View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Queen's University Research Portal

Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning

Sun, J., Vandierendonck, H., & Nikolopoulos, D. S. (2017). Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning. In ICPP2017: The 46th International Conference on Parallel Processing: Proceedings (pp. 181-190). (International Conference on Parallel Processing (ICPP): Proceedings). IEEE . DOI: 10.1109/ICPP.2017.27

Published in:
ICPP2017: The 46th International Conference on Parallel Processing: Proceedings

Document Version:
Peer reviewed version

Queen's University Belfast - Research Portal:
Link to publication record in Queen's University Belfast Research Portal

Publisher rights
© 2017 IEEE.
This work is made available online in accordance with the publisher's policies. Please refer to any applicable terms of use of the publisher.

General rights
Copyright for the publications made accessible via the Queen's University Belfast Research Portal is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The Research Portal is Queen's institutional repository that provides access to Queen's research output. Every effort has been made to ensure that content in the Research Portal does not infringe any person's rights, or applicable UK laws. If you discover content in the Research Portal that you believe breaches copyright or violates any law, please contact openaccess@qub.ac.uk.

Download date:09. Sep. 2018

Accelerating Graph Analytics by Utilising the Memory Locality of Graph Partitioning

Jiawen Sun
Queen's University Belfast
Email: jsun03@qub.ac.uk

Hans Vandierendonck
Queen's University Belfast
Email: h.vandierendonck@qub.ac.uk

Dimitrios S. Nikolopoulos
Queen's University Belfast
Email: d.nikolopoulos@qub.ac.uk

Abstract

This paper investigates how to improve the memory locality of graph-structured analytics on large-scale shared memory systems. We demonstrate that a graph partitioning where all in-edges for a vertex are placed in the same partition improves memory locality. However, realising performance improvement through such graph partitioning poses several challenges and requires rethinking the classification of graph algorithms and preferred data structures. We introduce the notion of *medium-dense* frontiers, a type of frontier that is sufficiently dense for a bitmap representation, yet benefits from an indexed graph layout. Using three types of frontiers, and three graph layout schemes optimized to each frontier type, we design an edge traversal algorithm that autonomously decides which type to use. The distinction of forward vs. backward graph traversal folds into this decision and need no longer be specified by the programmer.

We have implemented our techniques in a NUMA-aware graph analytics framework derived from Ligra and demonstrate a speedup of up to 4.34× over Ligra and up to 2.93× over Polymer.

I. Introduction

Many important problems in social network analysis, artificial intelligence, business analytics and computational sciences can be solved using graph-structured analysis. There is increasing evidence that large-scale shared-memory machines with terabyte-scale main memory are well-suited to solve these graph analytics problems as they are characterized by frequent and fine-grain synchronization [1], [2], [3], [4], [5], [6].

The memory locality of graph analytics has been repeatedly recognized as an important issue [7], [8], [9], [10], yet few authors have proposed solutions to it. The locality issues in graph analytics result from the irregularity of the graph. Graph analytics operations compute a value associated to the edges or vertices of a graph by iteratively updating the values until convergence, or a fixed number of iterations have been performed [11], [12]. Values are updated as a function of the values on connected vertices. As such, each update requires to read and/or write values associated to the end-points of edges, which may be randomly dispersed through memory depending on the structure of the graph. Memory locality of graph analytics is thus highly data-dependent.

Graph partitioning is a powerful technique to isolate memory accesses to specific parts of the graph data. Graph partitioning allows to stage graph data in main memory from backing disk [5], [13], allows to distribute graph data in compute clusters [14] and allows to direct memory accesses to the locally-attached memory node in Non-Uniform Memory Access (NUMA) machines [3]. Prior works focus on how to direct memory accesses to specific subsets of the data. They have, however, not investigated the impact of graph partitioning on *temporal locality* of memory accesses.

X-stream [6] uses graph partitioning to improve *spatial locality*. To this end, X-stream places edge updates sequentially in buffers rather than applying them immediately to spatially scattered vertices. The buffers are sorted and shuffled by updated vertex ID such that the updates can be applied with excellent spatial locality. The shuffle stage, however, significantly increases execution time. While spatial locality is high, performance is sub-optimal.

In this work we leverage graph partitioning to improve temporal locality. We utilize a simple and efficient partitioning technique where all incoming edges of a vertex are assigned to the same partition [5], [3]. For such a partitioning scheme, we demonstrate that *reuse distances* are reduced as the graph is more finely partitioned.

We furthermore investigate how far we can scale graph partitioning in order to maximally benefit from temporal locality. Prior works have satisfied themselves with matching the number of graph partitions to the number of nodes in a distributed (scale-up) system [14], or to the number of NUMA nodes in a scale-out system [3]. Out-of-core systems determine the partitioning factor such that individual partitions fit in core memory [5], [13], [6]. In this work we target scale-up systems and we identify multiple good reasons to strive for high degrees of partitioning:

1) Improving temporal locality of memory accesses;
2) Confining all updates to a value to one partition and one thread, which boosts performance by avoiding hardware atomic operations;
3) Load balancing work across threads as the amount of work per partition varies with graph structure and active edge set.

Implementing a large number of partitions, however, requires careful design of the graph data structures. Two types of data structures are commonly used. The coordinate list (COO) [15] lists all edges as a pair of source and destination vertices. However, all edges must be traversed during each iteration of the graph algorithm, which is prohibitively expensive when few edges are active [1], [3], [16].

Representations such as Compressed Sparse Rows (CSR) and Compressed Sparse Columns (CSC) effectively provide an index into the edge list, allowing efficient lookup of the edges incident to active vertices. This representation, however, does not scale to a large number of partitions due to either edges or vertices crossing partitions, requiring replication of those edges or vertices in all relevant partitions [14]. We develop a composite graph storage scheme that combines CSC/CSR with COO in the best possible way to enable a large number of graph partitions. Our graph storage scheme moreover adapts to properties of the graph algorithm.

ALGORITHM 1: Partitioning by destination

input : Graph $G = (V, E)$; number of partitions P
output : Graph partitions $G_i = (V, E_i)$ for $i = 0, \ldots, P-1$

```
1 avg = |E|/P;                                          // target edges per partition
2 E_i = {} for i = 0, ..., P-1;
3 i = 0;
4 for v : V do
5 |   if |E_i| >= avg and i < P-1 then
6 |   |   ++i;                                          // i has exceeded target edges
7 |   E_i = E_i ∪ in-edges(v);                          // i is home partition of v
```

The remainder of this paper is organized as follows. Section II discusses our graph partitioning algorithm and its ensuing properties. Section III presents the rationale and design of our graph processing system. Section IV reports on our experimental evaluation of graph partitioning and shows how it improves performance. Section V summarizes related work.

