X-Stream: Edge-centric Graph Processing using Streaming Partitions

Amitabha Roy, Ivo Mihailovic, Willy Zwaenepoel
{amitabha.roy, ivo.mihailovic, willy.zwaenepoel}@epfl.ch
EPFL

Abstract

X-Stream is a system for processing both in-memory and out-of-core graphs on a single shared-memory machine. While retaining the scatter-gather programming model with state stored in the vertices, X-Stream is novel in (i) using an edge-centric rather than a vertex-centric implementation of this model, and (ii) streaming completely unordered edge lists rather than performing random access. This design is motivated by the fact that sequential bandwidth for all storage media (main memory, SSD, and magnetic disk) is substantially larger than random access bandwidth.

We demonstrate that a large number of graph algorithms can be expressed using the edge-centric scatter-gather model. The resulting implementations scale well in terms of number of cores, in terms of number of I/O devices, and across different storage media. X-Stream competes favorably with existing systems for graph processing. Besides sequential access, we identify as one of the main contributors to better performance the fact that X-Stream does not need to sort edge lists during pre-processing.

1 Introduction

Analytics over large graphs is an application that is beginning to attract significant attention in the research community. Part of the reason for this upsurge of interest is the great variety of information that is naturally encoded as graphs. Graph processing poses an interesting systems challenge: the lack of access locality when traversing edges makes obtaining good performance difficult [39].

```
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

This paper presents X-Stream, a system for scale-up graph processing on a single shared-memory machine. Similar to systems such as Pregel [40] and Powergraph [31], X-Stream maintains state in the vertices, and exposes a scatter-gather programming model. The computation is structured as a loop, each iteration of which consists of a scatter phase followed by a gather phase. Figure 1 illustrates the common *vertex-centric* implementation of the scatter-gather programming model. Both the scatter and the gather phase iterate over all vertices. The user provides a scatter function to propagate vertex state to neighbors and a gather function to accumulate updates from neighbors to recompute the vertex state. This simple programming model is sufficient for a variety of graph algorithms [40, 31, 54] ranging from computing shortest paths to ranking web pages in a search engine, and hence is a popular interface for graph processing systems.

The accepted (and intuitive) approach to scale-up graph processing, for both in-memory [33] and out-of-core [43] graphs, is to sort the edges of the graph by originating vertex and build an index over the sorted edge list. The execution then involves random access through the index to locate edges connected to a vertex. Implicit in this design is a tradeoff between sequential and random access, favoring a small number of random accesses through an index in order to locate edges connected to an

SOSP '13, Nov 03-06 2013, Farmington, PA, USA ACM 978-1-4503-2388-8/13/11. http://dx.doi.org/10.1145/2517349.2522740

```
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

active vertex, over streaming a large number of (potentially) unrelated edges and picking up those connected to active vertices. It is this tradeoff that is revisited in this paper.

Random access to any storage medium delivers less bandwidth than sequential access. For instance, on our testbed (16 core/64 GB 1U server, 7200 RPM 3TB magnetic disk and 200 GB PCIe SSD), bandwidth for sequential reads compared to random reads is 500 times higher for disks and 30 times higher for SSDs. Even for main memory, as a result of hardware prefetching, sequential bandwidth outstrips random access bandwidth by a factor of 4.6 for a single core and by a factor of 1.8 for 16 cores. See §5.1 for more details.

We demonstrate in this paper that the larger bandwidth of sequential access can be exploited to build a graph processing system based purely on the principle of streaming data from storage. We show that this design leads to an efficient graph processing system for both in-memory graphs and out-of-core graphs that, in a surprising number of cases, equals or outperforms systems built around random access through an index.

To achieve this level of performance, X-Stream introduces an *edge-centric* approach to scatter-gather processing, shown in Figure 2: the scatter and gather phase iterate over edges and updates on edges rather than over vertices. This edge-centric approach altogether avoids random access into the set of edges, instead streaming them from storage. For graphs with the common property that the edge set is much larger than the vertex set, access to edges and updates dominates the processing cost, and therefore streaming the edges is often advantageous compared to accessing them randomly. Doing so comes, however, at the cost of random access into the set of vertices. We mitigate this cost using *streaming partitions*: we partition the set of vertices such that each partition fits in high-speed memory (the CPU cache for in-memory graphs and main memory for out-of-core graphs). Furthermore, we partition the set of edges such that edges appear in the same partition as their source vertex. We then process the graph one partition at a time, first reading in its vertex set and then streaming its edge set from storage. A positive consequence of this approach is that we do not need to sort the edge list, thereby not incurring the pre-processing delays in other systems [33, 37].

Graphchi [37] was the first system to explore the idea of avoiding random access to edges. Graphchi uses a novel out-of-core data structure that consists of partitions of the graph called 'shards'. Unlike streaming partitions, shards have to be pre-sorted by source vertex, a significant pre-processing cost, especially if the graph is not used repeatedly. Graphchi also continues to use the vertex-centric implementation in Figure 1. This requires the entire shard - vertices and all of their incoming and outgoing edges - to be present in memory at the same time, leading to a larger number of shards than streaming partitions, the latter requiring only the vertex state to be in memory. Streaming partitions therefore take better advantage of sequential streaming bandwidth. Shards also require a re-sort of the edges by destination vertex in order to direct updates to vertices in the gather step.

This paper makes the following contributions through the design, implementation and evaluation of X-Stream:

- We introduce edge-centric processing as a new model for graph computation and show that it can be applied to a variety of graph algorithms.
- We show how the edge-centric processing model can be implemented using *streaming partitions* both for in-memory and out-of-core graphs, merely by using different partition sizes for different media.
- We demonstrate that X-Stream scales well in terms of number of cores, I/O devices and across different storage media. For instance, X-Stream identifies weakly connected components in graphs with up to 512 million edges in memory within 28 seconds, with up to 4 billion edges from SSD in 33 minutes, and with up to 64 billion edges from magnetic disk in under 26 hours.
- We compare X-Stream to alternative graph processing systems, and show that it equals or outperforms vertex-centric and index-based systems on a number of graph algorithms for both in-memory and out-of-core graphs.

2 The X-Stream Processing Model

X-Stream presents to the user a graph computation model in which the mutable state of the computation is stored in the vertices, more precisely in the data field of each vertex. The input to X-Stream is an unordered set of directed edges. Undirected graphs are represented using a pair of directed edges, one in each direction.

X-Stream provides two principal API methods for expressing graph computations. Edge-centric scatter takes as input an edge, and computes, based on the data field of its source vertex, whether an update value needs to be sent to its destination vertex, and, if so, the value of that update. Edge-centric gather takes as input an update, and uses its value to recompute the data field of its destination vertex.

The overall computation is structured as a loop, terminating when some application-specific termination criterion is met. Each loop iteration consists of a scatter phase followed by a gather phase. The scatter phase iterates over all edges and applies the scatter method to each edge. The gather phase iterates over all updates produced in the scatter phase and applies the gather method to each update. Hence, X-Stream's edge-centric scatter gather is *synchronous* and guarantees that all updates from a previous scatter phase are seen only after the scatter is completed and before the next scatter phase is begun. In this sense it is similar to distributed graph processing systems such as Pregel [40].

2.1 Streams

X-Stream uses streaming to implement the graph computation model described above. An input stream has one method, namely read the next item from the stream. An input stream is read in its entirety, one item at a time. An output stream also has one method, namely append an item to the stream.

