GraphGrind: addressing load imbalance of graph partitioning

GraphGrind: Addressing Load Imbalance 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

We investigate how graph partitioning adversely affects the performance of graph analytics. We demonstrate that graph partitioning induces extra work during graph traversal and that graph partitions have markedly different connectivity than the original graph. By consequence, increasing the number of partitions reaches a tipping point after which overheads quickly dominate performance gains. Moreover, we show that the heuristic to balance CPU load between graph partitions by balancing the number of edges is inappropriate for a range of graph analyses. However, even when it is appropriate, it is sub-optimal due to the skewed degree distribution of social networks. Based on these observations, we propose GraphGrind, a new graph analytics system that addresses the limitations incurred by graph partitioning. We moreover propose a NUMA-aware extension to the Cilk programming language and obtain a scale-free yet NUMA-aware parallel programming environment which underpins NUMA-aware scheduling in GraphGrind. We demonstrate that GraphGrind outperforms state-of-the-art graph analytics systems for shared memory including Ligra, Polymer and Galois.

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], [23], [28], [19], [15], [21]. Recently, graph partitioning has been proposed to isolate memory accesses to specific parts of the graph data. Graph partitioning allows to stage graph data in main memory from backing disk [15] and allows to direct memory accesses to the locally-attached memory node in Non-Uniform Memory Access (NUMA) machines [28]. Moreover, graph partitioning is essential in distributed memory systems to spread the computation evenly across all nodes [9].

Several studies have proposed efficient heuristic partitioning techniques for social network graphs [9], [15], as near-optimal partitioning is excessively time-consuming. A common approach is to partition the edge set with the aim to place an equal number of edges in each partition. This results in balanced computation per partition as many graph analyses perform work proportional to the number of edges [9].

While graph partitioning is a crucial building block for graph analytics, little is known about the various ways in which it affects performance. This paper analyzes heuristic graph partitioning in detail and identifies side effects that limit achievable performance. In particular, we show that graph partitioning incurs an innate performance overhead, which stems from increased control flow and from the decreased connection density of the partitions.

Moreover, we find that partitioning the edge set results in an imbalance in the number of vertices appearing in each partition. Alternatively, partitioning the vertex set results in an imbalance in the number of edges. Thus, significant load imbalance exists between partitions, either for loops iterating over vertices, or for loops iterating over edges.

This paper makes the following contributions:

- We analyze the characteristics of graph partitions and identify how these limit performance.
- We present GraphGrind, a NUMA-aware graph analytics framework that reduces the performance impact of graph partitioning. Key highlights of GraphGrind are an improved graph representation, tuning the partitioning to the characteristics of the algorithm and improving the NUMA memory mapping of key data structures.
- We develop an extension to the Cilk parallel programming language [8], [12] that allows expression of NUMA affinity for parallel loops. Our extension simplifies the design of GraphGrind and is generally applicable to enforce NUMA-aware scheduling in parallel programs.
- We experimentally evaluate the performance of GraphGrind on 6 real-world graphs and 3 synthetic graphs. We show that GraphGrind improves performance by up to 82% over Polymer and up to 326% over Ligra.

The remainder of this paper is organized as follows. Section II introduces the background of graph analytics. Section III motivates this work through analyzing the adverse impact of graph partitioning. Section IV describes the design and implementation of GraphGrind, a graph processing system that significantly reduces the overhead and load imbalance of traversing partitioned graphs. Section V presents an experimental evaluation of GraphGrind. Section VI discusses further related work.

II. Background

Graph analytics provide abstract, *vertex oriented* and/or *edge oriented* programming models that iteratively calculate a value associated to a vertex.

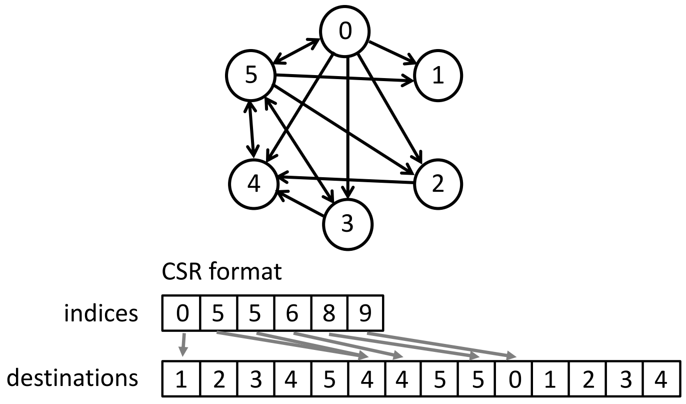


Fig. 1: A graph with skewed degree distribution and its representation in CSR format.

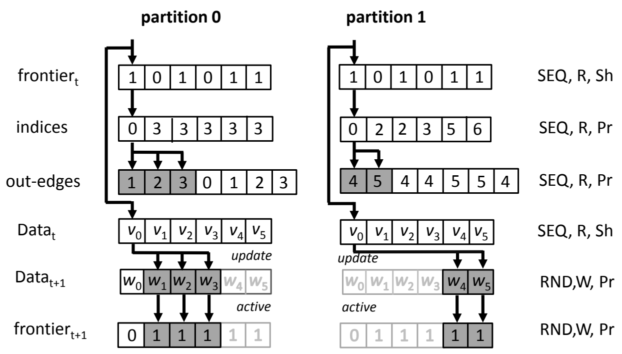


Fig. 2: Traversal of the graph in Figure 1 partitioned by destination.

