Report

X-Stream Edge-centric Graph processing

Yassin Hassan hassany@student.ethz.ch

Abstract. X-Stream is an edge-centric graph processing system, which provides an API for scatter gather algorithms. The system supports in-memory graphs as well as out-of-core graphs. Existing scatter gather system are implemented vertex-centric, they iterate through the list of vertices while accessing the related edges using an index (random access). X-Stream introduces a new edge-centric implementation approach by reading the list of edges in a sequential order and random accessing the vertices. The edge-centric approach does not need a presorted edge list, this reduces the preprocessing overhead. The design decision is motivated by the fact that in most graphs the number of edges is larger than the number of vertices and the sequential access bandwidth is much higher than the random bandwidth for all storage media.

This report is based on the paper "X-Stream: Edge-centric Graph Processing using Streaming Partitions".

1 Introduction

Graphs can be used to model a large variety of real world problems and information. For example:

- Links between websites
- Network top
- Road maps
- Financial transactions
- Social networks

Using graph algorithms answers to questions like "What is the shortest path between two cities?" can be extracted from these graphs. Such graphs can have billions of vertices and edges. The processing of graphs at this scale exceed the computational power of a single processor core. In order to overcome this limit graph processing has to be paralleized. Parallizing graph processing algorithms is challenging, because graphs represent irregular relations between vertices, any vertex could be connected with and other vertex. Based on the graph topology it may difficult to partition the graph processing in to smaller disjunct sub tasks with a high memory locality.

Implementing a specific graph algorithm one can choose between two options: create a custom parallel implementation for the specific algorithm or use

a general-purpose graph processing system like X-Stream, Pregel or GraphChi. These systems allow the user to implement a vertex-program which is executed in parallel on each vertex, the program can interact with the adjacent vertices using a messaging channel or a shared state. The high-level framework handles and resolves the resulting synchronisation and message passing issues.

1.1 Pregel

Pergel is graph processing framework introduced by Google in 2010. Pregel processes a graph by running a series of so called super-steps on the graph. In each super-step Pregel executes the vertex program with the incoming message from the previous iteration as a input parameter. If there are multiple incoming messages, the messages are combined in to one message using a combiner. The vertex program can modify the state of the current vertex or that of its outgoing edges and send messages to any other vertexes. This vertex-centric approach is similar to the map-reduce approach which processes each item independently. This model is well suited for parallel execution, since each vertex can be processed independently from the other vertices.

The computation is stopped when all the vertexes have reached an inactive state. Each vertex can change its state to inactive by calling a vote function. If a message is sent to an inactive vertex, the vertex will be reactivated. [3]

Listing 1.1: Shortest path vertex programm

```
class ShortestPathVertex
: public Vertex<int, int, int> {
    void Compute(MessageIterator* msgs) {
        int mindist = IsSource(vertex_id()) ? 0 : INF;
        for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
        if (mindist < GetValue()) {
            *MutableValue() = mindist;
            OutEdgeIterator iter = GetOutEdgeIterator();
            for (; !iter.Done(); iter.Next())
                SendMessageTo(iter.Target(), mindist + iter.GetValue());
        }
        VoteToHalt();
    }
};
```

Listing 1.2: Combiner which selects the minimal message value

```
class MinIntCombiner : public Combiner<int> {
    virtual void Combine(MessageIterator* msgs) {
        int mindist = INF;
        for (; !msgs->Done(); msgs->Next())
            mindist = min(mindist, msgs->Value());
        Output("combined_source", mindist);
    }
};
```

1.2 GraphLab

GraphLab executes the vertex program in parallel with a context as parameter. The vertex program then can access the adjacent vertexes using the scope and

update the own vertex data, the message passing appends implicitly. In contrast to Pregel, GraphLab uses an asynchronous computational model. The Graphlab model combines the combinator and the vertex-program in to one single step. [1].