The scatter phase of the computation takes the edges as the input stream, and produces an output stream of updates. In each iteration it reads an edge, reads the data field of its source vertex, and, if needed, appends an update to the output stream. The gather phase takes the updates produced in the scatter phase as its input stream. It does not produce any output stream. For each update in the input stream, it updates the data value of its destination vertex.

The idea of using streams for graph computation applies both to in-memory and out-of-core graphs. To unify the presentation, we use the following terminology. We refer to caches in the case of in-memory graphs and to main memory in the case of out-of-core graphs as *Fast Storage*. We refer to main memory in the case of in-memory graphs and to SSD or disks in the case of out-of-core graphs as *Slow Storage*.

Figure 3 shows how memory is accessed in the edge streaming model. The appeal of the streaming approach

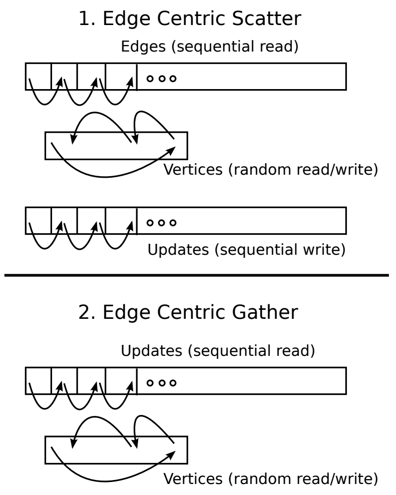
1. Edge Centric Scatter
Edges (sequential read)
Vertices (random read/write)
Updates (sequential write)
2. Edge Centric Gather
Updates (sequential read)
Vertices (random read/write)

Figure 3: Streaming Memory Access

to graph processing stems from the fact that it allows sequential (and therefore much faster) access to *Slow Storage* for the (usually large) edge and update streams. The problem is that it requires random access to the vertices, which for large graphs may not fit in *Fast Storage*. To solve this problem, we introduce the notion of streaming partitions, described next.

2.2 Streaming Partitions

A streaming partition consists of a vertex set, an edge list, and an update list. The vertex set of a streaming partition is a subset of the vertex set of the graph. The vertex sets of different streaming partitions are mutually disjoint, and their union equals the vertex set of the entire graph. The edge list of a streaming partition consists of all edges whose *source vertex* is in the partition's vertex set. The update list of a streaming partition consists of all updates whose *destination vertex* is in the partition's vertex set.

The number of streaming partitions stays fixed throughout the computation. During initialization, the vertex set of the entire graph is partitioned into vertex sets for the different partitions, and the edge list of each partition is computed. These vertex sets and edge lists also remain fixed during the entire computation. The update list of a partition, however, varies over time: it is recomputed before every gather phase, as described next.

2.3 Scatter-Gather with Partitions

With streaming partitions, the scatter phase iterates over all streaming partitions, rather than over all edges, as described before. Similarly, the gather phase also iterates over all streaming partitions, rather than over all

```
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

updates. Figure 4 details pseudocode for edge-centric scatter-gather using streaming partitions.

For each streaming partition, the scatter phase reads its vertex set, streams in its edge list, and produces an output stream of updates. This output stream is appended to a list `Uout`. These updates need to be re-arranged such that each update appears in the update list of the streaming partition containing its destination vertex. We call this the *shuffle phase*. The shuffle takes as its input stream the updates produced in the scatter phase, and moves each update to the update list `Uin(p)`, where *p* is the streaming partition containing the destination vertex of the update. After the shuffle phase is completed, the gather phase can start. For each streaming partition, we read its vertex set, stream in its update list, and compute new values for the data fields of the vertices as we read updates from the update list.

Streaming partitions are a natural unit of parallelism for both the scatter and the gather phase. We will show how to take advantage of this parallelism in §4.

2.4 Size and Number of Partitions

Choosing the correct number of streaming partitions is critical to performance. On the one hand, in order to produce fast random access to the vertices, all vertices of a streaming partition must fit in *Fast Storage*. On the other hand, in order to maximize the sequential nature of access to *Slow Storage* to load the edge lists and the update lists of a streaming partition, their number must be kept as small as possible. We restrict the vertex sets of streaming partitions to be of equal size. As a result, we choose the number of the streaming partitions such that, allowing for buffers and other auxiliary data structures, the vertex set of each streaming partition fills up *Fast Storage*.

2.5 API Limitations and Extensions

Unlike vertex-centric graph processing APIs, there are no means to iterate over the edges or updates belonging to a vertex in X-Stream's edge-centric API. However, in addition to allowing the user to specify edge scatter and gather functions, X-Stream also supports vertex iteration, which simply iterates over all vertices in the graph, applying a user-specified function on each vertex. This is useful for initialization and for various aggregation operations.

X-Stream also supports interfaces other than edge-centric scatter-gather. For example, X-Stream supports the semi-streaming model for graphs [26] or graph algorithms that are built on top of the W-Stream model [14]. Although these models allow a richer set of graph algorithms to be expressed, we focus on the more familiar scatter-gather mode of operation in this paper.

3 Out-of-core Streaming Engine

The input to the out-of-core graph processing engine is a file containing the unordered edge list of the graph. In addition, we store three disk files for each streaming partition: a file each for the vertices, edges and updates.

As we saw in Section 2, with the edge-centric scatter-gather model, sequential access is easy to achieve for the scatter and gather phases. The hard part is to achieve sequential access for the shuffle phase. To do so for out-of-core graphs, we slightly modify the computation structure suggested in Figure 4. Instead of a strict sequence of scatter, shuffle and gather phases, we fold the shuffle phase into the scatter phase. In particular, we run the scatter phase, appending updates to an in-memory buffer. Whenever that buffer becomes full, we run an *in-memory shuffle*, which partitions the list of updates in the in-memory buffer into (in-memory) lists of updates for vertices in the different partitions, and then appends those lists to the disk files for the updates of each partition.

3.1 In-memory Data Structures

Besides the vertex array used to store the vertices of a streaming partition, we need in-memory data structures to hold input from disk (edges during the scatter phase and updates during the gather phase), the input and the output of the in-memory shuffle phase, and output to disk (updates after each in-memory shuffle phase). In order to avoid the overhead of dynamic memory allocation, we designed a statically sized and statically allocated data structure, the *stream buffer*, to store these variable-sized data items. A stream buffer consists of a

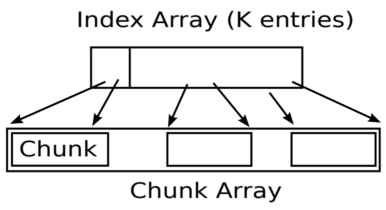

Figure 5: Stream Buffer (`K`: number of partitions)

```
merged scatter/shuffle phase:
  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

(large) array of bytes called the chunk array, and an index array with `K` entries for `K` streaming partitions (see Figure 5). The i-th entry in the index array describes the *chunk* of the chunk array with data relating to the i-th partition.

Using two streaming buffers, the in-memory shuffle becomes straightforward. One streaming buffer is used to store the updates resulting from the scatter phase. A second streaming buffer is used to store the result of the in-memory shuffle. We make one pass over the input buffer counting the updates destined for each partition. We then fill in the index array of the second streaming buffer. Finally, we copy the updates from the first buffer to the appropriate location in the chunk array of the second buffer.

3.2 Operation

The disk engine begins by partitioning the input edge list into different streaming partitions. Interestingly, this can be done efficiently by using the in-memory shuffle. The successive parts of the input edge list are read in from disk to a streaming buffer, shuffled into another streaming buffer, and then written to the disk files containing the edge lists of the streaming partitions.

