Scale-up Graph Processing: A Storage-centric View

Eiko Yoneki
University of Cambridge
eiko.yoneki@cl.cam.ac.uk

Amitabha Roy
EPFL
amitabha.roy@epfl.ch

ABSTRACT

The determinant of performance in scale-up graph processing on a single system is the speed at which the graph can be fetched from storage: either from disk into memory or from memory into CPU-cache. Algorithms that follow edges perform random accesses to the storage medium for the graph and this can often be the *determinant* of performance, regardless of the algorithmic complexity or runtime efficiency of the actual algorithm in use. A storage-centric viewpoint would suggest that the solution to this problem lies in recognizing that graphs represent a unique workload and therefore should be treated as such by adopting novel ways to access graph structured data. We approach this problem from two different aspects and this paper details two different efforts in this direction. One approach is specific to graphs stored on SSDs and accelerates random access using a novel prefetcher called RASP. The second approach takes a fresh look at how graphs are accessed and suggests that trading off the low cost of random access for the approach of sequentially streaming a large set of (potentially unrelated) edges can be a winning proposition under certain circumstances: leading to a system for graphs stored on any medium (main-memory, SSD or magnetic disk) called X-stream. RASP and X-stream therefore take - diametrically opposite - storage centric viewpoints of the graph processing problem. After contrasting the approaches and demonstrating the benefit of each, this paper ends with a description of planned future development of an online algorithm that selects between the two approaches, possibly providing the best of both worlds.

1. INTRODUCTION

Large scale graph processing represents an interesting systems challenge due to the problem of lack of locality. Executing algorithms that follow edges inevitably results in random access to the storage medium for the graph and this can often be the *determinant* of performance, regardless of the algorithmic complexity or runtime efficiency of the actual algorithm in use. This problem is particularly acute with scale-up graph processing: using a single system to process graphs. Scale-up graph processing using a single multicore machine with the graphs held in memory or on disk represents an attractive alternative to clusters [1] but is affected even more by the locality problem as large graphs must often be held on disk, rather than being distributed across the random access memories of a large cluster of machines with scale-out approaches [2, 3]. Figure 1 illustrates the different speeds of storage media for sequential and random access, for an example system with 32 AMD Opteron 6272 cores, a 3TB magnetic disk and one 200GB PCI Express SSD. Moving from RAM to SSDs to magnetic disks means larger and cheaper storage at the expense of much reduced performance for random access.

A storage-centric viewpoint would suggest that the solution to this problem lies in recognizing that graphs represent a unique workload and therefore should be treated as such by adopting novel ways to access graph structured data. We approach this problem from two different aspects and this paper details two different efforts in this direction. One approach is specific to graphs stored on SSDs and accelerates random access using a novel prefetcher: the (R)un(A)head (S)SD (P)refetcher or RASP [4]. The second approach takes a fresh look at how graphs are accessed and suggests that trading off the low cost of random access for the approach of sequentially streaming a large set of (potentially unrelated) edges can be a winning proposition under certain circumstances: leading to a system for graphs stored on any medium (main-memory, SSD or magnetic disk) called X-stream. RASP and X-stream therefore take - diametrically opposite - storage centric viewpoints of the graph processing problem. RASP is a system being developed at University of Cambridge, UK led by the first author; while X-stream is under development at EPFL in Switzerland led by the second author. However, they share common roots and are being developed collaboratively with the common vision of improving storage subsystem performance when processing graphs. After contrasting the approaches and demonstrating the benefit of each, this paper ends with a description of planned future development of an online algorithm that selects between the two approaches, possibly providing the best of both worlds.

2. IVEC

Designing storage related data structure for graph processing requires requires an *appropriate abstraction of graph algorithms*. This lets us ignore details of specific algorithms

Proc of the First International Workshop on Graph Data Management Experience and Systems (GRADES), June 23, 2013, New York, NY USA.