II. Graph Partitioning

The key goal of graph partitioning in this work is to improve temporal locality of graph processing. This Section defines our graph partitioning approach and discusses the many ways in which graph partitioning affects performance.

A. Preliminaries

Graph-structured algorithms iteratively calculate attributes for each edge or vertex in a graph. These algorithms follow the Pregel [11] or gather-apply-scatter model [14] where updated values for a vertex are propagated along all its out-going edges to update the destinations. The vertices whose values are propagated are called *active vertices*. Out-going edges of an active vertex are *active edges*. The *frontier* is the current set of active vertices. Each iteration calculates a new frontier, which consists of the updated vertices. Graph algorithms iterate the same procedure until convergence, which is typically signified by an empty frontier.

Frontiers may be *sparse* or *dense* [16]. A sparse frontier contains few active vertices. A frontier is typically considered sparse if fewer than 5% of the edges are active. Other frontiers are considered dense. A sparse frontier is typically represented as a list of vertex IDs. A dense frontier is represented as a bitmap. As such, the algorithms to traverse sparse frontiers iterate only over the active vertices, while algorithms to traverse dense frontiers iterate over all vertices and check for each of them whether they are are active.

When frontiers are dense, the graph may be traversed in either *forward* or *backward* order. The forward traversal iterates over all active source vertices in the graph. The backward traversal iterates over all destination vertices, then iterates over all in-coming edges of the vertex and check whether the source vertices of these edges are active. While the backward traversal is more involved, a number of algorithms execute faster when using backward traversal [17]. In general, one needs to determine experimentally when an algorithm executes faster with a backward or a forward traversal [2].

B. Graph Partitioning

Graphs can be partitioned by either partitioning the vertex set or the edge set. Partitioning the vertex set implies that some edges cross partitions, which requires additional reduction or communication steps when processing these edges. Alternatively, one can partition the edge set. Ghost vertices model replicas of vertices in other partitions [18]. We focus our exposition on the graph partitioning proposed in GraphChi [5]. We partition the edge set by (i) partitioning the vertex set, (ii) deciding a home partition for each vertex and (iii) partitioning the edge set using the home partition of either the source or destination vertex of the edge. Each vertex is replicated in every partition that holds an edge incident to the vertex. Formally, assume a graph $G(V, E)$ where V is the vertex set and $E \subset V \times V$ is the edge set. The set of vertices V is partitioned in k non-overlapping sets by $\mathcal{P} = P_i, i = 0, \ldots, k-1$. $P_i \subset V$ and $\bigcup_{i=0}^{k-1} P_i = V$ for all i, and $P_i \bigcap P_j = \emptyset$ for all $i \neq j$. This partitioning of vertices supports two options for partitioning the edge-set:

- **Partitioning by destination**: all in-edges of a vertex are in the home partition of the vertex

$$G_P^{dst} = (V, \{(u, v) \in G.E : v \in P\}) \tag{1}$$

- **Partitioning by source**: all out-edges of a vertex are in the home partition of the vertex

$$G_P^{src} = (V, \{(u, v) \in G.E : u \in P\}) \tag{2}$$

These partitioning algorithms can be performed with a single pass over the edge list. Algorithm 1 shows the algorithm for partitioning by destination.

C. Locality of Partitioning

To understand why locality improves temporal locality, we study an example graph (Figure 1). The graph has 6 vertices and 14 edges. The CSR and CSC layouts of the whole graph are shown at the top of the figure. The same layouts are shown below when the graph is partitioned by destination.

When traversing the graph, we consult two types of data for each edge: For the source vertex we consult *current* data which includes a current frontier stored in bitmap format as well as an application-specific array storing per-vertex data. For the destination vertex we consult *next* data which includes the next frontier and the next version of the application-specific data. In the remainder we will simply label these *current arrays* and *next arrays*.

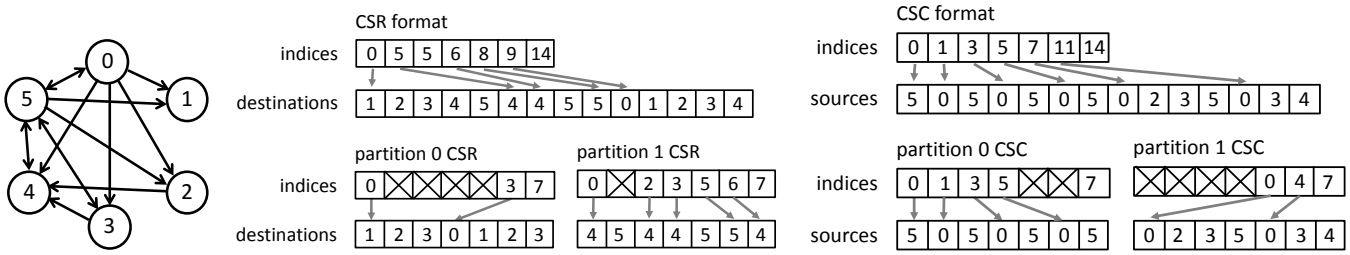


Fig. 1: Graph layout in CSR and CSC formats and the corresponding graph partitions when partitioning by destination.

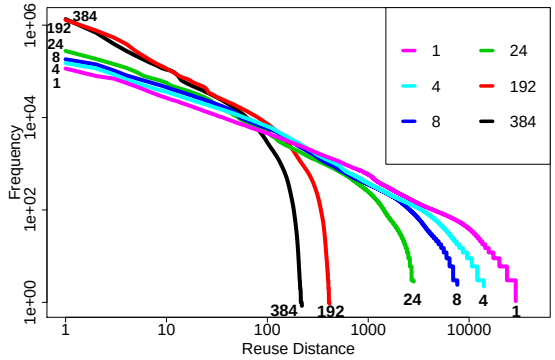


Fig. 2: Reuse distance distribution of updates to the next frontier in PRDeltafor the Twitter graph. The CSR layout is partitioned using partitioning-by-destination.

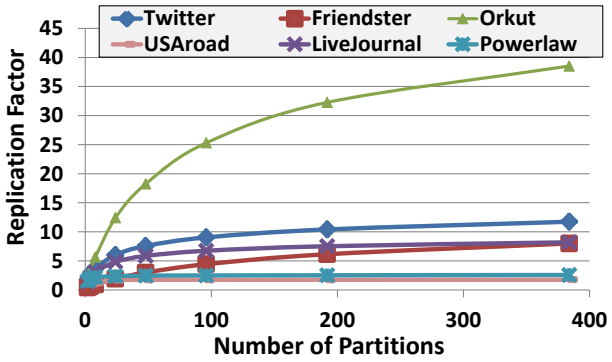


Fig. 3: The replication factor for a number of graphs and varying number of graph partitions. Results hold for partitioning by destination.