After this pre-processing step, the graph processing engine then enters the main processing loop, depicted in Figure 6.

Finally, we implemented a couple of optimizations. First, if the entire vertex set fits into memory, the vertex array need not be written out to disk at the end of the gather phase. Second, if all updates for the entire scatter phase fit into a streaming buffer, then the updates are not written back to disk after the in-memory shuffle. Instead, the gather phase simply reuses the in-memory output of the shuffle.

3.3 Disk I/O

X-Stream performs asynchronous direct I/O to and from the stream buffer, bypassing the operating system's page cache. We use an additional 4K page per streaming partition to keep I/O aligned, regardless of the starting point of a chunk in a streaming buffer.

We actively prefetch from a stream, exploiting the sequentiality of access. As soon as a read into one input stream buffer is completed, we start the next read into a second input stream buffer. Similarly, the writes to disk of the chunks in one output buffer are overlapped with computing the updates of the scatter phase into another output buffer. This requires allocating an extra stream buffer for input and an extra one for output. We found this prefetch distance of one, both on input and output, sufficient to keep the disks 100% busy in our experiments. A deeper prefetch can effectively be achieved with a larger chunk array, if necessary.

X-Stream transparently exploits RAID architectures. The sequential writes stripe the files across disks, and the sequential reads of these striped files achieve a multiplication of bandwidth compared to single disks. We can also exploit parallelism between the input and output files, which can be placed on different disks. X-Stream does asynchronous I/O using dedicated I/O threads and spawns one thread for each disk. One can therefore put the edges and updates on different disks, doing I/O to them in parallel.

X-Stream's I/O design is also well suited for use with SSDs. All X-Stream writes are sequential and therefore avoid the problem of write amplification [34], in a manner similar to log-structured filesystems [52] and memory allocators built specifically for flash devices [16]. Many modern flash translation layers in fact implement a log-structured filesystem in firmware. X-Stream's sequential writes can be a considered a best case for such firmware. In addition, we always truncate files when the streams they contain are destroyed. On most operating systems, truncation automatically translates into a TRIM command sent to the SSD, freeing up blocks and thereby reducing pressure on the SSD garbage collector.

3.4 Number of Partitions

Given a fixed amount of memory, the need to size stream buffers properly creates additional requirements on the number of streaming partitions beyond simply the need to fit the vertex set of the streaming partition in memory. We need to issue large enough units of I/O to approach streaming bandwidth to disk. *Assuming a uniform distribution of updates across streaming partitions*, and assuming we need to issue I/O request of S bytes to achieve maximum I/O bandwidth, we need to size the chunk array to *at least* $S*K$ bytes for K streaming partitions. Our design requires two stream buffers each for the input and output streams in order to support prefetching. In addition we need a stream buffer for shuffling: a total of 5. If we assume that N is total space taken by the vertices and M is the total amount of main memory available then our requirements translate to: $\frac{N}{K} + 5SK \leq M$.

Viable solutions to this inequality exist even for very large graphs. The left hand side reaches a minimum at $K = \sqrt{\frac{N}{5S}}$ at which point the minimum amount of memory needed is $2\sqrt{5NS}$. For an I/O unit of $S = 16$MB (justified in §5) the minimum amount of main memory required for a graph with total vertex data size as large as $N = 1$TB is therefore only $M = 17$GB with under $K = 120$ streaming partitions. This ignores the overhead of the index data which would have come to an additional 5KB in our design.

4 In-memory Streaming Engine

The in-memory engine is designed for processing graphs whose vertices, edges and updates fit in memory. Our main concern in designing the in-memory streaming engine is parallelism. We need all available cores in a system to reach peak streaming bandwidth to memory. In addition, parallelism was important in order to use all available computational resources (such as floating point units). We therefore discuss the important building blocks for parallelism in the in-memory streaming engine. A second concern was that the in-memory engine must deal with a larger number of partitions than the out-of-core streaming engine. This necessitates the use of a multi-stage shuffler, described in §4.2.

The in-memory engine considers the CPU caches when choosing the number of streaming partitions and aims to fit the vertex data corresponding to each partition in the CPU cache. Unlike disks, we do not have direct control on block allocation in CPU caches. The in-memory streaming engine must also consider additional data that must be brought in without displacing the vertices. Observing that an edge and an update must refer to a vertex

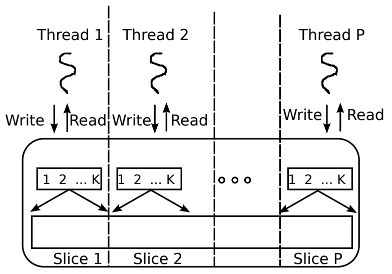

Figure 7: Slicing a Streaming Buffer

without displacing it from cache, the in-memory streaming engine therefore calculates the vertex footprint as the sum of vertex data size, edge size and update size. It then divides the total footprint of all vertices in the graph by the size of the available CPU cache to arrive at the final number of streaming partitions.

The engine needs exactly three stream buffers, one to hold the edges of the graph, one to hold generated updates and one more to be used during shuffling. We start by loading the edges into the input stream buffer and shuffling them into a chunk of edges for each streaming partition. We then process streaming partitions one by one generating updates from the streaming partitions in the scatter phase. *All* the generated updates are appended into a *single* output stream buffer. The output stream buffer is then shuffled into chunks of updates per streaming partition, which are then absorbed in the gather phase.

4.1 Parallel Scatter-Gather

Our approach to parallelizing the scatter and gather phases rests on the observation that the streaming operation can be done independently for different streaming partitions. Supporting parallel scatter-gather requires awareness of shared caches when calculating the number of streaming partitions. We assume underlying cores receive equal shares of a shared cache.

The threads executing different streaming partitions are still required to append their updates to the same chunk array. Each thread first writes to a private buffer (of size 8K), which is flushed to the shared output chunk array, by first atomically reserving space at the end and then appending the contents of the private buffer.

Executing streaming partitions in parallel can lead to significant workload imbalance as the partitions can have different numbers of edges assigned to them. We therefore implemented work stealing in X-Stream, allowing threads to steal streaming partitions from each other. This lets us avoid the work imbalance problem that requires specialized solutions for in-memory [44] or scale out [31] graph processing systems.

4.2 Parallel Multistage Shuffler

The in-memory engine must deal with a larger number of partitions than the out-of-core engine as CPU caches are small in size with respect to main memory and current architectural trends indicate that they are unlikely to grow further [27]. Repeating the analysis in § 3.4, for a graph with `1TB` of vertex data (on a system with more than `1TB` RAM) and a `1MB` CPU cache we need at least `1M` partitions to ensure that the randomly accessed vertex data for each partition fits in CPU cache (even excluding the rest of the vertex footprint). Shuffling into a large number of partitions leads to a significant challenge in maintaining our design goal of exploiting sequential access bandwidth to memory.

Sequential access from the CPU core provides higher bandwidth to memory for two reasons. The first is that microprocessors (such as the current generation x86 one used in our experiments) usually come with hardware prefetchers that can track multiple streams. Having the prefetchers track the input and output streams is beneficial in hiding the latency of access to memory (the primary source of higher bandwidth for sequential accesses). Increasing the number of partitions beyond a point means that we lose the benefit of hardware prefetchers. The second benefit of sequential accesses is due to maximum spatial locality as each cacheline is fully used before being evicted. This is only possible if we can fit a cacheline from the input stream and all output streams in the cache. With a larger number of partitions this is no longer the case (SRAM caches closest to the core typically fit only `512` to `1024` 64-byte cachelines).