Medium	Read (MB/s)		Write (MB/s)	
	Random	Sequential	Random	Sequential
RAM	567	2605	1057	2248
SSD	22.64	355	49.16	298
Disk	0.61	174	1.27	170

Note: 64 byte cachelines, 4K blocks (disk random), 16M chunks (disk sequential)

Figure 1: Sequential access vs. Random access

and implementations, while focusing purely on the tradeoffs made on the storage front. We therefore introduce the iterative vertex-centric (IVEC) programming model, a level of abstraction that we believe satisfies this purpose. The programming model is illustrated in Figure 2. The IVEC model consists of two basic functions: the scatter function takes a vertex's state and using it generates updates to be propagated along the edges (lines 9–15). The gather function takes the updates on a vertex's incoming edges and uses them to update the state of the vertex (lines 17–24). The main loop alternately runs through an iterator over vertices that need to scatter state and one over those that need to gather state (lines 2–6). The scatter function conditionally adds to the gather iterator and the gather function conditionally adds to the scatter iterator.

The interface to a fairly large number of systems can be mapped onto the IVEC model. For example, distributed graph processing systems such as Pregel [2] and Powergraph [3] explicitly support this model. Systems for single machines such as Graphchi [1] also export - as an option - a scatter-gather model. Higher level abstractions for analyzing the semantic web can also, interestingly enough, be mapped onto this abstraction [5]. Systems that do not explicitly do scatter gather can also be mapped onto this model. For example most implementations of graph traversal algorithms such as breadth-first search (BFS) scatter updates from the BFS horizon: the currently discovered but not processed nodes. These are then gathered by receiving vertices to expand the horizon. The IVEC model therefore also describes work that focuses purely on graph traversals [6, 7, 8]. Since the IVEC model allows specification of policy (what needs to be done) but does not restrict mechanism (how to do it), it adequately abstracts many of the differences in implementations across these systems. For example, most of the aforementioned implementations of breadth-first search directly follow edges in the scatter step, access the destination vertex and add it to scatter iterator for processing. The IVEC model also applies to systems heavily focused on optimizing iterators such as Green Marl [9]. Although a limitation of the IVEC model is that it cannot capture algorithms that require asympototically more state than vertices, such as all pairs shortest paths and triangle counting, we note that it can still be used to express approximate versions of the same algorithms.

The IVEC model suffers from random access to the edges of the stored graph in lines 10 and 19. This is regardless of how the iterators are implemented and a more serious problem than access to vertex data by virtue of the fact that *edge data is usually far more voluminous than vertex data* for most graph processing workloads today.

3. RASP

The run-ahead SSD prefetcher (RASP) is targeted at implementations of the IVEC model that execute on graphs

```
while not done {                                                    1
  // Iterators                                                      2
  for all vertices v in ScatterIterator { scatter(v) }              3
  for all vertices v in GatherIterator { gather(v)  }               4
}                                                                   5
                                                                    6
//scatter step                                                      7
scatter(v)                                                          8
{                                                                   9
    for all edges e out of v {                                      10
      e.update = User_defined_scatter(v, e)                         11
      if e.update.need_gather {                                     12
        GatherIterator.add(e.dest)                                  13
}}}                                                                 14
                                                                    15
//gather step                                                       16
gather(v)                                                           17
{                                                                   18
  for all edges e into v {                                          19
    v = User_defined_gather(v, e.update)                            20
    if v.need_scatter {                                             21
      ScatterIterator.add(v)                                        22
}}}                                                                 23
```

Figure 2: Iterative vertex-centric programming

```
while not done {                                                    1
  // Iterator                                                       2
  for all vertices v in ScatterIterator { scatter(v) }              3
}                                                                   4
                                                                    5
//scatter step                                                      6
scatter(v)                                                          7
{                                                                   8
    for all edges e out of v {                                      9
      e.update = User_defined_scatter(v, e)                         10
      v = User_defined_gather(v, e.update)                          11
      if v.need_scatter {                                           12
        ScatterIterator.add(v)                                      13
}}}                                                                 14
```

Figure 3: IVEC model used in RASP

stored in solid state drives. RASP is heavily focused on graph iteration, particularly the challenging problem of graph traversals and implements a subset of the IVEC model shown in Figure 3. The key difference is that the gather phase is folded into the scatter phase, with vertices at the other end of outgoing edges being accessed directly to propagate updates. RASP addresses the problem of random access to outgoing edges at line 9 in Figure 3.

