Managing Large Graphs on Multi-Cores With Graph Awareness

Vijayan Prabhakaran, Ming Wu, Xuetian Weng
Frank McSherry, Lidong Zhou, Maya Haridasan[†]*
Microsoft Research, [†]Google

Abstract

Grace is a graph-aware, in-memory, transactional graph management system, specifically built for real-time queries and fast iterative computations. It is designed to run on large multi-cores, taking advantage of the inherent parallelism to improve its performance. Grace contains a number of graph-specific and multi-core-specific optimizations including graph partitioning, careful in-memory vertex ordering, updates batching, and load-balancing. It supports queries, searches, iterative computations, and transactional updates. Grace scales to large graphs (e.g., a Hotmail graph with 320 million vertices) and performs up to two orders of magnitude faster than commercial key-value stores and graph databases.

1 Introduction

Last decade has witnessed an increase in the number and relevance of graph-based applications. Finding shortest path over road networks, computing PageRank on web graphs, and processing updates in social networks are a few well-known examples of such real-world workloads, which access graphs with millions and billions of vertices and edges.

Some of these workloads – such as PageRank [10] – run in the background, without any direct user interactions. Since they do not have latency constraints, they can be run on existing data-parallel architectures such as MapReduce [12], Hadoop [4], or DryadLinq [32]. Given the importance of these workloads, new distributed architectures such as Pregel [21] are built to run them more efficiently. Typically, such batch-processed workloads run on read-only graph snapshots and therefore, their platforms do not support graph updates.

On the other hand, certain graph workloads are latency sensitive. For example, finding directions in a map or a search query accessing the social network of a user to find relevant information require responses that do not exceed strict time constraints (typically, in the order of few 10s of milliseconds), even though each such query (or workloads) can access millions of random vertices and edges.

*Work done at Microsoft Research

Whereas batched workloads run on static graph snapshots, real-time queries often run on graphs that can change continuously (such as in social network). Existing graph processing platforms are largely unsuitable to host these workloads because they are optimized for batched, read-only workloads. Other general purpose systems such as key-value stores or relational databases are graph-agnostic and offer sub-optimal performance.

Grace is a graph-aware, in-memory, transactional graph management system that is specifically designed for supporting low-latency graph applications. It exposes a simple graph querying interface and a platform for running iterative graph computations. Grace supports transactions, guaranteeing ACID semantics on graph modifications. Consistent graph snapshots can be created instantaneously in Grace, allowing read-only workloads – such as computing the shortest path – to run concurrently with other transactional updates.

Two design aspects of Grace distinguish it from existing systems, such as key-value stores and relational databases that are used for storing graphs. First, Grace is *graph-aware*, that is, Grace's design including its data structures, algorithms, policies, and interfaces are chosen to support graphs and graph workloads rather than being general purpose. Second, given that many graph workloads are highly parallelizable, Grace is optimized to run on large-scale multi-cores, taking advantage of the underlying parallelism and memory layout to improve the overall performance.

We show that this graph-awareness and support for parallelism can boost performance significantly when compared to key-value stores such as Berkeley DB (BDB) [1] and even when compared to other commercially available graph databases such as Neo4j [6]. Grace runs up to two orders of magnitude faster than BDB and Neo4j under a single thread; under multi-threaded mode, Grace runs up to 40 times faster than its unoptimized, single threaded mode. Grace scales well on large multi-cores and large graphs; for example, it runs on a 48-core machine, managing a 320 million node Hotmail graph.

In the rest of the paper, we present the system as follows. Next, we present the high-level design of Grace. In Section 3, we describe the algorithms used for graph partitioning and ordering vertices in memory. Following that, we explain the platform for running iterative computations in Section 4 and present the details of transactions in Section 5. We then explain the implementation details in Section 6 and present results from evaluation in Section 7. Finally, we discuss related work in Section 8 and conclude in Section 9.

2 Design for a Graph Management System

Grace's overall design is driven by the following two characteristics exhibited by several graph workloads.

2.1 Imparting Graph Awareness

A graph's inherent structure – which reflects how its vertices are connected – gives us certain hints about how the vertices will be accessed. Most graph workloads follow a strong *graph-specific locality* in their access patterns; specifically, when a workload accesses a vertex, it is highly likely to access the vertex's neighbors in the near future. For example, in a social network, sending updates to friends (and friends of friends) involves traversing the social graph and updating neighbors' data. Even iterative computation such as PageRank, which does not follow any graph-traversal pattern, computes the rank of a page and passes a fraction of its rank to its neighbors (thereby, accessing its neighbors' data).

However, general-purpose solutions – such as key-value stores and relational databases – used today for storing and accessing graph datasets are often agnostic to the underlying graph structure and therefore rely only on default access patterns and policies to improve performance.

Grace's graph-awareness spans the whole system starting from its low-level cache and memory layouts to high-level interfaces exposed to applications. Some of its graph-aware features include a fast graph partitioning algorithm, which can be used to split a graph into smaller sub-graphs and decide where to assign new graph updates, a graph-aware layout where vertices are ordered such that neighboring vertices are placed close to each other, and a set of querying and updating APIs for accessing a graph.

2.2 Embracing Parallelism

Second, graph workloads are often partitionable and therefore, parallelizable. For example, in a social network, two different users can update their status in parallel without affecting each other. Even computations such as PageRank can be run in parallel on a partitioned graph such that updates from a partition are sent to another partition only if they share an edge. This partitionable nature of graphs and their workloads is well-aided by the evolution of hardware into the multi-core era.

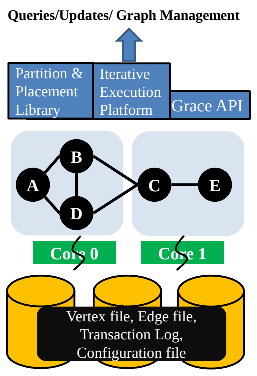

Queries/Updates/ Graph Management
Partition & Placement Library
Iterative Execution Platform
Grace API
A
B
C
D
E
Core 0
Core 1
Vertex file, Edge file, Transaction Log, Configuration file

Figure 1: **Grace Architecture.**

However, general purpose systems often lack support for large-scale multi-cores; for example, BDB and Neo4j do not support partitioned data or computation. Even if they are built for multi-cores, because of the absence of graph-awareness, key-value stores and relational databases cannot efficiently partition or layout a graph on multi-cores.