Inspired by solutions to similar problems in cache-conscious sorting [51] and in systems such as Phoenix [45] or tiled map-reduce [24], we implemented a multi-stage shuffler for the in-memory engine in X-Stream. We group partitions together into a tree hierarchy, with a branching factor (we term this the *fanout*) of `F`. This is done in X-Stream by enforcing that the number of streaming partitions for the in-memory engine is a power of two and also setting the fanout of the tree to a power of two. The tree is then implicitly maintained by using the most significant `b` bits of the partition ID to choose between between groupings at a tree level with 2^b nodes. We then do one shuffle step for each level in this tree. The input consists of a stream buffer with as many chunks as nodes at that level in the tree. Each input chunk is shuffled into `F` output chunks. Given a target of K partitions, the multi-stage shuffler can therefore shuffle the input into K chunks in $\lceil \log_F K \rceil$ steps down the tree. We use exactly two stream buffers in the shuffle process, alternating them between the input and output roles.

The fanout `F` can be set keeping in mind the constraints described above. For the experiments in this paper, we bounded it to the number of cachelines available in the CPU cache. In most cases, however, we are also within the limit of the number of tracked streams in the hardware prefetcher, which is a micro-architectural detail we did not specifically tune for.

We observe that a stream buffer typically carries many more objects than partitions. Our experiments (§5) on graphs result in in-memory streams with over a billion objects but never require more than 1K partitions. This causes shuffling to be cheaper than sorting even with multiple stages, a point we return to in the evaluation.

In order to enable parallelism in the multi-stage shuffler, we required the ability to have threads work in parallel on stream buffers, without needing synchronization. Our solution is to assign X-Stream threads disjoint equally sized slices of the stream buffer. Each thread receives exactly one slice. Figure 7 shows how a stream buffer is sliced across threads. Each thread has an independent index array describing chunks in its slice of the stream buffer. A thread is only allowed to access its own slice during shuffling. We parallelize the shuffle step by assigning each thread to shuffle its own slice. Since the number of target partitions is the same across all the threads, they all end up with the final slice in the same output stream buffer, at which point they synchronize.

The chunk corresponding to a streaming partition is the union of the corresponding chunks from all the slices. A thread can therefore recover a chunk during the scatter and gather steps from the sliced stream buffer using sequential accesses plus at most P random accesses, where P is the number of threads. Usually the number of threads P is far smaller than the number of objects in the chunk, rendering the random access negligible.

4.3 Layering over Disk Streaming

The in-memory engine is logically layered above the out-of-core engine. We do so by allowing the disk engine to independently choose its count of streaming partitions. For each iteration of the streaming loop in Figure 6, the loaded input chunk is processed using the in-memory engine that then independently chooses a further in-memory partitioning for the current disk partition.

This allows us to ensure that we maximize usage of main memory bandwidth and computational resources with the out-of-core streaming engine. The slower disk continues to remain a bottleneck in all cases, even when using faster disks such as SSDs and using floating-point computation in some graph algorithms.

5 Evaluation

5.1 Experimental Environment

Our testbed is an 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. The system is equipped with 64GB of main memory; two 200GB PCI Express SSDs arranged into a software RAID-0 configuration and two 3 TB SATA-II magnetic disks also arranged into a software RAID-0 configuration.

We first measured the streaming bandwidth available from main memory. We used a microbenchmark, in which each thread read from or wrote to a thread-private buffer of size 256 MB (well beyond the capacity of the L3 cache and TLBs). The results in Figure 8 show that for reads memory bandwidth saturates with 16 cores at approximately 25GB/s. Adding 16 more cores increases the available bandwidth by only 5%. We therefore use only 16 cores of the system, as memory bandwidth, the critical resource for graph processing, is already saturated with just 16 cores. We run a single thread on one core of each pair of clustered cores, leaving the other core of each pair unused. When determining the number of partitions for in-memory graphs, we assume that each core has exclusive access to its 2MB shared L2 cache

We used the `fio` [1] tool to benchmark the streaming bandwidth of the RAID-0 SSD and disk pairs in our testbed. Our workload issues a single synchronous request at a time, and varies the size of the request. The results are shown in Figure 9. An interesting aspect there is that SSD write bandwidth shows a temporary drop with a 512K request size. This is likely due to the flash translation layer not being able to keep up at this write request size. Bandwidth for both SSD and disk shows a sharp increase at 1M request sizes. The RAID stripe unit is 512K, and hence past 1M the request is striped across the SSDs or disks in the RAID-0 pair, explaining the increase in bandwidth. Figure 9 also shows that for reads both the SSDs and the disks are saturated with sequential requests of size 16MB. We therefore chose 16MB as our preferred I/O unit size for the out-of-core engine.

We also measured random access bandwidth by accessing entirely a randomly chosen cacheline from an in-memory buffer or by doing synchronous 4K transfers from an out-of-core file. Figure 11 shows that sequential access beats random access for every medium, with an increasing gap as we move to slower media. For main memory, it is necessary to use all available cores to saturate memory bandwidth. Random write performance is better than random read performance. For main memory, this is due to the write-coalescing buffers present in the AMD 6272 micro-architecture. For the out-of-core case, this is due to the write cache on the disk absorbing the writes, allowing the next write to be issued while the previous one is outstanding.

5.2 Algorithms and Graphs

We evaluate X-Stream using the following algorithms:

- Weakly Connected Components (WCC).
- Strongly Connected Components (SCC), using [47]. Requires a directed graph.
- Single-Source Shortest Paths (SSSP).
- Minimum Cost Spanning Tree (MCST) using the GHS [30] algorithm.
- Maximal Independent Set (MIS).
- Conductance [20].
- SpMV: Multiply the sparse adjacency matrix of a directed graph with a vector of values, one per vertex.
- Pagerank [42] (5 iterations).
- Alternating Least Squares (ALS) [55] (5 iterations). Requires a bipartite graph.
- Bayesian Belief Propagation (BP) [35] (5 iterations).

We used both synthetic and real-world (Figure 10) graph datasets to evaluate X-Stream. We generated synthetic undirected graphs using the RMAT generator [23] and used them to study the scaling properties and evaluate configuration choices of X-Stream. RMAT graphs have a scale-free property that is a feature of many real-world graphs [17]. We use RMAT graphs with an average degree of 16 (as recommended by the Graph500 benchmark [9]). We use the term scale n to refer to a synthetic graph with 2^n vertices and 2^{n+4} edges. For SCC, we assigned a random edge direction to the synthetic RMAT and Friendster graphs.

All the graphs are provided to X-Stream as unordered lists of edges. For inputs without an edge weight, we added a random edge weight (a pseudo-random floating point number in the range $[0\ \ 1)$). The footprint of vertex data varied from a single byte in the case of MIS (a boolean variable) to almost 250 bytes in the case of ALS.