3.1 Storage

RASP is targeted at graphs stored on SSDs. SSDs are much faster than magnetic disks but are still much slower than memory as Figure 1 illustrates. It therefore makes a design decision: place vertex data entirely in main memory and leave edge data on the SSD, fetched on-demand into main memory when processing the scatter step. This is motivated again by the fact that edge data is more voluminous that vertex data and therefore storing vertex data in main memory is both feasible and eliminates the problem of random access to vertex data were it to be stored in the SSD. Figure 4 illustrates this tradeoff in RASP. The vertex data in memory includes an index that specifies, for each vertex, the location of its clustered edges on the SSD.

This is a practical tradeoff, illustrated by Figure 5 that shows the memory usage for computing weakly connected components (WCC). The majority of data resides on the SSD and a much smaller fraction of data must reside in

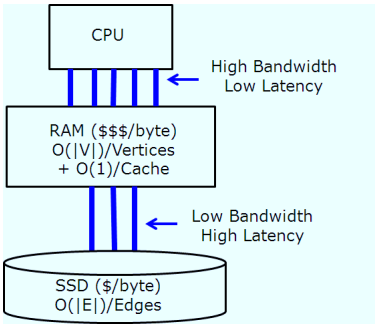

Figure 4: Graph Data Distribution in RASP

Graph	Vertices	Edges	RAM	SSD
Twitter [10]	52M	1.6B	1.18GB	8.4GB
Erdos-Reyni [11]	20M	2B	0.45GB	10.4GB
Scale-free-small [12]	32M	1.28B	1.10GB	12GB
Scale-free-large [12]	1B	40B	24GB	315GB

Figure 5: RASP Memory usage WCC

RAM.

3.2 Prefetcher

The core component of RASP, the prefetcher, is driven by the observation that an SSD can simultaneously handle a number of outstanding *random* requests without penalizing individual ones. This is very unlike traditional magnetic media where the penalty of having to move disk heads to service a stream of random requests requires queuing them and servicing them one by one. Figure 6 shows the effect of increasing the number of outstanding requests for 4K pages to the SSD (known as queue depth), measured on the same system as used for Figure 1. Using 32 outstanding requests results in a peak random access bandwidth of approximately 177 MB/s, about 8 times the random access bandwidth with one outstanding request shown in Figure 1.

Therefore, although the requests at line 9 in Figure 3 may be random, if we could know the request stream in advance prefetching from the SSD becomes possible due to its ability to service these random requests simultaneously without affecting the latency of servicing any individual request.

The second key building block (and contribution) of RASP is an elegant way to derive this request stream by looking ahead during execution of the graph algorithm. The information needed in already encapsulated in the `ScatterIterator` that is used by the system to locate the next vertex to process and therefore an *algorithm independent*

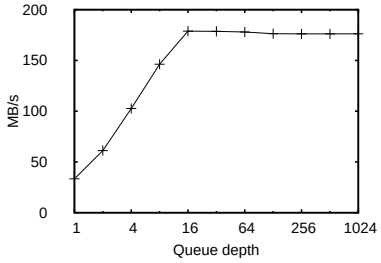

Figure 6: Increasing Random Read Bandwidth with Queue Depth

Graph	Base	RASP	MT(8)	RAM
Twitter	36.08	6.36	7.17	2.31
Erdos-Reyni	80.96	6.03	11.30	4.11
Scale-free-small	88.56	10.77	15.74	3.95
Scale-free-large	>2.5 days	402.56	>2 days	cannot fit

Figure 7: Runtime (mins) for WCC

way to derive the needed vertex stream is to look ahead in the scatter iterator to locate the set of vertices, whose edges will be required in the immediate future when processing the scatter step. The prefetcher then runs ahead of the computation and issues prefetch requests for the edges corresponding to these vertices and hence the name: (R)unahead (S)SD (P)refetcher or RASP. For example, in algorithms such as breadth-first search, the iterator is a FIFO queue that is read by the prefetcher to issue requests to the SSD, while being careful to remain within a window that does not exceed the capability of the SSD and the virtual memory page-cache. This capability is the number of outstanding requests without queuing delay (usually 32 outstanding requests, as shown in Figure 6).

