Trends in Large-Scale Graph Processing

Jasmina Malicevic
LABOS, I&C, EPFL

Abstract—Graphs represent an excellent data-structure for representing objects and relations among them. With the emergence of social networks, efficient mining of large graphs has turned these structures into an important analytics problem. The irregular structure of graphs make them hard to parallelize and solving this problem has become an interesting systems research area.

The different approaches to this problem cover in-memory processing in both distributed as well as single machine systems and out-of-core systems storing the graphs on secondary storage. In this work we provide an overview of the state of the art in each of these categories. Galois is a generic parallel processing system with a recent modification allowing for efficient shared-memory graph processing. Being a general programming model it allows greater flexibility and adaptability to certain problems than restricted DSL systems. PowerGraph represents the state of the art for in-memory distributed graph processing systems introducing a different approach to graph partitioning leading to reduced communication and memory overheads. Unlike the previous systems, X-Stream adapts its programming model to maximize the achieved bandwidth to secondary storage using the fact that sequential access to any type of storage always outperforms random access.

Finally we briefly present our own work and future directions in this area.

Index Terms—graph processing, big data, algorithms, distributed computing, out-of-core systems, shared memory systems

Proposal submitted to committee: June 5th, 2014; Candidacy exam date: June 11th, 2014; Candidacy exam committee: Prof. Anastasia Ailamaki, Prof. Willy Zwaenepoel, Dr Edouard Bugnion

This research proposal has been approved:

Date: __________________________

Doctoral candidate: __________________________ (name and signature)
(J.Malicevic)

Thesis director: __________________________ (name and signature)
(W.Zwaenepoel)

Doct. prog. director: __________________________ (name and signature)
(B.Falsafi)

I. INTRODUCTION

Efficient mining of graph structured data has attracted intense interest in the research community. From social media through science, advertising and the web, graphs as a data structure are used to represent relationships between people, facts, interests and ideas. Triggered by the high availability of graph structured data, exploring ways for extracting meaningful information from such data has been the subject of many recent research publications.

As stated in [1] graphs have specific, inherent characteristics that make them hard to process with generally applicable concurrent approaches. They suffer from an irregular and unpredictable structure, poor locality and are highly data-dependent.

Most of the graphs used in analytics problems are so called natural or power-law graphs. The peculiarity in such graphs lies in the fact that there is a power-law like distribution of connections leading to a small subset of vertices being connected to almost 50% of the graph. This is a particularly difficult problem for efficient partitioning.

In order to build an efficient system it is essential to understand these peculiarities and incorporate them in the overall system design. As shown in the case of X-Stream it is also beneficial to know the properties of the underlying hardware. The tradeoffs made in order to exploit the features of the hardware show worthy for efficient processing.

In this paper we will show improvements to a more general model, the Galois system [2]. The system tries to understand the structure of the graph and apply problem specific computation models on a general shared-memory parallel system. The work shows the benefits of supporting more general computation models over strict DSL systems. A more specific overview of how the system achieved good results in such a generic environment will be given in Section II.

Being that Galois is a single machine shared-memory system, it is limited by the size of the graph that can fit in such a scale-up shared memory system. In order to overcome such limitations, the PowerGraph [3] computation model introduces a different approach to distributed graph placement, reducing the communication and storage cost for high degree vertices. Section III provides more details of the benefits of such a partitioning scheme as well as on the three computation models provided in PowerGraph.

Another way of overcoming the limitation of available main memory and making large-scale graph processing possible on one machine with less cost, is by storing the graph on secondary storage. Unpredictable accesses during
computation and high disk access latencies have led to believe that it is inefficient to process graphs directly from disks/SSDs. However recent systems, such as X-Stream [4] have adapted their computation model and achieved results comparable to those of main memory systems. In Section IV we give an overview of this system and an analysis of how it compares to the other out-of-core systems. We will also describe an analysis of the performance of X-Stream in the cloud and an implementation of a compression scheme on top of X-Stream that showed beneficial in such a setting.

In Section V we briefly discuss the described systems, the benefits and shortcomings of each and describe a scale-out system based on X-Stream, incorporating the ideas of both X-Stream and PowerGraph offering a new approach to achieve better network throughput.

