Everything you always wanted to know about multicore graph processing but were afraid to ask

Jasmina Malicevic
EPFL

Baptiste Lepers
EPFL

Willy Zwaenepoel
EPFL

Abstract

Graph processing systems are used in a wide variety of fields, ranging from biology to social networks, and a large number of such systems have been described in the recent literature. We perform a systematic comparison of various techniques proposed to speed up in-memory multicore graph processing. In addition, we take an end-to-end view of execution time, including not only algorithm execution time, but also pre-processing time and the time to load the graph input data from storage.

More specifically, we study various data structures to represent the graph in memory, various approaches to pre-processing and various ways to structure the graph computation. We also investigate approaches to improve cache locality, synchronization, and NUMA-awareness. In doing so, we take our inspiration from a number of graph processing systems, and implement the techniques they propose in a single system. We then selectively enable different techniques, allowing us to assess their benefits in isolation and independent of unrelated implementation considerations.

Our main observation is that the cost of pre-processing in many circumstances dominates the cost of algorithm execution, calling into question the benefits of proposed algorithmic optimizations that rely on extensive pre-processing. Equally surprising, using radix sort turns out to be the most efficient way of pre-processing the graph input data into adjacency lists, when the graph input data is already in memory or is loaded from fast storage. Furthermore, we adapt a technique developed for out-of-core graph processing, and show that it significantly improves cache locality. Finally, we demonstrate that NUMA-awareness and its attendant pre-processing costs are beneficial only on large machines and for certain algorithms.

1 Introduction

Interest in processing graph-structured data has grown over the last few years, especially for mining relationships in social network graphs. Many graph processing systems have been built, including single-machine, cluster-based, in-memory and out-of-core systems [7, 8, 12–14, 16, 17, 19, 20, 22, 23, 26, 27, 29, 33, 36, 37]. In this paper we focus on single-machine in-memory graph processing systems. With the recent increase in main memory size and number of cores, such machines can now process very large graphs in a reasonable amount of time.

With few exceptions [4, 28], most papers on graph processing systems present a new system and compare its performance (and occasionally its programmability) to previous systems. While interesting, these comparisons are often difficult to interpret, because systems are multi-dimensional, and therefore a variety of features may contribute to observed performance differences. Variations in hardware and software infrastructure, input formats, algorithms, graphs and measurement methods further obscure the comparison.

In this paper we take a different approach. Rather than comparing different systems, we compare different techniques used in graph processing systems, and we try to answer the question: what techniques provide what benefits for what types of algorithms and graphs? We implement various techniques proposed in different papers in a single system. We then selectively enable the different techniques, and compare the performance of the resulting approach on the same hardware platform for the same algorithms and graphs.

In particular we take an end-to-end view of graph processing, often absent in other papers. Graph processing involves loading the graph as an edge array from storage, pre-processing the input to construct the necessary data structures, executing the actual graph algorithm, and storing the results. Most papers focus solely on the algorithm phase, but we demonstrate that there is an important trade-off between pre-processing time and algorithm execution time. While we recognize that pre-processing can potentially be amortized over repeated executions, we show that gains in algorithm execution time can be completely undone by increases in pre-processing time.

We structure our investigation of algorithm execution time along two dimensions. In a first dimension, we distinguish between a vertex-centric approach, in which the algorithm iterates over vertices, and an edge-centric approach, in which the algorithm iterates over edges. In addition, we propose a new iteration approach, adapted from out-of-core systems [37], in which the algorithm iterates over grids, with improved cache locality as a result. In a second dimension, we distinguish between algorithms that push information to their neighbors, or pull information from them. We also consider algorithms that dynamically choose between push and pull.

To illustrate through a simple example the importance of an end-to-end view, we analyze the push-pull approach to Breadth First Search (BFS)[1]. Earlier papers [2, 3, 29] have demonstrated that, for BFS, a push-pull approach results in better algorithm execution time than the conventional push approach. Figure 1 shows the end-to-end execution time of BFS on the well-known Twitter follower graph [18] using both approaches. While the algorithm execution time is indeed 3× smaller for push-pull, the overall execution is completely dominated by pre-processing. The pre-processing time is 2× larger for push-pull, resulting in 1.5× worse overall end-to-end time.

In addition to different methods of iteration and information flow, various optimizations have been proposed to take advantage of memory locality on NUMA machines. These optimizations often take the form of partitioning data structures during pre-processing, such that most accesses during algorithm execution are local to a NUMA node. Continuing the theme of the trade-off of pre-processing versus algorithm execution times, we investigate whether such pre-processing pays off for graph processing.

The main results of this paper are:

- An illustration of the fundamental trade-off between pre-processing and algorithm execution time in graph processing.

- An evaluation of different techniques for building adjacency lists, showing that radix sort provides the best performance when the graph is in memory or when it is loaded from a fast storage medium.

- An evaluation of the pre-processing vs. algorithm execution time trade-off for vertex-centric vs. edge-centric computation, showing that the construction of adjacency lists for vertex-centric processing may or may not pay off, depending on the algorithm execution time.

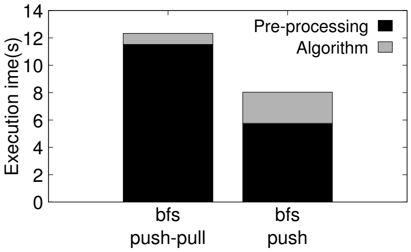


Figure 1: Example of the trade-off between pre-processing and algorithm execution time for BFS on the Twitter graph: push-pull improves algorithm execution time, but the required pre-processing time leads to overall worse end-to-end execution time (measured on Ligra [29]).

- An evaluation of a push vs. pull information flow, illustrating the benefits of reduced computation for push vs. reduced synchronization for pull.

- An evaluation of the pre-processing vs. computation trade-off for combined push-pull information flow, showing that the extra pre-processing costs associated with this combination outweigh gains in algorithm execution time.

- The adaptation of an out-of-core technique for improving the cache locality and the synchronization overhead of an in-memory graph processing system.

- An evaluation of the pre-processing vs. computation tradeoff for NUMA-aware optimizations, demonstrating that their large pre-processing times can be compensated by gains in algorithm execution time only on large NUMA machines and only for certain algorithms.