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 i = 0;
3 for $v$ : $V$ do
4     if $|E_i|$ >= avg and $i < P-1$ then
5         ++i;                                    // $i$ has exceeded target edges
6     $E_i$ = $E_i \cup$ in-edges(v);              // i is home partition of v
```

A. Graph Representation

The two key data structures are graphs and frontiers. A graph $G = (V, E)$ has a set of vertices V and a set of directed edges $E \subset V \times V$ represented as pairs of end-points. A frontier is a subset of the vertices which are active. Graph algorithms visit the destination vertices of the active edges ($\{v \in V : (u, v) \in E \wedge u \in F\}$) and apply an algorithm-specific function to update the value computed for v taking into account the current value for u. This operation is repeated until all values have converged.

Figure 1 illustrates a graph with skewed degree distribution and its representation in the Compressed Sparse Rows (CSR) format [22]. The CSR format stores two arrays: an edge array with IDs of the destination vertices and an index array storing for each vertex the index into the edge array where the destinations of its edges are recorded. The index array has length $|V|$ and the edge array has length $|E|$.

The Compressed Sparse Columns (CSC) representation is analogous and stores the incoming edges to each vertex as opposed to the outgoing edges.

B. Edge Traversal

The efficient implementation of graph algorithms is sophisticated and requires deep knowledge of the characteristics of the algorithms. First, the frontier is a set of vertices and may be implemented either as a bitmap or as an array storing vertex IDs. The most efficient implementation depends on the *density* of the frontier [11]. In a *dense frontier* more *edges* are active, while a *sparse frontier* has few active *edges*. The threshold is typically set at 5% active edges.

Secondly, edges may be traversed in *forward* or *backward* manner. In each case, the goal is to traverse the destination vertices of active edges. A *forward* traversal first traverses source vertices $u \in V$ and checks if they are active ($u \in F$). If they are, then their out-going edges are traversed. A *backward* traversal iterates over destination vertices $v \in V$ as well as their incoming edges $(u, v) \in E$. Only then can it check that the source vertex u is active.

Some algorithms execute faster with forward traversal, while others with backward traversal. The distinction is to a large extent motivated experimentally [23]. Beamer et al. motivate the distinction by the number of visited edges [2].

The graph representation is designed for efficient forward *and* backward iteration. Hereto, a dual representation is used for directed graphs (incoming and out-going edges are equal for undirected graphs), i.e., the graph is stored once in CSC format and once in CSR format [23].

III. Motivation

A low-overhead algorithm to partition the edge set is listed in Algorithm 1 [15], [28]. The graph is partitioned as $G_i = (V, E_i)$ where E_i is a partitioning of E: $\cup_i E_i = E$ and all E_i are non-overlapping. The algorithm assigns each vertex to a home partition such that (i) each partition is home to a range of subsequent vertex IDs and (ii) an edge $(u, v) \in E$ is assigned to the home partition of v. It follows that $E_i \subset V \times V_i$: each partition only has edges pointing to its own home vertices, but the sources may be any vertex.

An often-used criterion for balancing CPU load is to equalize the number of edges per partition, as many graph analytic algorithms perform an amount of work that is proportional to the number of edges. We refer to this partitioning technique as *partitioning by destination* as edges are assigned to the home partition of the destination vertex. Alternatively, *partitioning by*

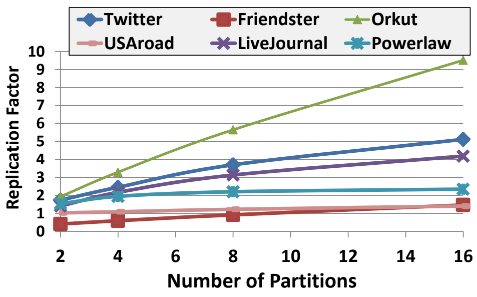


Fig. 3: Compressed vertices replication factor varying partition number

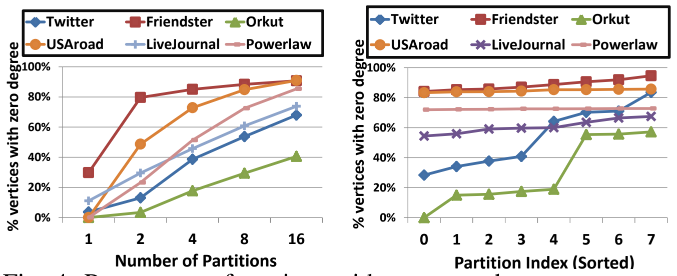


Fig. 4: Percentage of vertices with zero out-degree averaged across all partitions (left) and variation across each of 8 partitions (right).

source assigns an edge (u, v) to the home partition of u. Both algorithms achieve nearly the same number of edges in each partition [28].

Figure 2 shows how Algorithm 1 partitions the graph in Figure 1 in two parts. Partition 0 contains 7 edges and is home to vertices 0, 1, 2 and 3. Partition 1 also contains 7 edges and is home to vertices 4 and 5. Figure 1 furthermore shows how a single traversal of the graph proceeds, assuming a *dense forward* traversal. This traversal first checks whether each vertex is active, i.e., it has a 1 value in the *frontier* array. This is the case for vertex 0, so it traverses the out-edges of vertex 0 in each partition in parallel. It computes updated values for the vertices 1, 2 and 3 in partition 0 and for vertices 4 and 5 in partition 1. It updates the frontier accordingly. Note that each partition updates distinct values as edges with the same destination appear in the same partition.

A. Extra Work Induced by Partitioning

When partitioning the edge set, the list of edges of a vertex is split with parts of the list appearing in different partitions. As such, the edges for some vertices are stored in distinct partitions. Graph traversal must thus visit the vertex once for each replication. The additional cost of this is a small amount of control flow, lookups in the graph representation and checking whether the vertex is active. While these actions require only a few dozen assembly instructions, it is important to keep in mind that graph analytics perform little computation, typically less than a dozen assembly instructions per edge. Moreover, the overhead involves several main memory accesses as these algorithms are memory intensive.

Figure 3 shows the average replication factor of vertices for various degrees of partitioning. The graphs are described in Section V. We show data for 6 of the 9 graphs as the remaining 3 behave similarly. Graphs with few edges per vertex (USARoad and Friendster) have the lowest replication factors while highly skewed graphs (Twitter and Orkut) have the highest. Assuming 4 partitions, replication factors are often in the range 2–3, which implies that the control flow overhead of graph traversal is repeated 2 to 3 times. This results in an instruction count increase of up to 18%.