Consider a forward traversal over the whole graph using the CSR layout. We assume all vertices are set in the frontier for reasons of simplicity. As such, the iteration visits all edges. The order in which edges are visited is given by the *destinations* array, where source vertices are implicitly assigned in increasing order. As such, the current arrays are traversed sequentially and have excellent spatial locality. The next arrays are accessed with a random access pattern and may have bad spatial and temporal locality, depending on the structure of the graph, especially for large-scale graphs.

When partitioning the graph, in this example in two parts, we will independently traverse the partitions. Within each partition, we maintain sequential access to the current arrays and random access to the next arrays. The accesses to the next arrays are, however, confined to a subset of the vertices (those whose home partition is traversed). As such, the working set is smaller and temporal locality is improved. Looking across partitions, we traverse the current arrays twice (in general, once per partition). Each traversal is sequential but skips over some array elements. Depending on how many elements are skipped, spatial locality may be reduced.

Figure 2 shows the reuse distance distribution of accesses to the next frontier for the PRDelta algorithm applied to the Twitter graph (see Section IV for details on graphs and algorithms). As the number of partitions is increased, the range of reuse distances contracts, limiting the worst-case reuse distance. This restriction to shorter reuse distances is an immediate consequence of restricting the range of destination vertices. Moreover, the shorter reuse distances appear more frequently as the number of partitions is increased. This demonstrates the improved memory locality of graph partitioning.

The story is different for a backward traversal, which uses the CSC layout. Current arrays are accessed with a random access pattern, while next arrays are accessed sequentially. Due to partitioning by destination, however, edges are accessed in exactly the same order in the partitioned graph as in the whole graph. Partitioning-by-destination does not affect the memory locality of graph traversal. So, for CSC layout, we use the whole CSC graph layout with partitioned computation range.

Similar conclusions hold for partitioning-by-source: forward traversal visits edges in the same order as the whole graph while backward traversal observes improved temporal locality. We do not consider partitioning-by-source further in this paper as it incurs significant performance overhead. The overhead is the same in nature as that discussed below for partitioning-by-

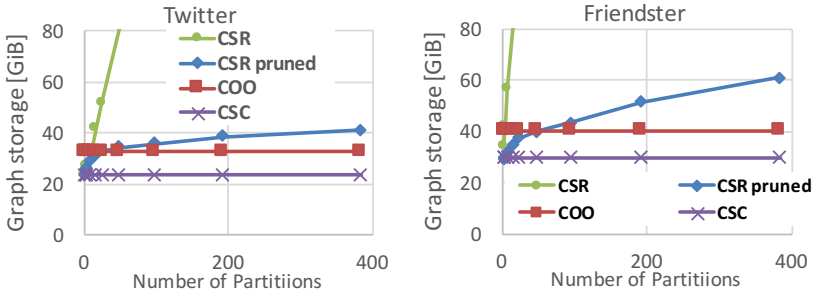


Fig. 4: Graph storage size for CSC/CSR and COO schemes for the Twitter and Friendster graphs.

destination. However, as backward traversal is most beneficial when frontiers are rather sparsely populated, it almost always affects performance negatively.

D. Vertex Replication

When partitioning the edge set, some vertices will appear in multiple partitions. These are called *replicated vertices* [14] or *ghost vertices* [3]. The replication factor quantifies the number of partitions where a vertex is replicated because it has incoming edges or outgoing edges that are assigned to that partition. A replication factor larger than one is a logical consequence of edge partitioning [14]. For instance, in Figure 1, the average replication factor is 7/6 (≈ 1.16) for the partitioned CSR layout.

The replication factor grows slower than a linear function of the number of graph partitions (Figure 3). E.g., with 16 partitions, a vertex in the Twitter graph has outgoing edges in 5 out of 16 partitions on average. This is comparable to the vertex-cut algorithm [14].

The worst-case replication factor is $r(|V|) = |E|/|V|$: when every vertex is placed in a distinct partition, then a vertex is replicated once per incident edge. The worst-case replication factor for Twitter is 35.2, while for Orkut it is 76.2. Figure 3 shows that the replication factor already becomes significant for 384 partitions: 11.7 for Twitter and 38.5 for Orkut. This effectively makes it appear as if the partitioned Twitter graph has 11.7 times as many vertices as the actual graph, which has adverse impact on memory usage and execution time.

E. Graph Storage Size

An immediate consequence of vertex replication is that the graph storage size grows with the number of partitions. We characterise the storage size for each graph layout assuming directed unweighted graphs to show the impact of graph partitioning. Let $r(p)$ be the replication factor for a p-way partitioned graph. We assume that zero-degree vertices are not stored in the CSR format. We store the vertex ID along with the vertex data in order to save space for zero-degree vertices [19]. The total storage cost for p partitions in CSR format can thus be expressed as:

$$r(p)\ |V|\ (b_e + b_v) + |E|\ b_v$$

where b_e is the storage in bytes for an index in the edge list and b_v is the storage in bytes for a vertex ID. Effectively, the number of vertices stored grows with a factor $r(p)$.

When both CSC and CSR formats are stored alongside each other [2] to implement the direction-reversing technique, the actual storage size needs to be doubled. Due to partitioning-by-destination does not affect the memory locality of graph traversal, during *backward* CSC traversal, it needs one whole graph CSC format for partitioning computation chunk. The storage size of the CSC format is $|E|b_v$+$|V|b_e$.

The storage size of the COO format is $2|E|b_v$ and is independent of the number of partitions. The factor 2 results as each edge stores both source and destination vertex ID.

Figure 4 shows how storage size varies with the number of partitions. The COO representation is independent of the degree of partitioning and shows a flat line. The CSR representation grows gradually in size, following a curve with the same incline as the replication factor (curve "CSR pruned"). Note that while the replication factor of Friendster is smaller than that of Twitter, the Friendster graph has many more vertices. This causes the storage size for the pruned CSR representation to grow more quickly.

When zero-degree vertices are not pruned from the CSR data structure, the storage size grows linearly with the number of partitions as $p\ |V|\ b_e + |E|\ b_v$. In this case, every vertex appears in every partition and the vertex ID need not be stored, reducing the storage per vertex to b_e bytes. Polymer does not prune zero-degree vertices from the representation [3].

This analysis shows that creating many partitions using the CSR formats requires significantly more storage space than working with one partition. This is prohibitive as the available main memory may be considered an inflection point: performance is high if we can work within the available main memory; it is poor when we need to work in an out-of-core manner. As such, one may manage using the CSR layout when sufficient memory is available. However, only the COO layout is scalable to large numbers of partitions.

F. Work Increase

Instruction count required to traverse the graph increases proportionally to the replication factor. Every vertex is visited as many times as it is replicated, i.e., $r(p)$ times on average. This replicates the actions of loading the vertex, checking its degree and whether it is active and iterating over (a subset of) its edges. As graph analytics typically require very little work per active edge, this control overhead has a noticeable impact already with 2 or 4 partitions.