The outline of this paper is somewhat unusual. We start in Section 2 with an overview of the hardware and software used in this paper. We discuss data structures and pre-processing costs in Section 3. In Section 4 we look at the relationship between the data layout and vertex-centric or edge-centric computation. Section 5 discusses methods for improving cache locality. In Section 6 we evaluate the choice between push and pull approaches and its implications for algorithm execution time, pre-processing time and synchronization overhead. Section 7 evaluates graph partitioning approaches to take advantage of NUMA characteristics. Section 8 summarizes results on graphs and algorithms not discussed in previous sections. Section 9 provides an overview of all the results in one place. Section 10 discusses the graph processing systems from which we draw inspiration for this work. Section 11 concludes the paper.

[1] see Section 6 for a precise definition of push-pull

The code used for the experiments in this paper and instructions on how to run them is available at: `https://github.com/epfl-labos/EverythingGraph`.

2 Experimental setup

Experimental environment. We evaluate the pre-processing and algorithm execution times on two machines, each representative of a large class of machines. Machine A has 2 NUMA nodes, and is less sensitive to NUMA effects than machine B, which has 4 NUMA nodes. More precisely, machine A has 2 Intel Xeon E5-2630 processors, each with 8 cores (16 cores in total) and a 20MB LLC cache, and 128GB of RAM. Machine B has 4 AMD Opteron 6272 processors, each with 8 cores (32 cores in total) and a 16MB LLC cache, and 256GB of RAM. Unless otherwise stated, all experiments are run on Machine B.

The pre-processing times, unless otherwise stated, assume the graph is already loaded in memory. The costs of loading the graph into memory and its implications on pre-processing are discussed separately.

The subset of vertices or edges to be processed during a computation step is kept in a work queue. Threads take work items from the queue in large enough chunks to reduce the work distribution overheads. We parallelize both pre-processing and computation using the Cilk 4.8 parallel runtime system. When needed, Cilk balances the work among threads by allowing threads to steal work items from one another. Our experiments using OpenMP and PThreads show comparable execution times and are therefore not reported.

Algorithms. We select six algorithms with different characteristics in terms of functionality (traversal, machine learning, ranking), vertex metadata, as well as the number of vertices active during computation steps (iterations).

We evaluate the following three traversal algorithms. **Breadth-first search (BFS)** traverses a graph from a given source vertex and builds a tree in breadth-first order. **Weakly connected components (WCC)** discovers connected vertices within a graph and classifies them into components using label propagation. **Single source shortest path (SSSP)** finds the (length of the) shortest path between a given source vertex and every other reachable vertex in the graph. We also evaluate two algorithms that compute over the entire graph: **Pagerank (PR)** [24] is a ranking algorithm used to rank web pages based on their popularity. **Sparse matrix vector multiplication (SpMV)** multiplies the adjacency matrix of a graph with a vector of values. The matrix entries are stored as edge weights. Finally, **Alternating Least Squares** (**ALS**) is an optimization method used in recommender systems.

Datasets. Table 1 gives an overview of the graphs used along with their number of vertices and edges. We use both synthetic and real-world datasets. The synthetic datasets are power-law graphs generated by the RMAT graph generator [5]. We generate graphs of different sizes to evaluate the scalability of optimizations in terms of graph size. RMAT26 is the biggest RMAT graph that we can fit on all machines for all approaches. As a real-world power-law dataset, we use the Twitter follower graph [18], which is the largest real-world dataset that fits on all machines for all approaches.

In addition to these two graphs, we also use the US-Road graph from the DIMACS challenge [1]. This graph has a different shape than power-law graphs: it has a high diameter, and all vertices have a small in/out degree. We use it to study the impact of the shape of the graph on different computation approaches. Finally, for ALS we use the bipartite Netflix graph [35].

Graph	**Vertices**	**Edges**
RMAT-N	2^N	2^{N+4}
Twitter	62M	1468M
US-Road	23.9M	58M
Netflix	0.5M	100M

Table 1: Graphs used in the evaluation, with their number of vertices and edges.

Due to space constraints, in Sections 3 to 7, we primarily present results for BFS and Pagerank (with 10 iterations). These algorithms represent opposite ends of the spectrum, both in terms of the percentage of the graph that is active during each step of the computation and in terms of computation complexity. Furthermore, we report results primarily for the RMAT26 graph. We include results for other algorithms and graphs only when they provide additional insights that depend on the algorithm or the shape of the graph. Section 8 completes the picture by presenting data on the combinations of algorithms and input graphs not discussed in earlier sections.

3 Data layouts and pre-processing costs

In this section we first present different data layouts and their associated pre-processing costs.

3.1 Data layouts

Edge arrays are the simplest and the default way to distribute graphs [27] and are used by many systems [6, 12, 27]. Graphs are stored as an array containing pairs of integers corresponding to the source and the destination vertex of each edge. In the remainder of the paper, we assume the graph input takes the form of an edge array and needs to be further converted into other formats.

Adjacency lists store edges in per-vertex edge arrays. Each vertex points to an array containing the destination vertices of its outgoing edges, and possibly also to an array containing the source vertices of its incoming edges.

3.2 Pre-processing costs

Edge array. The layout of edge arrays matches the format of the input file, and it suffices to map the input file in memory to be able to start computation. As such, edge arrays incur no pre-processing cost.

Adjacency lists. We explore two techniques to build adjacency lists.

The simplest technique consists of reading the input file and *dynamically* allocating and resizing the edge arrays of vertices as new edges are discovered.

The second technique avoids reallocations by loading the graph as an edge array and then sorting it by source vertex. Vertices use an index in the sorted edge array to point to their outgoing edge array. The incoming edge array is created by sorting the edge array by destination vertex. This way the edges are stored contiguously in memory, corresponding to compressed sparse row format (CSR). The performance of this approach depends on the sorting algorithm.

The most common approach to sort edges is to use a *count sort*. In a first pass over the edge array, we count the number of outgoing (incoming) edges for each vertex. In a second pass over the edge array, we place edges at the correct location in the sorted edge array. Most existing graph analytics frameworks use this approach, as it is optimal in terms of complexity (the input array is only scanned twice).

An alternative approach is based on *radix sort*. Radix sort treats keys as multi-digit numbers, and sorts the keys into buckets one digit at a time. In the parallel version, each thread recursively sorts a subset of edges into a small number of buckets [32]. The advantage of radix sort is that buckets are written sequentially, and therefore have good locality. The complexity of the sort is relatively low. We use a radix size of 8 bits (256 buckets) which only requires $log_2(\#vertices)/8$ recursions to sort the edge array (e.g., 4 recursions for a graph with 4 billion vertices, 8 recursions with 2^{64} vertices).