II. A GENERAL-PURPOSE INFRASTRUCTURE FOR GRAPH PROCESSING

In terms of generalizing the problem of graph computation, the Galois system introduced the notion of amorphous data parallelism (ADP). ADP is a data-centric programming model that can also be applied to regular structures but has proven to be very efficient for irregular structures which are subject of this paper. Such a programming model defines the following terminology:

- **Active vertices** present the set of vertices that are currently processed in the system.
- **Operator** is the operation being applied to those vertices. These operations can possibly be applied to multiple active vertices at the same time.
- **The Neighborhood** are the vertices affected by the changes in active vertices. The parallel execution of operators must follow neighboring constraints. That is, it must efficiently handle the overlapping of neighborhoods of active vertices.

Using this terminology, ADP can be defined as the parallelism that can be achieved when processing active vertices in parallel taking into account neighboring constraints. This neighborhood constraints are one of the main differences between Galois and other processing systems that tend to use vertex programs as the unit of parallelism. Namely, in vertex program, the neighborhood consists only of direct neighbors of an active vertex. In Galois, however, the neighborhood is not restricted and may consist of the entire graph.

This is beneficial in that it allows for easy morph operations such as adding and deleting a node. In addition to this, the model allows for implementation of different scheduling policies including priority scheduling leading to better performance on graphs with a high diameter.

The improvements to the existing Galois include:
- A priority scheduler
- A library of scalable data structures that include
  - A scalable memory allocator
  - Topology-aware synchronization
  - Code size optimizations

A. The Galois programming model

Before presenting specific improvements to the Galois system, we will first introduce the main programming principles of the system.

Application programmers work with a sequential programming language and do not explicitly handle any parallel programming constructs.

The application programmer is given a structure called an unordered-set iterator. [2] The iterator iterates over a list of active nodes. The body of the iterator specifies the operator that is being applied on the active node currently processed. The structure supports adding new nodes to the same work list if they are produced during an iteration. This enables data-dependent computations as well as changing the structure of the graph during execution by adding new nodes.

In addition to an unordered set-iterator the system provides an ordered set-iterator in order to support priority scheduling.

The parallelism consists of parallelizing the iterations. In order to support such execution, the programmer must use the library of concurrent data-structures defined in the system. These structures support a sufficient level of synchronization in order to guarantee a serializable execution. These data-structures internally acquire logical locks when reading or writing to a node or edge hence ensuring there is no conflict for overlapping neighborhoods. An important optimization in this sense is disabling locking if the application applies updates to only one machine word or does not need any form of locking(for read-only systems). In the first case, the locking is done automatically by the hardware support for transactional memory whereas in the second approach, the node is always in a consistent state. This optimization is supported by the system by setting a parameter in the iterator that indicates weather this application requires locking. Although the former requires an experienced programmer very familiar with the application it is nevertheless a useful optimization for large-scale systems.

In order to achieve efficient parallel execution, the scheduling of individual algorithm steps plays a crucial role. In the following section we will describe the scheduler supported by the Galois system.

B. Support for priority scheduling

Galois offers a library of schedulers (FIFO, LIFO, Priority..) and the application programmer specifies at a high level how the operator should be scheduled. This high-level implementation is then used by the system and transformed into a concrete scheduling implementation.

One option for priority scheduling would be maintaining a priority queue. But for graph processing tasks, where the actual
computation is very fine-grained the overhead of maintaining such queues would eventually cause a large computation overhead.

Hence, the priority scheduler was built upon an existing scheduling structure in Galois called concurrent bag. The structure of a bag is displayed in Fig 1a. Its implementation allows for concurrent retrieval of unordered tasks by taking into account load balancing among cores. As shown in Fig 1b, each core has its own buffer of tasks called a chunk. A core retrieves the tasks from its own chunk. Once the chunk is empty, the tasks are retrieved from a package, a shared data structure containing a stack of chunks. If the package is also empty, the core looks for chunks in packages shared among other cores. In this case, this core performs the search for chunks in the name of all cores associated to that package.

A simple approach for supporting priority scheduling is by using a sequence of bags. Each bag contains tasks with the same priority. However the problem with this approach is the large overhead when searching for non-empty, urgent priority bags.

An improvement that builds upon the idea of having one bag per priority is a scheduler called Ordered by Metric (OBIM).

The basic structure of OBIM is shown in Fig 1b. The implementation of the scheduler addresses the issues of synchronization in task processing by keeping a local cached version of the sequence and by distributing a bag between cores.