The COO layout is again independent of vertex replication. The amount of work remains constant regardless of the number of partitions as each edge is visited once. The work is independent of how many vertices appear in a partition.

III. System Design

A. Graph Layout Options

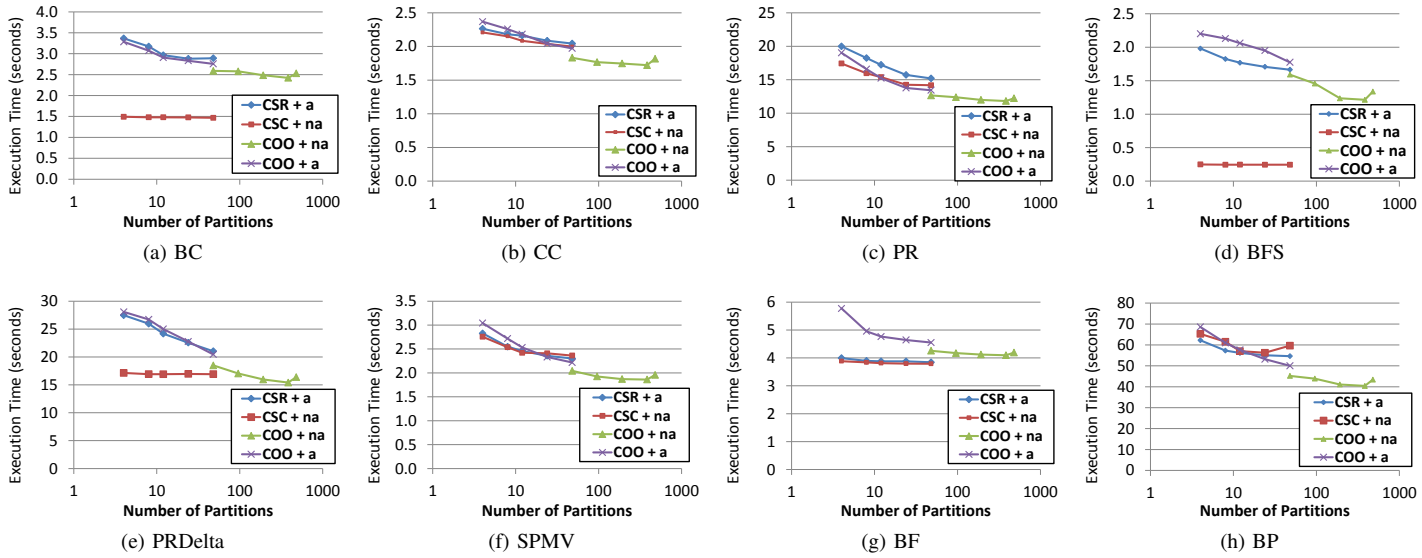
The graph layout is tuned to the density of the frontiers. Contrary to prior work, which distinguishes between sparse and dense frontiers [2], [3], [16], we introduce a third case where frontiers are "*medium-dense*".

1) Sparse Frontiers: When the frontier is sparse (typically less than 5% of vertices are active), little computation is done during traversal of active edges. A significant part of the time spent is covered by overhead in control flow. In this case, we found that there is little point in partitioning the graph [19] As there is not much useful work performed, the opportunity to improve locality is low. As such, we store a copy of the *unpartitioned graph in CSR layout* for the purpose of a traversal with sparse frontier.

ALGORITHM 2: Edge-map decision procedure. $deg_{out}(v)$ is the out-degree of vertex v.

input : Graph $G = (V, E)$ in multiple formats, frontier F in bitmap format and operation op
side effect : Operation op applied to all outgoing edges of the active vertices in F
output : New frontier storing active vertices in bitmap

if $|F| + \sum_{v \in F} deg\_out(v) > |E|/2$ **then** `// dense frontier`
  traversal of partitioned COO(G,F,op);
else if $|F| + \sum_{v \in F} deg_{out}(v) > |E|/20$ **then** `// medium-dense frontier`
  backward traversal of unpartitioned CSC(G,F,op);
else `// sparse frontier`
  forward traversal of unpartitioned CSR(G,F,op)
end



(a) BC (b) CC (c) PR (d) BFS

(e) PRDelta (f) SPMV (g) BF (h) BP

Fig. 5: Execution time as a function of the number of partitions and graph layout for Twitter. "+a" is with atomics, "+na" is without atomics. Atomics can be disabled only when each partition can be processed sequentially by one thread.

TABLE I: Characterization of real-world and synthetic graphs used in experiments.

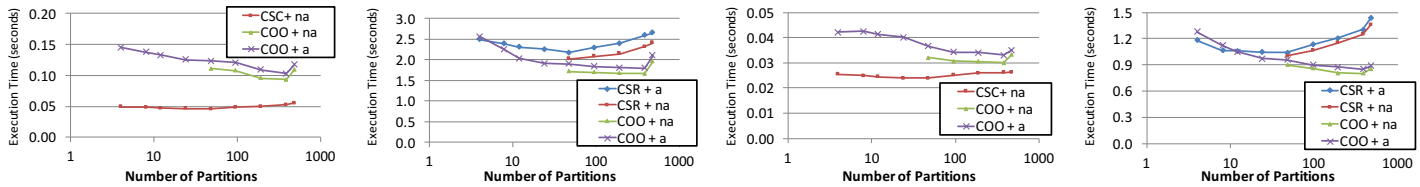
Graph	Vertices	Edges	Type
Twitter [21]	41.7M	1.467B	directed
Friendster [22]	125M	1.81B	directed
Orkut [23]	3.07M	234M	undirected
LiveJournal [22]	4.85M	69.0M	directed
Yahoo_mem [24]	1.64M	30.4M	undirected
USAroad [3]	23.9M	58M	undirected
Powerlaw ($\alpha = 2.0$)	100M	1.5B	directed
RMAT27	134M	1.342B	directed

TABLE II: Graph algorithms and their characteristics. V/E indicates vertex (V) or edge (E) orientation.

Code	**Description**	**Edge traversal**	**V/E**
BC	betweenness-centrality [2]	backward	V
CC	connected components using label propagation [2]	backward	E
PR	simple Page-Rank algorithm using power method (10 iterations) [20]	backward	E
BFS	breadth-first search [2]	backward	V
PRDelta	optimized Page-Rank forwarding delta-updates between vertices [2]	forward	E
SPMV	sparse matrix-vector multiplication (1 iteration)	forward	E
BF	Bellman-Ford algorithm for single-source shortest path [2]	forward	V
BP	Bayesian belief propagation [3] (10 iterations)	forward	E

2) Dense Frontiers: When the frontier is dense the majority of edges will be traversed. In this case, the COO layout is very efficient. We store the graph in a *large number of partitions in COO layout*. The number of partitions is in principle bounded only by the size of the graph. We will experimentally determine a good number of partitions.

