A Hybrid Approach To Processing Big Data Graphs on Memory-Restricted Systems

Harshvardhan, Brandon West, Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger
Parasol Laboratory
Dept. of Computer Science and Engineering
Texas A&M University
$\{$*ananvay, west, fidel, amato, rwerger*$\}$*@cse.tamu.edu*

***Abstract*—With the advent of big-data, processing large graphs quickly has become increasingly important. Most existing approaches either utilize in-memory processing techniques that can only process graphs that fit completely in RAM, or disk-based techniques that sacrifice performance.**

In this work, we propose a novel RAM-Disk hybrid approach to graph processing that can scale well from a single shared-memory node to large distributed-memory systems. It works by partitioning the graph into subgraphs that fit in RAM and uses a paging-like technique to load subgraphs. We show that without modifying the algorithms, this approach can scale from small memory-constrained systems (such as tablets) to large-scale distributed machines with $16,000+$ cores.

***Keywords*-parallel graph processing; out-of-core graph algorithms; big data;**

I. Introduction

In the past decade, processing large-scale graphs has become increasingly important in a large variety of domains, from scientific computing to extracting information from social networks. Processing these graphs in a reasonable time usually requires parallelism. However, parallelizing graph algorithms efficiently is a challenging problem that has received significant attention for several decades [18].

Over the past few years, many systems specialized for graph-processing have emerged to address this issue ([7], [8], [10], [13], [14]). These systems are either designed for in-memory or disk-based graph-processing. In-memory systems, whether distributed or shared-memory, store and process the entire graph in RAM. While this can provide good performance, the size of the graph that can be processed is limited by the amount of RAM available in the system. Disk-based systems, on the other hand, are not limited by RAM availability, but sacrifice performance due to disk I/O. Recently, a hybrid system (GraphChi [12]) was proposed to allow a single-node system to process large graphs using a parallel sliding-window approach. However, this was limited to a single shared-memory node.

In this paper, we propose a graph processing system that can scale well from small memory-restricted systems to large distributed-memory machines. This RAM-disk hybrid graph-processing system provides a unified approach to utilize the available resources (RAM, disk, cores) efficiently and seamlessly. Our system decouples algorithms from the details of the machine, allowing users to write fine-grained vertex-centric algorithms, that can run efficiently without modification on different systems. We use an approach similar to *memory paging*, but applied to subgraphs, that loads subgraphs of the graph in memory, processes them, and then stores them back to disk. A crucial difference from paging is that while paging uses fixed-size pages, our approach uses partitions based on graph structure. This preserves structural locality that can reduce disk I/O. We use an *asynchronous push model* that allows paged-in vertices to be processed, while updates to their neighbors are asynchronously pushed, or deferred, until the neighbors are loaded. We also propose system-level optimizations that do not require changes to algorithms, but can reduce disk I/O. An implementation of our approach in the STAPL Graph Library [8] allows us to process large graphs on systems ranging from small-scale systems such as off-the-shelf PCs or Android tablets, to large high-end clusters. Our results show that our subgraph-paging based approach and asynchronous push model, together with optimizations, provides $3-12\times$ faster graph processing on a single node than the best alternative, GraphChi, and extends efficiently to multiple nodes which GraphChi cannot.

Our contributions include:

- A hybrid subgraph-paging based approach to processing large graphs, allowing both in-memory and out-of-core processing.
- An implementation that transparently allows fine-grained level-synchronous graph algorithms to benefit from our hybrid approach without code modifications.
- An experimental evaluation showing good scalability and showing improved performance over existing disk-based and in-memory graph processing frameworks on systems ranging from small Android tablets to off-the-shelf PCs to large clusters with $16,000+$ cores.

II. Overview of Our Approach

We provide a graph-processing engine (the *graph_paradigm*) to control the execution of algorithms, that can effectively utilize available resources for fast, scalable processing of graphs. To do this, we allow users to express their algorithms in a vertex-centric fine-grained manner that abstracts them from details of parallelism and exposes the maximum amount of parallelism available in the algorithm. The graph algorithms themselves do not change and are decoupled from execution policies (distributed/shared, disk/in-memory, etc.), allowing the execution strategies to change, without changing the algorithm.

Our approach for out-of-core processing is similar to paging, as we partition the input graph into subgraphs (logical partitions), such that each subgraph may fit in main-memory. When needed, the subgraphs are paged-in from disk to main-memory and processed. In this *asynchronous push model*, vertices of a subgraph are only processed in-memory, and updates to neighboring vertices are asynchronously pushed to the subgraphs as follows. If the subgraph containing a neighboring vertex is also in memory, the vertex is updated. If the subgraph is stored on disk, the update is written to disk and applied the next time the subgraph is loaded. The approach differs from paging, as we reflect our pages at a logical (subgraph) level, versus a non-semantic (kilobyte, megabyte) level. This allows us to take advantage of temporal locality in subgraphs.

We also implement multiple optimizations to reduce disk I/O. These optimizations include caching *hub vertices* (i.e., vertices with very large degrees) when they are written to disk, skipping graph-structure writes for unmodified graphs, and over-partitioning subgraphs to utilize the RAM to the fullest extent. The optimizations are system-level, and therefore do not involve modifying the algorithms.

This technique and its implementation is described in more detail in Section IV. The next section describes how algorithms are expressed in the asynchronous push model.

III. Preliminaries

Our approach is applicable to frameworks using the asynchronous push model. In this paper, we assume that algorithms are expressed using the k-level-asynchronous (KLA) two-operator algorithmic specification presented in [9], which is one such push model that was shown to be generally applicable. This section provides an overview of the KLA graph-processing paradigm and how algorithms are expressed in it.

A. The KLA Graph Paradigm

KLA is a graph-processing paradigm that unifies Bulk Synchronous Parallel (BSP) [21] and asynchronous paradigms, allowing each BSP superstep to execute up to k levels of the algorithm asynchronously. KLA allows users to express fine-grained vertex-centric graph algorithms. KLA algorithms are decoupled from parallelism and communication details, as well as from the processing of the graph, whether it is level-synchronous or asynchronous, or stored on disk or in RAM.