The global map is a data structure accessed by all cores and maps a bag to its priority. Threads execute the tasks from the bags starting with the bag that has a higher priority. It is only once all the tasks from that bag are processed that the threads start processing tasks from the next bag.

In order to know the order of the bags, each thread keeps a local cached copy (local map) of the priority mappings for the bags it is responsible for.

In order to address the problem of determining the most urgent bag OBIM implements the following consensus heuristic: locally each thread writes the priority of the bag it is currently working on. When a thread is looking for work, it scans these values for threads sharing the same package and takes the earliest priority. In addition to that, in order to propagate the priority work among packages, one thread per package scans the other packages. This way the thread determines the earliest priority it has access to. The next step is simply popping the value from the bag with this priority.

Only, when there is no work for the bags locally assigned to the thread, the thread scans the global map for additional work.

In addition to retrieving the task, each thread can push a new task as a result of the current computation. If the thread has a priority mapping for that task, it simply assigns it to the bag associated with that priority. If there is no local mapping, the thread scans the global map and if a mapping is found, the task is added to the corresponding bag. In case there is no mapping in the global map, the thread inserts a new mapping into the global map, updates its local map with the latest version of the global map and a new bag is created that matches tasks with the new priority.

C. Scalable NUMA aware memory allocation

An interesting optimization in terms of memory allocation in NUMA systems was implemented in Galois. The system offers two concurrent data structures called the slab allocator and a bump-pointer region allocator.

The former is used to allocate memory in runtime leading to faster memory allocation during execution. Namely each allocator structure keeps track of a large page pool available to the application. A certain number of these pages is pre-allocated by the application before they are actually needed to avoid allocating pages from the operating system. This page pool is NUMA aware in the sense that after releasing the page, it is returned to the memory node it was allocated from.

Each thread has a list of free fixed-size blocks from this memory pool.

In addition to the slab allocator each thread has its own instance of the bump allocator. The amount of memory allocated by this allocator is not fixed but the bump allocator is used only for allocating memory which will be needed only during the execution of a single activity. The allocated memory is backed by a page from the page pool. Once the activity finishes, the part of memory is released and all memory can be reclaimed.

D. Evaluation and Conclusion

Most existing graph processing DSLs perform computations as a vertex program. The parallelism lies in running multiple vertex programs simultaneously and synchronizing the state after a round of multiple runs. In some specific cases this has shown to be a less efficient way of graph computation. Namely, if the graph has a high diameter and is skinny, for some algorithms, such as BFS, such systems end up with many rounds of execution but little parallelism in one round.

The more general programming model supported by Galois allows for better algorithm to be written adapting to the structure of the graph. One such example is the delta stepping algorithm for determining the Single Source Shortest Path (SSSP) which leads to much faster convergence especially for graphs with high-diameter. Another example is finding the Connected Components using the union find data structure which is not a vertex program. The programming model supports both coordinated as well as autonomous scheduling and enables these options to be set by the application developer.

This also enables for implementing existing DSLs on top of Galois being that they are more specific cases of the general system.
A comparison of finding the SSSP using the delta stepping algorithm which requires priorities and the Bellman-Ford implementation in PowerGraph using vertex programs is shown in Figure X. The first graph shows performance on a low diameter graph, twitter50[8]. But when running the same algorithm on a high-diameter graph such as the US road network the improvement by using autonomous scheduling and a better algorithm achieves speedups of over 1000X. The figure compares the speedup when using Galois over PowerGraph. It is though worth noticing that even though PowerGraph exploits thread parallelism, it is designed as a distributed system, hence on one machine it will have degraded performance due to the specific partitioning schemes optimized for a distributed environment.

III. DISTRIBUTED GRAPH-PARALLEL COMPUTATION

The infrastructure described in II argues that supporting only vertex operators is too restrictive when general purpose graph analytics are needed. By general-purpose they refer to a system with high diversity in both algorithms as well as the type of graphs supported. And truly in some cases the abstractions used in most systems have shown to have better performance if implemented on top of a more general infrastructure.

However, in a distributed setting where the graph has to be partitioned in order to support efficient parallelism, the tradeoff of a computation overhead in such single cases is made.

After demonstrating the success of vertex programs in Pregel[9], graph processing systems have since implemented this model in their frameworks. In such systems the parallelism lies in parallel execution of a vertex program on each vertex in the graph. The different vertex programs interact through some predefined model of communication. This computation model can be expressed through an abstraction called the GAS model of graph computation.