3) Medium-Dense Frontiers: We introduce a new category of frontier density. A *medium-dense* frontier is dense enough to warrant representing the frontier as a bitmap, yet it is not dense enough to make a traversal over the COO layout fully efficient. Edge traversal on a medium-dense frontier is more efficient when using a CSR or CSC layout as it allows to skip over edges incident to inactive vertices. We will demonstrate that, when memory locality is addressed, algorithms with medium-dense frontiers perform best when using a backward traversal and a CSC layout. As partitioning by destination has no effect on the



(a) LiveJournal BFS (b) LiveJournal BP (c) Yahoo_mem BFS (d) Yahoo_mem BP

Fig. 6: Performance varying number of partitions for LiveJournal and Yahoo_mem. Backward traversal algorithms do not require atomics as the destination vertex is updated by a single thread. "+a" is with atomics, "+na" is without atomics.

CSC layout, it is immaterial whether the CSC layout is partitioned or not. We choose not partitioned.

B. Graph Traversal Decision Algorithm

The edge traversal procedure is summarised in Algorithm 2. We make the following choices: (i) sparse frontiers traverse the whole graph; (ii) medium-dense frontiers use the CSC layout with a backward traversal; (iii) the most dense frontiers use the COO layout. We employ two experimentally defined thresholds to decide which traversal is most appropriate for each frontier. The 5% threshold for sparse frontiers is commonly used in the literature. We experimentally determined that a 50% threshold to differentiate medium-dense frontiers from dense frontiers works reliably across algorithms and graphs. The number of graph partitions in the CSC and COO layouts has an important impact on performance. The best degree of partitioning differs between COO and CSC. As the COO layout takes storage independent of the number of partitions, we choose an aggressive partitioning degree.

Our design trades off memory usage against execution speed. Where the state-of-the-art stores 2 copies of the graph (CSC and CSR) [2], [3], [17], we store 3 copies. Note that the memory requirements are independent of the number of partitions as vertex replication does not increase memory consumption when partitioning the CSC and COO schemes by destination. As such, the memory requirement of our system is less than double the memory of Ligra.

C. Auxiliary Performance Benefits

Graph partitioning controls parallelism besides locality. When using *partitioning-by-destination* all the incoming edges of a vertex are located in the same partition and all partitions have **non-overlapping update sets**.

We ensure at least as many partitions as there are processing cores in the hardware. We process each partition by a single thread. Due to the non-overlapping nature of update sets of partitions, we can calculate updates **without using hardware atomic operations** such as compare-and-set. Atomic operations are very costly as graph analytics are already memory bound and the interconnection network between CPUs is highly loaded. We observed a speedup between 6.1% and 23.7% by removing atomic operations.

D. Implementation

We implement the graph partitioning techniques in GraphGrind, a NUMA-aware graph analytics framework [19]. Contrary to Polymer [3], another derivative of Ligra, GraphGrind is fully compatible with the Ligra [2] API and does not require users of the API to code in a POSIX threading model. GraphGrind relies on an extension to the Cilk programming language and runtime to indicate NUMA-aware scheduling of loops. The details of the baseline system are described elsewhere [19]. We will distinguish the modifications described below through the name "GraphGrind-v2". We will compare performance against Ligra and Polymer to demonstrate that GraphGrind is a state-of-the-art graph analytics framework.

We partition graphs *by destination*. We distinguish two distinct criteria to create balanced partitions, depending on the properties of the algorithm. We call a vertex-oriented graph algorithm an algorithm that performs nearly constant work per vertex during a traversal of the graph. In contrast, an edge-oriented graph algorithm performs nearly constant work per edge. Vertex-oriented algorithms include BFS, BC and Bellman-Ford. For these, we load-balance the traversal such that each thread processes an equal number of distinct source vertices, as this correlates with the amount of work per traversal. For edge-oriented algorithms, we load-balance the traversal to balance the number of edges per thread. The COO layout is always partitioned such that each partition has the same number of edges.

Each graph partition is allocated on one NUMA domain. Edge traversal using the dense operators are performed exclusively by CPU cores attached to the NUMA domain that stores the graph partition. Graph partitions are spread over all NUMA domains. As we have 4 NUMA domains on our experimental platform, we consider only multiples of 4 and allocate the same number of partitions on each NUMA domain. Frontiers represented as bitmaps and application-specific arrays storing attributes of vertices are allocated across the NUMA domains such that the attributes are stored on the NUMA domain that will update those values.

IV. Experimental Evaluation

We evaluate the locality benefits of graph partitioning experimentally on a 4-socket 2.6GHz Intel Xeon E7-4860 v2 machine, totaling 48 threads (we disregard hyperthreading due to its inconsistent impact on performance). It has 256 GB of DRAM. We compile all codes using the Clang compiler with Cilk support. We evaluate 8 graph analysis algorithms (Table II), using 8 widely used graph data sets (Table I). We exclusively present results using 48 threads and present averages over 20 executions.

Our analysis focusses primarily on the Twitter and Friendster graphs as these are the largest real-world graphs in our study. The other graphs respect the same conclusions.

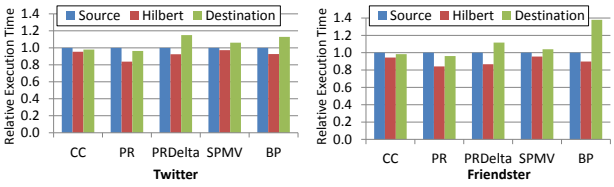


Fig. 7: Performance impact of sort order of edges. Experiments performed for 384 partitions with 48 threads. Execution times are normalised to sorting edges by source (CSR order).

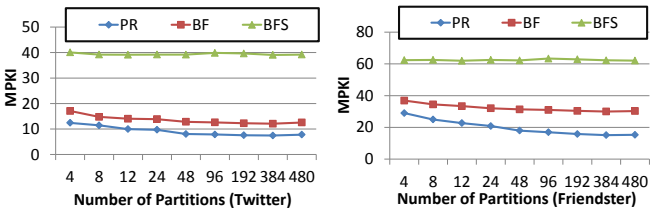


Fig. 8: Misses per kilo instructions (MPKI) of Hilbert-sorted COO.

A. Graph Layouts

We investigate partitioning of graph layouts using the 8 graph algorithms and the Twitter graph (Figure 5). We consider 4 configurations: the CSC and CSR layouts, the COO layout with atomic operations, and the COO layout without atomic operations. Atomics are unnecessary in the CSC layout due to grouping edges by their destination. They are unavoidable when using CSR due to partitioning by destination. Atomics can be avoided for COO when a partition is processed by one thread exclusively. In our experiments, this happens for 48 partitions or more as we evaluate using 48 threads.