This section shows how an example algorithm, breadth-first search (Figure 2) is expressed in the KLA paradigm. Breadth-first search (BFS) is an important graph algorithm due to its extensive usage in traversing graphs, and due to its indirect usage as a part of numerous other graph algorithms, such as betweenness centrality. A BFS of a graph marks each of its vertices with their distance from a given source vertex.

```
void graph_paradigm(Graph graph, VertexOp wf, NeighborOp uf)
  bool active = true;

  while(active) {
    pre_compute(graph);

    // apply vertex-operator to each vertex, reduce to
    // find #active vertices. wf returns true (active),
    // or false (otherwise), spawns neighbor-operators.
    active =
      reduce(map(vertex_wf(wf, visitor(uf)), graph), logical_or());
    global_fence();

    post_compute(graph);
  }
```

Figure 1. Pseudocode for the graph paradigm.

To express an algorithm, the user provides two operators – a *vertex-operator* (Figure 2(a)) which performs the computation of the algorithm on a single vertex and a *neighbor-operator* (Figure 2(b)) that updates the neighbor-vertices of the source vertex with the results of the computation. These two operators are then provided to the KLA graph paradigm along with the input graph (Figure 2(c)). The graph paradigm (Figure 1) executes the provided operators on active vertices of the input graph and handles communication, termination-detection of the algorithm, current active vertices, and the execution strategy (level-synchronous, asynchronous, KLA). The user's operators are decoupled from these details and can focus on expression of the algorithm.

For BFS, the vertex-operator checks if a vertex is active (grey) and propagates its distance to its neighbors through the neighbor-operator. The neighbor-operator updates the distance of the neighbor if needed, and marks it as active (grey). Active neighbors are processed by vertex-operators in the next iteration. Other algorithms such as connected components, k-core, PageRank, graph coloring, topological sort can also be expressed in a similar manner.

B. PageRank and Other Algorithms

The PageRank algorithm [3], [16] is a representative random-walk algorithm used to rank web-pages on the internet in order of relative importance. The PageRank computation proceeds in iterations, where each vertex calculates its rank in iteration i based on the ranks of its neighbors in iteration $i-1$, and then sends its new rank to its neighbors for the next iteration. Termination happens upon convergence of ranks or upon reaching a predetermined threshold for iterations. The STAPL GL implementation of PageRank's vertex and neighbor operators is shown in Figure 3.

Other algorithms, such as k-core decomposition and connected components can be expressed in a similar two operator fashion. More details can be found in [9].

```
bool bfs_vertex_op(vertex v)
  if (v.color == GREY)          // Active if GREY
    v.color = BLACK;
    VisitAllNeighbors(bfs_neighbor_op(_1, v.dist+1), v);
    return true;                // vertex was Active
  else  return false;           // vertex was Inactive
```

(a) vertex-operator

```
bool bfs_neighbor_op(vertex u, int new_distance)
  if (u.dist > new_distance)
    u.dist = new_distance;   // update distance
    u.color = GREY;          // mark to be processed
    return true;             // vertex was updated
  else  return false;
```

(b) neighbor-operator

```
void BFS(Graph graph, vertex source)
  source.color = GREY;
  graph_paradigm(bfs_vertex_op(), bfs_neighbor_op(), graph);
```

(c) Algorithm-driver

Figure 2. The fine-grained BFS algorithm.

```
bool pagerank_vertex_op(Vertex v)
  if (v.iteration < 20)
    v.rank = 0.15/num_vertices + 0.85*v.sum_ranks;
    v.iteration++;
    v.sum_ranks = 0;
    int n = v.neighbors().size();
    VisitAllNeighbors(v, pr_neighbor_op(_1, v.rank/n));
    return true;                // vertex was Active
  else  return false;           // vertex was Inactive
```

(a) vertex-operator

```
bool pr_neighbor_op(Vertex u, double rank)
  u.sum_ranks += rank;
  return true;
```

(b) neighbor-operator

Figure 3. The PageRank algorithm.

IV. Hybrid Graph Processing

In this section, we describe our hybrid approach to processing in-memory and out-of-core graphs. While our approach is applicable to frameworks using the asynchronous push model, for this work we assume that algorithms are expressed using the KLA two-operator algorithmic specification presented in Section III. Using this specification, our approach can utilize the vertex- and neighbor-operators to keep the algorithm unchanged from its in-memory variant, while our paradigm handles the execution (in-memory or out-of-core) and storage (which parts of the graph should be in-memory vs. on-disk).

A. Graph Storage

We start with an input graph partitioned into p subgraphs. Each subgraph is assigned to a *location* (a single processing element), and a location can have multiple subgraphs assigned to it. The location to which a subgraph is assigned is its *home location*, which is responsible for managing the subgraph and executing requests on its vertices. Each subgraph can be present either in memory (*loaded*) or on disk (*un-loaded*), and each location can independently decide when to load and un-load which subgraphs based on memory availability.

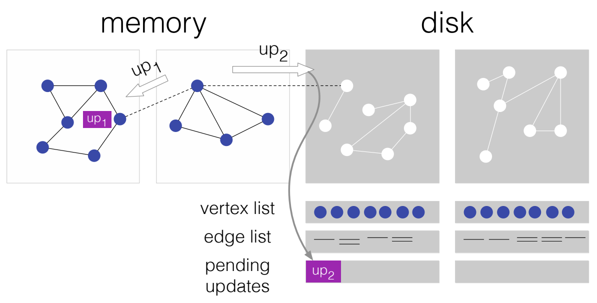


Figure 4. Diagram of the storage paradigm with two subgraphs in memory and two subgraphs on disk. An update on a loaded vertex is being applied in memory while an update to an un-loaded vertex is being stored in the pending updates shard.