The GAS model consists of three phases: Gather, Apply and Scatter. During the gather phase, each vertex program gathers data from the neighboring vertices. The information is being applied to the central vertex during the apply phase. If there was an update in the vertex state generated during the apply phase, the update is being propagated to the neighboring vertices during the scatter phase.

The gather and scatter phase define the fan-in and fan-out of a single vertices. These values will be high for high degree vertices causing a communication bottleneck and in some

```java
interface GASVertexProgram{
    //run on gather_nbrs(u)
    gather(D_{u,\alpha \leftarrow \Delta a}, D_{\alpha \leftarrow \Delta a}) \rightarrow Accum
    sum(Accum left, Accum right) \rightarrow Accum
    apply(D_{u, Accum}) \rightarrow D_{\alpha \leftarrow \Delta a}
    //run on scatter_nbrs(u)
    scatter(D_{\alpha \leftarrow \Delta a, D_{u, D_{\alpha \leftarrow \Delta a}}} \rightarrow (D_{\alpha \leftarrow \Delta a, \text{Accum}})
}
```

**Figure 3** The interface of a vertex program in the PowerGraph programming model

specific systems a storage overhead since the vertices need to keep track of the neighborhood.

A. The PowerGraph approach to GAS

The PowerGraph[3] abstraction allows for decomposing the vertex program by distributing the GAS steps. Each vertex program is defined by the functions gather, sum, apply and scatter as shown in Fig. 3. Each of the functions is invoked by the PowerGraph engine according to the algorithm in Fig. 4. Using these semantics one vertex program can be distributed across multiple computation units. It is the idea of distributing the vertex program itself and the implementation of vertex cuts (sub-section C) that has caused PowerGraph to stand out in the graph processing community.

1) Computation model

In such a setting the gather and sum functions are used to collect information about the neighborhood. The result of these two functions is stored in a local accumulator $a_u$ as shown in Fig. 4.

The apply phase then takes this accumulator and computes a new value for the active vertex.

Once the results of the gather phase are applied to the vertex, in case there was a change, the new value is propagated to the neighboring vertices in the scatter phase.

PowerGraph introduces an optimization to this process called **delta caching**. Namely each copy of the vertex keeps its local accumulator value in the cache. If the gather phase did not cause for a change in the accumulator for that subset of edges there is no need to invoke the same phase in the subsequent iteration because there are no new input values coming from those neighbors. The cached accumulator value is simply reused in the apply step.

The PowerGraph engine keeps track of active vertices on which the vertex program will be executed. The user can
explicitly declare vertices as activated adding them the engines' list of active vertices. But it is the engine itself that determines the exact time when an operator will be applied on those vertices. PowerGraph allows for both synchronous and asynchronous execution adapting, like Galois to certain algorithms.

The synchronous engine executes each of the sub steps of a vertex program one after another and changes to both vertex and edge data are committed after completion of each of these sub steps. However, the operator is not applied to the newly activated vertices until the completion of the entire vertex program. As already discussed in Section II, such frequent barriers can cause slow convergence of certain algorithms especially for graphs with high-diameter.

Therefore PowerGraph allows for two types of asynchronous scheduling. If the application permits and the programmer is experienced, the system can be ran entirely asynchronously meaning that an operator is executed on active vertices as soon as the data and resources become available. In such a system the changes made to graph data are committed during the execution of the sub steps.

Since this type of execution can lead to a high degree of non-determinism PowerGraph offers the option for asynchronous execution with guaranteed serializability. This is done by acquiring parallel locks over adjacent vertices preventing the operator be applied to neighboring vertices. This system is adapted for the partitioning scheme of PowerGraph (see subsection 2) where each replica of a vertex lock only the local edges on the single machine. The computation starts once all the replicas reach a consensus that they have succeeded and acquired locks for their local edges.

2) Vertex-cuts

For power-law graphs it is essential to find a good partitioning scheme that does not lead to bottlenecks in communication and storage for high degree vertices. The vertex-cut implementation in PowerGraph and the analysis following that work shows that for large, power-law degree graphs, the standard edge-cut partitioning scheme is not the optimal solution.