Grace makes several optimizations for multi-core processors. For example, it minimizes inter-core communications by batching updates across partitions. Grace's load-balancer can dynamically shed load from a thread that is handling a 'hot' graph partition to other free threads.

An overall result of these optimizations is that Grace outperforms other comparative systems; in a single-threaded mode, Grace runs up to two orders of magnitude faster than Berkeley DB and Neo4j; in a multi-threaded mode, Grace gains a speed up of up to 40 times relative to its single-threaded performance.

2.3 Grace Architecture

A high-level sketch of Grace is presented in Figure 1. Grace runs on a single machine. Since low-latency of queries is one of our main objectives, we specifically choose to keep the entire graph in-memory. Grace accesses the durable media only when loading a graph initially and logging transactional updates.

Grace exposes a set of APIs for querying and updating a graph. Queries can be vertex-specific (*e.g.*, GetVertexAndNeighbors()), or partition or graph-specific (*e.g.*, GetNumberOfVertices()). A graph can be updated by adding or deleting vertices and edges.

Grace also presents a platform for running iterative computations – which in turn use the querying APIs – on graph partitions. While iterative computations (such as PageRank) are typically treated as batch-processed workloads, we show that when run on Grace, they take a few minutes instead of several hours to complete.

Within Grace, a graph can be partitioned and stored as smaller sub-graphs. Grace provides a library for running management tasks such as partitioning a graph or rearranging its vertices in-memory; parameters for partitioning such as the algorithm to use and number of partitions to create can be specified via a configuration file. Each partition is handled by a thread, which can be pinned to a core; the thread is responsible for processing queries and updates to vertices and edges of that partition.

Next, we present more details on how Grace partitions a graph and arranges its vertices in memory.

3 Partitioning and Placement

One of the key opportunities we have in designing a specialized store for graph structured data is to exploit the expected locality of reference in the graph. Users accessing one vertex are often interested in adjacent vertices, and by carefully laying out the graph we can make such follow-up queries much faster than an otherwise random request.

We investigate graph layout at two granularities. First, we are interested in *partitioning* the vertices of the graph into a few parts, allowing simpler concurrency and parallelization between multiple workers. Second, we are interested in the relative *placement* of the vertices within a part, ideally with proximate vertices placed near one another to exploit caches at various levels. Partitioning is a very binary separation, two vertices are either in the same part or not, whereas placement is more continuous, in that two vertices can be placed within a spectrum of distances.

3.1 Partitioning and Placement Algorithms

Graph partitioning is a well-studied problem. The objective of a graph partitioning algorithm is to split a graph into smaller sub-graphs such that the number of edges that run between the sub-graphs (*i.e.*, the edge-cut between the partitions) is small. Another important criteria to consider while partitioning a graph is to create balanced partitions, that is, sub-graphs of roughly equal size. The number of vertices alone does not guarantee balanced partitions; for example, even with similar sizes, a partition with more high-degree vertices may become 'hotter' than other partitions. In addition, it is desirable to have a partition algorithm that can run faster, can be parallelized, and can be applied on incremental updates without having to look at the entire graph.

Optimizing all these criteria is a challenging, and unsolved problem. Moreover, different algorithms can have very different behavior on different graphs. Consequently, Grace provides an extensible library of partitioning algorithms, initially stocked with three simple but useful candidates: 1. a hash-based scheme oblivious to the graph structure, 2. a folklore heuristic, based on placing vertices in parts with fewest non-neighbors, and 3. a spectral partitioning algorithm based on the second eigenvector of the normalized Laplacian.

3.1.1 Hash Partitioning

Partitioning a data set based on the hash of the data is well-known. We hash by the vertex ID and distribute the vertices to different partitions. The nice features of hash partitioning are: it is fast; it does not require an entire graph to be loaded into memory and therefore, can be applied on incremental graph updates; it creates well-balanced partitions, and since it does not look at the degree of a vertex, high-degree vertices are mostly uniformly distributed, which reduces the chance of a particular partition becoming overloaded. However, because of its graph-agnostic nature, sub-graphs created using hash partition have high edge-cuts.

3.1.2 Heuristic Partitioning

One folklore heuristic for partitioning repeatedly considers vertices and places them in the part with fewest non-neighbors. All other things being equal, this is very similar to placing them in the part with the most neighbors, but avoids the degenerate case where all vertices join the same part. Instead, there is a tension between choosing an appealing part and balancing the parts.

More formally, for each vertex v with neighbors $N(v)$, the heuristic selects the part P_i minimizing the number of vertices not in $N(v)$: $|P_i \setminus N(v)|$. This number can be easily tracked by maintaining the sizes of each part, and a vector of current assignments of vertices to parts. Evaluating a vertex v requires only looking up the parts of its neighbors, and then subtracting these totals from each part size. The algorithm streams sequentially over the edge file, and only needs to perform random accesses to the vector of current assignments. Despite its simplicity, we find that the heuristic provides significant improvements in our experiments.

3.1.3 Spectral Partitioning

Spectral graph partitioning [29] uses the eigenvectors of the connectivity matrix of the graph to partition nodes.

Despite the intimidating name, it has a fairly simple description: we associate a real value with each vertex, and repeatedly update the value of each vertex to the average of the values of its neighbors. This process provably converges to values reflecting an eigenvector of the connectivity matrix, and intuitively assigns positive values more often to vertices whose neighbors are positive, and less often to those whose neighbors are negative, and vice-versa. Any threshold (we use zero) partitions the graph into two parts, nodes whose values are less than and greater than the threshold, for which we expect fewer edges between the parts than within the parts. One can then repeat the process recursively, until a desired size is reached.

We use hash and heuristic-based partitioning algorithms to create graph parts and spectral partitioning to order the graph vertices in-memory by their associated real values. Even though spectral partitioning can be used for graph partitioning, we did not use it because it is hard to generate balanced partitions and it runs slower than the other two.

On top of vertex placement, we also order the edges within a partition. Edges are first grouped according to their source vertices, following the same vertex ordering, and then within each group (with the same source), edges are ordered by their destination partition. While this placement improves most query performances, we also explore a different edge ordering for iterative computations, which is detailed in the next section.