3.3 Evaluation

Table 2 presents, for all three approaches (dynamic, count sort and radix sort), the execution times for creating outgoing per-vertex edge arrays and for creating both incoming and outgoing per-vertex edge arrays, for the Twitter graph and assuming the graph is already in memory. Using a radix sort is 4.8× faster than count sort. Surprisingly, sorting using a radix sort is also 4.9× faster than dynamically building the per-vertex edge arrays. Radix sort is faster, because it has better cache locality than the other solutions. Both the dynamic approach and count sort sequentially read the input edge array, but the subsequent steps have poor cache locality. The dynamic approach requires jumping between per-vertex arrays to insert a newly read edge. Count sort requires jumping between vertices as well in order to count their degree. It then does another scan of the input to place edges at their corresponding offsets in the sorted edge array. This step jumps between distant positions in the array.

Adj. list pre-processing variation	Twitter out	Twitter in-out	LLC misses
Dynamic	20.0	27.2	69%
Count sort	19.5	23.9	71%
Radix sort	4.0	8.5	26%

Table 2: Adjacency list creation cost (in seconds) and percentage of LLC misses on machine B when the graph is in memory.

Figure 2 presents the evolution of the pre-processing time for RMAT graphs depending on the graph size. All approaches scale as the graph size increases. The radix sort approach is always faster than the count sort and the dynamic sort approach (3.3× and 3.8×, respectively, on RMAT26).

For smaller graphs, count sort is slower than both the dynamic and radix approaches. The approach requires reading the edge array twice (once for counting, and then once to place edges in the sorted array). As the graph grows, however, the fact that the second pass in count sort does no reallocations makes it slightly better than the dynamic approach (e.g. there are 32 million reallocations for an RMAT26 graph).

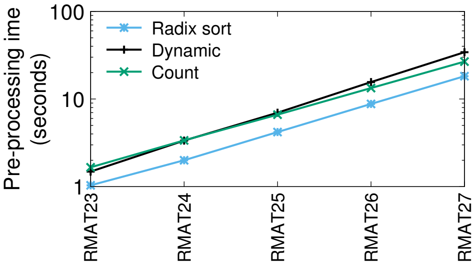


Figure 2: Scaling of pre-processing methods for adjacency list creation. All methods scale linearly with the graph size. RMAT-(N+1) is double the size of RMAT-N, and so is the pre-processing time.

3.4 Loading and pre-processing

The previous discussion assumes that the graph is already loaded into memory. Conclusions are different when the graph is to be read from storage or over the network. Indeed, doing a radix sort can only be partially overlapped with loading the graph in memory. In contrast, the dynamic approach of allocating and resizing per-vertex edge arrays can be fully overlapped with loading. For count sort, only the first pass can be overlapped with loading.

3.5 Evaluation with loading included

Table 3 presents the combined loading and pre-processing time when the graph is loaded from an SSD (380MB/s maximum bandwidth) and from a regular hard drive disk (100MB/s).

If we take loading speed into account, dynamically allocating per-vertex edge arrays becomes faster than radix sort when the storage medium is slow. On the SSD the total time for the radix sort approach is shorter than or more or less the same as the dynamic approach. The results for count sort are, as before, inferior, and are not included for that reason.

Pre-processing approach	**RMAT26 out**	**RMAT26 in-out**
Dynamic, loaded from SSD	20.7	40.0
Radix-sort, loaded from SSD	21.2	27.0
Dynamic, loaded from disk	61.0	61.1
Radix-sort, loaded from disk	65.0	71.0

Table 3: The cost of pre-processing for adjacency list creation with loading time included. Results show the time when building only the outgoing per-vertex edge arrays, and when building both the outgoing and incoming per-vertex edge arrays. The pre-processing is overlapped with loading when the adjacency list is created dynamically.

Summary. Costs associated with loading and building data structures in memory are non-negligible, and different approaches shine in different situations. Surprisingly, using radix sort to build adjacency lists is the fastest approach when the input file is in memory or loaded from a fast medium. When the graph is loaded from a slow medium, building adjacency lists dynamically is a better option, because it can be overlapped with loading.

4 Data layout and graph traversal

4.1 Vertex-centric vs. edge-centric

The choice of data layout impacts the decision of how to traverse the graph. In this section, we show that the best performing data layout and corresponding traversal model depend on the algorithm.

Computation on edge arrays happens in an **edge-centric** manner, and is quite simple: at every iteration of the computation the whole edge array is scanned, and the graph algorithm is called on every edge. This computation model is efficient, because scanning an edge array is cache-friendly: most of the accessed data is prefetched before being used. The drawback of this layout is that it offers no easy way to work on a subset of the vertices: a full scan of the edge array is required to find the edges of a vertex.

Adjacency lists are a natural solution to this problem. They enable **vertex-centric** computation, in which work is only performed on the subset of active vertices.

4.2 Evaluation

To illustrate the impact of data layout and traversal model on the end-to-end execution time, we show in Figure 3 the pre-processing and algorithm execution times of BFS, Pagerank, and SpMV on RMAT26. For BFS, vertex-centric computation performs the best, because during an iteration BFS only works on a limited subset of the graph. Edge arrays are not well suited for this type of computation, as all edges of the graph are read at every iteration.

In contrast, Pagerank accesses the entire graph in every iteration. Looking only at algorithm execution time, vertex-centric computation still performs a bit better, because it has better cache locality (all edges from a vertex are processed on the same core). When taking into account the pre-processing time, however, the end-to-end execution time is the same as for edge-centric computation.

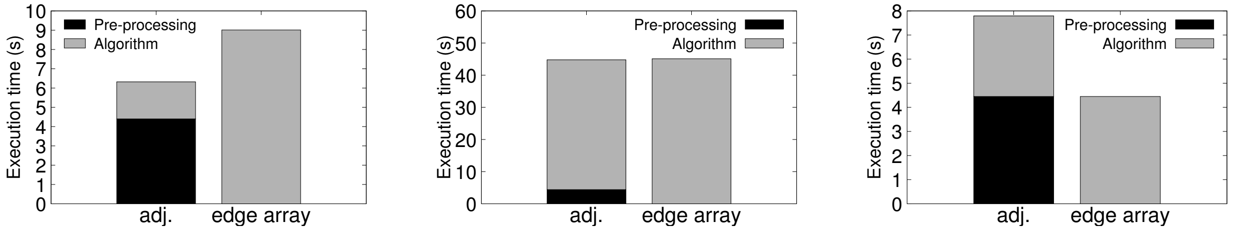
Finally, SpMV is an algorithm that makes only a single pass over the graph. Here, edge-centric computation produces the best end-to-end result, since the cost of building adjacency lists for vertex-centric execution is not amortized by any gains in algorithm execution time.