File format for sub-graphs. Each subgraph is stored in a binary format across three shards (Figure 4). The first shard is the *vertex list*, which contains the vertices of the subgraph, along with any vertex data (such as level for BFS, rank for PageRank, and ID for connected components). The second shard is the *edge list*, which contains the list of edges for each vertex, in order, including any data for the edges. These are stored separately to allow for optimizations when a vertex's edges need not be loaded, for indexing into the vertex list, and also to enable us to bypass rewriting the edge list shard unless the graph structure changes (Section IV-C). The third shard, *pending updates*, stores any incoming and pending updates to vertices belonging to that subgraph. When the subgraph is loaded, the updates shard is read and all outstanding updates are applied to corresponding vertices.

B. Processing Out-of-Core Graphs

We use an approach similar to paging to process out-of-core graphs (Figure 6). The computation proceeds in bulk-synchronous supersteps, and for every superstep, each location loads one or more of its active subgraphs (based on available RAM) in some schedule (for our experiments, we choose a round-robin policy), processes the vertices and stores the subgraphs back to disk, including any modified state. A different subset of subgraphs is then loaded and processed, until all active subgraphs have been processed for the given superstep. The process is then repeated for the next superstep until there are no remaining active vertices (i.e., vertices that will actively compute in the current superstep).

If the machine has sufficient RAM to store the graph and metadata in-memory, all subgraphs can be kept in RAM, and no penalty is paid for disk access. If the RAM is not sufficient, the paradigm will load and unload subgraphs to allow the seamless execution of the algorithm.

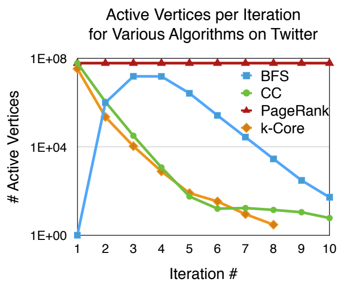
Inactive vertices or sub-graphs. Graph algorithms are often iterative, and for a large class of graph algorithms, not every vertex is actively processed in each iteration. An example of this is shown in Figure 5, which plots an execution profile in terms of the number of active vertices, for the first ten iterations of various representative algorithms. As can be observed, the algorithms shown, with the exception of PageRank, only process a very small fraction of their vertices in any given iteration, and only a few iterations process any significant fraction of the vertices. This provides an opportunity to skip reading and loading vertices that will not be processed.

In our approach, if a vertex's value is updated, it is marked as *active* for the next superstep. This implies that the vertex may potentially be processed in the next superstep using a vertex-operator, which can access its adjacent edges. For such vertices, it is required to load their adjacent edges. However, for *inactive* vertices in a given superstep, we can skip the loading of their adjacent edges, as it is guaranteed that such vertices will not have the vertex-operator applied to them. The inactive vertices and their values are still loaded in memory, however, to allow incoming updates to be applied to them from their neighboring vertices.

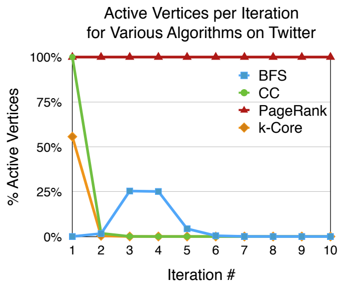
As an extension, if all vertices of a given subgraph are *inactive* for a given superstep, we skip the loading of the entire subgraph for that superstep. This is common for many graph traversals (such as BFS), which start from a single vertex, and only a few vertices are active in most iterations. This is also commonly observed in other graph algorithms (Figure 5). In such cases, a significant amount of time can be saved by not reading and loading entire inactive subgraphs from disk.

Figure 7(a) shows the impact of skipping inactive subgraphs (Opt1), along with the optimization of skipping inactive vertices (Opt1+2) vs. the baseline execution time for the Graph 500 benchmark input with 16 million vertices and 256 million edges on a PC with 4GB of RAM. Skipping inactive subgraphs by itself may not provide a significant improvement as opportunities of skipping entire subgraphs for Graph 500 inputs are few. However, for graphs that can be partitioned well, this can provide substantial benefits.

Updates. We use a policy which defers updates to subgraphs not present in memory to avoid unnecessary paging. Updates produced during processing of in-memory subgraphs are asynchronously forwarded to the home location of the target vertex. If the target vertex is present in RAM, the update is applied to it. If the target vertex is stored on disk, the update is written to a *pending updates* shard corresponding to the target vertex's subgraph, and applied when that subgraph is next loaded. This process is illustrated in Figure 4. The deferred paging policy works, as our asynchronous update model only guarantees that the effect of updates in superstep i will be visible in superstep $i+1$, and therefore, the effects



(a)



(b)

Figure 5. Variation in active vertices in the first ten iterations of various algorithms on the Twitter graph, shown as (a) count on log-scale, and (b) percentage of total vertices.

```
for_each(superstep ∈ algorithm)
  for_each(subgraph ∈ graph)
    subgraph.load();   // load subgraph to RAM
    subgraph.apply_pending_updates();

    for_each(v ∈ subgraph.vertices())
      updates = vertex−operator(v);

      Asynchronously forward updates to home locations:
        Apply updates to vertices in in−RAM subgraphs;
        Store updates to on−disk subgraphs as pending;

    subgraph.store();   // store to disk
```

Figure 6. Pseudocode for the off-core graph paradigm.

of updates need not be immediately visible. Another benefit of the push-model is that it does not have to load and read neighboring vertices' states, and can simply send updates to neighbors asynchronously.

Dynamic graphs. Our paradigm supports dynamic graph operations such as adding and deleting edges or adding and deleting vertices as the computation progresses. These operations are treated in the same way as regular updates described above. Modifications to graph structures are applied directly to subgraphs loaded in RAM, or are stored to an *outstanding modifications* shard if the subgraph is stored on disk and applied immediately upon loading. If a subgraph is modified during a superstep, a dirty-bit is set to indicate the modification, ensuring that the modified subgraph will be written to disk with the updated changes.