4 Iterative Execution Platform

On top of the simple querying interface in Grace, we built an execution platform, which can help users program and run highly parallel iterative graph computations, without worrying about complex thread management and system-level optimizations. Similar to Pregel [21], Grace adopts the Bulk Synchronous Parallel (BSP) [30] abstraction and implements a vertex-based data propagation model, which is effective for exploiting parallelism and convenient for programming.

In this programming model, a graph computation consists of a sequence of iterations, separated by global barriers. Conceptually, each iteration has three steps: first, updates from a previous iteration are received; second, user-defined functions are invoked to apply the updates and compute new values for each vertex; and finally, updates are propagated to other vertices for the next iteration. The sequence of iterations are stopped when every vertex – that is, the user-defined function running for every vertex – votes to halt.

Since Grace operates exclusively in a multi-core environment with cache coherent shared memory, propagation of updates simply involves applying the updates directly to other vertices' data. Specific to iterative computations, Grace implements two optimizations: first, updates can be batched per destination partition to minimize data shuttling among cores; second, load from a thread, which handles a 'hot' partition, can be dynamically shed to other threads resulting in balanced computation across cores.

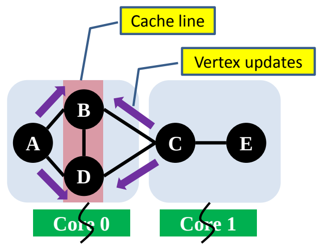


Figure 2: **An Example for Batching Inter-Core Messages.**

4.1 Batching Updates

For a given partitioned graph and a general-purpose iterative computation, Grace has limited opportunities to reduce the updates propagated among partitions. However, by carefully orchestrating *when* updates are propagated, Grace can substantially reduce unnecessary data copies among various cores.

Most modern multi-core architectures present a cache-coherent shared memory. That is, when a core modifies a piece of data, the data is populated on its cache and invalidated on other cores' caches that may have a copy of the data. Since a single cache line can hold multiple data items – which is especially true for graph computations, where a vertex data can be small such as the rank in PageRank or component Id in weakly-connected components – batching updates to all such data items together can minimize the number of cache copies and invalidations. This batching is especially useful in Grace, where vertices can be laid out according to their proximity in the graph.

Figure 2 illustrates a specific example of the above scenario, where batching is beneficial. It presents a graph with two partitions, 0 and 1, each handled by a thread pinned to its core. Consider a case where vertices B and D are placed in contiguous memory addresses and indexed into the same cache line. During an iterative computation, if vertices A and C propagate their updates to B and D, there are at least two possible schedules for applying the updates: in the first schedule, A sends its update to B, then C sends its update to B, which is followed by A's and C's updates to D; in the second schedule, A sends its updates to B and D, which is followed by C's updates to B and D. Assuming that the cache line with B and D was present in core 0 initially, in the first schedule, B and D are shuttled between partition 0 and 1 for three times, whereas in the second schedule, they are copied only once.

Strictly scheduling threads in a choreographed manner to avoid redundant data copies is difficult and can incur additional overheads. Instead, Grace implements a simple, yet effective memory layout and schedule for propagating updates. Within each partition, all the edges are grouped according to their destination partition, and within each group of destination partition, edges are ordered according to their source vertices' in-memory ordering. This is similar to the edge ordering proposed earlier in Section 3, except that edges are first ordered by destination and then by source. During an iterative computation, whenever updates must be propagated, a thread running in partition i, first selects the edges targeting itself and then selects the next partition in a round-robin fashion (*i.e.*, $(i + 1)\ mod\ N$, where N is the total number of partitions). In our evaluation, we find that batching updates as mentioned above can greatly improve the performance for dense graphs with large average vertex degree.

4.2 Balancing Load

In BSP programming model, iterative computations are separated by global barriers; that is, a partition can execute its next iteration only after all other partitions complete their current iteration. Although such synchronous execution simplifies programming, it introduces a cause for concern. If one partition's execution significantly lags behind others, then the whole computation is delayed proportionally.

In order to maximize the parallelism, it is necessary to balance the load among worker threads such that they are all busy during the entire computation. A standard way of achieving this is by partitioning the data equally among the workers; for example, a graph can be statically partitioned into sub-graphs of equal size. However, there are several drawbacks in static partitioning. First, it is difficult to define 'balance' because it is application-dependent. For example, partitions with same number of vertices may run computations for different time. Balancing the number of edges does not guarantee similar runtime either because not all edges propagate updates during each iteration. Second, enforcing balanced partitioning can degrade partitioning quality with respect to the size of edge-cut and reduce benefits from graph-specific locality.

Instead of statically balancing the sub-graphs, Grace dynamically balances the load by an efficient work sharing mechanism. During an iterative computation, instead of statically assigning all the vertices in a partition to a thread, Grace allows a thread to grab a set of vertices from other partitions, if necessary. Each thread starts processing vertices in its own partition. After all the vertices have been processed in its current partition, a thread grabs a set of vertices from another partition that has not yet been fully processed. For example, if there are two overloaded partitions and three free worker threads, the three threads repeatedly take a portion of vertices from the two partitions until everyone completes.

5 Transactional Graph Updates

The need to support graph updates stems from our initial goal of building a fast graph management system for graphs that may change continuously. Although updates can be applied by simply locking a graph, we chose to implement it with transactions in order to improve the concurrency between graph queries and updates; for example, Grace can run long iterative computations, while concurrently allowing a graph to change.

Grace supports structural changes to a graph. That is, a vertex or an edge can be added or deleted and edge weights can be changed. To update a graph, a thread starts a transaction, issues a set of changes, and finally tries to commit the transaction. The transaction may commit or abort depending on other conflicting changes that may have been applied to the graph. Once a transaction is committed, any new snapshot created from the graph will reflect the changes.

Under the covers, Grace implements transaction using snapshot-isolation [31]. When a transaction is started, Grace creates a consistent, read-only snapshot of the graph and allocates a temporary buffer for storing the updates. Grace ensures that the snapshot is consistent by making its creation atomic with respect to other transactional updates. All reads issued by the transaction are served from its snapshot and changes made by the transaction are logged into the temporary buffer.