Listing 1.3: GraphLab PageRank

```
void GraphLabPageRank(Scope scope) :
        float accum = 0;
        foreach (nbr in scope.in_nbrs):
                accum += nbr.val / nbr.nout_nbrs();
                vertex.val = 0.15 + 0.85 * accum;
```

1.3 X-Stream

Conceptually booth approaches (GraphLab and Pregel) can be abstracted to the GAS model introduced by powergraph. The GAS consists of three conceptual phases of a vertex program: Gather, Apply and Scatter. [1] X-Stream provides a similar programming interface similar but it consists of only two phases the scatter and the gather phase. An other big difference to the Powergraph model is that X-Stream is edge centric.

Scatter
: The scatter method takes an edge as input, based on the edge and the source vertex, the function return the value which will be propagated to the destination vertex.

Gather
: The gather method takes an Update message as input and based on this value, it recomputes the state for the destination vaertex.

X-Stream guarantees that the update from the last gather phase are visible before the next scatter function starts.

1.4 Edge-centric implementation

The edge-centric model avoids random accesses on the edge list by streaming the unsorted edge list directly from the storage. This model does not provide any method to iterate over all the edges or updates belonging to a vertex. In order to reduce the slowdown caused by the random access on the vertex set, X-Stream introduces stream partitions. Stream partitions partition the vertex sets in to smaller disjoint subsets such that they fit in to the high speed memory. The edges appear in the same partition as their source vertex. By streaming the edges X-Steam exploits the fact that the sequential reads are much faster than random access, this is also true for solid state disks (SSD) and main memory. The X-Stream paper shows that the new edge-centric approach can equal or even out preform the vertex-centric approach.

```
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)
```

Fig. 1: Edge-centric Scatter-Gather

```
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
                vertex_scatter(v)
        for all vertices v
                vertex_gather(v)
```

Fig. 2: Vertex-centric Scatter-Gather

1.5 In memory data structure

In order to avoid the dynamic memory allocation overhead, the X-Stream paper proposes a stream buffer data structure. A stream buffer consists of a large byte array called chunk array and an index array with K entries, one for each streaming partitions. Each entry in the index points to the chunk associated with the streaming partition.

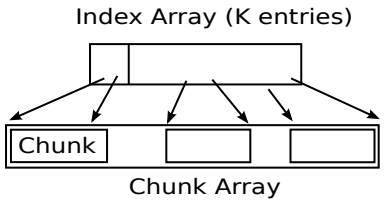

Fig. 3: Stream Buffer for K streaming partitions)

1.6 Streams

The the scatter phase takes an input stream of unsorted edges as input and produces an output stream of updates. The gather phase takes a input stream of updates and updates the destination vertices. All the accesses to the streams are sequential, this way X-Stream benefits from the higher sequentiality bandwidth. Sequential access also allows prefetching, this can hide a part of the disk latency.

1.7 Stream partitions

Each stream partition consists of three components

Vertex set
Holds all the vertices which are processed in the streaming partition.

Edge List
This list consists of all the edges with a source vertex in the vertex set.

Update List
This list holds all the incoming updates with the destination vertex is in the vertex set.

1.8 Number of stream partitions

The choice of the number of streaming partitions is performance critical. If the number is to low, the vertex set will not fit in to memory (out-of-core) or cpu cache (in-memory), this will slow down the processing since the vertexes are accessed randomly. If the chosen number is too high, the processing will not profit from the sequential disk (out-of-core) or memory (in-memory) access.

K Total space take by vertices
S Number of bytes until I/O achieves the maximum bandwidth
M Total amount of main memory required

Each chunk in the chunk array must have at least S bytes. The minimal size of the stream buffer is $S * K$. The X-Steam design requires four stream buffers for the input and output stream, two for each in order to support prefetching. I addition it needs an buffer stream for shuffling, explained later in the report. Based on these formulas the processing requires that $\frac{N}{K} + 5SK \leq M$. The minimal memory usage is reached by setting the number of streaming partitions to $K = \sqrt{\frac{N}{5S}}$.