C. Optimizations To Reduce Disk I/O

The majority of time for processing out-of-core graphs is spent in disk I/O. Therefore, we propose and implement several optimizations to reduce the number of bytes that need to be read from or written to disk. These optimizations are internal to the storage paradigm and therefore do not require any changes to the user algorithm.

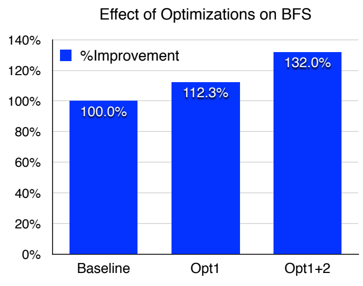
Graph structure writes. Many algorithms do not change the graph structure (i.e., they do not add or delete vertices or edges) but only update the values stored on vertices and edges of the graph. For such algorithms, we do not need to write back the structural information of the graph when storing a subgraph. In our implementation, we achieve this by separating the structure (edges) and values into separate shards (Section IV-A). The edge list shards are only updated if the graph structure was modified. The vertex list shards contain the vertex IDs and vertex values, and therefore are updated if the values change.

Multiple sub-graphs (over partitioning). Multiple subgraphs can be loaded in RAM at the same time. Hence, using smaller subgraphs allows finer-grained control, while still utilizing the available RAM to the fullest extent by loading multiple small subgraphs at once. This also increases the probability of a subgraph being inactive and therefore skipped. However, there is a trade-off between the size of a subgraph and the overhead of metadata for each subgraph.

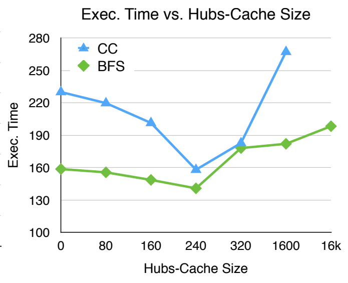
In-memory cache for hub vertices. Many social-networks and web-graphs exhibit small-world scale-free behavior with a power-law degree distribution. This implies the existence of *hub vertices* which are connected to a large number of neighbors. For such vertices, especially if they have a large in-degree, there may be multiple updates being applied to them from their neighbors in the same superstep. In such cases, if the subgraph containing the vertex is stored on disk, the updates will be written to the updates shard. These numerous writes can be avoided by using an in-memory write-back cache for such hubs. In our implementation, the hub vertices are identified at the time of graph creation (or modification) using a simple linear scan of the vertex degrees. An in-memory cache corresponding to each subgraph maps the hub vertices within the subgraph and stores its cached value.

When the subgraph is present in RAM, the cache is inactive and any updates are directly applied to the hub vertex. However, if the subgraph exists on disk, the cache is active and applies all incoming updates to the cached value. When the on-disk subgraph is next loaded, the cached value is written back to the original hub. Since the number of such hubs is usually small, the cache does not use a significant amount of resources. However, the time saved in writing, and then later reading and applying the updates can be significant. The size of the cache may be user specified, or determined from profiling the system.

Figure 7(b) shows the effect of cache size on algorithm performance for the connected components (CC) and BFS algorithms on a Graph500 benchmark graph. We found a value of 15 hubs per 1 million vertices to provide performance benefits of $15-40\%$ for a system with 4GB RAM. Increasing the cache size beyond this results in larger overhead of maintaining the cache and diminishing benefits due to the power-law degree of Graph500 inputs.



(a)



(b)

Figure 7. Effects of optimizations on performance: (a) Skipping inactive subgraphs (Opt1) and inactive vertices (Opt2), (b) Effect of varying size of hubs-cache for BFS and connected components (CC). Both plots run on Graph 500 input graph with 16 million vertices and 256 million edges, on a PC with 4GB RAM.

D. *Analysis of I/O Costs*

We use the I/O model described by Vitter [15], [5] to analyze the I/O costs of our approach. This model expresses the cost of an algorithm using the number of block-transfers of size B bytes from disk to main memory. An assumption in their model is that a disk can transfer a contiguous block of B bytes of data about as fast as it can transfer a single bit. Our analysis makes a worst case assumption that none of the optimizations in Section IV-C are applied and that all vertices in the graph are active.

We assume a graph of $|V|$ vertices and $|E|$ edges, of total size N bytes, is partitioned into m subgraphs, and each subgraph of size $\frac{N}{m}$ bytes contains the subset of the graph's vertices and edges that make up the partition. Based on this, all subgraphs of a graph can be transferred using $P = \frac{N}{B}$ block transfers. We also assume p_{node} compute nodes in the system, each with local disk and with main memory of M bytes, where $\frac{N}{m} \leq M$, and that each compute node stores $\frac{m}{p_{node}}$ subgraphs on local disk. Then, the total I/O cost (C_B) of our approach per superstep of the algorithm is given by summing the cost of reading and writing for the superstep:

$$C_B(G) = C_B^{Read}(G) + C_B^{Write}(G) \tag{1}$$

For each superstep, in the worst case, we process every subgraph. Therefore, the total number of block transfers for reading per node is at most $\frac{P}{p_{node}}$. Similarly, for each superstep, in the worst case, we would update the neighbor for every edge in the graph, giving a total number of block transfers for writes per node of $\frac{|E|}{B \cdot p_{node}}$. Therefore:

$$C_B(G) \leq \frac{P}{p_{node}} + \frac{|E|}{B \cdot p_{node}} \tag{2}$$

Finally, if the algorithm performs S supersteps, the total I/O cost for processing the graph is given by:

$$C_B^{Algo}(G) \leq S \cdot (\frac{P}{p_{node}} + \frac{|E|}{B \cdot p_{node}}) \tag{3}$$

For comparison, as shown in [12], using the same model, GraphChi has an upper-bound of $\frac{4|E|}{B} + \Theta(P^2)$ for the I/O cost of a *single superstep*. This is significantly more than in Equation 2, which is linear in the number of partitions. This is due to GraphChi's parallel sliding-window approach, where each superstep is executed in P execution intervals, and each execution interval requires $O(P)$ non-sequential disk reads to load the edges from $P - 1$ sliding shards for each of the P execution intervals in that superstep, giving a quadratic complexity. We also note that as our approach supports distributed-memory machines, each with independent disks, we can reduce our I/O costs by a factor of p_{node}, whereas GraphChi is limited to a single node.

