GraphChi: Large-Scale Graph Computation on Just a PC

Author(s): Aapo Kyrola, Guy Blelloch, Carlos Guestrin

This paper is an effort at democratizing large-graph processing by developing an algorithm that allows for sophisticated graph computing for graphs on the scale of modern internet graphs to be processed on a single, memory-limited (DRAM) computer instead of being dependent on large, expensive distributed networks.

At the core of the authors' solution, which they call the Parallel Sliding Windows method for processing large graphs from disk (SSD, hard drive), lies a gripe with the bulk-synchronous parallel (BSP) model used in (at the time, the most cutting-edge) vertex-centric graph processing systems such as Pregel and GraphLab (which we discussed in the Tuesday class). The BSP method is subject to bottlenecks and costly synchronization steps that may prevent it from converging at all, issues that could be avoided by using asynchronous methods that allow each update function to use the *most recent* version of the edge and vertex values rather than those from the *previous state*. These asynchronous methods open up a world of possibilities, including selective updating of vertices.

The first issue the authors tackle is one of data representation. CSR (Compressed Sparse Row) format allows for rapid access of out-neighbors, and CSC (Compressed Sparse Column, the CSR of the transposed graph) allows for rapid access of in-neighbors. When vertex values update, however, random-read and random-write are required to ensure alignment between the vertex value and that which its neighbors can access. This is not an easy problem, since real-world graphs are highly interconnected. One way of getting around this would be by simply using SSD, which, as a dedicated heap space, would support faster random-read and -write than a hard disk. Unfortunately its capacity is likely to be insufficient for storing the graph. Other solutions, like graph compression, are effective but less useful when there is data associated with vertices and edges.

The updating solution presented in this paper is neither to "simply have more/faster RAM" nor to compress the graph, but rather to process computations and update the graph by partition. The structure of each partition is as follows: the vertices V of graph $G = (V, E)$ are split into P disjoint intervals. The edges of the graph E are subdivided amongst these intervals according to their destination vertex, and ordered by their source vertex (e.g. if vertex n is in interval p, edge (m, n) will be allocated to a shard in p, and be ranked behind edge (l, n), which precedes it). Each interval constitutes a subgraph of G which can be fully loaded into memory. Thanks to the sorting, the source vertices of adjacent edges (say, (m, n) and (l, n)) will often be adjacent in neighboring intervals, so $P - 1$ block reads are required to fully update the graph given changes in p, or, alternately, P reads to fully process interval p. This significantly reduces the random-reads and random-writes to disk. It also allows for clever sequencing of the operations (e.g. if we know that m and l are in the same interval, PSW will process them in sequence to avoid race conditions, but k, in a different interval, may be updated in parallel).

Dynamic graphs with updating edges are also supported in a memory-access efficient way using edge-buffers. The edge buffers are written to disk once the buffer is full, and if writing the buffer overfills a shard beyond the local memory limit (a requirement for any shard), the shard is split in two.

For a full iteration of PSW, the number of non-sequential disk seeks is $O(P^2)$ as opposed to a possible $O(V^2)$ (where $V >> P$, since if the graph is fully connected, we would require V writes per vertex operation).

Among the implementation details, which are numerous, a few that interested me were:

- The time-intensity of dynamic allocation v.s. predesignating memory to an array (something I've thought about in the past but isn't as natural to encode in in Julia as it is in C++, but that I have increasingly being more mindful of).

- Building and maintaining auxiliary data structures as part of a pre-processing and dynamic-updating procedure.
- Selective scheduling that not only allows some parts of the graph to compute ahead of others, but to organize PSW as a whole. I don't think I fully understand the asynchronous scheduling mechanism, but it seems to me like it would limit the types of iterative calculations to ones that *do* work in a vertex-centric model (i.e. a vertex depends on neighbors to update, so maybe it only gets scheduled once the neighboring vertices report that they are done with their tasks? What prevents one side of the graph from completely running away with the computation and processing too fast relative to another part of the graph?)
- This hypothesis seems at least partially ratified by the types of algorithms the authors present as examples of PSW implementation.
- (My mom also had a Mac Mini running MACOS Lion in 2012, RIP little computer you stored so many CFA flashcards).

I think by modern expectations of processing times the reported values are not particularly impressive, but it is genuinely remarkable that the authors could get a runtime in the same order of magnitude as distributed graph processing algorithms like GraphLab on an 8GB-main-memory local machine. Clearly, the goal of democratizing large-graph processing is accomplished using GraphChi.

Some of the weaknesses the paper admits includes inefficiency for key algorithms such as graph traversals (BFS, etc.) since the shard-based access makes it necessary to load the entire shard and scan across it to find the out-neighbors of a given vertex (i.e. limited priority scheduling). Additionally, one may not want each shards to be written and read to disk every time it is processed if one has sufficient memory to load multiple shards, and the time complexity, while far better than it could be, is still limited by the size of P (which has to be pretty large to fit big graphs into memory). I mentioned my questions about the types of algorithms this method can process earlier (limitation to iterative algorithms). It would also have been nice to see experiments on other machines (e.g. a laptop) to see how performance drops off with increased P etc.