The smallest partition number evaluated is 4 and corresponds to the left-most point on the curves. The CSC/CSR layout with 4 partitions is similar to Polymer and serves as a reference. With the CSC/CSR layout we quickly run out of memory. As such we can evaluate at most 48 partitions for the Twitter graph on our machine.

Whenever we have 48 partitions or more, we can avoid the use of hardware atomics. These operations are demanding on the coherence protocol. As graph analytics are memory bound, hardware atomics have an important performance impact. Figure 5 shows that at 48 partitions avoiding atomics results in 6.1% to 23.7% speedup (comparing COO+a vs COO+na).

The COO layout scales to a large number of partitions. All algorithms and graphs observe incremental performance benefits up to 384 partitions. Execution time starts to increase at 480 partitions due to increased scheduling overhead.

While prior work has labeled algorithms as *forward* (having faster traversal over CSR) or *backward* (having faster traversal over CSC), our results contradict the ruling classification reported in the literature and summarized in Table II. In contrast, we observe that vertex- vs. edge-orientation explains the results. Vertex-oriented algorithms perform best when using the CSC layout, while edge-oriented algorithms perform best using the COO layout with a high number of partitions. Whether the CSR, CSC or COO layout performs better in other situations depends on a complicated combination of factors including the density of frontiers, the cost of atomic operations, memory locality and, unavoidably, graph structure.

Moreover, the CSC layout has no, or hardly any, improved memory locality due to partitioning for vertex-oriented algorithms. In contrast, there is a performance improvement by increasing the number of partitions for edge-oriented algorithms. This is not due to enhanced locality. Instead, it follows from improved load balancing. Parallelizing traversal over a non-partitioned CSC or CSR implies that threads receive an equal number of vertices to traverse, which may correspond to a highly imbalanced number of edges per thread. Graph partitions, however, are designed to balance the edges, which enhances parallel efficiency.

The nature of vertex- vs. edge-orientation correlates strongly with the density of the frontier: in a vertex-oriented algorithm, updates of edges are by necessity propagated in a small fraction of the edges in order to achieve a time complexity proportional to the number of vertices. By not propagating most updates, the number of vertices activated each round will be small. As such, we can expect the CSC layout to out-perform COO on some iterations, while COO is faster during iterations with more dense frontiers. This is the case for PRDelta, an edge-oriented algorithm, where 8 frontiers are dense, 3 are medium-dense and 22 are sparse.

B. Emulating Unrestricted Memory Capacity

The results indicate that the benefits of the CSR layout are restricted by memory consumption. In order to understand what would be feasible with more main memory, we take a closer look at two small graphs: LiveJournal and Yahoo_mem (Figure 6). For these, we can scale up the number of graph partitions in CSR layout. We observe, however, that edge-oriented algorithms quickly see diminishing returns and a slowdown. This is a consequence of vertex replication which increases the amount of work. Vertex-oriented algorithms (e.g., BFS) do not observe significant performance variation when increasing the number of partitions. Other algorithms not shown here confirm these conclusions.

In all cases, avoiding atomics, which is possible with 48 partitions or more, reduces execution time. Note that in BFS there is no need to use atomics in the CSC case as it uses a backward edge traversal.

C. Sorting Edge Lists

The COO layout may store edges in various orders. Up to now, we have stored edges in the same order as the CSR representation, i.e., they are sorted by the source vertex. Alternatively, edges may be sorted by destination vertex, which corresponds to the same traversal order as CSC. Another option is to sort edges using a space-filling curve such as Hilbert order to improve memory locality [25], [26].

Comparing these three options for the Twitter and Friendster graphs shows that Hilbert sorting has consistently lowest execution time (Figure 7). It is up to 16.2% faster. While graph partitioning makes a large improvement to memory locality, traversing edges in Hilbert order further improves it.

Interestingly, the results reflect the preferred traversal order, showing that the CC and PR algorithms, which are edge-oriented backward algorithms, see better performance when sorting edges by destination (CSC order) compared to sorting by source.