In the standard approach to graph partitioning the vertices are distributed over a set of machines performing edge cuts in the graph. During the execution of a vertex program we have to iterate sequentially over its neighborhood in order to gather the necessary information. In standard message passing algorithms this generates a lot of messages that need to be propagated along the edges whereas in shared-state systems it leads to storing a substantial amount of meta-data and neighborhood information which eventually saturates a single machine for high-degree vertices. The overall network communication can thereby be expressed as a function of edge-cuts. But most frameworks cut randomly which in power-law graphs leads to a large percentage of the graph being cut.

The PowerGraph abstraction takes a different approach. They take high degree vertex and split it across machines whereby declaring on of the vertices to be a master vertex and the remaining vertices are called mirrors. The edges are then evenly distributed onto these machines. The gather is then executed entirely in parallel, computing for example partial sums in the case of Pagerank. This sum is then sent out to the master. It is applied to the master vertex followed by sending this new value to the mirror vertices. The scatter phase can also be computed entirely in parallel since each mirror triggers only its local neighbors by updating edge metadata and triggering neighboring vertices to update their values.

This leads to substantially less data that needs to be synchronized.

As a further optimization to this model, the system offers three approaches to vertex cuts.

Random vertex cuts evenly assign the number of edges to the available machines without any additional considerations.

Greedy vertex cuts assign edges to machines which already have one of the vertices of the edge that is currently being assigned. If more than one machine contain one of the vertices, the edge is assigned to the less loaded one in order to ensure balance. There are two implementations of this approach: coordinated and oblivious. During coordinated greedy vertex cuts each machine coordinates on the history of the vertex cuts across all other machines before assigning edges to machines. This approach leads to optimal cuts but comes at a high preprocessing cost. As an alternative, each machine locally remembers the history of vertices assigned to it and tries to find the best fit for the upcoming edge. This is the main principal of oblivious greedy cuts.

B. Evaluation and Conclusion

The PowerGraph abstraction implements many machine learning techniques and algorithms from collaborative filtering, to graph analytics such as PageRank, Shortest Path etc.

Fig. 5 shows the improvements of Powergraph over current distributed graph processing systems in terms of

![Figure 5 Comparison of PowerGraph to other distributed graph processing systems](image)

![Figure 6 Sensitivity of vertex-cuts to changes in the power law degree of the graph compared to the sensitivity of two other systems](image)
communication and runtime. In both cases PowerGraph was able to outperform both of the systems.

In terms of scalability, PowerGraph was able to process the Yahoo Altavista WebGraph[10] (1.4 billion vertices, 6.6 billion edges) in 7 seconds per iteration.

As proof for the efficiency of vertex-cuts over edge cuts Fig. 6 shows the sensitivity of different abstraction towards the power-law degree of the graph. The evaluation is done for a synthetic scale-free graph where a lower α represents more high-degree vertices. The tests clearly show the robustness of PowerGraph to the presence of high degree vertices.

This is an important feature of PowerGraph which we integrated in our work on a scale-out system as well.

IV. AN OUT-OF CORE APPROACH

The size of graphs that are of interest today by far exceeds the main-memory capabilities of common single machines. Therefore systems such as the one described in Section III were considered the only acceptable approach for efficient processing of large-scale graphs. The main reason were the unacceptable latencies of secondary storage devices.

As can be observed from Sections II and III the current systems access vertex and edge data in a random manner which performs badly on any kind of external storage devices (SSD or HDD).

Therefore, the standard vertex-centric approach used in systems so far was not acceptable for running from disk and thus overcoming the barrier of available storage on a single machine was not possible.

Recently a new programming model called X-Stream was introduced making access to graph data sequential. The main contributions of such a system come in the form of an edge-centric scatter-gather programming model and a different way of partitioning the data called streaming partitions.

A. Edge-centric scatter-gather

The scatter-gather approach as described above has been the standard of computation from the Pregel[9] system in 2010 up until the most recent, PowerGraph[3]. The pseudo-code for the standard scatter-gather is considered vertex-centric as can be seen from Fig. 7. Here we iterate through the vertices propagating updates through each outgoing edge of that vertex. Following the logic that the number of vertices in power-law graphs is smaller than the number of edges X-Stream offered a different approach to the scatter-gather model shown in Fig. 8. Here, instead of iterating through the vertices, we scan the edges and if the source vertex change, we propagate the update to the destination vertex of an edge.

However, in this scenario, the vertex set is accessed randomly. In order to eliminate random access to secondary storage X-Stream stores the vertex set in main memory where random access does not come at a great cost.

This comes with the overhead of always scanning the entire edge list from storage whereas the standard approach only accesses the edges that are needed at the moment.

