X-Stream: Edge Centric Graph Processing using Streaming Partitions

Amitabha Roy, Ivo Mihailovic, Willy Zwaenepoel

Division name appears here

Background

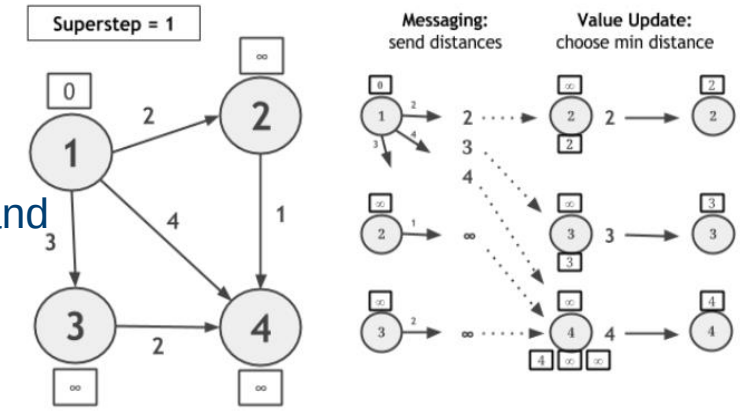
- System for processing large graphs on a single machine
- Aimed at scaling the “scatter-gather” programming model for graph algorithms
- Current approaches to “scatter-gather” iterate over vertices, and consist of:
  - Scatter: a user-provided scatter function propagates vertex state to all its neighbours
  - 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

```
vertex_scatter(vertex v)
  send updates over outgoing edges of v

vertex_gather(vertex v)
  apply updates from inbound edges of v

while not done
  for all vertices v that need to scatter updates
    vertex_scatter(v)
  for all vertices v that have updates
    vertex_gather(v)
```

Figure 1: Vertex-centric Scatter-Gather



From [2]: Iterative Graph Processing

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

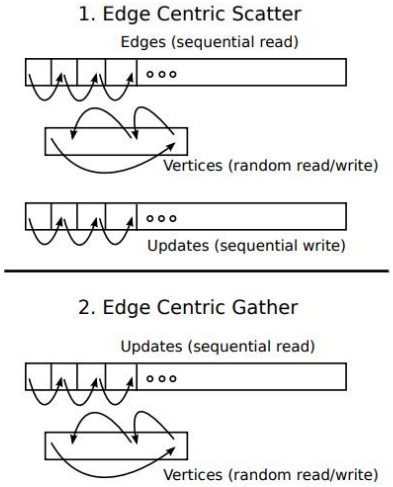


Figure 3: Streaming Memory Access

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”

```
scatter phase:
  for each streaming_partition p
    read in vertex set of p
    for each edge e in edge list of p
      edge_scatter(e): append update to Uout

shuffle phase:
  for each update u in Uout
    let p = partition containing target of u
    append u to Uin(p)
  destroy Uout

gather phase:
  for each streaming_partition p
    read in vertex set of p
    for each update u in Uin(p)
      edge_gather(u)
    destroy Uin(p)
```

Figure 4: Edge-Centric Scatter-Gather with Streaming Partitions

```
  for each streaming partition s
    while edges left in s
      load next chunk of edges into input buffer
      for each edge e in memory
        edge_scatter(e) appending to output buffer
        if output buffer is full or no more edges
          in-memory shuffle output buffer
          for each streaming partition p
            append chunk p to update file for p

gather phase:
  for each streaming_partition p
    read in vertex set of p
    while updates left in p
      load next chunk of updates into input buffer
      for each update u in input buffer
        edge_gather(u)
    write vertex set of p
```

Figure 6: Disk Streaming Loop

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 (eg. 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

	WCC	SCC	SSSP	MCST	MIS	Cond.	SpMV	Pagerank	BP
memory									
amazon0601	0.61s	1.12s	0.83s	0.37s	3.31s	0.07s	0.09s	0.25s	1.38s
cit-Patents	2.98s	0.69s	0.29s	2.35s	3.72s	0.19s	0.19s	0.74s	6.32s
soc-livejournal	7.22s	11.12s	9.60s	7.66s	15.54s	0.78s	0.74s	2.90s	1m 21s
dimacs-usa	**6m 12s**	**9m 54s**	**38m 32s**	4.68s	9.60s	0.26s	0.65s	2.58s	12.01s
ssd									
Friendster	38m 38s	1h 8m 12s	1h 57m 52s	19m 13s	1h 16m 29s	2m 3s	3m 41s	15m 31s	52m 24s
sk-2005	44m 3s	1h 56m 58s	2h 13m 5s	19m 30s	3h 21m 18s	2m 14s	1m 59s	8m 9s	56m 29s
Twitter	19m 19s	35m 23s	32m 25s	10m 17s	47m 43s	1m 40s	1m 29s	6m 12s	42m 52s
disk									
Friendster	1h 17m 18s	2h 29m 39s	3h 53m 44s	43m 19s	2h 39m 16s	4m 25s	7m 42s	32m 16s	1h 57m 36s
sk-2005	1h 30m 3s	4h 40m 49s	4h 41m 26s	39m 12s	7h 1m 21s	4m 45s	4m 12s	17m 22s	2h 24m 28s
Twitter	39m 47s	1h 39m 9s	1h 10m 12s	29m 8s	1h 42m 14s	3m 38s	3m 13s	13m 21s	2h 8m 13s
yahoo-web	—	—	—	—	—	16m 32s	14m 40s	1h 21m 14s	8h 2m 58s

(a)

	# iters	ratio	wasted %
memory			
amazon0601	19	2.58	63
cit-Patents	21	2.20	50
soc-livejournal	13	2.13	57
dimacs-usa	**6263**	1.94	**98**
ssd			
Friendster	24	1.06	63
sk-2005	25	1.04	67
Twitter	16	1.04	55
disk			
Friendster	24	1.04	63
sk-2005	25	1.04	67
Twitter	16	1.04	55
yahoo-web	—	—	—

(b)

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.

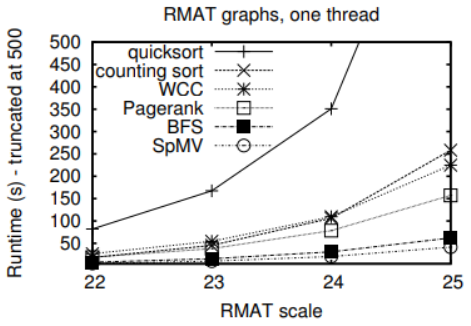


Figure 18: Sorting vs. Streaming

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

- 1.

[1] Roy, A., Mihailovic, I., Zwaenepoel, W., 2013. X-Stream: Edge-Centric Graph Processing Using Streaming Partitions, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13. Association for Computing Machinery, New York, NY, USA, pp. 472–488. https://doi.org/10.1145/2517349.2522740

[2] "Iterative Graph Processing." Flink. [Online]. Available: https://nightlies.apache.org/flink/flink-docs-release-1.15/docs/try-flink/datastream/

[3] Kyrola, A., Blelloch, G., Guestrin, C., 2012. GraphChi: Large-Scale Graph Computation on Just a PC, in: 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12). USENIX Association, Hollywood, CA, pp. 31–46.