During commit, Grace detects conflicts using version numbers. Version numbers can be maintained for each vertex, and although it reduces transaction conflicts, such fine granularity can add a lot of overhead; on the other hand, version can be managed at a large granularity for each graph, which will conflict on every concurrent transaction. Grace finds a middle ground and detects conflicts at the granularity of partitions. When a transaction prepares to commit, it checks if the partitions, which will be changed, were modified since the snapshot was taken. If not, the transaction proceeds to commit. During commit, contents of the temporary buffer are logged into the durable media and then, the changes are applied to the graph. After the transaction commits, the snapshot is deleted.

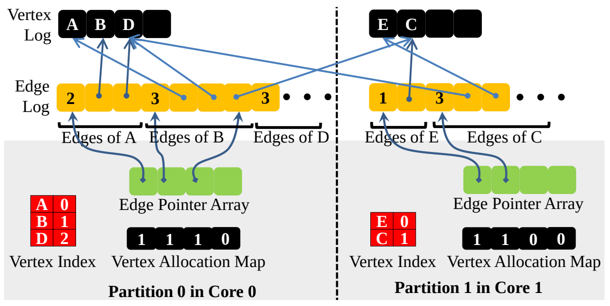


Figure 3: **In-memory Data Structures in Grace.**

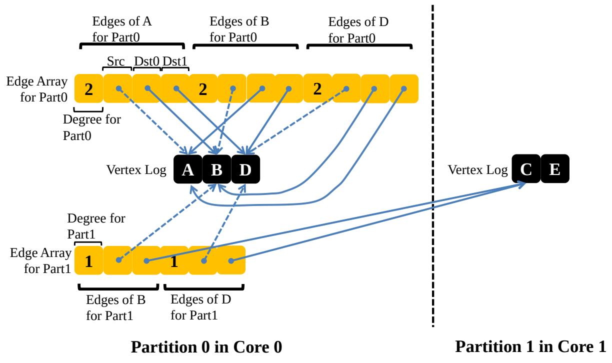


Figure 4: **Data structures of partition-based edge grouping.**

6 Implementation

When implementing Grace, we took careful measures to avoid features and data structures that can affect the performance. Grace is implemented in C++, which gives us the freedom to manipulate the memory layout to our convenience. In this section, we give an overview of the in-memory structures and how they are used to run transactions; finally, we briefly explain the on-disk data.

6.1 In-memory Data Structures

Figure 3 presents an in-memory data organization of the graph shown in Figure 1.

A graph object contains a set of partitions. Within each graph partition, vertices and their edges are stored in separate arrays, which are managed as *in-memory logs* (Vertex Log and Edge Log, in Figure 3). Vertex Log stores per-partition vertex records and the Edge Log stores the edge set – which is the degree and edges of a vertex – of all the vertices in that partition. An edge itself is a composite value consisting of a partition ID and the position of the destination vertex in Vertex Log.

In addition to the Vertex and Edge Logs, Grace creates and manages a few other in-memory structures. An Edge Pointer Array is used to store the position on the Edge Log where a vertex's edges set is stored. For example, in Figure 3, in Partition 0, vertex B's (whose index is 1 in its Vertex Log) edge set is pointed by the contents of Edge Pointer Array at position 1. Grace also maintains an index, mapping a vertex ID to its location on Vertex Log. Finally, Grace uses a Vertex Allocation Map to track whether a position on the Vertex Log contains a valid vertex or not; for example, a vertex can be marked as deleted by clearing the corresponding bit in Vertex Allocation Map.

Grace uses several other data structures, which are not shown in Figure 3 for simplicity. Grace separates any data associated with a vertex (such as the 'rank' in PageRank) and stores them on a separate array, at the same index position as the vertex on the Vertex Log. This separation of data and metadata improves performance considerably, especially for queries that only look at the graph structure. Grace maintains a light-weight bitmap locks for coordinating access to a vertex's data from different partitions.

6.1.1 Implementing Iterative Computations

Iterative computation platform is built on top of the simple APIs exposed by Grace and batching the updates and balancing the load are implemented separately at this layer.

As we detailed earlier, for batching updates, edges are first grouped according to their destination partitions and then within each group, they are ordered by their source vertices. To implement this, each partition uses one edge array for each destination partition as shown in Figure 4. For simplicity, we only show the data structures in partition 0, which has two edge arrays, one for partition 0 and 1. In addition to storing the edges, we also store the number of edges each vertex has for a specific destination partition and a pointer to the vertex's data (dashed arrow). If a vertex does not have any updates to propagate, it can be quickly skipped (based on the number of edges that vertex has on the edge array); however, if a vertex has some updates, its data can be quickly accessed and propagated.

It is important to note that iterative computations are executed on top of graph snapshots and therefore, use separate data structures from the in-memory ones; as a result, iterative computations are unaffected by transactional updates.

6.1.2 Implementing Transactions

Grace uses temporary in-memory buffers for collecting transactional updates and a version vector for a graph, where each element of the vector represents the version of each partition.

Consistent, read-only graph snapshots can be created instantaneously in Grace using copy-on-write techniques. Therefore, at any instant there will be one read-write graph version (on which updates will be applied) and there may be several read-only snapshots. Some of the data structures are shared between all the graph copies, while others are specific to each version. The Vertex and Edge Logs are common across read-write and read-only versions of the graph. Any change made on the graph (such as adding a vertex or modifying an edge set) is appended to the end of the corresponding logs, without affecting other snapshots; that is, no vertex or edge is modified in-place in the log. However, Edge Pointer Array, Vertex Index, and Vertex Allocation Map are unique to different versions of snapshots.

When a new vertex is added, Grace determines the partition where the new vertex should go by rerunning the partition algorithm. Then, the new vertex is appended to the end of the Vertex Log; a copy of the Vertex Index is created, where the new vertex's mapping is included.

If an existing vertex's edge set is modified, the newly modified edge set is appended to the end of the Edge Log; a copy of the Edge Pointer Array is created and its entry corresponding to the modified vertex is changed to point to the new edge set. Finally, if a vertex is deleted, Grace copies the Vertex Allocation Map and clears the corresponding vertex bit in it. In all cases, none of the entries in Vertex Log or Edge Log are updated in place and only the copy-on-write versions of Vertex Index, Edge Pointer Array, and Vertex Allocation Map are modified.