5 Cache-locality

Due to their irregular access patterns, graph algorithms usually exhibit poor cache locality. Last-level cache (LLC) misses may happen during three key steps of the computation: fetching an edge, fetching the metadata associated with the source vertex of the edge, and fetching the metadata associated with the destination vertex of the edge. In this section, we study how to lay out the data in memory to reduce the number of LLC misses, and we explain the pre-processing costs associated with creating those layouts.

5.1 Impact of the data layout

Edge array. In edge-centric computation, since edges are streamed, they are prefetched efficiently and do not incur cache misses. Fetching the metadata of the vertices, however, leads to random accesses with poor spatial and temporal locality.



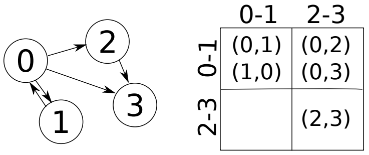
(a) BFS on RMAT26 (b) Pagerank on RMAT26 (c) SPMV on RMAT26

Figure 3: Pre-processing and algorithm execution time for BFS, Pagerank and SpMV on RMAT26, using vertex-centric computation on an adjacency list or edge-centric computation on an edge array.

Adjacency lists. In adjacency lists, computation is performed from the point of view of a vertex: a core iterates over all edges of a given vertex before processing another vertex. As a consequence, the metadata of the source vertex is read only once, after which it is cached. This is beneficial for vertices that have a large number of edges. Fetching edges may introduce a cache miss for the first edge, but subsequent edges are prefetched, as with the edge array. Also similar to the case of the edge array, the metadata of the destination vertices exhibits poor cache behavior.

Grids: optimizing edge arrays. To improve the cache locality of edge arrays, data is laid-out as a grid of cells. Each cell contains the edges from a range of vertices to another range of vertices. Figure 4 shows an example of a graph transformed into a grid. This data structure is inspired by the grid data structure first introduced in GridGraph [37], which aimed at maximizing reuse of data read from disks. Computation then iterates over cells. The goal is that the metadata associated with the vertices in the cell stays in cache and can therefore be reused. We construct the grid using the same radix sort approach as for building adjacency lists. Instead of bucketing edges by source vertex, we bucket them by the cell to which they belong. The optimal number of cells in the grid depends on the graph shape and size. We experimentally find that a grid of 256x256 cells performs best on the Twitter and RMAT26 graphs. Building a grid is slightly more expensive than building an adjacency list (the number of cells in the grid is equal to $(\#vertices/256)^2$, which is higher than the number of vertices for large graphs).

We compare using radix sort with a dynamic approach for buiding the grid, and the conclusions regarding different pre-processing approaches made in Section 3.2 are applicable to grids as well: radix sort is faster when the graph is in memory or loaded from a fast medium, while dynamically building the grid is faster otherwise.



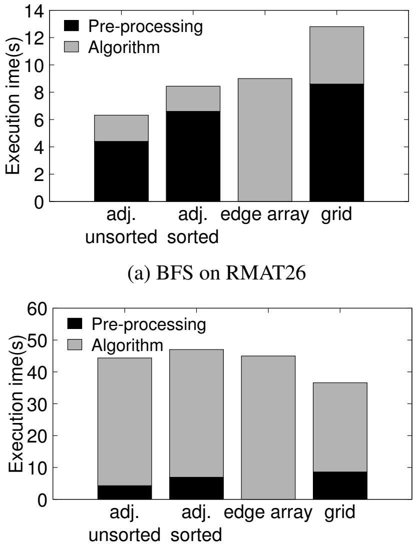
	0-1	2-3
0-1	(0,1) (1,0)	(0,2) (0,3)
2-3		(2,3)

Figure 4: Transforming a graph into a grid representation.

Optimizing adjacency lists. An intuitive idea to improve cache locality in adjacency lists is to sort the per-vertex edge arrays by destination. Indeed, the metadata of vertices with contiguous IDs is also contiguous in memory, thus when accessing vertex 0 and then vertex 1, the metadata of vertex 1 is likely to be present in cache. Of course, sorting the per-vertex edge arrays increases the pre-processing cost.

5.2 Evaluation

Figure 5 compares the pre-processing and algorithm execution times of BFS and Pagerank on RMAT26, on the unsorted adjacency list, the sorted adjacency list, the edge array and the grid. Table 4 presents the cache miss rate for these four data layouts.



(a) BFS on RMAT26

(b) Pagerank on RMAT26

Figure 5: Impact of cache-related optimizations on pre-processing and algorithm execution time for BFS and Pagerank on RMAT26.

Data layout	**BFS**	**Pagerank**
Edge array	57%	83%
Grid	23%	35%
Adjacency list	63%	78%
Adjacency list sorted	63%	78%

Table 4: Cache miss ratio for BFS and Pagerank on RMAT26.

BFS. For BFS, the unsorted adjacency list remains the solution with the best end-to-end execution time. Looking at algorithm execution time alone, BFS is 2.4× faster with a grid than with unsorted per-vertex edge arrays. However, creating the grid adds significant pre-processing time (9s), making the grid the slowest solution overall for BFS. Sorting the per-vertex edge arrays also leads to end-to-end performance inferior to unsorted adjacency lists. The pre-processing time increases, and the algorithm execution time does not decrease. Table 4 shows that sorting the per-vertex arrays does not significantly impact the cache miss rate. The destination vertices are accessed in order, but in practice a cache line only contains the metadata associated with very few vertices (64 in the case of BFS). Even when sorted, the destination vertex identifiers in the per-vertex edge arrays are sufficiently far apart for their metadata to fall in different cache lines, which explains the limited impact of this optimization on the number of cache misses and therefore on algorithm execution time. The increased pre-processing time for sorting the per-vertex arrays increases end-to-end execution time.