Figure 3 moreover shows that the graph partitioning algorithm studied in this paper achieves a comparable replication factor for the Twitter graph as the more elaborate algorithm in [9]. We may thus assume that the conclusions of this paper are independent of the partitioning algorithm used, as our conclusions build on the observation that the replication factor is larger than one.

B. Sparsity of Graph Partitions

If vertices are not replicated across all partitions, then by necessity vertices will not have incoming or out-going edges in several of the partitions. Figure 4 (left) shows the average number of vertices with zero degree for varying degrees of partitioning by destination. Similar results hold for partitioning by source. The fraction of vertices with zero out-going edges shoots up quickly as more partitions are introduced, exceeding in many cases 50% for 4 partitions. Moreover, real-world social networks have strongly imbalanced partitions (Figure 4 (right)). In contrast, the partitions of synthetic graphs, intended to model real-world graphs, have equal numbers of unconnected vertices in each partition. Interestingly, the Friendster graph has fairly equal partitions. The sparsity of graph partitions leads to an opportunity: if we can avoid iterating over the absent vertices in a partition, then the instruction count increase for these vertices can be restricted only to the partitions where the vertex occurs. To this end, GraphGrind uses a variation of the CSR representation where zero-degree vertices are not recorded.

C. Balancing Edges vs. Vertices

It is hard to partition a social network graph in a balanced way due to its skewed degree distribution. Figure 5 shows the relative number of vertices per partition for various graphs and numbers of partitions. Social network graphs like Twitter and Friendster have highly different numbers of vertices per partition when balancing the number of edges.

The imbalance of the number of vertices per partition has an important impact on performance. First, many graph algorithms make passes over vertices apart from passes over the edges. As such, the work performed per graph partition is not only proportional to the number of edges, but also depends on the number of vertices.

Secondly, not all algorithms perform a fixed amount of work per edge. Instead, algorithms such as BFS, betweenness-centrality, Bellman-Ford and K-Core visit at most one active edge per active vertex. For them, balancing the edges between partitions does not result in a balanced CPU load.

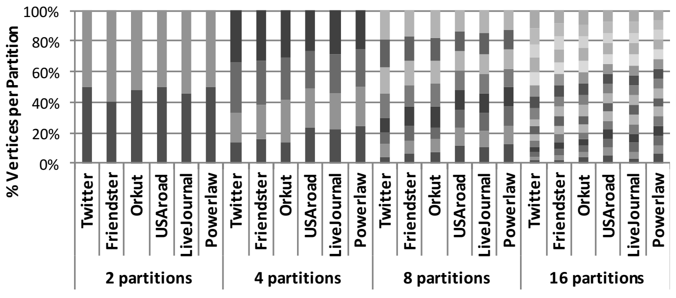


Fig. 5: Relative sizes of partitions for varying degree of partitioning.

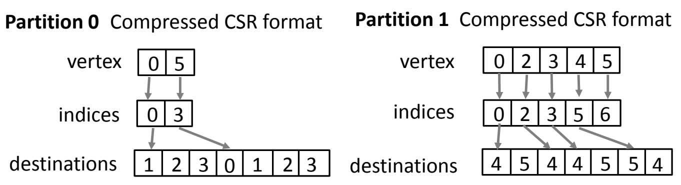


Fig. 6: Compressed CSR format.

Thirdly, an imbalance in the number of vertices per partition results in a skewed utilization of memory and creates hotspots for certain partitions. This unnecessarily drives to scale-out distributed systems to higher degrees of parallelism to drive the worst-case partition size down, even if the computation does not warrant scaling out. In shared memory systems the memory imbalance may be combated by storing data in a sub-optimal NUMA node, which results in the lesser evil of remote NUMA accesses.

Increasing the number of partitions may seem to avoid skewed partitions. This is however not true. As Figure 5 shows, the presence of highly-connected vertices remains an issue with higher degrees of partitioning as some partitions have twice as many vertices as others. We conclude that the graph partitioning needs to balance CPU load and should be adapted to characteristics of the algorithm.

IV. GraphGrind: Design and Implementation

GraphGrind is a NUMA-aware graph analytics framework that builds on the characteristics of graph partitions to optimise the memory layout of graphs and to reduce load imbalance. GraphGrind contains all the required features of graph analytics systems, including hierarchical parallel decomposition of the computation, NUMA-aware data placement and code scheduling [28], balanced vertex-cut partitioning [9] and adapting data structures [11] and search direction [2] to the size of the frontier. We discuss its key features below.

A. Application Programming Interface

GraphGrind is compatible with the Ligra programming model. It provides two data types: graphs and frontiers. A frontier is a subset of the vertices in a graph. The key functions apply operations to edges or vertices and calculate new frontiers in the process. They are defined as follows:

- *size()*: For a frontier F, $size(F)$ returns $|F|$.
- The *edge-map()* operator is the main work-horse. It applies an algorithm-specific function to every active vertex in the graph. Its arguments are a graph $G = (V, E)$, a frontier F, a function Fn and a condition C. An edge $(u, v) \in E$ is active if $u \in F$ and $C(v)$= *true*. The argument Fwd determines whether a forward or a backward traversal is likely to be faster. *Edge-map* returns a new frontier consisting of all visited vertices v for which $Fn(u, v)$ returned a true value.
- *vertex-map()* applies a function Fn to every vertex in the frontier F. It returns a new frontier consisting of all visited vertices u for which $Fn(u, v)$ returned a true value.

We extend the programming interface with a *cache* for ***backward*** *edge-map* traversals. While *edge-map* may execute in parallel, it traverses the incoming edges of a vertex sequentially when the number of vertices is not very large (less than 1000). Compilers should, in principle, be able to hold the intermediate updates for the destination vertex's value in registers. However, the complexity of control flow and pointer aliasing prohibits this in practice. GraphGrind allows the programmer to specify how to cache intermediate updates for the function Fn. This explicit notation allows compilers to allocate them to registers and involves a cache type definition and 3 functions to initialize the cache, to update it and to commit it to the main state.