V. IMPLEMENTATION

We implemented our approach in the STAPL framework, by extending the STAPL Graph Library (STAPL GL), which previously supported in-memory graph computations only, to support out-of-core processing. The API of the STAPL GL did not change, as our technique is transparent to the user. This also allowed us to use the existing STAPL GL algorithms *without modification*.

A. The STAPL Framework

STAPL [4] is a framework for parallel C++ code development. STAPL's core is a library of C++ components implementing parallel algorithms (`pAlgorithms`) and distributed data structures (`pContainers`). The STAPL Runtime System (RTS) and its communication library ARMI (Adaptive Remote Method Invocation) abstract the underlying platform, providing portable performance, thus eliminating the need to modify STAPL applications. The RTS abstracts the physical parallel processing elements into *locations*, components of a parallel machine such that each one has a contiguous memory address space and associated execution capabilities (e.g. threads). ARMI uses the remote method invocation (RMI) abstraction to allow asynchronous communication on shared objects (`p_objects`), while hiding the underlying communication layer (e.g MPI, OpenMP, threads, etc.). Users of the RTS, such as the `pContainers` and the `PARAGRAPH`, can create `p_objects` over locations and invoke RMIs on them.

B. The STAPL Graph Library

The STAPL Graph Library (STAPL GL) [8] consists of a generic parallel graph container (`pGraph`), graph `pViews`, and a collection of parallel algorithms to allow users to easily process graphs at scale. The `pGraph` container is built using the `pContainer` framework (PCF) provided by STAPL.

STAPL GL algorithms are expressed using the *KLA-paradigm* (Section III-A). The KLA paradigm applies user-provided operators on the input graph to execute the algorithm. It abstracts the execution, communication and parallelism from the user. As the KLA two-operator algorithmic specification implements an asynchronous push model, our hybrid approach can use the same algorithms provided in the STAPL GL, without modification.

The graph structure is stored in the `pGraph`, a distributed container which consists of one or more base containers per location. The subgraphs are stored in the base container, which we allow to be stored in-memory or on-disk. The base container also contains the in-memory hubs-cache, if the optimization is enabled. To allow for loading and storing subgraphs to disk, we extended the `pGraph` base containers to provide serializing and deserializing capabilities, and implemented the ability to read and write shards. The asynchronous forwarding of updates is processed by a two-level distributed directory which maps vertices to their home-locations.

VI. EXPERIMENTS

In this section, we evaluate the performance of our approach on various platforms, and compare it with existing in-memory and disk-based libraries. Our approach is evaluated on a set of important graph mining and graph analytics algorithms, and a variety of inputs. The input graphs include the Graph 500 benchmark inputs [1], a benchmark for data-intensive and graph applications, as well as real-world graphs available to us.

Our experiments were run on multiple platforms – a Cray XE6 machine with 153,216 cores (Hopper), and a smaller Cray XE6m machine with 576 cores available to us. In addition, we evaluated our technique on two 4-core PCs with 4GB RAM and 16GB RAM (respectively), both with 7200 RPM hard disk drives, and a 4-node commodity cluster with two quad-core processors per node. Finally, we ran on an EXYNOS 4 quad-core tablet running Android 4.0, containing an ARM Cortex-A9 with 1GB of DDR2 main memory and a 16GB MMC card for storage. Graphs were distributed as specified by their input files, using a block distribution of the vertices.

A. Comparison with in-memory libraries

We study our in-memory performance and compare it with existing in-memory graph libraries using the Graph500 benchmark. Our approach is the only one among the libraries that can perform out-of-core computations, however, our comparisons with both shared-memory and distributed-memory systems still show competitive in-memory performance with no penalty paid for disk-access when sufficient RAM is available.

Figure 8 shows the execution times of various libraries normalized to the time taken by our approach implemented in the STAPL GL. We note that the STAPL GL performs comparably to shared-memory libraries, and scales better and is faster than other distributed-memory graph libraries such as the Parallel Boost Graph Library (PBGL) [7] and GraphLab/PowerGraph [13]. While our approach is $2\times$

slower than the Graph500 benchmark implementation [1] up to 16 cores, once off-core, our approach scales better. Our approach is initially slower as the benchmark implementation uses a raw array of integers to store the graph, and as such does not have the overhead of a generic library.

As shown in Figure 8, and in Figure 9 on up to $16,384$ cores, our approach outperforms other libraries in distributed memory. PBGL uses ghost-vertices, which limits its scalability as the ghost cells need to be kept synchronized. GraphLab uses a pull-model, where each vertex synchronously reads the values from its neighbors, which does not allow deferred updates as in our asynchronous push model.

Finally, our approach is also seen to compare favorably with shared-memory libraries. Green-Marl [11] is a domain specific language (DSL) for graph computations in which the DSL compiler is able to avoid the overhead of a generic library and is around $1.5 - 2\times$ faster than our approach in shared memory. However, as the provided implementation is limited to shared-memory only, it cannot run beyond 16 cores (1 node) on our Cray XE6m. Galois [10] is another shared-memory graph library, that has performance comparable to our approach.

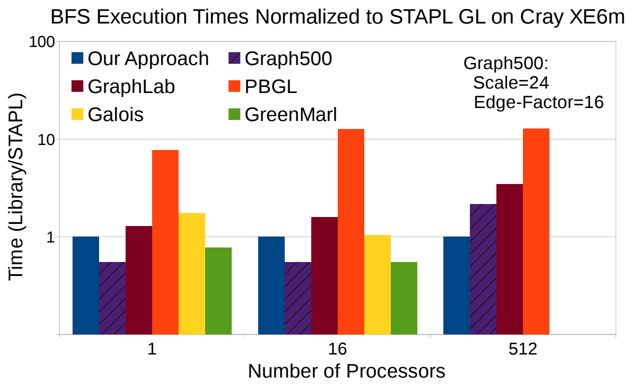


Figure 8. Execution times of Graph500 on various graph libraries normalized to STAPL GL on CRAY XE6m. Shared-memory graph libraries (Galois, GreenMarl) are shown to 16 cores.