But as is shown in the X-Stream runtimes, the improved bandwidth to storage used with sequential access makes up for the overhead in the amount of data we are actually reading in.

B. Streaming partitions

As described above, the solution for random access to vertices is to store them in main memory. But as the target application of X-Stream is mining of large graphs, the vertex set often won’t fit in the available main memory causing the need for graph partitioning.

X-Stream introduces the notion of streaming partitions where the size of one streaming partition is bounded by the amount of RAM memory available. The only constraint for a partition is that all the vertices in that partition have to fit in available RAM. This is followed by partitioning the edges so that one partition contains all edges whose source vertices are in that partition. X-Stream has no additional requirements as to the quality of partitioning enabling random partitioning without any additional preprocessing cost. This is a generally applicable system and can also be applied to single machine shared memory systems where the vertices from one partition fits into CPU caches and the edges are streamed from main-memory.

The partition consists of an edge list, an update list and a vertex set.

The vertices in each partition are a disjoint subset of the entire vertex set whereas the partition contains edges whose source vertex is in that partition and updates whose destination vertex belongs to the partition. This partitioning scheme makes it possible to entirely parallelize the processing of disjoint partitions.

C. Scatter Gather with Streaming Partitions

The gather and scatter phases in X-Stream have streaming partitions as the main level of abstraction over the graph structure.

During the scatter phase the program scans the edgelist and in case its source vertex generated an update, the new value is being appended to an update list.

The gather phase in X-Stream is equal to the gather and apply phases in PowerGraph. Here the program sequentially scans the update list and applies the updates to the corresponding vertices.

The application of updates to vertices as well as checking for updates during the scatter phase are straight forward being that all the data needed for the computation belongs to the same partition.

1) The shuffle phase

The only problem is that the destination vertex of an edge does not necessarily belong to the same partition. In this case, X-Stream introduces an additional phase into the scatter phase. In order to avoid frequent write-outs to disk, the updates are appended to an in memory buffer called the stream buffer. The

for each vertex v
if v has update
for each edge e from v
scatter update along e

Figure 7. Vertex-centric scatter-gather

for each edge e
if e.src has update
scatter update along e

Figure 8. Edge-centric scatter-gather

for each edge e
if e.src has update
scatter update along e

for each vertex v
if v has update
for each edge e from v
scatter update along e

Figure 7. Vertex-centric scatter-gather

for each edge e
if e.src has update
scatter update along e

Figure 8. Edge-centric scatter-gather
goal of such a structure is to append data to the update files of the partitions that contain the destination vertices.

Since memory allocation and de-allocation can be expensive the stream buffer is implemented using two statically allocated arrays: the chunk array and a corresponding index array. The chunk array is an array of bytes containing data generated during the scatter phase. The index array contains one entry per streaming partition and points to the chunk in the chunk array corresponding to that partition. The programming model therefore writes updates to another statically sized in-memory buffer. Once this buffer becomes full, the updates are sliced up using the index array to corresponding chunks in the chunk array and written out to disk.

Since these are all in memory structures, the communication to disk is delayed until the stream buffer becomes full.

D. Optimizations for efficient Disk I/O

Aside from the sequential access pattern to secondary storage, the X-Stream processing model supports several other optimizations in order to achieve maximum disk bandwidth.

X-Stream issues asynchronous direct I/O requests to disk. In order to keep the disks busy, the system supports prefetching by reserving two output and two input buffers.

While the data is being fetched into one input buffer, the other buffer is used for performing computation on the data that has previously been fetched.

Analogous to this, while data is being written out from one output buffer, the other buffer is being used for storing current updates generated during the scatter phase making it possible to start writing to disk as soon as the first buffer has been written out. This has shown to keep the disks a 100% busy [4].

As X-Stream has separate input and output buffers, the scanning of edges and write out of updates could be parallelized by having separate threads write to different disks. This would cause no overlap between the reads and writes for one partition and has shown to cause speed-ups of 30%.

The sequential access to disk enables the system to fully exploit RAID systems striping the data across multiple devices. The experiments performed in [4] show a speedup of 50% when using RAID arrays.

This access pattern is also convenient for SSDs which have shown to have worse performance for random writes due to write amplification. Most SSDs also support the TRIM function which causes unused blocks of data to be marked as free. X-Stream truncates the files that are not used which is usually translated into a TRIM command for SSDs. Another benefit is that, since all the files were sequentially written, consecutive memory locations are being ‘fired’.