B. Frontier Representation

We adapt the representation of frontiers between bitmaps and arrays of vertex IDs on-the-fly, depending on their density [11]. Frontiers are created either by constructors, or by the *edge-map* and *vertex-map* functions. From the users point of view, frontiers are immutable. One of the constructors creates a frontier containing all vertices. We explicitly record this property in the frontier to omit checks of the frontier and speed up graph traversal. Remember that graph analytics typically perform little work per edge. As such, any reduction in instruction count has a measurable impact.

This optimization affects traversal with dense frontiers. The backward traversal benefits much more from this optimization as it performs more lookups in the frontier, namely once per edge vs. once per vertex in the case of the forward traversal. We similarly optimize the *vertex-map* operation and any auxiliary loop iterating over the frontier.

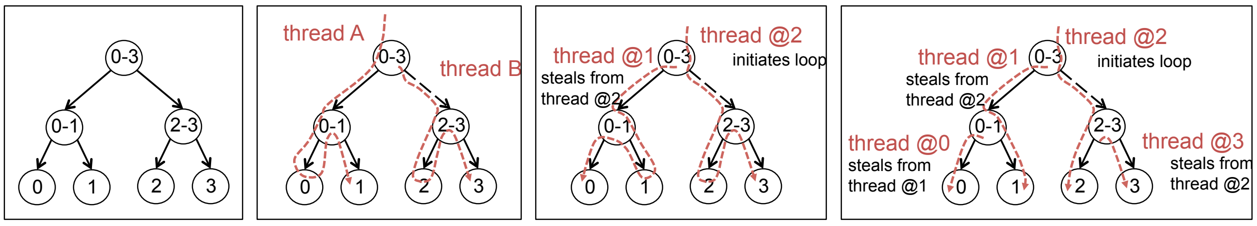
C. Compressed Graph Representation

We modify the CSR and CSC representation to combat efficiency issues with zero-degree vertices. We compress the index array by storing only information for vertices with non-zero degree and store the vertex ID with it. Figure 6 shows the modified CSR format for the graph partitions of Figure 2. In Partition 0, vertex 0 and 5 have out-edges which are home to

TABLE I: NUMA allocation and binding strategy

Data structure	NUMA allocation
full graph	interleaved
graph partition	allocate on one node
vertex arrays	match home partition

Operation	NUMA binding
edge-map (sparse)	none
edge-map (dense)	bind to holding node
vertex-oriented loops (e.g., vertex-map)	equally distribute loop iterations over NUMA nodes



(a) Cilk spawn tree (b) Cilk work-first (depth-first) execution (c) NUMA-aware execution order for threads 1 and 2 (d) NUMA-aware execution order with four threads

Fig. 7: NUMA-aware work-stealing. "Thread @n" represents any thread executing on socket "n".

vertex 0, 1, 2 and 3. In Partition 1, only vertex 1 has zero degree, so it is not stored. The representation reduces the size of the index array due to the high number of zero-degree vertices. The main benefit, however, is that a sequential edge traversal becomes more efficient as iteration over the index array automatically skips all zero degree vertices.

GraphGrind stores each graph partition in the CSR and CSC representations in order to support the direction-reversing technique. I.e., a *dense forward* traversal uses CSR while a *dense backward traversal* uses CSC. This representation is, however, not efficient for traversals with sparse frontiers as these are dominated by control flow, which is aggravated by the replication of vertices. As such, we store a *non-partitioned* copy of the original CSR representation of the graph specifically for sparse traversals. As such, GraphGrind stores three copies of the graph for undirected graphs, and two copies for directed graphs (as the CSR and CSC representations are equal for directed graphs).

D. Partition Balancing Criterion

We have argued that balancing the number of edges across partitions does not necessarily result in the best balancing of CPU time. Instead, some algorithms observe better CPU load balancing when the number of vertices in each partition is about equal. GraphGrind adds a parameter to the algorithm specification that shows its preference for a balanced edge partitioning vs. a balanced vertex partitioning. This parameter is checked during graph ingress in order to select the balancing criterion for graph partitioning. Our balanced vertex partitioning is similar to Algorithm 1, except that we strive for $|V|/P$ destination vertices in each partition.

Balancing vertices is appropriate for 3 of the 8 algorithms that we use in the experimental evaluation. The algorithms are commonly used in prior work. As such, this property is sufficiently important to ask programmers to record it. The property is easily derived from the algorithm specification.

E. NUMA Optimization

The state-of-the-art in NUMA-aware programming requires two coordinated actions: (i) data placement and (ii) thread placement. Common data placement strategies are to allocate data in a specific NUMA node or to distribute the data across nodes. Thread placement is optimized such that the thread has a low latency/high bandwidth connection to the NUMA domain holding its most frequently accessed data. This two-pronged strategy allows for many optimizations, such as co-locating threads with data and spreading data and threads across NUMA domains to enhance memory bandwidth.

Graph partitions can enforce NUMA-local access as each partition can be stored and processed within the confines of one NUMA node. Prior work has advocated to replicate frontiers and algorithm-specific data arrays on each NUMA node [28]. Accordingly, memory accesses are NUMA-local, except when interchanging data across nodes.

GraphGrind follows a different route, which is summarized in Table I. The full graph is stored in an interleaved fashion over the NUMA nodes. As the full graph is used with sparsely populated frontiers only, the memory accesses are few and hard to schedule optimally. Interleaved allocation provides a good compromise.

Graph partitions are spread over NUMA nodes in such a way that each partition is stored on one NUMA node and all NUMA nodes hold the same number of partitions. A graph traversal over a partition is scheduled on the NUMA node that holds that partition. This ensure that the majority of memory accesses are issued against the local NUMA node.

We distribute *vertex arrays* over NUMA nodes, storing the element for each vertex on the same NUMA node as its home partition. As such, the *edge-map* operation that is *writing* data to a vertex element performs NUMA-local accesses. This placement incurs some *false sharing*, as NUMA placement works on the granularity of virtual memory pages. As such, a small fraction of the vertices will be placed on a remote NUMA node. E.g., assuming 1 M vertices, at most 1 in 10,000 will be stored in a different node. The distribution of vertex arrays may be highly skewed due to the imbalance of vertices in each partition. Loops iterating over the vertex arrays, such as *vertex-map* and loops that analyze frontiers, are however scheduled