1.9 Scatter-Gather with stream partitions

The scatter phase is executed for each stream partition separately, instead of iteration over all the edges, X-Stream iterates over the stream partition edge list and produces the update stream. The update stream needs to be rearranged such that each update is appended to the update stream in the streaming partition of the destination vertex (shuffle phase). The shuffle phase uses two buffer streams. One streaming partition is used to store the unsorted updates resulting from the scatter phase, the second buffer stream is used to store the shuffled updates. The shuffle phase first counts the updates for each destination stream partition by scanning through stream buffer holding the updates. Based on the update counts per stream the chunks in the shuffle buffer are allocated for each stream partition. After that all the updates are copied to the destination stream partition. The shuffled updates are then consumed by the gather phase.

1.10 Out-of-core streaming engine

The out-of-core streaming engine is used when the graph does not fit in to the main memory. For each streaming partition the engine creates three files, a file for the vertices, the edges and one for the updates. Before processing the graph the edge list must be preprocessed, the disk engine partitions the edge input stream to in to the streaming partitions. This is done using the in-memory shuffle.

X-Stream preforms asynchronous direct I/O to and from the streaming buffers, bypassing the operating system's paging cache. Streams are actively prefetched in order exploit the sequentiality of access. As soon the read in one input stream is completed, X-Stream starts a read in to the second stream. To keep the disk busy the writing process of an output stream buffer overlaps with the computation in the scatter phase in the other output stream buffer.

Additional optimisations

RAID
X-Stream exploits the RAID architectures by striping sequencing writes across multiple disks, such that the data can be accessed concurrently. X-Stream also exploits read and write parallelism by placing the edges file and the updates file on a different disk.

SSD
Flash memory can not be simply overwritten, the block must first be erased then a write operation can be executed. The write amplification is defined as the number of writes by the flash memory divided by the number of writes by the host. For a sequential writes this factor is equal to 1, there is no write amplication. [2]. Since on delete the entire file can be block can be marked as invalid and no data needs to be moved.
X-Stream also always truncates the files when the steams are destroyed. On most operating system this automatically translates in to a TRIM command sent to the SSD.

1.11 In-memory streaming engine

In contrast to the out-of-core streaming engine the in-memory engine must deal with a much larger number of streaming partitions. Since the CPU cache is very small. If we have 1 TB of vertex data (on a system with more than 1TB of RAM) and a 1MB CPU cache X-Stream would need more then one million streaming partition in order to ensure that the vertex data fits in to the fast storage (cache). The large number of streaming partitions cause a significant performance loss while shuffling the the data.

The parallel multi-stage shuffler does not need any synchronisation. Each thread is assigned to work on its own disjoint slice, during the shuffle stage the thread only accesses its own data. After shuffling each thread ends up with the same number of chunks, since the number of target partitions is the same across all the thread. At this point the process is synchronized and all the chunks form each partition are merged.

2 Evaluation

2.1 Setup

X-Stream was evaluated the following testbed:

CPU
AMD Opteron (6272, 2.1Ghz), dual-socket, 32-core system. The CPU uses a clustered core micro-architecture, with a pair of cores sharing the instruction cache, floating point units and L2 cache.

Main Memory
64GB

SSD
Two 200GB PCI Express SSDs arranged into a software RAID-0 configuration

Disk
Two 3 TB SATA-II magnetic disks also arranged into a software RAID-0 configuration.

Based on the measured streaming bandwidth of the testbed the authors of the paper decided to use only 16 out of the 32 available cores, since adding the other cores will increase the bandwidth by only 5%. They run one thread on one core of each pair of cluster cores. While calculating the number of partition they assumed that each core has full access to the 2MB shared L2 cache.