3.3 Key Results

RASP greatly enhances the capability of storage subsystems such as SSDs when processing graph algorithms, effectively hiding their much larger latency than random access memory. To demonstrate this, we allocated a 2GB main memory cache for the data stored on SSD in Figure 5. The majority of data therefore continues to reside on the SSD. We then computed weakly connected components for the graphs in Figure 5, the absolute runtimes are shown in Figure 7. RASP results in a speedup varying (across graphs) from 8x-13X. Also included in the table is the running time of the baseline with multithreading (using all the available 8 cores) in the column labeled MT-8: an alternative way to work around large random access latencies with storage [8]. RASP using a single thread of computation is often more effective at working around storage bottlenecks. Finally the last column includes the running time with the entire graph loaded into memory (this is not possible for the largest graph). RASP allows the runtime to approach that of executing purely in memory, illustrating its effectiveness at hiding the random access bottleneck to the SSD.

3.4 Problems

RASP clearly provides impressive speedups for graphs processed from SSD, approaching the performance of random access memory at a fraction of the cost. This is made possibly by combating the inefficiency of random access through prefetching. There are however some aspects of RASP that stand in the way of it being a perfectly general solution to the problem of random access to edges in the IVEC model.

- RASP requires pre-processing the edges of a graph into a sorted and indexed format, the current implementation uses the compressed sparse row (CSR) format. Pre-processing is not cheap: the scale-free graph with 40 billion edges is initially generated as an unordered edge list and took approximately *6 hours* to pre-process into a sorted format.

```
while not done {                                                    1
  // Iterators (note: sequential access to edges)                   2
  for all edges e in Graph { scatter(e) }                           3
  for all edges e in GatherIterator { gather(e)  }                  4
}                                                                   5
                                                                    6
//scatter step                                                      7
scatter(e)                                                          8
{                                                                   9
  v = e.src                                                         10
  if v.need_scatter {                                               11
      e.update = User_defined_scatter(v, e)                         12
      if e.update.need_gather {                                     13
        GatherIterator.add(e)                                       14
}}                                                                  15
                                                                    16
//gather step                                                       17
gather(e)                                                           18
{                                                                   19
  v = e.dst                                                         20
  v = User_defined_gather(v, e.update)                              21
}}                                                                  22
```

Figure 8: Iterative vertex-centric programming in X-stream

- RASP is specific to SSDs. It is, by design, inapplicable to magnetic disks and to graphs stored completely in main memory. In the latter case, random access to fetch data into the CPU cache can become a problem.

4. X-STREAM

X-stream is a system that is built around the observation (supported in Figure 1) that sequential access to any medium is always faster than random access. X-stream therefore restructures the IVEC model to replace iteration over vertices with iteration over edges, shown in Figure 8. We eliminate random access to edges in the scatter phases by removing the scatter iterator and instead streaming all the edges of the graph. We retain the gather iterator, which becomes a sequential list of edges with updates instead of an iterator over vertices. Although we have changed the manner of execution of the IVEC model, we have not changed the interface exposed to the programmers, which continues to be vertex-centric scatter gather.

X-stream does not use a scatter iterator to avoid having to use random access to locate outgoing edges of vertices that have updates to send out. This can result in accesses to very many more edges than necessary in the scatter step. In return, it can operate using a purely streaming model, exploiting the available sequential bandwidth of storage in the system (Figure 1).

4.1 Data Structures

At the core of X-stream are streaming partitions. The vertices of the graph are divided into partitions, the only criterion being an equal number of vertices in each partition. There is no need manage the quality of the partitions such as minimizing the number of edges that cross them. The operation of a streaming partition in shown in Figure 9. It includes the vertex data for the partition, a list of edges with source vertex in the streaming partition and a list of updates (separated from the edges they belong to) with target vertex in the streaming partition. During the scatter phase, the list of edges is streamed and, if necessary an update is generated.