Pagerank. Even with the added pre-processing cost, the grid outperforms all other data layouts for Pagerank: it is 1.4× faster than an edge array and 1.3× faster than an unsorted adjacency list. This improvement is a direct result of the reduced cache miss rate when using a grid. As shown in Table 4, the cache miss ratio for the grid is less than half of that for the other data layouts. As for BFS, sorting the per-vertex edge arrays provides no benefit for Pagerank, for the same reasons. A cache line can fit at most 6 vertices for Pagerank, leading to an even smaller improvement in spatial locality than for BFS.

Summary. Creating a grid improves cache reuse and has a significant impact on algorithm execution time. Yet, this comes at the cost of an extra pre-processing, which is not always amortized. Different layouts also shine in very different situations. For instance, the grid is the best solution for Pagerank, but the slowest on BFS.

6 Information flow: Push and Pull

One of the core design decisions for a graph processing system is the information flow model it adopts. Information propagates through the graph in one of two ways: a vertex either **pushes** data along its out edges, writing to the state of its neighbors, or it **pulls** data along its incoming edges and updates its own state. These two approaches have important implications on computation, synchronization and pre-processing that we detail in this section.

6.1 Impact on end-to-end execution time

6.1.1 Impact on algorithm execution time

The **push** and **pull** approaches have different impact on the number of vertices and edges that need to be accessed during an iteration.

First, the **push** approach allows working on a subset of the vertices, while the **pull** approach does not. When pushing, vertices that do not need to propagate their value can be safely ignored. In contrast, the pull approach requires a vertex to scan all its incoming edges for neighbors that could potentially propagate a value. It also requires a pass over all vertices to check whether they need to look at their incoming edges (e.g., whether they have already been discovered in BFS).

Second, for some algorithms, the **pull** approach allows stopping the computation for a vertex in the middle of an iteration, while the **push** approach does not. Indeed, while **pulling** data a vertex may stop pulling before exploring all its incoming edges. For instance in BFS, if a vertex marks itself as discovered in the middle of an iteration, it stops exploring its remaining incoming edges. This guarantees that the vertex is discovered only once. In the **push** approach, vertices need to check that all their neighbors have been discovered, which leads to redundant work if multiple vertices have the same neighbors.

Figure 6 shows the per-iteration execution time of pushing vs. pulling for BFS on an RMAT26 graph. During the first iteration and after the third iteration, pushing is faster than pulling. During iterations 2 and 3, pulling is faster than pushing. This difference is explained by the percentage of the graph that is accessed during the iterations: most vertices in the graph are discovered during iterations 2 and 3. When pushing data, lots of redundant work is done in these iterations.

Because pushing data and pulling data perform best at different phases of the computation, some frameworks

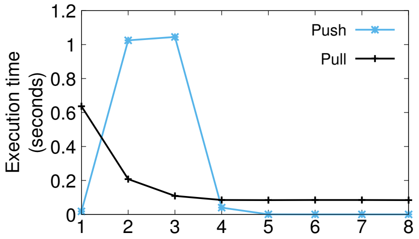


Figure 6: Per-iteration algorithm execution time for push vs. pull for BFS on RMAT26.

dynamically switch between pushing and pulling, depending on the number of active vertices in an iteration [2, 3, 29].

6.1.2 Impact on synchronization

A significant part of the algorithm execution time may involve synchronization. For example, in Pagerank on an RMAT26 graph with 16 cores, 40% of the algorithm execution time is spent in code protected by locks. The goal of this section is to evaluate the possibilities for lock removal, how they depend on the data layout and the information flow, and what if any pre-processing costs they induce.

In push mode, a vertex pushes updates to all its neighbors, and thus needs to lock them to update their metadata. In pull mode, a vertex only updates its own state. Thus, lock removal with adjacency lists requires execution in pull mode.

The grid offers a natural partition of the graph: edges in different rows have different source vertices, and edges in different columns have different destination vertices. To perform computation without locks in push mode, it suffices to assign different columns to different cores. To perform computation without locks in pull mode, it suffices to assign different rows to different cores.

6.1.3 Impact on pre-processing

Adjacency lists. To use push-pull, a system needs to iterate over both outgoing and incoming edges. As a result, when the graph is directed, we need to build both the outgoing and incoming per-vertex edge arrays. In contrast, for push we only need to build the outgoing, and for pull only the incoming per-vertex edge arrays. As a result, for directed graphs push-pull comes with an increased pre-processing cost, compared to push or pull, as seen in Section 3.2. When the graph is undirected, it suffices to build the outgoing per-vertex edge arrays (outgoing and incoming edges are the same), and push-pull induces no extra pre-processing cost.

Edge array. Computation over an edge array always requires scanning all the edges in the graph, so there is no advantage to using either push or pull. Furthermore, since the computation is edge-centric and not vertex-centric, locks need to be acquired for all updates. For these reasons, edge arrays are not considered any further in this section.

Lock removal. Lock removal does not require any additional pre-processing, beyond what is otherwise necessary for adjacency lists and grids, but it cannot be used with edge arrays, which have zero pre-processing cost.

6.2 Evaluation

6.2.1 BFS

Figure 7 presents the end-to-end execution times for BFS running on a directed RMAT26 graph, with adjacency lists, using push-pull, push (with locks) and pull (without locks). We do not show any results for edge array or grid for BFS, as we have shown in Section 5 that these approaches lead to inferior results compared to adjacency lists.

Push-pull is much faster in terms of algorithm execution time, but it is 1.5× slower than the push approach in terms of end-to-end execution time because of the extra pre-processing time. When taking pre-processing time into account, we find no combination of graphs, algorithms and machines in which push-pull is beneficial on directed graphs. On undirected graphs, push-pull does not add any pre-processing time, and is thus much faster than just pulling or pushing data. Furthermore, due to the fact that, on average, only a small percentage of vertices is processed per iteration, BFS in push mode performs 20% better than BFS in pull mode, even though push uses locks and pull does not.

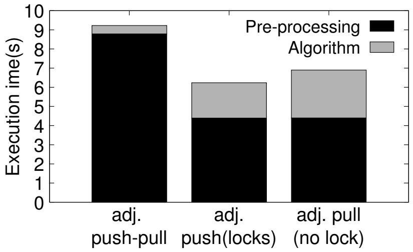


Figure 7: Pre-processing and algorithm execution time for BFS on RMAT26 using push-pull, push (with locks) and pull (without locks).

6.2.2 Pagerank

Figure 8 shows the end-to-end execution times for Pagerank in push mode on an adjacency list (with locks), in pull mode on an adjacency list (without locks), in push mode on a grid (with locks), and in pull mode on a grid (without locks). Here, the advantages of removing locks can be clearly seen. On adjacency lists, the version without locks is 40% faster than the push version when looking at end-to-end time. On a grid, the version without locks shows a gain of 1.5× in end-to-end time when comparing to the version with locks.