6.2 On-disk Data

Grace's on-disk data is maintained in a fairly straightforward manner. Each partition stores its vertices and edges in respective files. When loading a graph, all the vertices and edges are read parallelly by the partitions, resulting in an overall fast graph loading. In addition to the vertex and edge files, Grace also maintains an on-disk log of committed updates, to recover from crashes.

7 Experiments and Evaluations

We evaluate Grace on two commodity multi-core machines running Windows Server 2008. First machine has 96 GB of memory and four 2.29GHz AMD Opteron 6176 processors, each of which has 12 cores. The other one has 24 GB of memory and two 2.4GHz Intel Xeon E5645 processors, each having 6 cores.

We run representative graph applications such as iterative graph computations, graph traversal algorithms, and online graph queries as benchmarks. For iterative computations, we choose PageRank, Weakly Connected Component (WCC), and Single Source Shortest Path (SSSP). For graph traversal, we select Depth-First Search (DFS) and Breadth-First Search (BFS). Finally, for online queries, we use n-hops queries that retrieve n-hops neighbors' information for a random vertex.

App	**Orkut graph**			**Web graph**		
	BDB	**Neo4j**	**Grace**	**BDB**	**Neo4j**	**Grace**
PgRk	6,548.7	2,428.0	71.1	6,271.8	31,600.0	280.1
WCC	3,510.9	4,315.0	69.3	8,861.9	48,092.0	671.6
SSSP	700.8	2,000.0	12.4	6,336.7	12,702.2	210.1
BFS	1,227.6	2,732.0	7.6	3,743.2	32,500.0	68.5
DFS	1,329.5	1,882.0	19.1	3,707.1	33,416.5	51.6
q4hop	2,052.7	1,791.2	9.1	4.2	32.0	0.162
q3hop	43.9	52.1	0.636	0.473	15.1	0.02

Table 1: **Comparison With Existing Systems.** This table presents the workload running time (seconds) on BDB, Neo4j, and Grace. PgRk refers to PageRank (5 iterations) and q$n$hop refers to a query for n-hops neighbors. To make query run time measurably long, we randomly pick 8 vertices to query from Orkut graph, and randomly select 128 vertices to query from web graph, which has a smaller average degree.

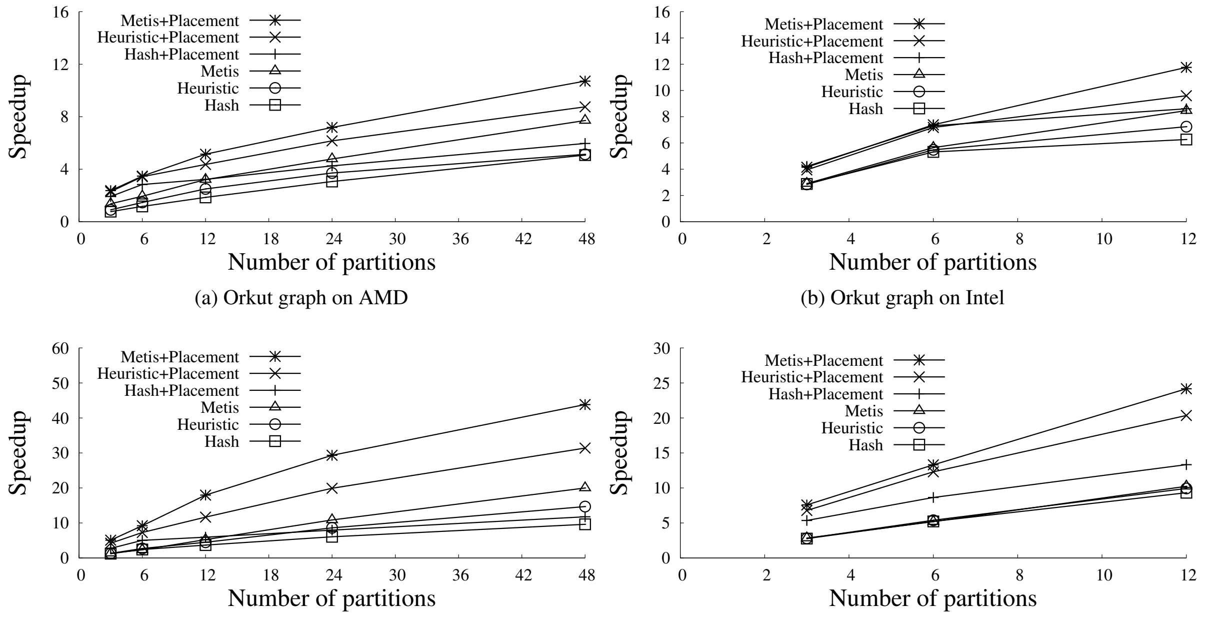
We run our benchmarks on two different graphs: a social network graph and a web graph. For the social network graph, we use a dataset of Orkut which was collected between Oct 3 and Nov 11 2006, and can be publicly accessed [24]. It contains 3,072,441 vertices (users) and 223,534,301 edges. For web graph, we use a dataset from ClueWeb09 [2], which contains the first 10 million English pages crawled during January and February 2009. The corresponding graph consists of 88,519,880 vertices (URLs) and 447,001,255 edges.

In all experiments, applications are run after the whole graphs are loaded into memory and the graphs are kept in memory until the applications finish. Our experiments are designed to answer the following questions. 1. Does Grace outperform existing graph management systems? 2. How effective are the graph-aware and multi-core-specific optimizations? 3. How transactional copy-on-writes impact the runtime? and 4. Does Grace scale to large graphs? We explore these questions in the following sections.

7.1 Comparison with Existing Systems

In order to demonstrate the effectiveness of Grace, we compare it with two existing systems: Berkeley DB (BDB), a widely used open-source key-value store and Neo4j, an open-source transactional graph database implemented in Java.

In BDB, both the key and value are organized as raw bytes. Therefore, to store a graph, we serialize the vertex Id and edges into the key and value payload. For fair comparison, we use the in-memory mode of BDB and set its index as hash table. In Neo4j, we use its vertex and edge creation APIs to import the graphs into its storage. Unlike BDB, Neo4j does not provide an in-memory mode and therefore, we scan the entire graph to preload it into memory before starting applications.