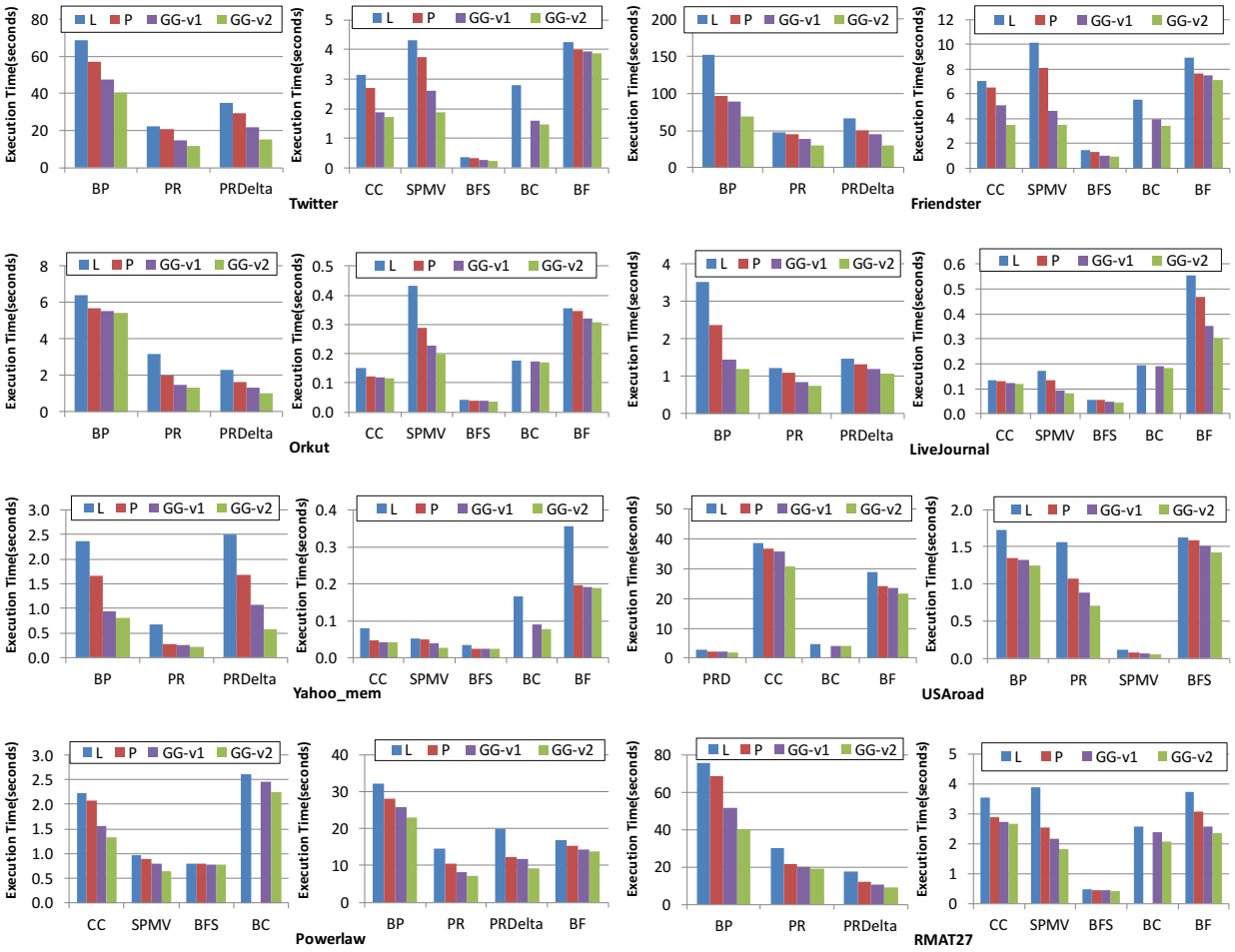


Fig. 9: Comparison of our graph traversal algorithm against Ligra (L), Polymer (P) and GraphGrind-v1 (GG-v1). Polymer and GraphGrind-v1 use 4 partitions to match the number of NUMA nodes in our machine, GraphGrind-v1 only uses CSC/CSR format. GraphGrind-v2 (GG-v2) uses 384 partitions for the CSC computation chunk size and the COO layout.

For the other algorithms, preferring forward traversal, the CSR order performs better. We conclude that the performance benefit of backward traversal is explained in part by improved memory locality. Prior work has identified that backward traversal is sometimes faster than forward traversal, but did not provide a thorough explanation [17].

D. Memory Locality

Figure 8 shows last-level cache misses per kilo instructions (MPKI) for a varying number of partitions. MPKI values are high as graph analytics is a memory-intensive workload. However, graph partitioning significantly reduces the MPKI values. E.g., for PR on the Friendster graph, MPKI is reduced by half: from 29.0 at 4 partitions to 15.1 at 384 partitions. This demonstrates that shorter reuse distances (Figure 2) translate to fewer main memory accesses, and higher performance. For BFS, a vertex-oriented algorithm, graph partitioning does not reduce cache misses.

E. Comparison to State-of-the-Art

Finally we compare against the state-of-the-art: Ligra [2], Polymer [3] and GraphGrind-v1 [19] (Figure 9). We limit the comparison to these systems as it has been established [3] that Polymer out-performs Galois [27] and X-stream [6]. Results for BC on Polymer are missing as Polymer does not provide an implementation for this algorithm.

GraphGrind-v2 out-performs by a significant margin Ligra, Polymer and GraphGrind-v1. For vertex-oriented algorithms, this is due to (i) using a non-partitioned CSR layout for traversals with sparse frontiers and (ii) enhanced memory locality by using 384 partitions in the CSC computation chunk size. Speedup for the vertex-oriented algorithms ranges from 2.35% to 37.31% over Ligra, from 0.28% to 19.9% over Polymer and from 0.6% to 15.1% over GraphGrind-v1.

Edge-oriented algorithms benefit from the enhanced memory locality due to graph partitioning and, to a lesser extent, sorting edges. Using a non-partitioned CSR layout for the sparse iterations is again beneficial. The speedup for PRDelta using our scheme is 138% for Twitter and 129% for Friendster. BP is sped up by 47.7% for LiveJournal and 108% for Yahoo_mem. The speedup ranges up to 4.34× over Ligra, up to 2.93× over Polymer and up to 45% over GraphGrind-v1, in both cases for PRDelta on Yahoo_mem.

Our technique moreover achieves high speedup on the USAroad graph, a road network graph that is hard to process for graph analytics frameworks. We achieve 23.9% speedup over Polymer for PRDelta and 22.6% for SPMV.

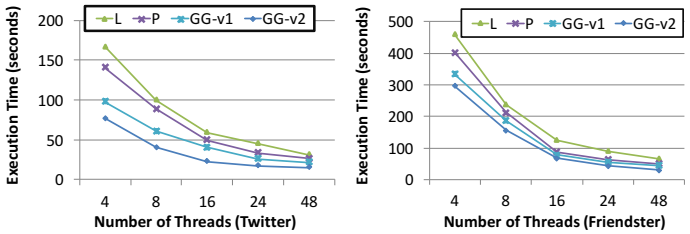


Fig. 10: Parallel scalability compared to Ligra (L) Polymer (P) and GraphGrind-v1 (GG-v1) for PRDelta.

F. Parallel Scalability

Finally, we demonstrate the scalability of our graph traversal algorithm for PRDelta (Figure 10). We show execution time starting at 4 threads, as fewer threads do not work well with Polymer and our system as they require at least 1 thread per NUMA node. Additional threads are spread uniformly across NUMA nodes. Increasing parallelism reduces execution time commensurately. E.g., on the Friendster graph, Polymer speeds up 6x from 4 to 48 threads, while our system speeds up 10x.

G. Selecting The Degree of Partitioning

Our framework has a hidden parameter that determines how many partitions are employed for the COO layout. This parameter may be exposed to programmers and users. However, it would be convenient to determine them heuristically. Our results show that graph partitioning scales to about 384 partitions for all graphs and algorithms. Further investigation is required to understand the mechanisms that determine performance as a function of partitioning degree.

V. Related Work

It has been documented that generic tools such as METIS [28] to partition graphs by vertex or edge cut do not produce good partitions for social network graphs. Moreover, they take much more time to compute than many graph algorithms. Sheep [29] is a distributed graph partitioner that produces high quality edge partitions an order of magnitude faster than METIS. Alternatively, linear-time heuristics have been proposed. The vertex cut is a greedy edge partitioning algorithm that minimizes the number of cut vertices [14].

The closest related work is X-stream [6], a graph processing framework that improves spatial locality. X-stream relies on streaming (sequential memory access) to maximize spatial locality. They parallelize the computation and stage data in the memory hierarchy using partitioning-by-source. Vertex updates are not applied immediately as these would result in random memory accesses with bad temporal and spatial locality. Instead they stream vertex updates into a buffer and perform an additional sorting and shuffling stage such that updates can be applied with high spatial locality. We perform graph partitioning differently: using partitioning-by-destination increases temporal locality in destination vertices, while accesses to source vertices retain high spatial locality. Moreover, each vertex is updated by one thread only and updates are applied to the locally-attached NUMA node.

Hilbert space filling curves have been applied to optimize the iteration order of edge list traversal [26], [25]. Space filling curves improve memory locality for various algorithms [30], [31], [32]. We have experimented with Hilbert curves and have achieved speedup compared to CSC and CSR layouts when the number of partitions is sufficiently high and hardware atomics can be avoided.