TABLE II: Graph algorithms and their characteristics. Frontiers: S=sparse, D=dense.

Algorithm	Description	Edge traversal	Frontiers	Cache	Balance
BC	betweenness-centrality [23]	backward	SDS	Yes	Vertices
CC	connected components using label propagation [23]	backward	DS	Yes	Edges
PR	simple Page-Rank algorithm using power method (10 iterations) [20]	backward	D	Yes	Edges
BFS	breadth-first search [23]	backward	SDS	No	Vertices
PRDelta	optimized Page-Rank forwarding delta-updates between vertices [23]	forward	DS	No	Edges
SPMV	sparse matrix-vector multiplication (1 iteration)	forward	D	No	Edges
BF	Bellman-Ford algorithm for single-source shortest path [23]	forward	SDS	No	Vertices
BP	Bayesian belief propagation [28] (10 iterations)	forward	D	No	Edges

such that the loop iterations are equally spread across NUMA nodes. While this induces some remote NUMA accesses, it is far more important to load-balance these loops than it is to optimize NUMA-awareness.

An alternative strategy is to replicate the vertex arrays on each NUMA node [28]. We found this to be sub-optimal due to the additional memory traffic that is required to replicate and to merge vertex arrays. In contrast, our NUMA placement and scheduling rules guarantee that an *edge-map* operation on a graph partition only writes to vertex array elements stored on the local NUMA node. Read operations may be remote, but these have lower impact on performance. As such, we obtain good NUMA locality without incurring the overhead of replicating data.

F. A NUMA-Aware Cilk Extension

GraphGrind is built on Cilk [8], an efficient work-stealing scheduler for parallel programs. Cilk, however, is agnostic of the memory hierarchy as it promotes cache-obliviousness [7], [27]. We modify the Cilk language and runtime system to support NUMA-aware scheduling and work stealing. We have deliberately searched for a minimalistic modification as to not affect space- and time-efficiency [3] and implement this in Intel Cilkplus version 1.2 [12]. We delegate a proof of the space and time bounds to future work.

We focus exclusively on parallel loops, which in Cilk are expressed with the `cilk_for` keyword, asserting that all iterations of the loop may execute in parallel. We extend the programming language with a pragma "`#pragma cilk numa(strict)`" that can be supplied immediately preceding a `cilk_for` loop, similarly to the existing *grainsize* pragma. The NUMA pragma indicates that loop iteration i should preferably be executed on cores associated to NUMA domain i. The assumption that the number of loop iterations does not exceed the number of NUMA domains is a pragmatic one. Programmers may split loops over a NUMA-aware outer loop and a normal **cilk_for** innner loop that executes only on the NUMA domain encoded in its calling context.

Cilk implements parallel loops using a helper function that recursively splits the iteration range of the loop in half. Once the iteration range is shorter than a heuristically determined threshold the helper function executes the loop sequentially over this part of iteration range.

Figure 7 (a) shows the call tree of the helper function for a loop with 4 iterations. Each node represents an invocation of the helper function. Edges indicate a parent-child relationship between function calls. Nodes in distinct subtrees are independent and may execute concurrently. Cilk uses a work-first scheduler [3] which translates into a depth-first traversal of the tree (Figure 7 (b)). Idle threads attempt to steal work from a randomly selected victim thread. Threads steal the continuation of the oldest function on their victim's call stack, i.e., the one nearest to the root of the call tree. E.g., if thread A starts execution of the range 0-3 in depth-first order it will first execute the sub-range 0-1. Meanwhile, thread B may steal the continuation of the oldest function and execute the sub-range 2-3.

We provide a NUMA-aware helper function that changes the execution order of loop iterations. The thread that executes an instance of the helper function checks its current NUMA domain and first executes the sub-range that matches its NUMA domain. E.g., if a thread on NUMA domain 2 initiates execution of the loop, it executes the range 2-3 before the range 0-1 (Figure 7 (c)). This strategy is applied recursively: a thread on NUMA domain 3 will first execute loop iteration 3. This way, work is distributed to the correct NUMA domain with a minimal work stealing (Figure 7 (d)).

Work stealing is modified to respect the NUMA constraints. Every dynamic function call is marked by the helper function with the range of NUMA nodes where the function may execute. This range reflects the iteration sub-range of the loop. The range is copied over to recursively called functions. A worker that selects a victim thread inspects the NUMA range of the victim's oldest function and aborts the work stealing attempt if the NUMA range does not contain its own NUMA node. By default, NUMA ranges are not set and work stealing proceeds as normal.

The algorithm is robust against anomalous conditions such as absence of active threads on a NUMA domain and a mismatch between the number of NUMA domains specified by the program and those in hardware. In both cases, pending iterations are executed on sub-optimal NUMA domains.

The NUMA extension supports non-commuting reductions [6] and pedigrees [16]. Both constructs depend on the execution order of function calls, which the helper function disrupts. The solution is beyond the scope of this paper.

V. Experimental Evaluation

We evaluate GraphGrind on a 4-socket 2.6GHz Intel Xeon E7-4860 v2, totaling 96 threads, with 256 GB of DRAM. We compile all codes using our modified version of the Clang compiler which implements the NUMA extension to Intel Cilkplus [12]. We evaluate 8 graph algorithms (see Table II) using 9 widely used graph data sets (see Table III). All reported results are averaged over 5 executions.

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