Summary. Push and pull on adjacency lists have conflicting benefits. Push works better for algorithms that only access a subset of the vertices in a given iteration, while pull allows vertices to be updated without locks. With grids, locking can be avoided regardless of whether push or pull is used, but the advantage of push remains

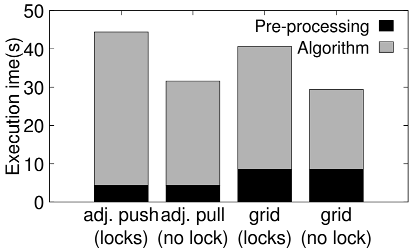


Figure 8: Pre-processing time and algorithm execution time for Pagerank on RMAT26 for push (with locks) on an adjacency list (with locks), for pull on an adjacency list (without locks), for push on a grid (with locks), for pull on a grid (without locks).

for algorithms that only access a subset of the vertices. Whether push or pull comes out ahead depends heavily on the nature of the algorithm. A combined push-pull approach requires extra pre-processing, which outweighs the benefits in terms of algorithm execution time.

7 NUMA-Awareness

We evaluate the trade-offs between the potential benefits of being NUMA-aware and the overheads it introduces in both the pre-processing and algorithm execution phase.

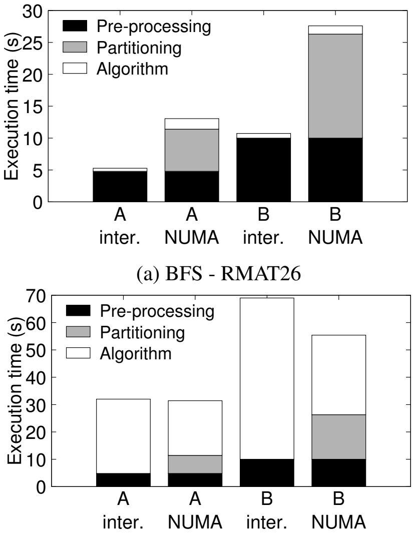
7.1 Data layout

In NUMA-aware solutions, the graph is *partitioned* across the NUMA nodes, and threads prioritize work from partitions that are local to their NUMA node. The partitioning scheme divides graph data evenly across NUMA nodes and places related data on the same NUMA node. Partitioning is performed so as to minimize the number of edges whose source and destination vertices are on different NUMA nodes, while still balancing the number of vertices and edges per NUMA node.

We evaluate in particular the partitioning schemes of Polymer [33] and Gemini [36]. The vertices are split into as many subsets as there are NUMA nodes. The outgoing edges of vertices are colocated with their target vertices. This approach avoids random remote writes and balances the number of edges across NUMA-nodes. Threads first process their local partitions. After that, they start working on remote partitions by updating the target vertices that are local to their NUMA node.

7.2 Evaluation

We evaluate the potential performance improvement of NUMA-aware data placement on the two machines presented in Section 2. Figure 9 shows the impact of NUMA-aware graph partitioning of an RMAT26 graph when running BFS and Pagerank. We compare NUMA partitioning to a solution that randomly interleaves the graph data on all NUMA nodes. We use, for each application, the best algorithm in terms of algorithm execution, as presented in the previous sections (push/pull for BFS and pull without locks for Pagerank). The end-to-end execution time is broken down into pre-processing, partitioning and algorithm execution.



(a) BFS - RMAT26

(b) Pagerank - RMAT26

Figure 9: Impact of NUMA-aware partitioning on machines A and B. For each machine we show the pre-processing, partitioning and algorithm execution time for BFS and Pagerank on RMAT26 with memory interleaving vs. NUMA-aware data placement.

Looking at Figure 9b, the NUMA-aware data layout improves the algorithm execution time for Pagerank 1.3× on Machine A and 2× on Machine B. However, only on the machine B, with 4 NUMA nodes, does the end-to-end execution time benefit from being NUMA-aware.

In contrast, looking at Figure 9a, for BFS the NUMA-aware version is 3.5× slower on Machine A and 1.8× slower on Machine B. For BFS the time spent in partitioning dwarfs the algorithm execution time on both machines. More surprisingly, even when looking only at algorithm execution time, the NUMA-aware version performs worse than the interleaved version. In BFS, in a given iteration, only a small number of vertices is processed, and these vertices often share a common ancestor (e.g., during the first iteration, all processed vertices are the children of the root vertex). As a consequence, vertices processed during a given iteration often reside in the same partition. This leads to all cores accessing the same NUMA node, which creates memory contention [9]. This undesirable effect is even more visible on high-diameter graphs with low-degree vertices, as shown in Figure 10 when running BFS on the US-Road graph. The NUMA-aware version is 12× slower than the interleaved version.

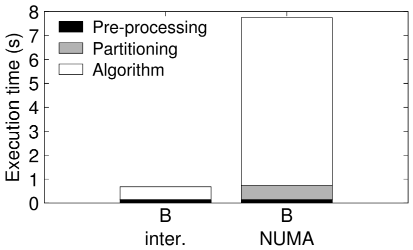


Figure 10: Effect of contention on memory bus on high diameter graphs. Pre-processing, partitioning and algorithm execution time for BFS US-Road graph with memory interleaving vs. NUMA-aware data placement

Summary. NUMA-aware data partitioning has a high pre-processing cost. This cost is amortized for algorithms that run for a long time and that work on most of the data during every iteration. For algorithms that run only for a short time, this may not be the case. For algorithms that only work on a subset of the data, NUMA-aware partitioning may exacerbate memory contention.

8 Additional algorithms and workloads

Table 5 shows the best solutions for BFS and Pagerank for graphs not evaluated in previous sections. The Twitter graph has a degree distribution similar to that of RMAT, and benefits from the same approaches: using an adjacency list while pushing data for BFS, and using a grid for Pagerank. The US-Road graph leads to slightly different conclusions. The best approach on Pagerank is to use an edge array and not a grid. Since the graph has a lower per-vertex degree than the RMAT and Twitter graphs, the grid data structure reduces only slightly the cache miss ratio, and therefore its pre-processing cost is not amortized.

In Table 6 we report the best approaches for WCC, SpMV, SSSP and ALS, their end-to-end execution time and its breakdown over pre-processing and algorithm execution time.