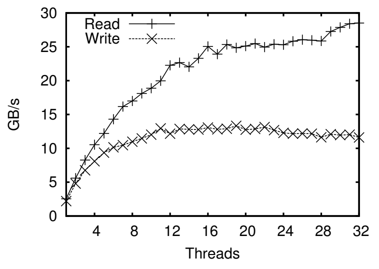

Figure 8: Memory bandwidth

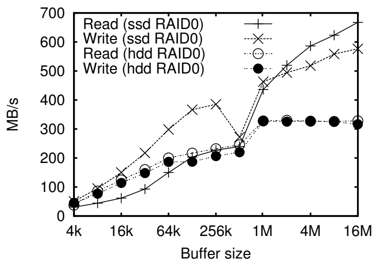

Figure 9: Disk bandwidth

Name	Vertices	Edges	Type
In-memory			
amazon0601 [2]	403,394	3,387,388	Directed
cit-Patents [3]	3,774,768	16,518,948	Directed
soc-livejournal [4]	4,847,571	68,993,773	Directed
dimacs-usa [5]	23,947,347	58,333,344	Directed
Out-of-core			
Twitter [36]	41.7 million	1.4 billion	Directed
Friendster [6]	65.6 million	1.8 billion	Undir.
sk-2005 [7]	50.6 million	1.9 billion	Directed
yahoo-web [8]	1.4 billion	6.6 billion	Directed
Netflix [55]	0.5 million	0.1 billion	Bipartite

Figure 10: Datasets

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

Figure 11: Sequential Access vs. Random Access

5.3 Applicability

We demonstrate that X-Stream's API can be used to express a large number of graph algorithms with good performance, in spite of the fact that the API does not allow direct access to the edges associated with a vertex. Figure 12a shows how X-Stream performs on a variety of graph algorithms, real-world datasets and storage media. The yahoo-web graph did not fit onto our SSD, so it is absent from SSD results.

The execution time on SSD is roughly half of that on magnetic disk reflecting the fact that the SSD delivers twice the sequential bandwidth of the magnetic disk (Fig 9), although at a considerably greater cost per byte.

X-Stream performs well on all algorithms and data sets, with the exception of traversal algorithms (WCC, SCC, MIS, MSCT and SSSP) for DIMACS and the Yahoo webgraph. DIMACS traversals take a long time relative to the size of the graph, and the Yahoo webgraph did not finish in a reasonable amount of time. We hypothesized that the problem lies in the structure of these graphs, and to confirm this, we implemented HyperANF [21] in X-Stream to measure the *neighborhood function* of graphs. The neighborhood function $N_G(t)$ is defined as the number of pairs of vertices reachable within t steps in the undirected version of the graph. Figure 13 shows the number of steps needed to converge to a constant value for the neighborhood function, which is equal to the diameter of graph. As Figure 13 shows, the Yahoo webgraph and DIMACS have a diameter much larger than other comparable graphs in our dataset. A high diameter results in graphs with an 'elongated' structure, causing X-Stream to execute a very large number of scatter-gather iterations, each of which requires streaming the entire edge list but doing little work.

In Figure 12b we report some additional information on the execution of WCC on the various graphs, including 1) the number of scatter-gather steps, 2) the ratio of total execution time to streaming time, and 3) the percentage of edges that were streamed and along which no updates were sent. For DIMACS we see, as discussed above, the very large number of scatter-gather steps. The ratio of total execution time to streaming time is approximately 1 for out-of-core graphs, confirming that the execution time is governed by the bandwidth of secondary storage. For in-memory graphs, the ratio ranges roughly between 2 and 3, indicating that here too streaming takes up an important fraction of the execution time, but computation starts playing a role as well. The 'wasted' edges, i.e., edges that are streamed in but produce no updates, are a direct consequence of the tradeoff underlying X-Stream. As Figure 12b shows, X-Stream does waste considerable sequential bandwidth for some algorithms. Exploring generic stream compression algorithms as well as those specific to graphs [11], or performing extra passes to eliminate those edges that are no longer needed are important avenues of exploration we are pursuing to reduce wastage in X-Stream.

We conclude that X-Stream is an efficient way to execute a variety of algorithms on real-world graphs, its only limitation being graphs whose structure requires a large number of iterations.

5.4 Scalability

We study the scalability of X-Stream from two different angles. First, we look at the improvement in performance for a given graph size as more resources are added. Second, we show that as more storage is added, larger graphs can be handled. For these experiments, we select two traversal algorithms (BFS and WCC) and two sparse matrix multiplication algorithms (Pagerank and SpMW).

Figure 14 (with both axes in log scale) shows how X-Stream's performance scales with increasing thread

	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.

Graph	**# steps**
In-memory	
amazon0601	19
cit-Patents	20
soc-livejournal	15
dimacs-usa	**8122**
Out-of-core	
sk-2005	28
yahoo-web	**over 155**

Figure 13: Number of Steps Taken to Cover the Graph by HyperANF

count for the largest graph we can fit in memory (RMAT scale 25 with 32M vertices and 512M undirected edges). For all algorithms, performance improves linearly as more threads are added. X-Stream is able to take advantage of more memory bandwidth with increasing thread count (Figure 8) and does not incur any synchronization overhead.

Figure 15 shows how X-Stream's performance scales with an increasing number of I/O devices. We compare three different configurations: with one disk/SSD, with separate disks/SSDs for reading and writing, and with two disks/SSDs arranged in RAID-0 fashion (our baseline configuration). In the case of magnetic disk, we use an RMAT scale 30 graph, and in the case of the SSD, we use an RMAT scale 27 graph. Putting the edges and updates on different disks/SSDs reduces runtime by up to 30%, compared to using one disk/SSD. RAID-0 reduces runtime up to 50-60% of the runtime of using one disk/SSD. Clearly, X-Stream's sequential disk access pattern allows it to fully take advantage of additional I/O devices.

The scalability of X-Stream in terms of graph size is purely a function of the available storage on the machine: the design principle of streaming is equally applicable to all three of main memory, SSD and magnetic disk. We limit the available memory to X-Stream to 16GB. Figure 16 (with both axes in log scale) illustrates that X-Stream scales almost seamlessly across available devices as graph size doubles, with the 'bumps' in runtime occurring as we move to slower storage devices.

The fact that X-Stream starts from an unordered edge list means that it can easily handle growing graphs - similar to distributed systems such as Kineograph [25]. We used X-Stream to add 330M edges at a time from the Twitter dataset [36] to an initially empty graph. After each addition we recomputed weakly connected components on the graph taking into account the new edges. Figure 17 shows the recomputation time as the graph grows in size. X-Stream is limited to only use 16GB of main memory, forcing the graph to go to SSD. Each batch of ingested edges is partitioned and appended to files on the SSD and weakly connected components are recomputed. The time required for this grows with the size of the accumulated graph as component labels need to be propagated across a larger accumulated graph. However, even when the last batch of 330M added edges is ingested before the graph reaches its peak size (1.9 billion edges) recomputation takes less than 7 minutes. In contrast, the full graph takes around 20 minutes (Figure 12a). In summary, X-Stream can absorb new edges with little overhead because it supports efficient recomputation on graphs with the newly added edges.

5.5 Comparison with Other Systems

In-memory We begin by considering the performance of the in-memory engine as compared to other systems that process graphs in memory. We study the effect of the design decision made in X-Stream to stream edges and updates rather than doing random access through an index into their sorted version.

First, we consider the costs of actually producing sorted formats for input graphs such as compressed sparse row.