X-stream replaces random access into the edges with random access into the vertex data of the streaming partition (Figure 8 lines 11 and 21). This is motivated by the fact that

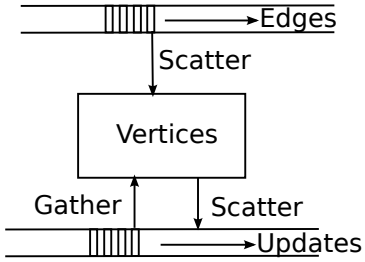

Figure 9: Streaming Partitions in X-stream

edges are usually far more numerous than vertices. Further, a key insight in X-stream is that the number of streaming partitions may be chosen such that the vertex data fits in cache (cpu cache for main memory graphs and main memory for on-disk graphs).

4.2 Key Results

X-stream is highly scalable and its basic principle applies across main-memory, SSD and disk as all three are amenable to streaming. Further, X-stream admits simple multithreading as streaming partitions can be processed independently of each other. Three properties of X-stream are of particular interest in this paper (with regard to the problems pointed out in Section 3.4).

First, X-stream does not depend on a sorted edge list, edges may be streamed in any order. This leads to the interesting consequence that X-stream can produce results from the graph faster than one can sort its edge list, as shown in Figure 14, where we compare the runtime to sort a graph in memory using quicksort as opposed to identifying weakly connected components from the unordered edge list. The single-threaded experiment uses scale-free graphs with the factor on the X-axis: a factor of n means a graph with 2^n vertices and 2^{n+4} edges.

Second, X-stream applies to all three types of storage: main-memory, SSD and magnetic disk. Figure 10 shows how X-stream scales smoothly across all three storage media on a system with 16 GB main memory, 200 GB SSD and 3 TB magnetic disk. We switch to slower and larger storage (marked by arrows) when the graph can no longer fit on the currently chosen faster but smaller storage.

Finally, X-stream does not require any part of the graph to fit in memory. The only requirement is choosing the number of partitions to be large enough to allow the vertex data of a partition to fit in memory. This requires less memory than RASP and is also competitive to the recent Graphchi system, that tries to do sequential accesses to partitions of the graph on disk, but requires the edges in any partition (far more numerous than vertices) to fit in memory. Figure 11 shows the result, a reproduction of the Graphchi experiments in [1] and a comparison with X-stream. Graphchi requires pre-processing time to sort shards, which X-stream does not. Further, execution time in Graphchi is also higher. This is attributable to two factors. First, partitions (shards) in Graphchi are sorted by source vertex. However locating the incoming edges to absorb updates in the gather step (Figure 2, line 19) requires re-sorting the shard by destination vertex (a component of runtime, shown in the last column). Another contributor is the fact that Graphchi requires a higher number of partitions. It needs to access all

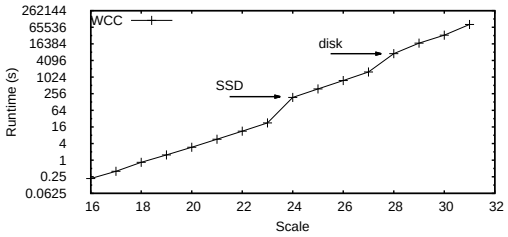

Figure 10: X-stream across different storage media

Pagerank [13] on Twitter [14]				
System	Partitions	Preproc.	Time	Re-sort
X-stream	1	*none*	545.28s	–
Graphchi	32	825.70s	1320.13s	989.75s
WCC on scale 27				
System	Partitions	Preproc.	Time	Re-sort
X-stream	1	*none*	1486.74s	–
Graphchi	24	2359.36s	4020.13s	2169.24s

Figure 11: X-stream compared to Graphchi

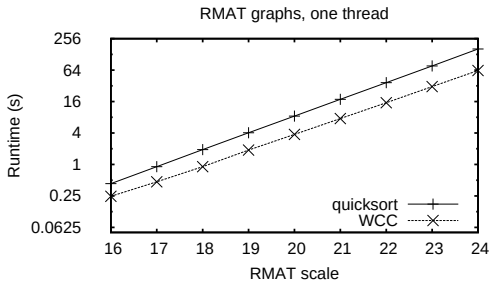

Figure 14: X-stream vs. Sorting

the partitions simultaneously (by design) and therefore its sequential accesses are more spread out, leading to lower utilization of the streaming bandwidth of the SSD than X-stream. This is demonstrated using a 4 minute snapshot from `iostat` in Figure 12.

