24] J. Wassenberg and P. Sanders, "Engineering a multi-core radix sort," in *Euro-Par Parallel Processing*, 2011, pp. 160–169.

[25] "Intel 64 and IA-32 architectures optimization reference manual," *Intel Corporation*, September 2015.

[26] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," *Parallel Computing*, vol. 35, no. 3, pp. 178–194, 2009.

[27] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari, "Optimal sparse matrix dense vector multiplication in the I/O-model," *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2007.

[28] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *ACM National Conference*, 1969.

[29] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer, "An algorithm for reducing the bandwidth and profile of a sparse matrix," *SIAM Journal on Numerical Analysis*, vol. 13, no. 2, pp. 236–250, 1976.

[30] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *International World Wide Web Conference (WWW)*, 2004, pp. 595–601.

[31] A.-J. N. Yzelman and R. H. Bisseling, "A cache-oblivious sparse matrix–vector multiplication scheme based on the hilbert curve," in *Progress in Industrial Mathematics at ECMI*. Springer, 2012, pp. 627–633.

[32] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what COST?" *Hot Topics in Operating Systems*, 2015.

[33] J. Park, M. Penner, and V. Prasanna, "Optimizing graph algorithms for improved cache performance," *Transactions on Parallel and Distributed Systems*, pp. 769–782, 2004.

[34] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke, "Fast iterative graph computation with block updates," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, 2013.

[35] V. Kiriansky, Y. Zhang, and S. Amarasinghe, "Optimizing indirect memory references with milk," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 299–312, 2016.

[36] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. Amarasinghe, "Optimizing cache performance for graph analytics," arXiv:1608.01362, August 2016.

[37] D. Buono, F. Petrini, F. Checconi, X. Liu, X. Que, C. Long, and T.-C. Tuan, "Optimizing sparse matrix-vector multiplication for large-scale data analytics," *International Conference on Supercomputing (ICS)*, 2016.

[38] A. Azad and A. Buluç, "A work-efficient parallel sparse matrix-sparse vector multiplication algorithm," *International Parallel & Distributed Processing Symposium (IPDPS)*, 2017.

[39] S. Beamer, "Understanding and improving graph algorithm performance," Ph.D. dissertation, University of California, Berkeley, 2016.

[40] S. Kamil, D. Coetzee, S. Beamer, H. Cook, E. Gonina, J. Harper, J. Morlan, and A. Fox, "Portable parallel performance from sequential, productive, embedded domain-specific languages," *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012.