Algorithms which where used to evaluate X-Stream

- Weakly Connected Components (WCC).
- Strongly Connected Components (SCC), using. Requires a directed graph.
- Single-Source Shortest Paths (SSSP).
- Minimum Cost Spanning Tree (MCST) using the GHS algorithm.
- Maximal Independent Set (MIS).
- Conductance.
- SpMV: Multiply the sparse adjacency matrix of a directed graph with a vector of values, one per vertex.
- Pagerank (5 iterations).
- Alternating Least Squares (ALS) (5 iterations). Requires a bipartite graph.
- Bayesian Belief Propagation (BP) (5 iterations)

2.2 Data Sets

Synthetic Dataset
RMAT graphs are scale-free, a feature of many real-world graph. In scal-free networks the degree distribution follows the power law. The graphs used in the evaluation are generated with an average degree of 16. The RMAT graphs are generated at multiple scales, the authors call a graph with 2^n vertices and 2^{n+4} edges a graph of scale n

Realworld Dataset

Name	Vertices	Edges	Type
	In-memory		
amazon0601	403,394	3,387,388	Directed
cit-Patents	3,774,768	16,518,948	Directed
soc-livejournal	4,847,571	68,993,773	Directed
dimacs-usa	23,947,347	58,333,344	Directed
	Out-of-core		
Twitter	41.7 million	1.4 billion	Directed
Friendster	65.6 million	1.8 billion	Undir.
sk-2005	50.6 million	1.9 billion	Directed
yahoo-web	1.4 billion	6.6 billion	Directed
Netflix	0.5 million	0.1 billion	Bipartite

Fig. 4: Realworld Dataset

3 Applicability

The authors of X-Stream have show that despite the fact that X-Stream does not allow direct access to the edges associated with a vertex, a variety of graph algorithms can be expressed using the edge centric model. X-Stream performed well on all the algorithms and datasets exempts the graph traversal algorithms (WCC,SCC,MIS,MSCT and SSP) for the DIMACS and the yahoo-web graph. This is due to the fact that the DIMACS and the yahoo-web graph datasets have a much larger diameter than the other graphs. In this case X-Stream performs many scatter-gather iterations while doing very little work, a high percentage of the streamed edges do not produce any updates. In the DIMACS case 98% of the loaded edges do not produce any updates.

	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

Fig. 5: Processing time running algorithms on real world graphs using X-Stream

4 Scalability

The X-Stream author investigated the scalability to their system regarding resources and graph size.

4.1 Resources

The Figure 6 show how X-Stream scales with increasing thread count for the biggest possible in-memory graph (RMAT scale 25 wit 32M vertices and 512 edges). For all algorithm the performance improved linearly, X-Stream takes advantage of the additional memory bandwidth for each added thread, there is noticeable synchronisation overhead.

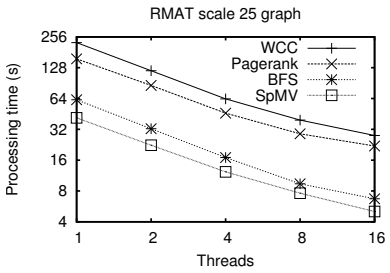

Fig. 6: Thread scaling

Figure 7 shows how X-Streams performance scales with an increasing number of I/O devices. The X-Stream author compared three different configurations:

- One disk or SSD
- Separate disks or SDDs for reading and writing
- Two disks or SSDs arranged in RAID-0 (baseline configuration)

For the SSD configuration they used RMAT scale 27 graph and RMAT scale 30 graph for the magnetic disk configuration. Putting the edges and updates on different disks reduced runtime by up to 30%, compared to using one disk. RAID-0 reduces runtime up to 50-60% compared to one disk. X-Streams sequential disk access pattern allows it to fully take advantage of additional I/O devices.

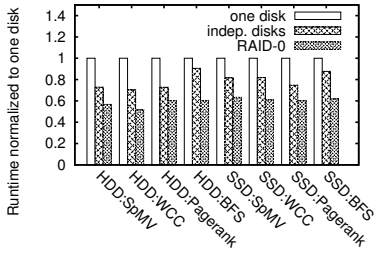

Fig. 7: I/O parallelism

4.2 Graph size

The scalability of X-Stream in terms of graph size is purely a function of the available storage. The Figure 8 shows that runtime scales linearly. The bump in the graph are due to the change from SSD to magnetic discs.