Graph	Vertices	Edges	Type
Twitter [14]	41.7M	1.467B	directed
Friendster [26]	125M	1.81B	directed
Orkut [18]	3.07M	234M	undirected
LiveJournal [26]	4.85M	69.0M	directed
Yahoo_mem [25]	1.64M	30.4M	undirected
USAroad [28]	23.9M	58M	undirected
Powerlaw ($\alpha = 2.0$)	100M	1.5B	directed
RMAT24	16.8M	168M	directed
RMAT27	134M	1.342B	directed

TABLE IV: Runtime in seconds of GraphGrind, Polymer, Ligra and Galois. The fastest results are indicated in bold-face. Execution times that differ by less than 1% are both labeled. Missing results occur as not all systems implement each algorithm. GraphGrind and Polymer use 4 partitions.

Algoritm	Graph	GG	Polymer	Ligra	Galois
CC	Twitter	**1.810**	2.580	2.878	16.660
	Friendster	**5.924**	8.030	7.330	6.210
	Orkut	**0.122**	0.180	0.138	0.311
	LiveJournal	**0.111**	0.177	0.125	0.206
	Yahoo_mem	**0.042**	0.049	0.063	0.046
	USAroad	35.348	36.730	38.910	**20.110**
	Powerlaw	**1.168**	2.110	1.680	3.113
	RMAT24	**0.455**	0.522	0.601	1.440
	RMAT27	**2.305**	3.220	2.444	10.120
BC	Twitter	**1.771**		4.130	4.160
	Friendster	**3.394**		5.490	6.110
	Orkut	**0.149**		0.160	0.178
	LiveJournal	**0.197**		0.334	0.388
	Yahoo_mem	**0.091**		0.110	0.150
	USAroad	**4.402**		5.174	6.010
	Powerlaw	**2.118**		2.300	2.860
	RMAT24	**0.482**		0.503	1.110
	RMAT27	**2.073**		2.360	15.110
PR	Twitter	**15.979**	20.400	23.660	20.120
	Friendster	**38.249**	41.8	43.300	61.200
	Orkut	**1.596**	1.660	2.240	2.120
	LiveJournal	**0.652**	0.688	0.708	0.700
	Yahoo_mem	**0.234**	0.262	0.278	0.255
	USAroad	**0.933**	1.220	1.582	1.180
	Powerlaw	**10.394**	12.716	13.600	11.614
	RMAT24	**2.730**	2.970	3.660	3.110
	RMAT27	**17.517**	23.21	28.600	30.220
BFS	Twitter	**0.254**	0.298	0.319	0.449
	Friendster	**0.896**	**0.899**	1.210	1.330
	Orkut	**0.039**	0.043	0.044	0.051
	LiveJournal	**0.050**	0.068	0.078	0.103
	Yahoo_mem	**0.025**	0.026	0.033	0.363
	USAroad	**1.750**	1.855	2.009	5.180
	Powerlaw	**0.595**	0.601	**0.599**	0.993
	RMAT24	**0.104**	0.119	0.118	**0.104**
	RMAT27	**0.412**	0.421	0.429	0.631

Algoritm	Graph	GG	Polymer	Ligra	Galois
PRDelta	Twitter	**20.560**	24.120	29.890	
	Friendster	**36.097**	36.600	62.100	
	Orkut	**1.244**	1.310	3.472	
	LiveJournal	**1.013**	1.110	1.138	
	Yahoo_mem	**0.831**	1.094	1.640	
	USAroad	**2.124**	2.260	2.905	
	Powerlaw	**10.659**	14.100	16.900	
	RMAT24	**1.845**	2.230	2.911	
	RMAT27	**8.645**	12.120	14.500	
SPMV	Twitter	**2.251**	2.860	4.610	
	Friendster	**3.624**	5.220	9.010	
	Orkut	**0.148**	0.208	0.630	
	LiveJournal	**0.060**	0.096	0.151	
	Yahoo_mem	**0.033**	0.045	0.063	
	USAroad	**0.077**	0.128	0.166	
	Powerlaw	**0.655**	**0.661**	0.707	
	RMAT24	**0.197**	0.221	0.288	
	RMAT27	**1.963**	2.210	2.830	
BF	Twitter	**1.489**	1.618	2.213	12.810
	Friendster	**6.498**	7.193	7.690	9.220
	Orkut	**0.213**	0.310	0.354	2.100
	LiveJournal	**0.258**	0.293	0.284	0.530
	Yahoo_mem	**0.146**	0.200	0.173	0.288
	USAroad	21.992	24.110	26.310	**16.330**
	Powerlaw	**10.326**	11.112	12.600	15.110
	RMAT24	**1.366**	1.390	1.410	1.880
	RMAT27	**1.665**	1.933	2.180	5.310
BP	Twitter	**38.896**	**38.900**	56.980	
	Friendster	**58.704**	66.210	129.000	
	Orkut	**2.223**	3.110	5.538	
	LiveJournal	**1.026**	1.420	1.940	
	Yahoo_mem	**0.448**	0.455	1.124	
	USAroad	**1.024**	1.660	1.462	
	Powerlaw	**15.264**	15.530	19.500	
	RMAT24	**4.788**	7.030	9.310	
	RMAT27	**32.994**	43.320	58.230	

A. Performance Comparison

We compare the performance of GraphGrind against leading graph analytics systems for shared-memory, namely Ligra[1] [23], Polymer [2][28] and Galois [19] version 2.0 (Table IV). GraphGrind and Polymer both use 4 partitions to match the NUMA characteristics of our hardware. All systems use 96 threads. We show the backward PageRank algorithm for Polymer as the forward version, presented in [28], contains errors. The absolute execution times depend on our hardware, compiler version and randomly generated graphs. Moreover, some algorithms are sensitive to the start vertex, which in our experiments is vertex 100 for all graphs. The reported trends match previously reported results.

Overall, GraphGrind outperforms the other systems for all algorithms and all graphs, except for CC and BF on the USAroad graph. In these cases, Galois is faster. This results from using different algorithms [19], [28]. Nonetheless, GraphGrind makes progress over Polymer and Ligra for these cases. In a few cases, GraphGrind performs on par with other systems. These are labeled in bold-face as well.

The performance improvements are significant: up to 326% faster than Ligra (SPMV with Orkut graph) and up to 82.2% faster than Polymer (BP with USAroad graph). The smallest speedups appear for BFS, as there is already little computation going on. The superior performance of GraphGrind results from a combination of optimizations. Next, we will tease out the main contributing factors.

[1]https://github.com/jshun/ligra.git

[2]http://ipads.se.sjtu.edu.cn:1312/opensource/polymer.git

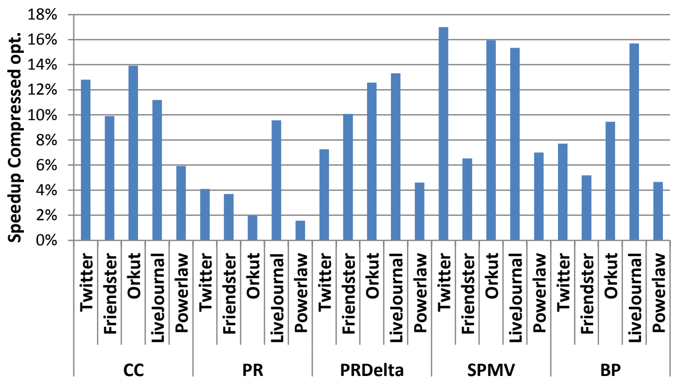


Fig. 8: Speedup of compressed graph compared to visit zero-degree vertices.

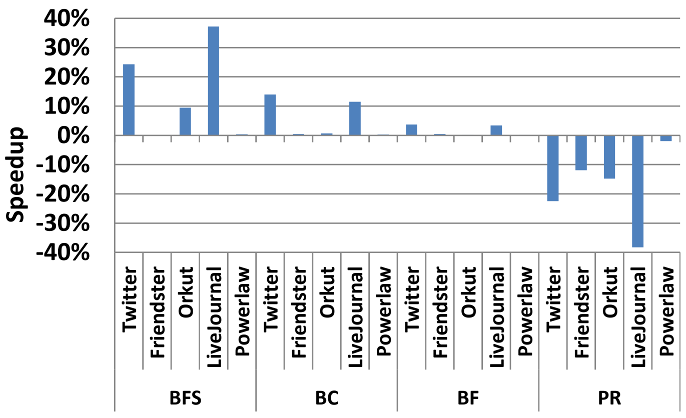


Fig. 9: Speedup of balancing vertices compared to balancing edges in graph partitions.

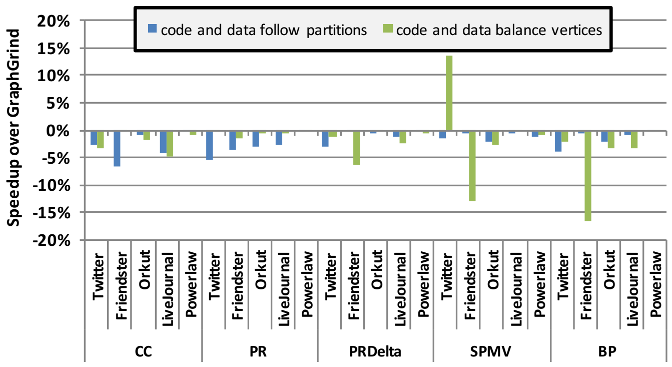


Fig. 10: Impact of NUMA decisions for vertex arrays. GraphGrind may be described as data follow partitions, code balances iterations.

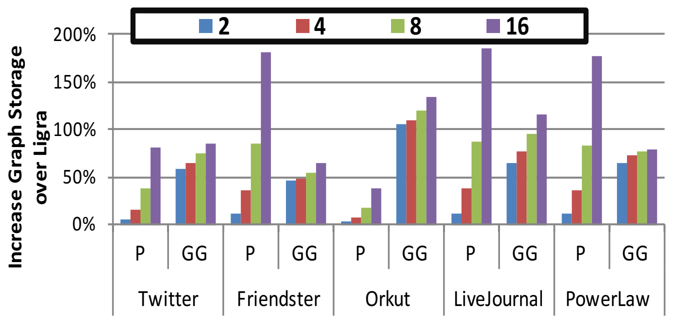


Fig. 11: Increase of graph storage for Polymer (P) and GraphGrind (GG) compared to Ligra.

B. Compressed Graph Representation

GraphGrind's graph data structure prunes vertices with zero degree from the representation. We will show later that this saves significant spaces compared to the CSC and CSR representations used by Polymer. Moreover, by not storing these vertices, *edge-map* traversals no longer need to visit them. Figure 8 shows the speedup resulting from the graph representation for 5 algorithms, which ranges between 2% and 16%. Twitter and LiveJournal benefit most due to the high sparsity of graph partitions.

C. Adapting Graph Partitioning