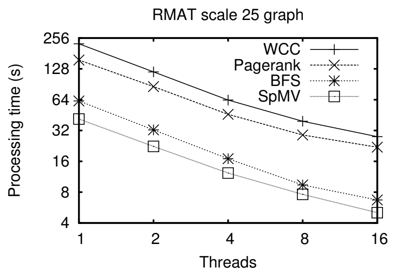

Figure 14: Strong Scaling

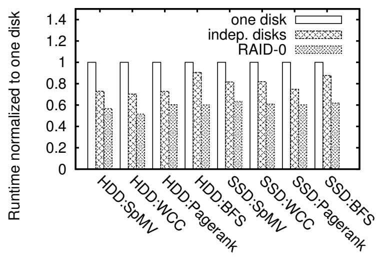

Figure 15: I/O Parallelism

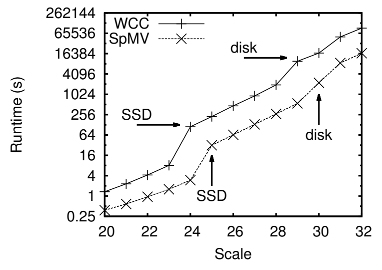

Figure 16: Scaling Across Devices

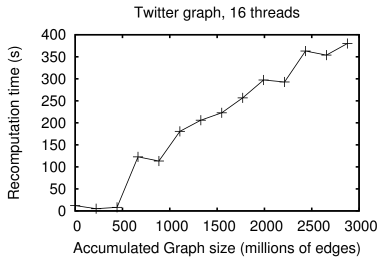

Figure 17: Re-computing WCC

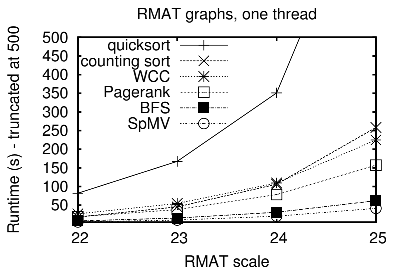

Figure 18: Sorting vs. Streaming

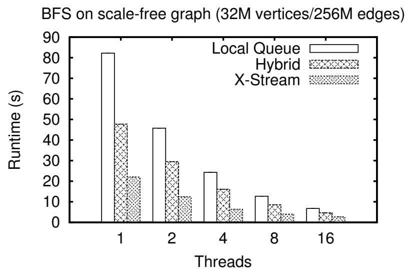

Figure 19: In-memory BFS

This requires sorting the edge list and producing an index over the sorted list. Figure 18 compares the time required for sorting the edge list of various RMAT graphs to the time required to compute results in X-Stream from the unsorted graph. We use both quicksort (from the C library) and counting sort (since the keyspace is known) to sort the graph. Both of these are single-threaded, and therefore we compare them with a single-threaded run of X-Stream. Sorting does not scale well with increasing graph size, and as a consequence X-Stream ends up completing all the graph benchmarks faster than either version of sorting at the largest graph size. This provides evidence that, where pre-processing times are a concern, streaming is a winning proposition in comparison to sorting and random access. For all comparisons with other systems in this paper, we allow the other systems to start with a sorted and indexed edge list, while also reporting pre-processing times where available. X-Stream uses the unordered edge list as input.

We next compare the performance of X-Stream to optimized in-memory implementations of breadth-first search that do random access through an index over the edges. The first method (`local queue`), due to Agarwal et. al. [12], uses a per-core queue of vertices to be visited and heavily optimized synchronization. The second method (`hybrid`), due to Hong et. al. [33], includes significant enhancements to the other (older) piece of work. In the comparisons we use the exact same graph as used in Hong et. al. [33].

Figure 19 shows the performance of X-Stream, `local queue`, and `hybrid`. The figure includes 99% confidence intervals on the runtime that are too small to be visible due to negligible variation in runtime. X-Stream performs better than both methods for all thread counts, albeit with a closing gap towards higher thread counts. The reason for this closing gap in runtime is that the gap between random and sequential memory access bandwidth is also closing with increasing thread count, from 4.6X at 1 core to 1.8X at 16 cores (Figure 11). At the same time, X-Stream sends updates on only about 35% of the streamed edges, thereby wasting 65% of the available sequential bandwidth (a tradeoff we described in §1).

Random access, however, enables highly effective algorithm-specific optimizations. Beamer et al. [18] demonstrated that for scale-free graphs large speedups 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. At these later stages, the set of discovered vertices has grown to cover a large portion of the graph, and therefore a number of updates go to vertices that are already part of the BFS tree. It is then cheaper to pull updates by scanning neighbors from the remaining vertices, rather than push updates by iterating over the BFS horizon. Ligra [48] is a recent main-memory graph processing system that implements this observation. We compare the performance of X-Stream with Ligra on a subset of the Twitter graph [36]. Comparing with Ligra is, unfortunately, not a strict apples-to-apples comparison. Ligra runs on the Cilk runtime [29] and is compiled with the Intel compiler, while X-Stream is built on top of `Linux pthreads` and is compiled with `gcc`.

We list the runtimes for the two systems in Figure 20 for BFS and Pagerank, separating the overall runtime in preprocessing time and runtime for the computation proper. For BFS, when considering only the computation proper, Ligra is much faster than X-Stream (10X - 20X), but this performance comes at a significant pre-processing cost. Using direction reversal requires pre-processing to produce an inverted edge list, in turn requiring random access to a large data structure in order to switch edges from a list sorted by source to one sorted by destination. This pre-processing dominates the overall runtime, and is about 7X-8X that of the overall running time for X-Stream. This pre-processing time in Ligra could be improved using counting sort instead of quicksort, or by storing the reversed and sorted edge list in order to amortize the pre-processing cost. For Pagerank, X-Stream is faster than Ligra at all thread counts. Pagerank's uniform communication pattern makes direction reversal ineffective.

Finally, we analyzed the impact of the memory access patterns on instruction throughput, comparing X-Stream to the other in-memory graph processing solutions discussed above. Figure 21 shows the average count of instructions per cycle (IPC) and the total number of memory references for BFS. X-Stream shows a far higher IPC than the other implementations. In general, a higher IPC results either from a smaller number of main memory references (misses in the last level cache) or from a lower average latency to resolve memory references. For BFS, the improved IPC cannot be explained by reduced memory references alone. In the case of Ligra, Figure 21 shows that X-Stream makes more memory references and yet demonstrates a higher IPC. We therefore conclude that the higher IPC in X-Stream is due to lower latencies for resolving memory access, a result of the fact that sequential access allows the prefetcher to hide some of the memory access latency.

In summary, X-Stream's in-memory engine demonstrates that sequential access can be a winning proposition even given the relatively small gap between random and sequential access bandwidth to main memory and even when compared to specialized multi-threaded implementations of graph algorithms. We also underlined an important property of X-Stream that makes it attractive for in-memory graph processing: the fact that it can return results immediately from unordered edge lists.

Out-Of-Core We now compare the performance of X-Stream to Graphchi [37]. Like X-Stream, Graphchi is a scale-up system that can process large graphs from secondary storage. Graphchi uses the traditional vertex-centric approach to graph processing, but it uses an innovative out-of-core data structure, called parallel sliding windows, to reduce the amount of random access to disk. We used the same algorithms and graphs as reported by Graphchi [37], constraining both systems to 8 GB of memory and using the SSD for storage (as done in that work).