(a) Orkut graph on AMD

(b) Orkut graph on Intel

(c) Web graph on AMD

(d) Web graph on Intel

Figure 5: **Hash vs. Heuristic Partitioning on PageRank**

Since both BDB and Neo4j do not support partitions, we configured Grace to load a graph in to a single partition, where the vertices are rearranged according to their graph proximity, and all three systems use a single thread to run the workloads.

Table 1 compares the execution time of all the applications on BDB, Neo4j, and Grace, running on the AMD machine. The result shows that Grace can perform significantly better than the other two systems – sometimes, up to two orders of magnitude faster – because of its graph-aware optimizations and compact data structures. Other reasons for the poor performance of BDB and Neo4j are: first, search for a vertex following an edge involves an indirect hash-table lookup on vertex Id; second, data structures representing a vertex or an edge are neither compact nor cache-aligned, which leads to larger memory footprint and more cache misses; finally, BDB incurs an additional overhead because it requires extra data copies between its internal key-value structures and the data structures used by applications.

7.2 Optimization Effectiveness

Given the large performance gap between Grace and other systems, we further explore to understand the effectiveness of our optimizations.

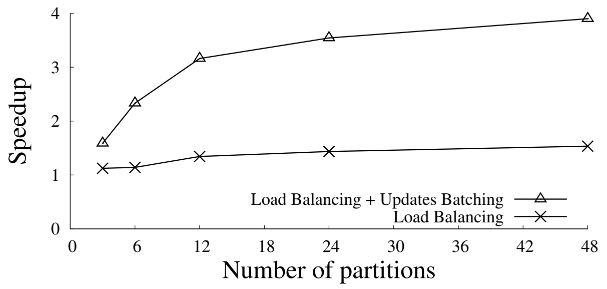
7.2.1 Partitioning and Placement

We run experiments to understand the effectiveness of graph-aware partitioning and vertex placement. For better understanding, we also integrate Metis, a public implementation of multilevel k-way graph partitioning algorithm [5, 19], into Grace's partitioning library for comparison. Figures 5a–5d show the performance of PageRank on Orkut and web graphs on Intel and AMD machines. Each figure plots the speedup from hash, heuristic, and Metis partitioning, with and without vertex reordering, relative to Grace's performance on an unoptimized single partition.

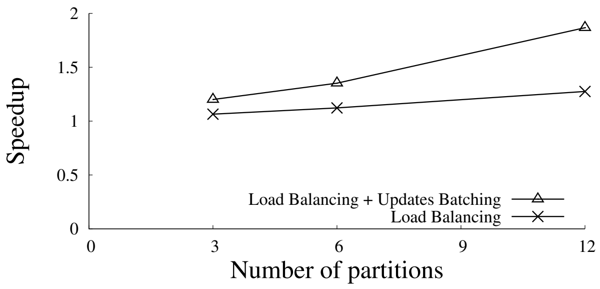
	L1	**L2**	**L3**	**DTLB**
Orkut-Intel	0.46 (6036)	0.50 (5326)	0.44 (1282)	0.31 (999)
Orkut-AMD	0.51 (6825)	0.41 (3815)	0.48 (19469)	0.33 (1118)
Web-Intel	0.23 (7908)	0.27 (12505)	0.20 (2997)	0.24 (6749)
Web-AMD	0.38 (14245)	0.24 (10005)	0.27 (47346)	0.24 (6378)

Table 2: **Relative Reduction in Cache and TLB Miss.** The table shows reduction in cache and TLB miss for PageRank on 12 partitions of Orkut and web graphs when heuristic partitioning and vertex placement optimizations are used. The results are presented relative to the case when only heuristic partitioning is used. Actual number of cache and TLB misses, in millions, is shown in brackets.

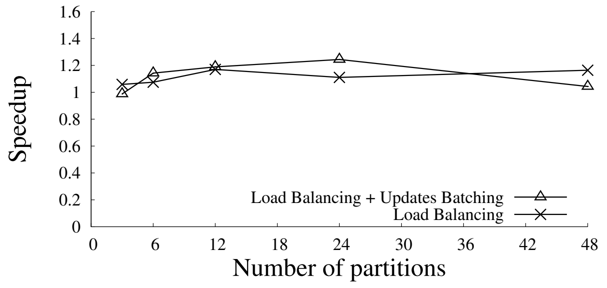
First, graph-aware partitioning alone does not improve the performance if the number of graph partitions are fewer. However, the differences between these partitioning techniques grow larger at higher number of partitions. Second, after the vertices and edges are ordered according to their graph-specific locality, both heuristic and Metis partitioning significantly outperform hash partitioning because when a graph is partitioned in a graph-aware manner, the rearrangement algorithm has more opportunities to place a vertex and its neighbors close to



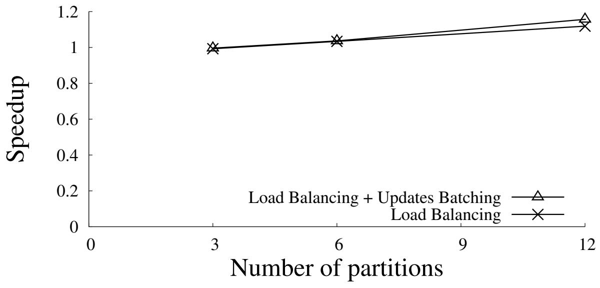
(a) Orkut graph on AMD



(b) Orkut graph on Intel



(c) Web graph on AMD



(d) Web graph on Intel

Figure 6: **Load Balancing and Updates Batching for PageRank**

each other, resulting in better locality.

To better understand the effects of vertex placement, we also measured the cache and TLB miss counts through VTune [7] on Intel machine and CodeAnalyst [3] on AMD machine. Table 2 presents the relative reduction in cache and TLB miss when vertices were rearranged after heuristic partitioning relative to the case when no vertices were rearranged. These results demonstrate that the graph-aware vertex placement can significantly improve the graph data locality.

However, the actual speedup from graph-aware partitioning and placement depends on both the graph and the architecture. Sparse graphs such as the web graph are easier to partition and order, whereas dense graphs such as Orkut are less amenable to vertex layouts. Also, we find that the AMD architecture, with its 48 cores, is more sensitive to cross core communications – perhaps due to the large number of cores – than the 12-core Intel. As a result, we find that the web graph on AMD enjoys a significant speedup from a graph-aware partitioning and placement, whereas Orkut graph on Intel only gains a relatively moderate improvement. Since the Intel and AMD architectures have other differences as well, more investigation is required to understand the reasons behind the different speedups. WCC and SSSP workloads, which are not shown here, also have similar performance curves.