B. *Out-of-Core Performance*

Single Node Performance. We compared STAPL GL's single node out-of-core performance with GraphChi, a graph-processing system designed to solve large graph problems on a single node efficiently. GraphChi is a sister project of GraphLab that focuses on allowing the processing of large graphs on a single-node computer using a parallel sliding-window technique. In [12], the authors show how GraphChi has comparable performance, even on a single node, to existing frameworks, such as Spark, Hadoop and GraphLab running on much larger machines.

GraphChi partitions the input graph into p partitions and processes the partitions in a sliding-window. However, much like in GraphLab, GraphChi's pull-model reads neighboring vertices' values, which may require $p - 1$ reads for p partitions in the worst-case. Furthermore, as noted in [12], the technique used in GraphChi is unable to take full advantage of available RAM, as their model is not adaptive to the resources.

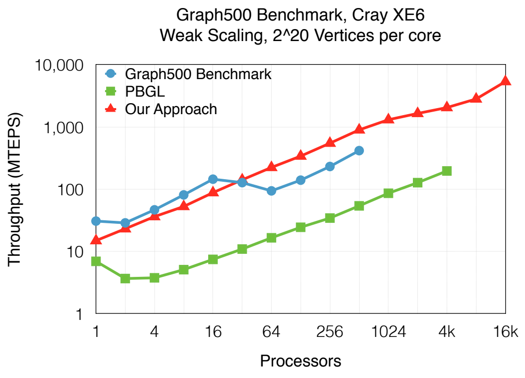


Figure 9. Weak scaling of STAPL GL on the Graph500 benchmark input on CRAY XE6 with 2^{20} vertices per core. Y-axis shows throughput in Mega-Traversed Edges per Second (MTEPS) in log-scale.

Figure 10(a) shows the out-of-core performance of various graph algorithms in the STAPL GL and GraphChi on a 4-core PC with 4GB RAM. For GraphChi, we ran the experiments on different configurations and explored the space of parameters to choose the fastest configuration for comparison. As can be observed, STAPL GL's push-model provides $2.5 - 6\times$ faster performance than GraphChi's pull-model. Further, our approach allows better scalability with increasing RAM sizes. Figure 10(b) shows the same graph on a 4-core PC with 16GB RAM. Increasing the RAM and resources allows a $4\times$ or better performance increase for the STAPL GL. However, GraphChi is not able to optimally utilize all available RAM, and therefore, STAPL GL is able to perform $4 - 12\times$ better with increased resources. GraphChi's k-core algorithm was designed to work with matrix inputs, and did not accept edge-list or adjacency-list representations, so thus we were unable to obtain comparative results for it. We evaluated our performance on two real-world graphs. Figure 14 shows performance of the STAPL GL on Twitter and Friendster social-networks on a PC with 4GB RAM.

We studied the behavior of our approach with respect to memory size in Figure 11 for breadth-first search and PageRank on the Graph500 graph. As shown, total time to completion decreases as we increase memory size. In fact, at the largest memory size, where the graph fits in memory, our approach performs the same as the in-memory STAPL GL variant. When the available memory is restricted, a majority of the subgraphs are present on disk. Therefore, a higher number of updates to vertex values generated during computation have to be written to disk, increasing the load and store times. As the available memory increases, larger portions of the graph can fit in memory, allowing more updates to happen in RAM. When the entire graph fits in memory, all updates happen in RAM, resulting in negligible overhead due to loading and storing. We observe this behavior for PageRank in Figure 11(b), where there is a substantial improvement in runtime from 4 GB to 8 GB, which is significantly larger than the benefit in BFS (Figure 11(a)). This is due to a larger volume of updates generated by PageRank, and thus written to disk for the out-of-core case, compared to BFS's fewer updates from only a subset of the graph's vertices per superstep, as shown in Figure 5. Due to this, the amount by which an increase in memory improves performance is both input and algorithm dependent.

Android Tablet Performance. To further test our approach with limited RAM, we also tested on an Android tablet with 1GB RAM, using two cores. We use this as an extreme example to test the performance of our method, and not necessarily as a viable graph-processing system. Even so, we were able to process the Graph500 input graph using our approach, as seen in Figure 12. The time taken to execute the algorithms was much higher than on a PC, due to the different architecture of the processor, lower clock-speed, lower RAM and far slower disk speed (the storage used was a low power MMC card). The size of the input graph was also limited due to the capacity of the memory-cards available. However, this experiment demonstrates that our approach can scale well to systems with limited memory as compared to the size of the input graph. We were unable to get GraphChi or other graph processing systems to run on our Android tablet.

Multi-Node and Cross-Platform Performance. As we use asynchronous forwarding and each subgraph is managed individually only by its home location, the technique naturally extends to multiple nodes. Figure 13 shows the execution of multiple algorithms on a distributed-memory system with a graph that does not fit fully in RAM. As we increase the number of nodes available, larger portions of the graph can fit in RAM and more processors are available, allowing faster execution. When the graph fits completely in RAM (on 8 nodes), no penalty is paid for disk access, and the fastest execution time can be observed.

We also tested our approach across platforms to demonstrate its effective use of resources. Figure 15 shows the Graph500 input graph scaling from an Android tablet to a Cray XE6m machine with 128 cores. Figure 16 shows the Twitter input graph on a PC, a cluster of 4 compute nodes, and a Cray XE6m machine on 512 cores. Our framework is able to accommodate processing these large graphs, while adapting well to the resources available on each machine. In this sense, the program is able to scale with the machine without user-code modification.