4.3 Problems

X-stream needs to trade off fewer random accesses to the edge list in the scatter phase for the sequential bandwidth of streaming a large number of potentially unrelated edges. This can lead to X-stream performing suboptimally with some graphs. Figure 13 shows the spread of X-stream's performance on various real world graphs. The table includes the number of supersteps and fraction of wasted edges. The former is the number of times the main set of loops (Figure 8 lines 1–5) are executed. The latter is the fraction of streamed edges (Figure 8 line 8) wasted because the vertex had no update to send (Figure 8 line 11). We note that X-stream performs poorly on the DIMACS and Yahoo-web graphs. In both the cases, the number of supersteps is extremely high. Although we did not get the data for the Yahoo webgraph, it ran for upwards of 500 supersteps. Also the DIMACS graph shows a high fraction of wasted edges.

This problem with X-stream can be understood in terms of the fact that the number of supersteps for WCC is bounded by the diameter of the graph (the length of the longest shortest path). For graphs of high diameter, not only does X-stream take a longer time to finish but also the number of active vertices at each superstep is low (lower diameter means better connectivity) in turn leading to more wasted edges. Our initial design for X-stream was based on the observation that such graphs are in fact rare in practice: most graphs have a small diameter and in fact many real world graphs demonstrate shrinking diameter with the addition of vertices [18].

Nevertheless we would like to mitigate the impact of high diameter graphs in X-stream. We note that RASP does not have any problem with such graphs as it accesses exactly the required edges and therefore does no wasted work on any iteration.

```
while not done {                                                    1
  // Iterators                                                      2
  one of {                                                          3
    for all vertices v in ScatterIterator {scatter(v)}              4
    for all edges e in Graph { scatter(e) }                         5
  }                                                                 6
  for all edges e in GatherIterator { gather(e)  }                  7
}                                                                   8
//scatter step for edge                                             9
scatter(e)                                                          10
{                                                                   11
  v = e.src                                                         12
  if v.need_scatter {                                               13
       e.update = User_defined_scatter(v, e)                        14
       if e.update.need_gather {                                    15
         GatherIterator.add(e)                                      16
}}                                                                  17
// scatter step for vertex                                          18
scatter(v)                                                          19
{                                                                   20
  if v.need_scatter {                                               21
       e.update = User_defined_scatter(v, e)                        22
       if e.update.need_gather {                                    23
         GatherIterator.add(e)                                      24
}}                                                                  25
//gather step                                                       26
gather(e)                                                           27
{                                                                   28
  v = e.dst                                                         29
  v = User_defined_gather(v, e.update)                              30
  possibly {                                                        31
    if v.need_Scatter {                                             32
      ScatterIterator.add(v)                                        33
}}}                                                                 34
```

Figure 15: Hybrid model with X-stream

5. A POSSIBLE UNIFIED APPROACH

We are currently exploring the possibility of unifying the approaches in RASP and X-stream. Our planned solution is to allow streaming partitions to sort their associated edges and access them randomly. This hybrid execution of the IVEC model is illustrated in Figure 15. The runtime chooses to either iterate through vertices to scatter updates from those vertices that need to scatter updates or simply streams all the edges scattering updates along those edges whose source vertex has an update. Some salient features of the planned approach are:

- The starting point is X-stream and the decision of whether to scatter using an index on the edges or to simply stream them is made dynamically on a per-partition basis
- On the first instance of scattering through an index the