	L1	**L2**	**L3**	**DTLB**	**Remote core**	**Remote chip**
Bal	5867	5172	1343	1019	661	1159
Bal+Bat	4057	2603	421	17	204	291

Table 3: **Hardware Event Counts for Load Balancing and Update Batching.** This table shows the hardware event counts for PageRank running on Orkut graph with 12 partitions on Intel machine. It compares load balancing (**Bal**) with both load balancing and updates batching (**Bal+Bat**). **Remote core** counts the number of retired memory load instructions that hit in the L2 cache of a sibling core on the same die. **Remote chip** measures the number of retired memory load instructions that hit the remote processor socket.

7.2.2 Batching Updates and Balancing Load

Next, we focus on understanding the benefits of load balancing and batching updates on iterative computations.

Figures 6a–6d show the additional benefits (or lack thereof) from load balancing and batching, relative to the performance of Grace with heuristic partitioning and placement, for PageRank workload on Orkut and web graphs on both the machines. Each figure shows two curves: the curve ‘Load Balancing’ represents the benefit just from sharing work among threads and the curve ‘Load Balancing + Updates Batching’ is the speedup from enabling both the optimizations. As can be noted from the figures, these optimizations perform differently for different graphs and architectures.

First, heuristic partitioning’s strength in producing balanced sub-graphs can be observed from the fact that

App	AMD (48 cores)		Intel (12 cores)	
	Orkut	Web	Orkut	Web
PgRk	34.1	36.5	17.9	23.5
WCC	40.4	19.2	14.5	12.5
SSSP	14.8	8.2	6.8	8.2

Table 4: **Maximum Speedup.** This table presents the maximum speedup Grace gets with all optimizations and maximum number of partitions (as supported by the architecture). Speedup is measured relative to a baseline with a single partition and no optimizations.

load balancing strategy did not offer substantial benefits. Second, updates batching is very effective for dense graph such as Orkut, which offers more destination edges for batching and hence shows better performance. Sparse graphs such as web graph do not have sufficient destination edges for updates batching to be effective; however, since the source vertex's data is accessed once for every destination partition (as explained in Section 6.1.1), the extra overhead can sometimes reduce the performance. We noticed similar results for other workloads as well and they are not shown here.

We measure the performance counters for PageRank on Orkut graph on Intel machine to better understand the effects of update batching (we do not measure the numbers for AMD machine since, to the best of our knowledge, it does not provide counters for cross-core access). Table 3 shows that the number of remote core and remote chip accesses significantly decrease when updates are batched. We notice that the cache and TLB miss counts also decrease considerably. We believe that this is mainly caused by the following two reasons: first, since each remote core access implies an L2 cache miss (L2 cache is private for each core) and each remote chip access implies an L2 and L3 cache miss (L3 cache is shared by the cores on the same chip), reduction in remote core or chip accesses will also decrease L2 and L3 cache misses; second, batching could lead to access patterns with better locality because vertices in the same partition are more likely to be in the same cache line than vertices from different partitions.

It is clear that not all optimizations work equally well on all graphs (or architectures); so, it is important to have a system configurable to specific scenarios. Grace provides the flexibility to (manually) choose the right configuration desired by the user. In the future, we can add logic that automatically selects optimizations appropriate for a graph; for example, for sparse graphs, Grace can turn off updates batching.

7.2.3 Overall Speedup

Table 4 presents the maximum speedup Grace achieves for a specific workload-graph-and-architecture combination. The speedup is calculated with respect to a Grace configuration with just one partition and without any other optimization. We can notice speedups up to 40 times (for the case of WCC on Orkut graph on AMD), reducing the runtime of those batch-processed workloads to a few minutes.

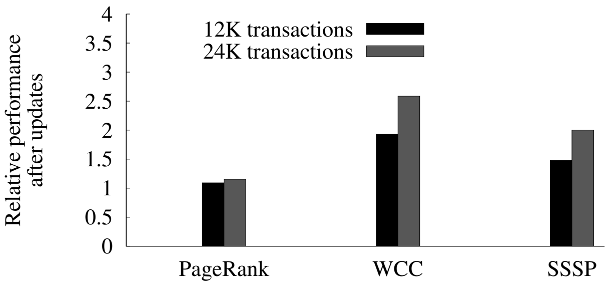


Figure 7: **Impact on Runtime due to Copy-on-Writes.**

7.3 Effect of Transactional Updates on Layout

Copy-on-write is a great mechanism for implementing instantaneous snapshots, consistent updates, and transactions. However, by their very nature, copy-on-writes can disrupt object layouts; specifically, when a data item is changed, it is copied to a different location voiding any careful cache or memory layout optimizations previously made. We evaluate the transactional support in Grace to understand how it affects the in-memory layout and therefore, the performance of workloads.

Figure 7 presents the increase in runtime for all the iterative workloads on Orkut graph, after the graph was updated with transactions; the measurements are presented relative to the runtime on Orkut graph without any changes. Each transaction either randomly deletes a vertex or adds a new vertex with the same set of neighbors as the last deleted one; although, this does not change the conceptual structure of the graph, it does affect Grace's memory layout significantly, where every neighbor and neighbor's edges of a deleted (or added) vertex is copied to the end of the Vertex Log or Edge Log. We vary the number of transactions and estimate the increase in the runtime. We can find that degradation in performance is workload dependent; whereas PageRank suffers only a small increase in runtime, others like WCC and SSSP incur a noticeable drop in performance (up to 2.5 times, for 24K transactions). To overcome such performance drops, Grace can periodically reorder its modified memory layout and regain its original performance.

7.4 Scalability of Grace

Our previous experiments showed that Grace can scale well on large multi-cores and our final experiment measures the scalability of Grace to large graphs. We used a

Graph	PageRank	WCC	SSSP
Hotmail	190.1	109.9	46.5

Table 5: **Scalability to Large Graphs.** This table presents the running time (seconds) of iterative workloads on Hotmail graph.

Hotmail user graph, which is a weighted graph with over 320 million vertices and 575 million edges and ran the iterative workloads on it. Table 5 presents the runtime of iterative computations, which shows that Grace can reduce the runtime of PageRank-like workloads (which we ran for 20 iterations) to close to 3 minutes.