Threads	Ligra (s)	X-Stream (s)	Ligra-pre (s)
	BFS		
1	11.10	168.50	1250.00
2	5.59	86.97	647.00
4	2.83	45.12	352.00
8	1.48	26.68	209.40
16	0.85	18.48	157.20
	Pagerank		
1	990.20	455.06	1264.00
2	510.60	241.56	654.00
4	269.60	129.72	355.00
8	145.40	83.42	211.40
16	79.24	50.06	160.20

Figure 20: Ligra [48] on Twitter (99%CI under 5%)

	BFS [33]	X-Stream
IPC Mem refs.	0.47 982 million	1.30 620 million
	Ligra,BFS [48]	X-Stream
IPC Mem refs.	0.75 1.3 billion	1.39 1.5 billion

Figure 21: Instructions per Cycle and Total Number of Memory References for BFS

Figure 22 shows the results in terms of execution time. Graphchi needs time to pre-sort the graph into shards before beginning execution. For three out of four algorithms we used to compare against Graphchi, X-Stream finishes execution on the same unsorted graph before Graphchi finishes sorting it into shards. This result extends the observations about sorting time for the in-memory case in the previous section to the out-of-core case. Moving further, we found that X-Stream finished execution of the graph algorithm faster than Graphchi, even excluding pre-processing time. We attribute X-Stream's shorter runtimes to two factors.

The first factor is the vertex-centric approach used in Graphchi, in which updates are absorbed by vertices by executing a loop over their in-edges. This requires Graphchi to *re-sort* the edges in the shard by destination vertex after loading the shard into memory. The creation of this reverse-sorted in-memory data structure consumes a significant amount of time, reported as `re-sort` in Figure 22.

The second contributor to Graphchi's longer runtime is its incomplete usage of available streaming bandwidth from the SSD. For the graphs used in this experiment,

	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	4589.52 ± 322.28	1717.50

Figure 22: Comparison with Graphchi on SSD with 99% Confidence Intervals. Numbers in brackets indicate X-Stream streaming partitions/Graphchi shards (Note: re-sorting is included in Graphchi runtime.)

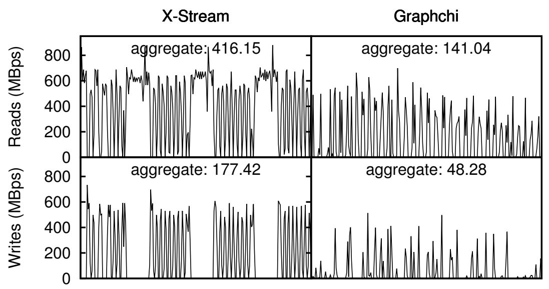

Figure 23: Disk Bandwidth

X-Stream needs only one streaming partition, by virtue of the fact that it only needs to fit the vertex data for the partition into memory. In contrast, Graphchi needs many shards, because it also needs to fit all edges of a shard into memory. This leads to more fragmented reads and writes that are distributed over many shards. Figure 23 illustrates this phenomenon with an I/O bandwidth report from the `iostat` tool. The report depicts a 4-minute interval, after the shard creation phase for Graphchi, of the execution of Pagerank on the Twitter graph. X-Stream aggregate bandwidth use is much higher than that of Graphchi. For X-Stream, the scatter phase exhibits a regular pattern, alternating between a burst of reads (from the edge file) and a burst of writes (to the update files). The gather phase exhibits a long burst of reads without any writes. In contrast, Graphchi's SSD accesses are far more bursty.

5.6 Design Decisions

First, we consider the effect of varying the number of partitions. The number of partitions needs to be large enough such that the vertex set of each streaming partition fits in the CPU cache for in-memory graphs or in main memory for out-of-core graphs. Too large a number of partitions, however, leads to excessive partitioning overhead and more random accesses. Figure 24 shows the effect of the number of partitions on the in-memory execution time for four algorithms using the RMAT-25 graph. The execution time is relatively stable for a large range of number of partitions, but increases substantially when too small or too large a number is chosen.

Second, Figure 25 shows the effect of varying the number of shuffler stages for the RMAT-25 graph with 1M (2^{20}) partitions. The figure shows *total* runtime for the same four algorithms, normalized to the runtime for a single-stage shuffler. Using a one-stage shuffler is clearly sub-optimal due to the reasons explained in §4.2. Using too many stages leads to unnecessary copying. The optimal choice in this case is a two-stage shuffle,

X-Stream automatically picks the number of streaming partitions for in-memory and out-of-core graphs, using the amount of main memory and the cache size as inputs. It also automatically picks the shuffler fanout for in-memory graphs, using the number of cache lines as input. Space constraints prevent us from presenting the algorithm for doing so, but in all cases that we have been able to verify, X-Stream succeeds in choosing optimal or near-optimal values.

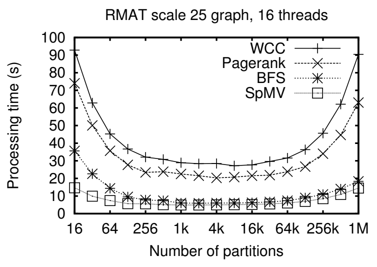

Figure 24: Effect of the Number of Partitions

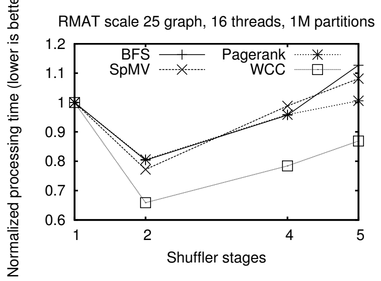

Figure 25: Multistage Shuffling

Approach	Partitions (K)	Pre-processing	One iteration	All iterations
X-Stream	$\frac{\lvert V\rvert}{M}$	Nothing	$\frac{\lvert V\rvert+\lvert E\rvert}{B}+\frac{\lvert U\rvert}{B}\log_{\frac{M}{B}}K$	$D\frac{\lvert V\rvert+\lvert E\rvert}{B}+\frac{\lvert E\rvert}{B}\log_{\frac{M}{B}}K$
Graphchi (as reported in [37])	$\frac{\lvert E\rvert}{M}$	Sorting	$\frac{\lvert E\rvert}{B}+K^2$	$D(\frac{\lvert E\rvert}{B}+K^2)$
Sorting [51] + Random Access [33]	$\lvert V\rvert$	$\frac{\lvert E\rvert}{B}\log_{\frac{M}{B}}\min(\lvert V\rvert,\frac{\lvert E\rvert}{M})$	–	$\lvert V\rvert+\lvert E\rvert$

Figure 26: Big-O Bounds in the I/O Model (U is the set of updates)

5.7 Generalization

To express X-Stream's capabilities and limitations in more general terms, we examined using the theoretical I/O model [13] the cost of propagating a message from (any) source vertex to all other (assumed reachable) vertices in a graph $G=(V,E)$, modeling label propagation in the graph. The *maximum number* of edge-centric scatter phases to complete this task 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. Fewer I/Os implies a faster algorithm. The results in Figure 26 confirm that X-Stream does well on low diameter graphs where it scales better than solutions that involve first sorting the graph. It also shows that for dense graphs, X-Stream uses fewer partitions than Graphchi uses shards and that it scales better than Graphchi on I/Os regardless of graph diameter.

6 Related Work

Exploiting sequential access bandwidth has been a long-standing theme in the algorithms community with cache-oblivious data structures [19, 28, 50]. X-Stream is not cache-oblivious, but aspires to the same goal of sequential scans over data. Another closely related area of work is stream processing. Stream processing aims to analyze unbounded information flows using only a constant amount of buffering. Specific to graphs, there has been some success on stream processing using small [$O(V\text{polylog}(V))$] space in the semi-streaming model [41], streaming just the edges, and more recently with the W-Stream model [14]. This is a good fit to streaming partitions, and we expect to implement semi-streaming and W-Stream algorithms on X-Stream as research on them progresses. X-Stream's shuffling phase also draws inspiration from work on sorting algorithms, such as polyphase merging [51].