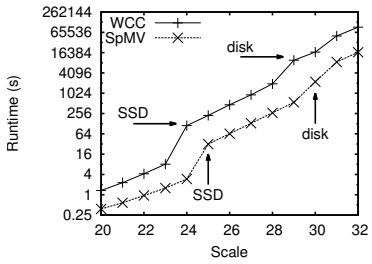

Fig. 8: Graph sizes

5 Comparison with other system

5.1 In-memory

Sorting The X-Stream engine does not need a sorted edge list, in contrast to the other in-memory graph processing engines. Before comparing the performance the X-Stream authors decided to compare the time needed to sort an edge list to the time X-Stream takes to processes the graph. X-Stream completes the actual graph processing faster than the edges sorting process, both the sorting process and X-Stream are single-threaded.

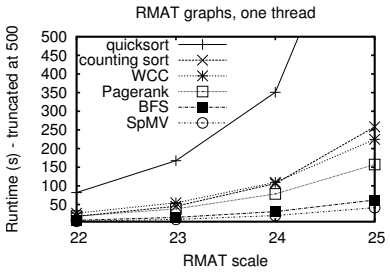

Fig. 9: Scaling Across Devices

Breath-first search The authors also compared X-Stream to optimized in-memory implementations, that performs random access through an index over the edges. The first method (local queue), uses a per core queue of vertices to be visited and heavily optimized synchronisation. The second method (hybrid), is

based on the paper "Efficient parallel graph exploration on multi-core CPU and GPU". X-Stream out performs the other methods for all thread counts (Figure 10). The runtime gap between the other method is closing with increasing thread count, because the performance gap between random access and sequential access bandwidth closes with increasing thread count (Figure 11).

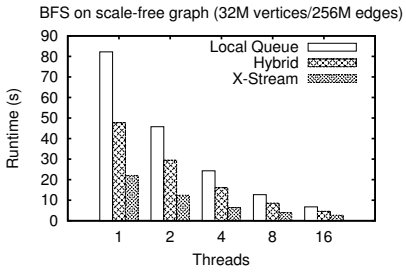

Fig. 10: In-memory BFS

Medium	Read (MB/s)		Write (MB/s)	
	Random	Sequential	Random	Sequential
RAM (1 core)	567	2605	1057	2248
RAM (16 cores)	14198	25658	10044	13384
SSD	22.5	667.69	48.6	576.5
Magnetic Disk	0.6	328	2	316.3

Fig. 11: Sequential Access vs. Random Access

Random access enables algorithm specific optimisations. Beamer et al. demonstrated that for scale-free graphs large speed ups can be obtained in the later execution stages of BFS by iterating over the set of target vertices rather than the set of source vertices.

Ligra is a in-memory processing system which implements this optimisation. Running Ligra and X-Stream on the Twitter dataset, Ligra outperforms X-Stream for BFS and Pagerank at all thread counts but this comes at a significatant pre-processing costs. Ligra requires an inverted edge list. The pre-processing dominates the Ligra overall runtime. Comparing the total runtime X-Stream is about 7x-8x faster.

Analysing the effect of the X-Stream design decisions on the instruction throughput, comparing X-Stream to other graph processing systems. X-Stream has a higher average instruction per cycle (IPC) and total number of memory access. In the higher IPC count can result from a smaller number of main memory references or from a lower latency to resolve memory addresses. The authors concluded that the lower latency to resolve the memory addresses is due to the sequential nature of the X-Stream memory accesses. Sequential accesses allow the prefetcher to hide some of the memory access latency.

5.2 Out-Of-Core

The X-Stream authors decided to compare X-Stream to Graphchi. Graphchi uses a certex-centric approach and uses a out-of-core data structure which uses parallel sliding window. This data structure reduces the number of needed random accesses.