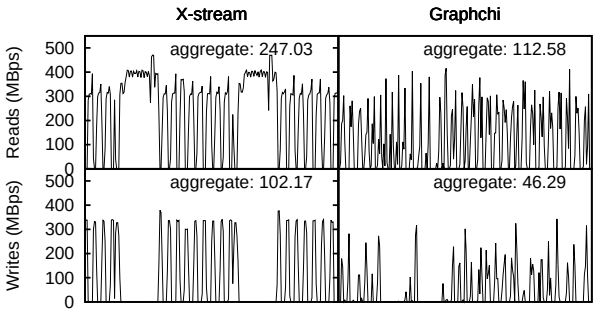

Figure 12: SSD Utilization for Pagerank on Twitter

Graph	Vertices	Edges	T	s	w
Memory					
amazon0601 [11]	403K	3.3M	00:00:00.34	18	0.63
cit-Patents [11]	3.7M	16M	00:00:02.06	20	0.48
dimacs-usa [15]	23M	58M	00:08:12.31	6262	0.98
soc-livejournal [11]	4.8M	68M	00:00:04.71	12	0.56
SSD					
Friendster [11]	65.6M	1.8B	01:03:57.75	23	0.61
sk-2005 [16]	50.6M	1.9B	01:14:07.93	24	0.65
Twitter [14]	41.7M	1.4B	00:28:31.53	15	0.53
Magnetic Disk					
Friendster	65.6M	1.8B	02:09:08.20	23	0.61
sk-2005	50.6M	1.9B	03:00:57.20	24	0.65
Twitter	41.7M	1.4B	01:08:33.46	15	0.53
yahoo-web [17]	1.4B	6.6B	> 1 week		

Figure 13: X-stream performance on various graphs: runtime hh:mm:ss.ss (T), Supersteps (s) and Wasted edges fraction(w)

edges in the partition are sorted and an index built on them

- We intend to start with a streaming approach and based on the fraction of wasted edges observed, switch to an indexed random access approach.

Once a streaming partition has switched to random access, we are free to apply RASP to mitigate the effect of random access through prefetching. We note that at this point all the needed (subset of) vertex data is in memory. This solves the problem of needing to fit *all* vertex data of RASP into memory. Also, we can then prefetch needed edges from main memory into CPU cache rather than being restricted to prefetching edges only from SSD into main memory. This partially resolves the restriction of RASP to SSDs, we still restrict ourselves to streaming for magnetic disks. Switching to random access allows X-stream to avoid problems with graphs of high diameter, where it can avoid streaming wasted edges and finish quicker.

6. CONCLUSION

We have argued in this paper that storage is the determinant for performance in graph processing. Along these lines, we have presented two systems under development that aim to improve the interaction between graph algorithms and storage. Each system individually represents a significant step forward in the state of the art and we believe that an amalgamation of the two ideas can finally remove storage bottlenecks from the critical path in high performance graph processing systems.

Acknowledgment. The research is part funded by EU grant FP7-ICT-257756 and EPSRC grant EP/H003959.

7. REFERENCES

[1] Aapo Kyrola and Guy Blelloch. Graphchi: Large-scale graph computation on just a PC. In *OSDI*. USENIX Association, 2012.

[2] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146. ACM, 2010.

[3] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *OSDI*, pages 17–30. USENIX Association, 2012.

[4] Amitabha Roy, Karthik Nilakant, Valentin Dalibard, and Eiko Yoneki. Mitigating I/O latency in SSD-based graph traversal. Technical Report UCAM-CL-TR-823, University of Cambridge, November 2012.

[5] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: graph algorithms for the (semantic) web. In *ISWC - Volume Part I*, pages 764–780. Springer-Verlag, 2010.

[6] Sungpack Hong, Tayo Oguntebi, and Kunle Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *PACT*, pages 78–88. IEEE Computer Society, 2011.

[7] Virat Agarwal, Fabrizio Petrini, Davide Pasetto, and David A. Bader. Scalable graph exploration on multicore processors. In *SC*, pages 1–11. IEEE Computer Society, 2010.

[8] Roger Pearce, Maya Gokhale, and Nancy M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *SC*, pages 1–11. IEEE Computer Society, 2010.

[9] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *ASPLOS*, pages 349–362. ACM, 2012.

[10] `http://twitter.mpi-sws.org/`.

[11] `http://snap.stanford.edu/`.

[12] `http://www.graph500.org/`.

[13] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[14] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600. ACM, 2010.

[15] `http://dimacs.rutgers.edu/Challenges/`.

[16] `http://www.cise.ufl.edu/research/sparse/matrices/LAW/sk-2005.html`.

[17] `http://webscope.sandbox.yahoo.com/`.

[18] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.