We remove CPU load imbalance through selecting an appropriate criterion to balance the graph partitions. We identified through code inspection that 3 of the evaluated algorithms (BFS, BC and BF) prefer an equal number of vertices in each partition. The others prefer a uniform number of edges. Figure 9 shows the speedup obtained by balancing vertices over balancing edges for these 3 algorithms and PR. We show results for a subset of the graphs, the remaining graphs behave similar to the ones shown. The partitioning has negligible impact for Friendster and PowerGraph, which have a balanced number of vertices per partition in either case (see Figure 4). Graphs with unbalanced partitions see important improvements with vertex-balanced partitions, with up to 37% speedup for LiveJournal.

Vertex-balanced partitioning is appropriate only for algorithms with fixed amount of work per vertex. Other algorithms, like PR, have a strong preference for edge-balanced partitioning. We conclude that it is crucial to balance partitions appropriately to the algorithm.

D. NUMA Optimization

Various choices can be made for the placement of vertex arrays, i.e., arrays storing frontiers or per-vertex application-specific data. GraphGrind places the vertex arrays such that each vertex is co-located with its home partition. Vertex-oriented loops, such as those in *vertex-map*, are typically short and have well-balanced work per iteration. As such, GraphGrind distributes the iterations equally across threads, even though this results in remote NUMA accesses.

We compare two variations on the NUMA policy (Figure 10): (i) placing vertex data and scheduling iterations on their home partition; (ii) equally spreading vertex data and iterations across all NUMA nodes. Option (i) aims to avoid remote NUMA access during vertex-oriented loops. This is however uniformly worse than GraphGrind's policy. It shows that CPU load balance is simply more important than NUMA locality for the vertex-oriented loops.

Option (ii) load-balances vertex-oriented loops and tries to minimize remote NUMA accesses by spreading vertex arrays to match the distribution of iterations. This results in worse performance in nearly all cases as the placement decision is sub-optimal for the *edge-map* operator. This operator performs the majority of main memory accesses and will incur excess remote memory accesses when vertices are not co-located with their home partition.

An interesting effect occurs when SPMV processes the Twitter graph, as in this case an increase in remote memory accesses during *edge-map* results in improved performance. We contrast this against Friendster, where the same effect results in performance degradation. We measured the local and remote memory accesses incurred and observe that both GraphGrind and option (ii) incur the same total number of memory accesses and that option (ii) incurs an increased number of remote accesses for both graphs.

The performance difference between the graphs, however, results as Twitter has highly skewed partitions: The number of elements of vertex arrays accessed on one NUMA node is much higher than on other NUMA nodes. Where GraphGrind directs those accesses to the local NUMA node, option (ii) spreads them across nodes. This way, option (ii) can share the unused memory bandwidth on one NUMA node with the computation on another node. On Friendster, GraphGrind is faster than option (ii) because Friendster has relatively uniform partitions and performs more memory accesses per unit of time. As such, all NUMA nodes are equally stressed and there is no benefit in making remote accesses.

These results show that a careful trade-off is required to optimize NUMA placement, as option (i) incurs fewer remote memory accesses than GraphGrind, yet has worse performance. In rare cases can remote accesses result in performance improvement due to imbalance in memory traffic.

E. Peephole Optimizations

GraphGrind marks frontiers that are initialized to contain all vertices such that an optimized *edge-map* can avoid memory accesses and control flow related to frontier access. Only algorithms that initialize frontiers this way can benefit. The algorithms using backward traversal (CC and PR) benefit most, up to 8%, as the backward traversal queries the frontier once for every edge, while the forward traversal queries it only once per vertex. The speedup is modest, but consistently positive. It moreover requires no user intervention.

GraphGrind allows programmers to define a cache, which allows the compiler to store intermediate values in registers (the cache) and avoid memory accesses. This optimization is relevant only during backward traversal. When applicable, the cache results in a speedup between 2 and 15%.

F. Memory Usage

Figure 11 shows the additional memory used on graph data for Polymer and GraphGrind compared to Ligra. Polymer stores each graph partition in CSR and CSC format (as in Ligra) using index arrays of length $|V|$. Because of this, the memory consumption of Polymer grows as $P|V|$ for P partitions. As GraphGrind stores only vertices with non-zero degree in the index arrays, its memory usage grows more slowly and follows the vertex replication factor (Figure 3). However, as GraphGrind stores an additional copy of the graph for sparse traversal, it starts at a 50% increase compared to Ligra for directed graphs. Overall, GraphGrind's memory consumption is more scalable than Polymer's.

VI. Further Related Work

It has been documented that generic tools such as METIS [13] 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 [17] 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 [9].

Bourse et al. [4] target distributed memory systems as it minimizes the number of edges crossing partitions, which involve messages. This is not immediately relevant to the performance of shared memory systems. It is not immediately clear that the algorithm would perform well in the context of our system. The algorithm moreover approximates edge and vertex balancing. Experimental evaluation shows deviations in the vertex balance up to 50%, which would have a prohibitively high impact on GraphGrind.

GraphChi [15] streams graph data from disk. It uses partitioning to obtain small vertex sets that fit in the main memory. It uses partitioning by destination with an equal number of edges per partition. The vertex data must be made to fit in memory by tuning the number of partitions.

