Processing Big Data Graphs on Memory-Restricted Systems

Harshvardhan Nancy M. Amato Lawrence Rauchwerger

Parasol Laboratory
Department of Computer Science and Engineering
Texas A&M University
{ananvay, 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, which can only process graphs that fit completely in RAM, or disk-based techniques that sacrifice performance.

Contribution. 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.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable Libraries*

Keywords

Parallel Graph Processing, Out-of-Core Graph Algorithms; Graph Analytics; Big Data; Distributed Computing.

1. TECHNIQUES

We provide a graph-processing engine (the *graph_paradigm*) to control the execution of algorithms, which 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 decouples them from details of parallelism and exposes the maximum amount of parallelism available in the problem. As the graph algorithms are decoupled from execution policies, the execution strategies can change without changing the algorithms themselves.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.
ACM 978-1-4503-2809-8/14/08.
http://dx.doi.org/10.1145/2628071.2671429 .

Our approach for out-of-core processing is similar to *paging*, but applied to subgraphs. 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. This is combined with an "asynchronous push model", where vertices of a subgraph are 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, where as paging uses fixed-size pages at a non-semantic (kilobyte, megabyte) level, our approach operates at a logical (subgraph) level, using partitions based on graph structure to take advantage of temporal locality in subgraphs and reduce disk I/O.

Our *asynchronous push model* 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. These optimizations include caching hub-vertices with 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.

Using the I/O model described by Vitter [5, 1] our approach uses a linear (in number of edges of the graph) number of block transfers from disk, which is optimal as all edges need to be accessed in the worst case.

An implementation of our approach in the STAPL Graph Library [2] 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 with $16,000+$ cores. 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 [4], and extends efficiently to multiple nodes which GraphChi cannot.

1.1 Implementation

We implemented our approach in the STAPL framework, by extending the STAPL Graph Library (STAPL GL) [2], a distributed-memory graph library that previously supported in-memory graph computations only, to support out-of-core processing. The API of 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.*

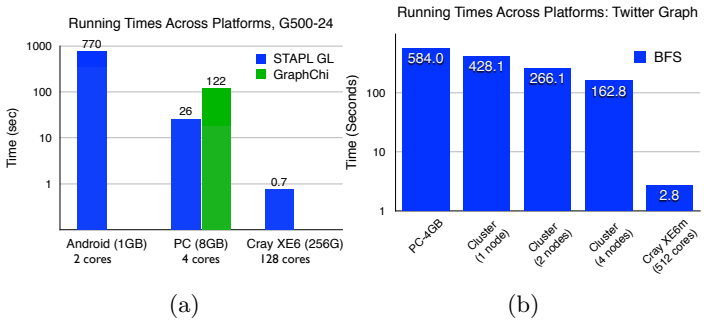
The graph structure was stored in the `pGraph`, a distributed container. A two-level distributed-directory asynchronously forwards updates to vertices in the graph, providing a shared-object view of the entire graph. We extend this concept to allow storing the subgraphs on each processor on disk, such that only a subset of subgraphs that can fit on RAM at any given time are active. Updates sent to a vertex are asynchronously forwarded to its home-location (processor responsible for storing the vertex). Updates to vertices loaded in RAM are immediately applied, while to those stored on disk are written to disk and applied when the corresponding subgraph is loaded. The distributed system allows us to scale to large systems, allowing each shared-memory node to manage how much of its local graph is stored in RAM vs. disk. If the machine has sufficient RAM, no penalty is paid for disk access.

STAPL GL's *KLA graph paradigm* [3] unifies the BSP and asynchronous computational models and allows graph algorithms to be expressed in a vertex-centric manner by using the *KLA two-operator algorithmic specification* [3], which implements an asynchronous push model, allowing us to take advantage of deferred paging. To express an algorithm, the user provides two operators – a *vertex-operator* which performs the computation of the algorithm on a single vertex and a *neighbor-operator* that updates the neighbor-vertices of the source vertex with the results of the computation. The KLA paradigm decouples algorithm writers from underlying parallelism, implementation, and execution details. We extend the paradigm to support disk-based computing. The KLA paradigm proceeds in supersteps, where each superstep processes active vertices in the algorithm, allowing asynchronous execution of up to k levels in each superstep. In our framework, each superstep loads the needed subgraphs on each location, processes its vertices and applies updates to neighboring vertices as previously described. When a subgraph is loaded, all pending updates are applied to it. Using asynchrony in our algorithm allows us to hide some of the latency associated with disk-access.

2. RESULTS

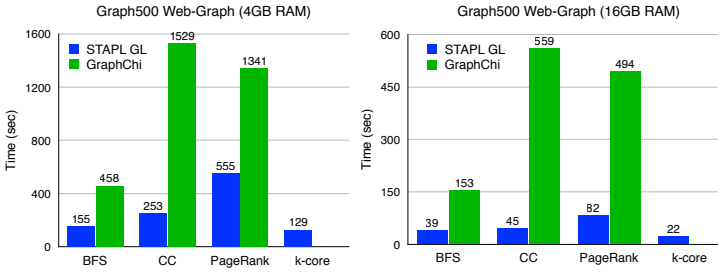
Figure 1(a) shows our running time on multiple platforms, including a 2-core tablet with 1GB RAM, a 4-core PC with 8GB RAM, and a CRAY XE6 supercomputer. We also show comparative times with GraphChi on the PC, though we were unable to run it on our tablet or the Cray XE6. We further show comparative running times with GraphChi on various graph-mining and graph-analytics algorithms on desktop PCs with 4GB and 16GB RAM in Figure 2. Both results show the scalability of our approach across platforms and the ability to best utilize available resources (e.g., cores, RAM). Further, our asynchronous push model with deferred paging requires fewer disk accesses and can better utilize available RAM than GraphChi's pull model and parallel sliding-window approach, allowing it to process graphs $3-12\times$ faster than GraphChi.

Our approach also naturally extends to distributed-memory machines, as shown in Figure 1(b) on the Twitter social-network graph. At 512 cores on a Cray XE6m machine, the entire Twitter graph can fit in RAM, and therefore our approach pays no penalty for disk-access. This can scale to $16,000+$ cores on a Cray XE6, as shown in [2].



(a) (b)

Figure 1: STAPL GL running time across platforms on (a) Graph500 input graph with 16 million vertices, 256 million edges, and (b) Twitter input graph with 65 million vertices, 1.2 billion edges.



(a) (b)

Figure 2: 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.

3. REFERENCES

[1] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, pp. 139–149, USA, 1995.

[2] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Graph Library. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp 46–60, Springer, 2012.

[3] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*, PACT '14, pp. 27–38, USA, 2014. ACM.

[4] A. Kyrola, G. Blelloch, and C. Guestrin. *GraphChi: Large-scale Graph Computation On just a PC.* In *Proc. of USENIX conf. on Operating Systems Design and Implementation*, OSDI '12, pp. 31–46.

[5] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pp. 222–232, USA, 1993.