X-Stream: Edge Centric Graph Processing using Streaming Partitions

Amitabha Roy, Ivo Mihailovic, Willy Zwaenepoel
Background

- System for processing large graphs on a single machine
- Aimed at scaling the “scatter-gather” programming model for graph algorithms
- Current approaches to “scatter-gather” iterate over vertices, and consist of:
  - Scatter: a user-provided scatter function propagates vertex state to all its neighbours
  - Gather: accumulates updates from all neighbors to update the current vertex state
  - Current approaches sort edges by originating vertex, and then use random access across an index of vertices to locate relevant edges connected to a vertex
Main Contributions

- Adapts “scatter-gather” model of state propagation across graphs to be edge-centric
- Takes advantages of large speedups from sequential vs. random memory access
  - ~500x for disk, 30x for SSD, 1.6-5x for main memory on tested system
  - Much larger speedup for “slow storage”, which is where large edge lists typically need to reside
- Rather than random access across vertices, X-Stream processes all edges sequentially, only propagating state where needed
- Streaming partitions are used to reduce random access overhead across vertices
- For typical graphs, number of edges >> number of vertices, and processing these dominates scatter-gather runtime, so sequential processing is useful optimization

```
edge_scatter(edge e)
    send update over e

update_gather(update u)
    apply update u to u.destination

while not done
    for all edges e
        edge_scatter(e)
    for all updates u
        update_gather(u)
```

**Figure 2: Edge-centric Scatter-Gather**

**Figure 3: Streaming Memory Access**
System Architecture

- X-Stream runs by processing a set of “streaming partitions”, which consist of: (vertex set, edges with source vertices in set, relevant updates)

- Each streaming partition is sized so that all vertices fit in “fast memory”, and to ensure enough I/O capacity available for full utilization of streaming bandwidth

- Trade off: many partitions destroys sequential access

- Large graphs are handled by an “out-of-core” streaming engine (Figure 6) – architecture allows this to be “stacked” with in-memory engine

- X-Stream parallelizes work across streaming partitions – constrained by I/O resources available by CPU

- Load balancing achieved by “work-stealing”
Benchmarks

- The “scatter-gather” framework is flexible enough to express a wide range of different graph algorithms, e.g. connected components, shortest paths, spanning trees, etc.

- X-Stream outperforms where sorting-based pre-processing is required (figure 18)

- Some graphs (e.g. Dimacs, figure 12) perform poorly as only very few edges need updates. Yahoo graph also failed to compute in reasonable time for many algorithms.

- X-Stream performs poorly for graphs with large diameter, which need a larger number of edge-centric scatter-gather iterations without much work.

![Figure 12: Different Algorithms on Real World Graphs: (a) Runtimes; (b) Number of scatter-gather iterations, ratio of runtime to streaming time, and percentage of wasted edges for WCC.]

![Figure 18: Sorting vs. Streaming]
Pros/Cons

• Pros:
  • Sequential processing of edges takes advantage of tremendous speed-ups available, and these increase for “slow memory”, and so scale with graph size
  • Streaming partitions allow parallelism and scaling with out-of-core streams gives X-Stream significant scalability and power
  • X-Stream avoids slowdowns from heavy pre-processing and other manipulations used by related work, e.g. the sharding process from Graphchi [3]

• Cons:
  • Overall processing is bounded by I/O capability of single-shared-memory-machine, e.g. paper could only scale to 16/32 cores due to bandwidth limitations
  • Optimality is dependent on architecture of the graph – high diameter graphs = bad
  • Optimality is also dependent on hardware – difficult to predict for cloud-based compute, and speedups might not be possible if too many partitions are needed
Citations