8 Related Work

Systems for Graphs. Early research systems for graphs were built for web graphs, where fast random access is a basic requirement; consequently, systems such as Connectivity Server [9] kept entire graphs in memory and focused on how to compress them efficiently. While early systems ran on a single machine, later systems such as Scalable Hyperlink Store [25] were distributed in design. Unlike these systems, Grace is general-purpose and do not rely on specific graph characteristics. Moreover, none of the early systems focused on multi-core optimizations or transactional updates.

To support efficient graph computations, systems like Pregel [21], Naiad [23], GraphLab [20], and Parallel Boost Graph Library [14, 15] provide a natural graph programming API, which is adopted by Grace. Of all these systems, Grace is more similar to GraphLab as they both operate in memory and run on multi-cores; however, the similarities stop there. Whereas GraphLab is optimized for machine learning algorithms, Grace is general purpose. Grace is unique in how it uses its graph awareness for efficient in-memory graph representations. Moreover, none of these systems, including GraphLab, provide transactional or snapshotting capabilities.

Neo4j [6] is an open-source, single-box graph database. It supports transactions but does not allow graph computations to run concurrently with transactions, unless the computation is wrapped in a big transaction, which makes it likely to be aborted. Moreover, Neo4j's internal in-memory data structures are not optimized for graphs, resulting in poor performance.

Graph-Aware Optimizations. Graph partitioning is a well-studied problem. Its goal is to produce partitions with fewer cross-partition edges. Several offline partitioning algorithms [18, 29] have been proposed, which typically take longer to run. Pujol *et al.* [27] proposed an online partitioning mechanism to obtain good partition quality with a small replication cost. While this works well on distributed systems, – where the cost of accessing a remote partition is far higher than what we observe in multi-cores – we find that a graph-aware partitioning alone does not perform better; however, when accompanied by vertex ordering, graph-aware partitioning works very well on multi-cores.

Imranul *et al.* [17] proposed to leverage the community structure of social graphs to optimize its layout on disk. Similarly, Diwan *et al.* [13] presented a clustering mechanism for tree structure to achieve good locality on disk. Although relevant, these past work optimize for specific graph characteristics (such as social graph or tree), whereas Grace remains general-purpose. Moreover, since Grace keeps an entire graph in memory, it focuses on memory and cache layouts, and does not bother with disk structures.

Multi-Core Optimizations. Since multi-cores are a de facto standard on modern servers, data-parallel computations and key-value stores have been evaluated in them [22, 28]. Recently, many graph algorithms [8, 11, 16] have been specifically designed and implemented for multi-core system.

Jacob *et al.* [26] developed a system runtime out of commodity processors to optimize graph applications. Although they focus on single machine performance, they assume that graph locality is hard to improve and therefore, propose to tolerate memory access latency with massive concurrency. They built a lightweight multithreading library, which accesses memory asynchronously using prefetch instructions. Their technique is orthogonal and can be complementary to Grace.

9 Conclusion

In the last 15 years – perhaps fueled by the stupendous growth of web and, more recently, social networks – systems researchers have been working on special purpose storage and computation platforms for graphs. Grace is another entry in this continuing effort. Grace, with its graph-awareness and optimizations for multi-cores, is unique among them. We plan to extend Grace to run on distributed frameworks, where we hope to apply its current optimizations and discover new ones in those contexts as well.

Acknowledgments

We thank the anonymous reviewers and Pei Cao for their tremendous feedback. We also thank the members of Microsoft Research Silicon Valley for their excellent suggestions and feedback. We are grateful to Frans Kaashoek and Nickolai Zeldovich for their valuable comments on this work, and Sean McDirmid for proof reading.

References

[1] BerkeleyDB. `http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html`.

[2] Clueweb09. `http://lemurproject.org/clueweb09/`.

[3] CodeAnalyst. `http://developer.amd.com/tools/codeanalyst/Pages/default.aspx`.

[4] Hadoop. `http://hadoop.apache.org/`.

[5] METIS. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.

[6] Neo4j. `http://neo4j.org`.

[7] VTune. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`.

[8] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[9] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The connectivity server: fast access to linkage information on the Web. In *Proceedings of the 7th international conference on World Wide Web 7*, pages 469–477, 1998.

[10] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7):107–117, 1998.

[11] G. Cong and K. Makarychev. Optimizing large-scale graph analysis on a multi-threaded, multi-core platform. In *IPDPS*, pages 688–697, 2011.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04*, pages 137–150, 2004.

[13] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *VLDB'96, September 3-6, Mumbai (Bombay), India*, pages 342–353, 1996.

[14] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *OOPSLA'05*, pages 423–437, New York, NY, USA, 2005. ACM.

[15] D. Gregor and A. Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC'05)*, 2005.

[16] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, PACT'11*, pages 78–88, 2011.

[17] Imranul Hoque and Indranil Gupta. Social Network-Aware Disk Management. Technical report, University of Illinois at Urbana-Champaign, 2010-12-03.

[18] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[19] G. Karypis, V. Kumar, and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[20] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. GraphLab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence*, 2010.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD'10*, pages 135–146, New York, NY, USA, 2010. ACM.

[22] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys'12, Bern, Switzerland*, pages 183–196, 2012.

[23] F. McSherry, R. Isaacs, M. Isard, and D. G. Murray. Naiad: The animating spirit of rivers and streams. Poster Session of Symposium on Operating Systems Principles (SOSP'11), 2011.

[24] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, pages 29–42, New York, NY, USA, 2007. ACM.

[25] M. Najork. The scalable hyperlink store. In *Proceedings of the 20th ACM conference on Hypertext and Hypermedia*, pages 89–98, 2009.

[26] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, HotPar'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[27] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The Little Engine(s) That Could: Scaling Online Social Networks. In *SIGCOMM'10*, New Delhi, India, Aug. 2010.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA'07*, pages 13–24. IEEE Computer Society, 2007.

[29] D. A. Spielman and S.-H. Teng. Spectral partitioning works: Planar graphs and finite element meshes. In *FOCS*, pages 96–105, 1996.

[30] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[31] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2009.

[32] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI'08, San Diego, California*, pages 1–14, 2008.