From the perspective of disks, sequential access has been a constant theme for both magnetic disks and SSDs [16, 46, 49]. The latency of servicing random requests in SSDs can be hidden by concurrency in servicing them, a feature not available in magnetic disks. This has become a viable route to graph processing from SSDs [32, 53].

A number of distributed graph processing systems [10, 31, 40] provide scale-out solutions for graph processing. X-Stream offers the alternative of using easier-to-manage single servers. It is competitive to Graphchi, which itself is competitive to many of these distributed systems [37].

X-Stream is best suited to graphs with low diameter in relation to size. There is evidence that the diameter of real world graphs often grows only sub-logarithmically ($O(\frac{\log(V)}{\log\log(V)})$) with the number of vertices [22] or even demonstrates *densification* [38], where the diameter shrinks with new vertices joining the network. On a similar note, Backstrom et. al. [15] report that the average path length between two people in the 721 million strong Facebook social network is smaller than 5.

7 Conclusion

X-Stream is an edge-centric approach to the scatter-gather model. X-Stream uses streaming partitions to utilize the sequential streaming bandwidth of the storage medium for graph processing, scaling seamlessly across graphs stored in main memory, on SSD and on magnetic disk. We have demonstrated that X-Stream's approach is in many cases a winning proposition when compared against the traditional approach of indexing the edge list and performing random access through the index. A release of X-Stream is available at:
`http://labos.epfl.ch/x-stream`

Acknowledgments: We would like to thank Aapo Kyrola and Julian Shun for their help in benchmarking Graphchi and Ligra respectively. We would also like to thank our reviewers and shepherd as well as Ed Bugnion, Stanko Novakovic, Eiko Yoneki and Florin Dinu for their feedback that helped improve the paper. X-Stream was supported by a grant from the Hasler Foundation.

References

[1] http://freecode.com/projects/fio

[2] http://snap.stanford.edu/data/amazon0601.html

[3] http://snap.stanford.edu/data/cit-Patents.html

[4] http://snap.stanford.edu/data/soc-LiveJournal1.html

[5] http://dimacs.rutgers.edu/Challenges/

[6] http://snap.stanford.edu/data/com-Friendster.html

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

[8] http://webscope.sandbox.yahoo.com/catalog.php?datatype=g

[9] http://www.graph500.org/

[10] http://incubator.apache.org/giraph/

[11] M. Adler and M. Mitzenmacher. Towards compressing web graphs. In *Proceedings of the Data Compression Conference*, pages 203–212. IEEE Computer Society, 2001.

[12] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[13] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[14] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 540–549. IEEE Computer Society, 2004.

[15] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. Four degrees of separation. *CoRR*, abs/1111.4570, 2011.

[16] A. Badam and V. S. Pai. SSDAlloc: hybrid SSD/RAM memory management made easy. In *Proceedings of the USENIX conference on Networked Systems Design and Implementation*, pages 16–. USENIX Association, 2011.

[17] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[18] S. Beamer, K. Asanović, and D. Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 12:1–12:10. IEEE Computer Society, 2012.

[19] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *Proceedings of the ACM symposium on Parallel Algorithms and Architectures*, pages 61–70. ACM, 2007.

[20] N. Biggs. *Algebraic Graph Theory*. Cambridge University Press, 2005.

[21] P. Boldi, M. Rosa, and S. Vigna. HyperANF: approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the International conference on World Wide Web*, pages 625–634. ACM, 2011.

[22] B. Bollobas and O. Riordan. The diameter of a scale-free random graph. *Combinatorica*, 24(1):5–34, 2004.

[23] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *In Proceedings of the SIAM International Conference on Data Mining*. SIAM, 2004.

[24] R. Chen, H. Chen, and B. Zang. Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the International conference on Parallel Architectures and Compilation Techniques*, pages 523–534. ACM, 2010.

[25] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen. Kineograph: taking the pulse of a fast-changing and connected world. In *Proceedings of the ACM European conference on Computer Systems*, pages 85–98. ACM, 2012.

[26] J. Feigenbaum, S. Kannan, A. Mcgregor, and J. Zhang. On graph problems in a semi-streaming model. In *Proceedings of the International colloquium on Automata, Languages and Programming*, pages 531–543, 2004.

[27] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads

on modern hardware. In *Proceedings of the International conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48. ACM, 2012.

[28] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious algorithms. In *Proceedings of the Annual symposium on Foundations of Computer Science*, pages 285–298. IEEE Computer Society, 1999.

[29] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the conference on Programming Language Design and Implementation*, pages 212–223. ACM, 1998.

[30] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

[31] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the conference on Operating Systems Design and Implementation*, pages 17–30. USENIX Association, 2012.

[32] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single PC. In *Proceedings of the International conference on Knowledge discovery and data mining*, pages 77–85. ACM, 2013.

[33] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core CPU and GPU. In *Proceedings of the International conference on Parallel Architectures and Compilation Techniques*, pages 78–88. IEEE Computer Society, 2011.

[34] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of the Israeli Experimental Systems Conference*, pages 10:1–10:9. ACM, 2009.

[35] U. Kang, D. Chau, and C. Faloutsos. Inference of beliefs on billion-scale graphs. *Workshop on Large-scale Data Mining: Theory and Applications*, 2010.

[36] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proceedings of the International conference on World Wide Web*, pages 591–600. ACM, 2010.

[37] A. Kyrola and G. Blelloch. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the conference on Operating Systems Design and Implementation*. USENIX Association, 2012.

[38] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data*, 1(1), 2007.

[39] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[40] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the International Conference on Management of Data*, pages 135–146. ACM, 2010.

[41] S. Muthukrishnan. Data streams: algorithms and applications. In *Proceedings of the Symposium on Discrete Algorithms*, pages 413–. SIAM, 2003.

[42] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[43] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proceedings of the International conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[44] V. Prabhakaran, M. Wu, X. Weng, F. McSherry, L. Zhou, and M. Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the Annual Technical Conference*, pages 4–. USENIX Association, 2012.

[45] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the International symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[46] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[47] S. Salihoglu and J. Widom. Computing strongly connected components in Pregel-like systems. Technical report, Stanford University.

[48] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, pages 135–146. ACM, 2013.

[49] D. N. Simha, M. Lu, and T.-c. Chiueh. An update-aware storage system for low-locality update-intensive workloads. In *Proceedings of the International conference on Architectural Support for Programming Languages and Operating Systems*, pages 375–386. ACM, 2012.

[50] A. Twigg, A. Byde, G. Milos, T. Moreton, J. Wilkes, and T. Wilkie. Stratified B-trees and versioned dictionaries. In *Proceedings of the Conference on Hot topics in Storage and File Systems*, pages 10–. USENIX Association, 2011.

[51] J. S. Vitter. *Algorithms and Data Structures for External Memory*. Now Publishers Inc., 2008.

[52] D. Woodhouse. JFFS : the Journalling Flash File System. Ottawa Linux Symposium, 2001.

[53] E. Yoneki and A. Roy. Scale-up graph processing: a storage-centric view. In *International workshop on Graph Data Management Experiences and Systems*, pages 8:1–8:6. ACM, 2013.

[54] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, pages 25–. IEEE Computer Society, 2005.

[55] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. In *Proceedings of the International conference on Algorithmic Aspects in Information and Management*, pages 337–348. Springer-Verlag, 2008.