The benchmarks use the same algorithms and datasets reported by the Graphchi paper. In each benchmark X-Stream finishes the graph processing before Graphchi finishes pro-sorting the data. Even if only the effective processing of the graph (excluding the pre-processing) is compared X-Stream is faster than Graphchi.

A significant amount of time is used by Graphchi re-sorting the edges in the shards by destination the edge, this is needed to process the updates sent through the edges. This time is reported in Figure 12 as re-sort. Graphchi also does not use all the available bandwidth provided by the SSD. Graphchi needs to fit all the edges in to memory this leads to a big number of shards. This leads to a high fragmented read write behaviour. The X-Stream scatter phase has a regular pattern alternating between burst reads from the edges file and burst write to the updates files. The gather phase has a long burst read phase without any writes (Figure 13).

	Pre-Sort (s)	Runtime (s)	Re-sort (s)
Twitter pagerank			
X-Stream (1)	none	397.57 ± 1.83	–
Graphchi (32)	752.32 ± 9.07	1175.12 ± 25.62	969.99
Netflix ALS			
X-Stream (1)	none	76.74 ± 0.16	–
Graphchi (14)	123.73 ± 4.06	138.68 ± 26.13	45.02
RMAT27 WCC			
X-Stream (1)	none	867.59 ± 2.35	–
Graphchi (24)	2149.38 ± 41.35	2823.99 ± 704.99	1727.01
Twitter belief prop.			
X-Stream (1)	none	2665.64 ± 6.90	–
Graphchi (17)	742.42 ± 13.50		

Fig. 12: Comparison with Graphchi on SSD

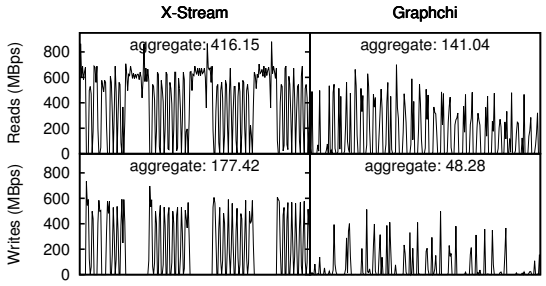

Fig. 13: Disk Bandwidth

5.3 Conclusion

Using the the theoretical I/O model, the I/O complexity can be analysed. The maximum number of edge scatters needed to propagate a message from any source vertex to all the reachable vertices in a graph G = (V, E) is the diameter of the graph (D). The I/O model uses a memory of M words backed by an infinitely sized disk from which transfers are made in aligned units of B words.

X-Stream utilizes $K = \frac{|V|}{M}$ partitions. A single iteration has

$$\frac{|V| + |E|}{B} + \frac{|U|}{B} \log_{\frac{M}{B}} K$$

I/O operations when U is the update set. All iterations have

$$D\frac{|V| + |E|}{B} + \frac{|E|}{B} \log_{\frac{M}{B}} K$$

I/O operations.

The results show that X-Stream performs well for graphs with a low diameter. However, the authors did not compare the X-Stream performance on high diameter graphs to Graphchi, three out of four algorithms chosen for the comparison had a fix number of iterations. Compared to other graph processing systems X-Stream has the great advantage that the edges do not need to be presorted. An other advantage which is not covered by the theoretical model is the nature of the X-Stream memory accesses. Even with a relatively small gap between random and sequential access on the main memory the there is a noticeable performance gain in the in-memory engine.

In conclusion the main contributions of the X-Stream paper are the introduction of an edge-centric computing model and the implementation of this processing model using streaming partitions which can be used for in-memory and out-of-core graphs.

References

1. Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.
2. XYHR Haas and X Hu. The fundamental limit of flash random write performance: Understanding, analysis and performance modelling. Technical report, IBM Research Report, 2010/3/31, 2010.
3. Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.