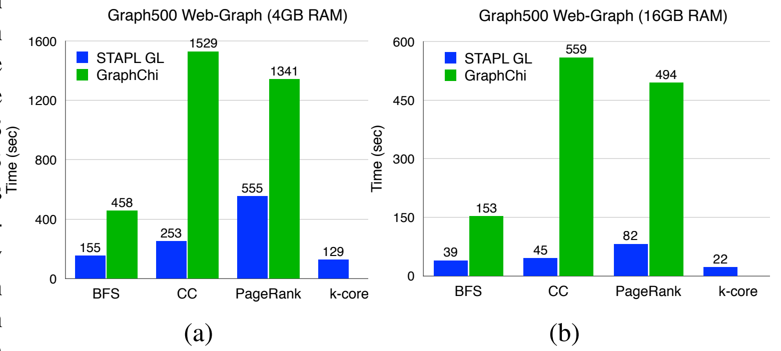


Figure 10. STAPL GL running time (various algorithms) vs. GraphChi on the Graph500 benchmark input with 16 million vertices, 256 million edges, running on PC with (a) 4GB RAM and (b) 16GB RAM.

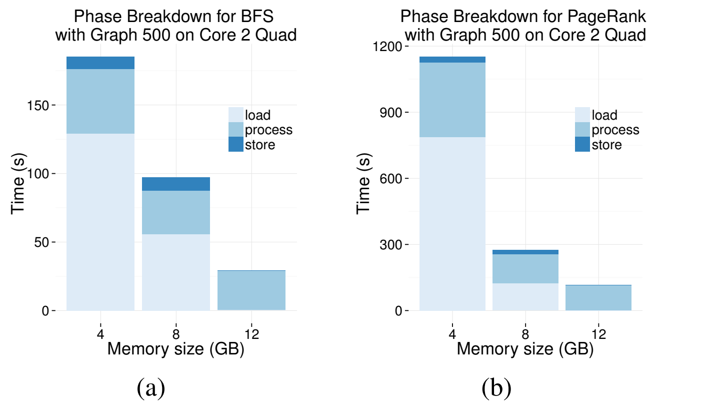


Figure 11. Breakdown of times for different phases (loading, storing and processing) for (a) BFS and (b) PageRank with respect to memory size.

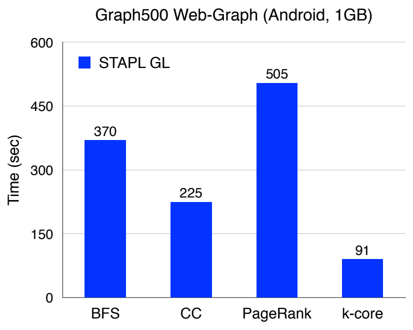


Figure 12. STAPL GL running time (various algorithms) on the Graph500 benchmark input on an Android tablet with 1GB RAM, 4 million vertices, 64 million edges.

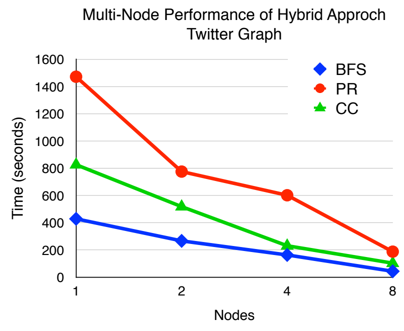


Figure 13. STAPL GL running time for BFS, PageRank (PR) and connected components (CC) on the Twitter input on multiple nodes of Cray XE6.

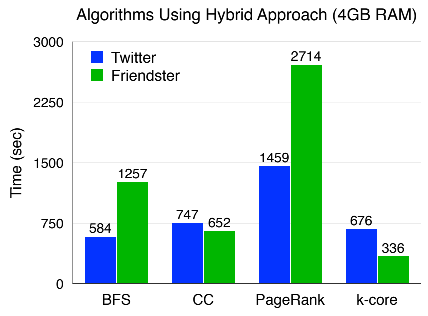


Figure 14. STAPL GL running time (various algorithms) on 4GB PC on the Twitter graph with 65 million vertices, 1.2 billion edges, and the Friendster graph with 118 million vertices, 2.6 billion edges.

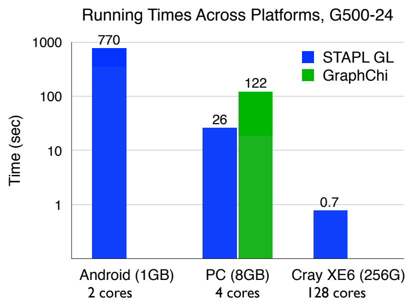


Figure 15. STAPL GL running time across platforms on the Graph500 input graph with 16 million vertices, 256 million edges.

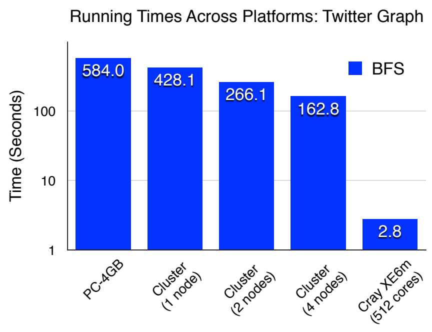


Figure 16. STAPL GL running time across platforms on the Twitter input graph with 65 million vertices, 1.2 billion edges.

VII. Related Work

In this section, we discuss some of the important existing graph processing systems, as well as theoretical work on processing out-of-core graphs.

In-memory graph libraries allow fast processing of graphs in parallel. These can be classified into two categories: distributed-memory and shared-memory. Shared-memory graph libraries such as Galois [10] and Multi-Threaded Graph Library (MTGL) [2] operate on a single node and are therefore restricted by the amount of RAM on the node, which severely limits the maximum size of the graph that can be processed. In contrast, distributed-memory graph libraries such as the Parallel Boost Graph Library (PBGL) [7], GraphLab [13] and Giraph can utilize multiple nodes to process graphs. However, even here, the maximum size of the graph that can be processed is limited by RAM. A detailed performance comparison of GraphLab, Giraph and other popular graph processing systems was shown in [6]. An in-memory system called GraphX [22] has been developed to allow multiple stages of graph-processing and pre- and post-processing to be addressed by a single framework. In their paper, the authors show GraphX to be slower than dedicated graph processing systems like GraphLab, but faster for the overall pipeline, which may include non-graph processing tasks. In Section VI, we compare our approach with GraphLab, PBGL, Galois and GraphChi.