1) Compressed Disk I/O

X-Stream supported the zlib[5] compression scheme for compressed I/O. Due to the large computation overhead that zlib produces, we implemented another compression library, Google Snappy[6].

The compression is an optional parameter in the system and our experiments have shown the choice of the compression scheme has impact on performance.

The efficiency of the compression scheme depends on the compression ratio, the compression/decompression speed and the speed of the underlying infrastructure.

On local machines, using zlib compression had a negative impact on the overall computation time. We believe that this is due to the fact that zlib handles data at ~200MB/s whereas the SSDs used in our system transmits data at up to 600MB/s. Snappy offers a trade-off between speed and compression ratio and has shown little or no improvement on local system but no degraded performance.

What we saw as interesting was the impact of compression when running in the cloud and the graph was stored to network attached storage. As stated in [7] the interfering network now becomes a factor in the overall runtime and decreasing the amount of data that needs to be sent over the network showed improvement.

We tested two sets of graphs. The synthetic graph which had a small compression ratio and the twitter graph which was compressible due to the ordering of the vertex IDs.

For the synthetic graph, Snappy showed an improvement in performance of approximately 12% even for an initially uncompressible edge-list.

On the other hand, for the twitter graph, using zlib showed speedups of about 30%. This also led to cost savings in the cloud.

E. Conclusion and Evaluation

X-Stream was compared to GraphChi[11], another system that has achieved good performance when strong graphs on secondary storage. However, Fig. 9 shows that X-Stream is able to achieve a better performance.

These are two fundamental differences between these two systems which also led to such a difference in performance.

GraphChi has a preprocessing step in order to sort the edges whereas X-Stream uses random edge splits. This preprocessing cost tends to be high and sometimes it is higher then the overall computation time of X-Stream.

But given that some applications may perform this step only ones, X-Stream was also compared itself to the runtime of GraphChi without the preprocessing time. The reasons for such speedups in X-Stream are that GraphChi needs edges, vertices and updates of a shard to fit into main memory. For large graphs this causes for more partitions leading to lower utilization of the available bandwidth to storage and more

![Figure 9 X-Stream speedup over Graphchi (without preprocessing time)](image)
computation steps.

In terms of scalability, the process of scaling-up using X-Stream is shown in Fig. 10 showing that the system was able to process a synthetic graph with 4 billion vertices, 64 billion edges in 26 hours.

V. DISCUSSION AND CURRENT WORK

All three systems described above have introduced interesting features for large-scale graph processing.

The arguments expressed in Section II for supporting more general queries is a valid argument. There are algorithms that just cannot be implemented as vertex programs or take too long to converge in bulk synchronous programming models.

X-Stream for example has noted poor performance for traversal algorithms on high-diameter graphs such as the Yahoo Web Graph.

Even though PowerGraph offers a solution to this problem implementing asynchronous scheduling it lacks the support for priority scheduling which enables the implementation of better algorithms for certain cases.

However, it is hard to implement the Galois unrestricted neighboring principle on a distributed environment with many nodes and a large graph.

On the other hand, even though X-Stream is also built upon the scatter-gather model, it easily supports morphing operations due to the streaming model it implements.

Our current work consists of combining the performance of X-Stream by using the vertex-cut algorithm explained in PowerGraph in order to support scale-out systems that store graphs on secondary storage.

The reason for this is that scaling out using main memory is costly and requires very large clusters which are not always available.

The system adapts to the available storage using X-Stream’s support for main memory if the vertex set of the graph fits in main memory or streaming from disk in case the graph is too large.

The full utilization of available bandwidth to disk in combination with a unique network barrier protocol for efficient communication among the nodes achieves impressive performance compared to current scale-out systems.

We are working on measuring the impact of compressed disk I/O but also network communication in such a system as well as adding support for more machine learning algorithms.

It would be interesting to measure the performance on high-diameter graphs in such a setting and compare it to the general approach explained in Section II.

Eliminating the edges that are not needed when scanning the edgelist in X-Stream would lead to better performance. There is recent work offering the application programmer to choose between the standard streaming model in X-Stream and an indexed approach similar to that of edge-centric systems.

Another interesting direction for future work is the exploration of energy efficiency of scale-out systems using secondary storage exploiting the feature of X-Stream to utilize multiple SSDs on one machine.

REFERENCES