X-Stream [21] uses what we call partitioning by source, but does not required edges to be pre-sorted. It aims for a uniform number of vertices per partition as it wants to keep only vertex data in fast memory (e.g., CPU cache), whereas edges are streamed in from slower memory (e.g., main memory).

GraphX [10] is a library for graph analytics. It partitions edge lists using Spark's resilient distributed datasets (RDD) and supports user-defined partitioning schemes.

Our observations are relevant for each of the systems discussed above. E.g., the reduced connectivity of partitions implies that memory locality is poor in a system like X-stream. A large variation in vertices per partition implies that partitions with few vertices will leave a large portion of main memory unutilized in GraphChi.

Frasca et al. [5] design NUMA-aware work queues for betweenness centrality. The work queues first execute locally generated work prior to stealing work from other queues. Work queues are visited in order of increasing NUMA distance. They demonstrate a 51.2% performance improvement compared to an OpenMP implementation.

Agarwal et al. [1] study the execution of breadth-first-search on NUMA systems. They too organize the computation around work queues, spread over multiple sockets. They use efficient spinning locks and lock-free channels to synchronize threads and they introduce peephole optimizations, e.g., avoiding atomic operations by first checking if they will fail.

Graph compression can significantly reduce memory requirements and with it memory bandwidth. Shun *et al* [24] compress the destination IDs of vertices stored in the edge array of the CSR and CSC representations. They reduce memory usage up to 56%. These techniques are orthogonal to the compressed representation of the CSC and CSR index arrays proposed in this work, as they pertain to edges only.

VII. Conclusion

Graph partitioning is an important technique to efficiently orchestrate the execution of graph analytics. In this paper, we study graph partitioning in the context of NUMA-aware data placement and code scheduling. We analyze the performance issues that graph partitioning inadvertently introduces, including load imbalance, increased work per vertex, and a significantly reduced connection density. Combined, these problems imply that graph partitioning is inherently unscalable to large partition counts.

We propose several techniques to counter-act the identified performance issues and implement these in GraphGrind, a novel NUMA-aware graph analytics framework that is compatible with the Ligra API. We moreover extend the Cilk language, in which GraphGrind is implemented, to enable NUMA-aware scheduling.

GraphGrind achieves significant speedup compared to prior work, out-performing Polymer, the most recent contender, by as much as 82%. We moreover show that fully minimizing remote memory accesses is not optimal in irregular computations. Instead, one needs to strike a careful trade-off between remote accesses and CPU load balancing.

We believe that this work makes important progress in making graph partitioning scalable. In future work, we will explore how to apply graph partitioning at much higher scales and translate these into enhanced performance.

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 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. 2010, pp.1–11.

[2] 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, 10 pages.

[3] R. D. Blumofe and C. E. Leiserson Scheduling multithreaded computations by work stealing, in *Proc. of the Annual Symp. on Foundations of Computer Science*. 1994, pages:356–368.

[4] B. Florian, L. Marc, and V. Milan. Balanced graph edge partition, in *Proc. of the 20th SIGKDD Intl. Conf. on Knowledge discovery and data mining*. 2014, pages 1456–1465.

[5] 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.

[6] M. Frigo, P. Halpern, C. E Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects, in *Proc. of the Annual Symp. on Parallelism in algorithms and architectures*. 2009, pages 79–90.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms, in *Proc. of the Annual Symp. on Foundations of Computer Science*. 1999, pages 285–.

[8] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of the Conf. on Programming Language Design and Implementation*. 1998, pages 212–223.

[9] 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.

[10] J. E Gonzalez, R. S Xin, A. Dave, D. Crankshaw, M. J Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow Framework, in *Proc. of the Intl. Symp. on Operating System Design and Implementation*.2014, pages 599–613.

[11] 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.

[12] *Intel Cilk Plus Language Extension Specification* (version 1.2. 324396-003us ed.). 2014, Intel.

[13] 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.

[14] 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.

[15] 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.

[16] C. E. Leiserson, T. B. Schardl, and J. Sukha. Deterministic Parallel Random-number Generation for Dynamic-multithreading Platforms, in *Proc. of the Symp. on Principles and Practice of Parallel Programming*. 2012, pages 193–204.

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

[18] 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.

[19] 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.

[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] 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.

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

[23] 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.

[24] J. Shun, L. Dhulipala, and G. E Blelloch. Smaller and faster: Parallel processing of compressed graphs with Ligra+, in *Data Compression*.2015, pp. 403–412.

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

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

[27] K. Yotov, Tom R., K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs, in *Proc. of the Annual Symp. on Parallel algorithms and architectures*. 2007, pp. 93104.

[28] 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.