Disk-based systems such as MapReduce and Hadoop can also be used to process graphs, however, the overhead of reading from and writing to disk prompted the development of more specialized graph libraries like Pregel and Giraph.

Recently, a system based on GraphLab called GraphChi [12] was developed to allow large graphs to be processed by a single-node computer by storing the graph on disk, and using a Parallel Sliding-Window approach to bring edges in memory to process them. The parallel sliding-window model and pull model implemented in GraphChi is not efficient for graph traversals, as loading the neighborhood of a single vertex requires scanning a complete memory shard. We provide a comparison of our theoretical model in Section IV-D, as well as compare our results with those of GraphChi in Section VI-B. A similar library, X-Stream [19], uses an edge-centric computational model, representing the data as a stream of edges. However, this too is limited to a single shared-memory node, and has an extra shuffle phase that takes $O(\frac{|E|}{|B|}log(P))$ extra time per iteration.

Pearce et al. [17] presented an asynchronous system for graph traversals for non-dynamic graphs on external and semi-external memory systems. However, it requires the vertex values to be in-memory (RAM), while the graph structure can be stored on disk. Due to this limitation, the realistic size of graphs is still limited, although to a much lesser extent than in the case of in-memory graph libraries.

In [15], Vitter proposes a theoretical model for external-memory graph searching using the technique of blocking. Their technique optimally uses disk blocks with replication of vertices on multiple blocks to provide fast performance for graph searching. However, due to replication, their algorithm is suited more for read-only graph algorithms, where the data for the vertex does not change. For supporting read-write algorithms, they need to update the multiple vertex replicas by bringing them in memory, the cost of which is analyzed in the paper. Their model uses a lower-level blocking and paging strategy than our subgraph-based paging. Further, their model loads the blocks on demand, while we use deferred asynchronous updates to allow better utilization of subgraphs that are already in memory.

VIII. Conclusion

We have presented a RAM-disk hybrid approach to processing large graphs that can scale from an Android tablet to off-the-shelf PCs to a large-scale distributed cluster with $16,000+$ cores. It uses an asynchronous push model in conjunction with subgraph-based paging and deferred updates to utilize available resources effectively, while decoupling algorithms from the underlying system. Our implementation extends an in-memory system (STAPL GL) to allow it to process out-of-core graphs, and shows improved performance over existing disk-based graph processing systems, as well as no penalty for execution when the system has sufficient RAM to store the entire graph in memory.

IX. Acknowledgments

We would like to thank Glen Hordemann for help with our initial design. We would also like to thank our anonymous reviewers. This research is supported in part by NSF awards CCF 0702765, CNS-0551685, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0917266, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Dept. of Energy under Contract No. DE-AC02-05CH11231.

References

[1] The graph 500 list. http://www.graph500.org, 2013.

[2] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Intl. Par. and Dist. Proc. Symp.*, 0:495, 2007.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, pp. 107–117, 1998.

[4] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: Standard Template Adaptive Parallel Library. In *Proc. Annual Haifa Experimental Systems Conference (SYSTOR)*, pp. 1–10, 2010.

[5] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. Symp. on Discr. Alg.*, SODA'95, pp. 139–149, 1995.

[6] Y. Guo, M. Biczak, A. Varbanescu, A. Iosup, C. Martella, and T Willke. How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis. In *Intl. Par. and Dist. Proc. Symp.*, pp. 395–404, 2014

[7] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing*, POOSC, 2005.

[8] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Graph Library. In *Lang. and Comp. for Par. Comp.*, Lecture Notes in Comp. Sc., pp. 46–60, 2012.

[9] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *Proc. of Intl. Conf. on Parallel Architectures and Compilation*, PACT'14, pp. 27–38, 2014.

[10] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered and unordered algorithms for parallel breadth first search. In *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques*, PACT'10, pp. 539–540, 2010.

[11] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proc. of Intl. Conf. on Architectural Support for Prog. Languages and Operating Syst.*, ASPLOS'12, pp. 349–362, 2012.

[12] A. Kyrola, G. Blelloch, and C. Guestrin. *GraphChi: Large-scale Graph Computation On just a PC*. In *Proc. of Conf. on Oper. Sys. Design and Impl.*, OSDI '12, pp. 31–46, 2012.

[13] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed Graphlab: A framework for machine learning and data mining in the cloud. *Proc. of VLDB Endowment*, pp. 716–727, 2012.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proc. Intl. Conf. on Management of data*, SIGMOD'10, pp. 135–146, 2010.

[15] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. In *Proc. Symp. on Princ. of Database Systems*, PODS'93, pp. 222–232, 1993.

[16] L. Page, S. Brin, R. Motwani and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. 1998.

[17] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proc. of Intl. Conf. for High Perf. Comp., Networking, Storage and Analysis*, SC'10, pp. 1–11, 2010.

[18] M. J. Quinn and N. Deo. Parallel graph algorithms. In *ACM Computing Surveys (CSUR)*, pp. 319–348, 1984.

[19] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-Stream: Edge-centric Graph Processing Using Streaming Partitions. In *Proc. Symp. on Oper. Sys. Princ.*, SOSP'13, pp. 472–488, 2013.

[20] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. Symp. on Princ. and Practice of Par. Prog.*, PPoPP'11, pp. 235–246, 2011.

[21] L. Valiant. Bridging model for parallel computation. *Comm. ACM*, pp. 103–111, 1990.

[22] R. S. Xin, J. E. Gonzalez, M. J. Franklin, I. Stoica. GraphX: A Resilient Distributed Graph System on Spark. In *Wkshp. on Graph Data Management Expr. and Sys.*, pp. 1–6, 2013.