SPMV is a very short algorithm, and edge arrays are always the fastest approach, as they induce no pre-processing cost.

Intuitively, **WCC** should perform best on adjacency lists, because it is a traversal algorithm (only a subset of the graph is processed during every iteration of the computation), but WCC runs on an undirected graph. We therefore first have to build an undirected version of the graph from the input file. In the case of adjacency lists, an edge has to be inserted in both the outgoing edge array of its source and its destination. Thus, the pre-processing cost for creating adjacency lists is increased. In contrast, no additional pre-processing is required for edge arrays and grids to perform computation on an undirected graph. As a consequence, on graphs with a low diameter, WCC works best with an edge array, because the pre-processing time of building adjacency lists is too high. On graphs that have a higher diameter, like the US-Road graph, WCC needs more iterations to converge, and an adjacency list works best.

SSSP is very similar to BFS, and previous conclusions regarding the trade-offs between algorithm execution time and pre-processing for BFS are applicable to this algorithm as well. The only difference is that BFS discovers a vertex only once, whereas in SSSP a vertex may update its path many times during the computation, leading to an increase both in the number of iterations and the number of vertices active in each iteration.

ALS computes recommendations from a bipartite graph. The left side of the graph represents users and the other side items being rated. During every iteration, a subset of the graph (the left or right side) is active, and hence adjacency lists are the best data layout.

9 Summary

Improvements in algorithm execution time often come at the cost of increased pre-processing time. As seen in the previous sections, no approach fits every graph, algorithm or machine. In this section we try to provide a roadmap for choosing between different data layouts and computation approaches.

The first step consists of choosing an appropriate data layout. The layout is chosen based on the algorithm and graph characteristics. Short algorithms, such as SPMV, that complete in one iteration, should use an edge array, as it incurs no pre-processing cost. When the computation works only on a small subset of the graph at every computation step, adjacency lists in push mode improve algorithm execution time. The cost of building them is usually amortized compared to computation over edge arrays, especially on graphs with a high diameter. Other algorithms that run on graphs that have a large average per-vertex degree and iterate over most of the graph at every iteration, may benefit from using a grid, because the grid improves cache locality.

Second, if the machine is a large NUMA machine and the algorithm execution time is predicted to be large, then partitioning the graph to be NUMA-aware is beneficial (Figure 9b).

Third, if the data layout and computation approach chosen during the first step allow for execution without locking (e.g., pull mode in grids), then it is always beneficial to remove locks. We do not find any algorithm or directed graph for which switching between a pull mode without locks and push mode is beneficial when looking at end-to-end execution time.

Finally, when pre-processing cannot be avoided, it induces a non-negligible cost, and it should be optimized by using appropriate sorting techniques.

Algo	Graph	Data layout	Propagation model	Pre-processing	Algorithm	Total
BFS	Twitter	Adj. list	Push	5.8	2.3	8.1
BFS	US-Road	Adj. list	Push	0.3	0.5	0.8
Pagerank	Twitter	Grid	Pull (no lock)	23.2	37.8	61.0
Pagerank	US-Road	Edge array	Pull	0.0	1.6	1.6

Table 5: Best approaches in terms of end-to-end execution time for BFS and Pagerank on the Twitter and US-Road graph.

Algo	Graph	Data layout	Propagation model	Pre-processing	Algorithm	Total
WCC	RMAT-26	Edge array	Push	0.0	11.0	11.0
WCC	Twitter	Edge array	Push	0.0	19.2	19.2
WCC	US-Road	Adj. list	Push	0.6	56.8	57.4
SpMV	RMAT-26	Edge array	Push	0.0	4.4	4.4
SpMV	Twitter	Edge array	Push	0.0	5.8	5.8
SpMV	US-Road	Edge array	Push	0.0	0.3	0.3
SSSP	RMAT-26	Adj. list	Push	4.4	2.8	7.2
SSSP	Twitter	Adj. list	Push	5.8	4.4	10.2
SSSP	US-Road	Adj. list	Push	0.5	30.7	31.2
ALS	Netflix	Adj. list	Pull (no lock)	0.8	7.7	8.1

Table 6: Best approaches in terms of end-to-end execution time for SpMV, WCC and ALS on different graphs.

System	Data layout	Iteration model	Push or Pull	Without locks	NUMA-Aware
Ligra	Adj list	Vertex-centric	Push&Pull	Yes	-
Polymer	Adj list	Vertex-centric	Push&Pull	Yes	Yes
Gemini	Adj list	Vertex -centric	Push&Pull	Yes	Yes
X-Stream	Edge array	Edge-centric	Push	-	-
GridGraph	Grid	Grid-cell	Push	Yes	-

Table 7: Overview of multicore graph processing systems that inspired this work and their features.

10 Related Work

Very few papers compare the benefits of different graph processing systems. Satish et al. [28] evaluate various single-machine and distributed systems and compare them to a hand-optimized baseline. The paper looks at complete systems rather than individual techniques. Capota et al. [4] introduce a benchmark for graph processing platforms.

A large number of graph processing systems have been proposed [7, 8, 12–17, 19, 20, 22, 23, 25–27, 29, 31, 33, 36, 37]. We cover here only those works that have directly inspired the techniques evaluated in this paper. For a brief summary of the main features of these systems, see Table 7.

Beamer et al. [2, 3] are the first to propose push-pull for BFS. Ligra [29] extends this idea to other graph algorithms. It also uses radix sort for creating adjacency lists. X-Stream [27] introduces edge-centric graph processing in the context of out-of-core systems. GridGraph [37] improves on this idea by organizing the edges into a grid. Polymer [33] and Gemini [36] optimize graph processing for NUMA machines. We use their data placement technique in Section 7. In addition to the techniques used in Polymer, Gemini adds work stealing to balance work across NUMA nodes.

Not explored in this paper, the use of GPUs for graph processing has been the subject of some recent works [10, 11, 21, 30, 34]. This approach could affect the relative magnitude of pre-processing vs. algorithm execution time, and thereby impact the conclusions for certain algorithms.

11 Conclusion

We have presented an analysis of various techniques aimed at improving the algorithm execution time in graph processing systems, and we have explained their impact on pre-processing time. Our main observation is that pre-processing often dominates the end-to-end execution time of graph analytics. Therefore, it is often better to work with simple graph data layouts that induce less pre-processing than to invest time in elaborate pre-processing to speed up the algorithm execution phase. We argue that future works on graph analytics frameworks must more carefully consider this trade-off between pre-processing and algorithm execution time.