Murray *et al* [25] present a PageRank implementation with a COO layout and space filling curves. Their work uses graph partitioning to create parallelism. They consider partitioning-by-source and partitioning of the edge list after rearranging edges in Hilbert order. While the Hilbert order improves their performance by an order of magnitude, they have not considered partitioning-by-destination to improve locality and to ensure vertices are updated by a single thread. They also have no specific support for sparsely populated frontiers.

Zhang *et al* [3] optimize graph analytics for NUMA systems by graph partitioning, as discussed in the main text. Agarwal *et al* [1] optimize BFS for NUMA systems by distributing vertices across NUMA domains (vertex partitioning). They use intra-socket and inter-socket queues to distribute the active vertices to their home node with minimal contention. Frasca *et al* [33] define an Adaptive Data Layout (ADL) to reorganize the graph after observing parallel access patterns on NUMA system. Dai *et al* [34] apply graph partitioning in the context of multi-FPGA systems. Chronos is a graph engine supporting temporally evolving graphs [35]. As such, locality may be optimized along the time dimension or along the spatial dimension. The authors optimize the time dimension due to the challenging nature of locality in the spatial dimension.

VI. Conclusion

This work presents a graph processing framework for shared memory machines with large main memory capacity. It differs from prior work by *improving temporal locality* through graph partitioning, which controls the order of iteration over edges. The partitioning algorithm is designed such that each vertex is updated by at most one thread, obviating the need for synchronisation and the use of costly hardware atomics.

In order to scale to a large number of partitions, we use a combination of COO, CSR and CSC graph layouts. Prior work distinguishes between *sparse* and *dense* frontiers to optimize graph traversal. We identify a third frontier type, *medium-dense* frontiers, which are highly populated but nonetheless benefit from an indexed graph representation such as CSR or CSC. We moreover remove the requirement that programmers decide whether an algorithm runs faster when traversing the graph in a

backward or in a forward direction. In fact, we observe that this distinction is not accurate and show that the traversal order may be selected based on frontier density.

We evaluate our graph processing framework on a set of commonly used algorithms and graphs and demonstrate a speedup up to 4.34× over Ligra, up to 2.93× over Polymer and up to 45% over our previous version of GraphGrind.

An open question, which prior work has not answered in a conclusive manner [2], [17], is why a backward or forward traversal order is faster. We have demonstrated that frontier density and memory access order are important factors. A complete understanding of this subject may result in additional speedup of graph analytics.

ACKNOWLEDGMENT

This work is supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the ASAP project, grant agreement no. 619706, and by the United Kingdom EPSRC under grant agreement EP/L027402/1.

REFERENCES

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proc. of the ACM/IEEE Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–11.

[2] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*. 2013, pp. 135–146.

[3] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proc. of the ACM Symp. on Principles and Practice of Parallel Programming*. 2015, pp. 183–193.

[4] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proc. of the ACM Symp. on Operating Systems Principles*. 2013, pp. 456–471.

[5] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC." in *OSDI*, vol. 12, 2012, pp. 31–46.

[6] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. of the ACM Symp. on Operating Systems Principles*. 2013, pp. 472–488.

[7] L. Yuan, C. Ding, D. Tefankovic, and Y. Zhang, "Modeling the locality in graph traversals," in *Proc. of the Intl. Conf. on Parallel Processing*. 2012, pp. 138–147.

[8] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an Ivy Bridge server," in *Proc. IEEE Intl. Symp. on Workload Characterization*. 2015, pp. 56–65.

[9] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[10] G. Cong and S. Sbaraglia, "A study on the locality behavior of minimum spanning tree algorithms," in *High Performance Computing-HiPC*. 2006, pp. 583–594.

[11] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proc. of the ACM SIGMOD Intl. Conf. on Management of data*. 2010, pp. 135–146.

[12] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proc. of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[13] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu, "TurboGraph: A fast parallel graph engine handling billion-scale graphs in a single PC," in *Proc. of the ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*. 2013, pp. 77–85.

[14] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[15] Y. Saad, "SPARSKIT: A basic tool for sparse matrix computations," NASA, Tech. Rep. NASA-CR-185876, May 1990.

[16] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and gpu," in *Intl. Conf. on Parallel Architectures and Compilation Techniques*. 2011, pp. 78–88.

[17] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 12:1–12:10.

[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endow.*, vol. 5, no. 8, pp. 716–727, Apr. 2012.

[19] J. Sun, H. Vandierendonck and D.S. Nikolopoulos, "GraphGrind: Addressing Load Imbalance of Graph Partitioning," in *Proc. of the ACM International. Conference on Supercomputing*. 2017, pp. 16:1–16:10.

[20] 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, previous number = SIDL-WP-1999-0120.

[21] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proc. of the 19th Intl. Conf. on World wide web*. 2010, pp. 591–600.

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

[23] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and Analysis of Online Social Networks," in *Proc. of the ACM/Usenix Internet Measurement Conf.*, October 2007.

[24] Y. Vigfusson, "Affinity in distributed systems," Ph.D. dissertation, Cornell University, 2010.

[25] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proc. of the Twenty-Fourth ACM Symp. on Operating Systems Principles*. 2013, pp. 439–455.

[26] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what COST?" in *Workshop on Hot Topics in Operating Systems (HotOS XV)*. May 2015.

[27] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo *et al.*, "The tao of parallelism in algorithms," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 12–25, 2011.

[28] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96 – 129, 1998.

[29] D. Margo and M. Seltzer, "A scalable distributed graph partitioner," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1478–1489, Aug. 2015.

[30] J. Mellor-Crummey, D. Whalley, and K. Kennedy, "Improving memory hierarchy performance for irregular applications using data and computation reorderings," *Intl. Journal of Parallel Programming*, vol. 29, no. 3, pp. 217–247, 2001.

[31] C. Ding and K. Kennedy, "Improving cache performance in dynamic applications through data and computation reorganization at run time," in *Proc. of the ACM Conf. on Programming Language Design and Implementation*. 1999, pp. 229–241.

[32] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear array layouts for hierarchical memory systems," in *Proc. of the 13th Intl. Conf. on Supercomputing*. 1999, pp. 444–453.

[33] M. Frasca, K. Madduri, and P. Raghavan, "NUMA-aware graph mining techniques for performance and energy efficiency," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 95:1–95:11.

[34] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "Foregraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proc. of the 2017 ACM/SIGDA Intl. Symp. on Field-Programmable Gate Arrays*. 2017, pp. 217–226.

[35] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: A graph engine for temporal graph analysis," in *Proc. of the Ninth European Conf. on Computer Systems (EuroSys)*. 2014, pp. 1:1–1:14.