Acknowledgments: This work was supported in part by Swiss National Science Foundation Grant No. 167157 and by an EPFL-INRIA postdoctoral fellowship. We thank our reviewers, our shepherd Rong Chen, Laurent Bindschaedler, Florin Dinu, Rachid Guerraoui, Tim Harris, Dushyanth Narayanan, Amitabha Roy and Nicolas Schiper for their valuable feedback.

References

[1] http://dimacs.rutgers.edu/Challenges/.

[2] Beamer, S., Asanović, K., and Patterson, D. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 12:1–12:10.

[3] Beamer, S., Asanovic, K., Patterson, D. A., Beamer, S., and Patterson, D. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. Tech. rep.

[4] Capotă, M., Hegeman, T., Iosup, A., Prat-Pérez, A., Erling, O., and Boncz, P. Graphalytics: A big data benchmark for graph-processing platforms. In *Proceedings of the GRADES'15* (New York, NY, USA, 2015), GRADES'15, ACM, pp. 7:1–7:6.

[5] Chakrabarti, D., Zhan, Y., and Faloutsos, C. R-MAT: A recursive model for graph mining. In *Proceedings of the SIAM International Conference on Data Mining* (2004), SIAM.

[6] Chen, R., Shi, J., Chen, Y., and Chen, H. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 15.

[7] Cheng, R., Hong, J., Kyrola, A., Miao, Y., Weng, X., Wu, M., Yang, F., Zhou, L., Zhao, F., and Chen, E. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the ACM European conference on Computer Systems* (2012), ACM, pp. 85–98.

[8] Ching, A. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara 11* (2011).

[9] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V., and Roth, M. Traffic management: a holistic approach to memory placement on NUMA systems. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 381–394.

[10] Davidson, A., Baxter, S., Garland, M., and Owens, J. D. Work-efficient parallel GPU methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium* (Washington, DC, USA, 2014), IPDPS '14, IEEE Computer Society, pp. 349–359.

[11] Fu, Z., Personick, M., and Thompson, B. Mapgraph: A high level API for fast development of high performance graph analytics on GPUs. In *Proceedings of Workshop on Graph Data Management Experiences and Systems* (New York, NY, USA, 2014), GRADES'14, ACM, pp. 2:1–2:6.

[12] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the Conference on Operating Systems Design and Implementation* (2012), USENIX Association, pp. 17–30.

[13] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. GraphX: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 599–613.

[14] Han, W., Miao, Y., Li, K., Wu, M., Yang, F., Zhou, L., Prabhakaran, V., Chen, W., and Chen, E. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 1:1–1:14.

[15] Hong, S., Chafi, H., Sedlar, E., and Olukotun, K. Green-Marl: A DSL for easy and efficient graph analysis. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, ACM, pp. 349–362.

[16] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS Operating Systems Review* (2007), vol. 41, ACM, pp. 59–72.

[17] Ju, W., Li, J., Yu, W., and Zhang, R. iGraph: an incremental data processing system for dynamic graph. *Frontiers of Computer Science* (2016), 1–15.

[18] Kwak, H., Lee, C., Park, H., and Moon, S. What is Twitter, a social network or a news media? In *Proceedings of the International conference on World Wide Web* (2010), ACM, pp. 591–600.

[19] Kyrola, A., Blelloch, G., and Guestrin, C. GraphChi: Large-scale graph computation on just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Imple-*

mentation (Berkeley, CA, USA, 2012), USENIX Association, pp. 31–46.

[20] MAASS, S., MIN, C., KASHYAP, S., KANG, W., KUMAR, M., AND KIM, T. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, ACM, pp. 527–543.

[21] MERRILL, D., GARLAND, M., AND GRIMSHAW, A. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2012), PPoPP '12, ACM, pp. 117–128.

[22] MURRAY, D. G., MCSHERRY, F., ISAACS, R., ISARD, M., BARHAM, P., AND ABADI, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[23] NGUYEN, D., LENHARTH, A., AND PINGALI, K. A lightweight infrastructure for graph analytics. In *Proceedings of the Symposium on Operating Systems Principles* (2013), ACM, pp. 456–471.

[24] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[25] PEREZ, Y., SOSIČ, R., BANERJEE, A., PUTTAGUNTA, R., RAISON, M., SHAH, P., AND LESKOVEC, J. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1105–1110.

[26] ROY, A., BINDSCHAEDLER, L., MALICEVIC, J., AND ZWAENEPOEL, W. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 410–424.

[27] ROY, A., MIHAILOVIC, I., AND ZWAENEPOEL, W. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the ACM symposium on Operating Systems Principles* (2013), ACM, pp. 472–488.

[28] SATISH, N., SUNDARAM, N., PATWARY, M. M. A., SEO, J., PARK, J., HASSAAN, M. A., SENGUPTA, S., YIN, Z., AND DUBEY, P. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, ACM, pp. 979–990.

[29] SHUN, J., AND BLELLOCH, G. E. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 135–146.

[30] WANG, Y., DAVIDSON, A., PAN, Y., WU, Y., RIFFEL, A., AND OWENS, J. D. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2016), PPoPP '16, ACM, pp. 11:1–11:12.

[31] WU, M., YANG, F., XUE, J., XIAO, W., MIAO, Y., WEI, L., LIN, H., DAI, Y., AND ZHOU, L. GraM: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (New York, NY, USA, 2015), SoCC '15, ACM, pp. 408–421.

[32] ZAGHA, M., AND BLELLOCH, G. E. Radix sort for vector multiprocessors. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing* (1991), ACM, pp. 712–721.

[33] ZHANG, K., CHEN, R., AND CHEN, H. NUMA-aware graph-structured analytics. In *ACM SIGPLAN Notices* (2015), vol. 50, ACM, pp. 183–193.

[34] ZHONG, J., AND HE, B. Medusa: Simplified graph processing on GPUs. *IEEE Trans. Parallel Distrib. Syst. 25*, 6 (June 2014), 1543–1552.

[35] ZHOU, Y., WILKINSON, D., SCHREIBER, R., AND PAN, R. Large-scale parallel collaborative filtering for the Netflix Prize. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management* (Berlin, Heidelberg, 2008), AAIM '08, Springer-Verlag, pp. 337–348.

[36] ZHU, X., CHEN, W., ZHENG, W., AND MA, X. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA* (2016).

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