L-Graph: A General Graph Analytic System on Continuous Computation

Weixiong Rao
School of Software Engineering
Tongji University, China
wxrao@tongji.edu.cn

Eiko Yoneki
Computer Laboratory
University of Cambridge, UK
eiko.yoneki@cl.cam.ac.uk

Lei Chen
Department of Computer Science and Engineering
Hong Kong University of Science and Technology
leichen@cse.ust.hk

ABSTRACT

Massive graph analytics have become an important aspect of multiple diverse applications. With the growing scale of real world graphs, efficient execution of entire graph analytics has become a challenging problem. Recently a number of distributed graph processing systems (Pregel [6], PowerGraph [1], Trinity [8]) and centralized systems (GraphChi [2] and XStream [7]) have been designed. Compared with high expense of distributed systems deployed on a cluster of commodity machines, the centralized systems on cheap PCs are very attractive propositions with low expense and comparable performance. By careful analysis, we fin that (*i*) the graph computation abstraction in the centralized systems inherently adopted a *batch* model similar to the distributed systems. The batch model could lead to suboptimal performance. (*ii*) The execution model in the centralized systems advocates sequential operations on Solid State Disk (SSD) which are still slower than memory-based operations.

In order to tackle the above efficiency issues in centralized systems, we firs propose a novel *continuous* graph computation abstraction. This model continuously processes edges and updates computation results. It allows much faster convergence than the batch model. Second, we propose to maintain vertex states in memory and advocates memory-based operations for much faster I/O operations than sequential operations on SSD. Finally, we design an adaptive memory layout to minimize overall I/O cost. We develop a proof of concept prototype L-Graph and implement four example graph analytic applications atop L-Graph. Preliminary evaluation on real and synthetic graphs have verifie that the proposed continuous model greatly performs the widely used batch model and L-Graph can achiever much higher efficiency than the state of arts GraphChi [2].

1. INTRODUCTION

Graph analytics are becoming increasingly important for numerous applications, ranging across the domains of bioinformatics, social networks, computer security and many others. Such domains frequently require the analysis of an entire graph and identify interesting patterns. Unfortunately, the growing scale of real world graphs has made efficient execution of entire graph analytics very challenging, especially when external storage is utilized. Poor locality of access [5] leads to high I/O overhead caused by slow random access to traditional magnetic disks.

To address this challenge, a number of distributed graph processing platforms, such as Pregel [6], PowerGraph [1] and Trinity [8], have been designed. In these distributed systems, each machine maintains its portion of the graph into fast memory. By updating their own portion of the graph and broadcasting the changes, the machines then work cooperatively to offer a distributed shared memory approach.

Compared with high expense to build the above large scale distributed systems on a cluster of commodity machines, some recent centralized systems GraphChi [2] and XStream [7] work on a single machine. They rely on the use of external storage, e.g., Solid State Disk (SSD), to perform the computation on a single machine. These systems often exhibit reasonable runtime performance but at a fraction of the expense compared to the clustered machines.

Unfortunately, we show that the centralized systems still suffer from the following performance issues. (*i*) **Suboptimal performance**: The key of general (distributed or centralized) graph systems is graph computation abstraction. The gather/scatter abstraction and its variations have been widely used, such as synchronous gather/scatter model in Pregel and asynchronous gather/scatter model in GraphLab [3, 4] and PowerGraph [1]. The distributed systems like Pregel, GraphLab and PowerGraph gather incoming messages from neighbor vertices, next process a bunch of gathered messages together to invoke user define functions (UDFs), e.g., the function used to compute PageRank, and finall scatter the updates of computed results also as a bunch of messages to outgoing neighbors. The batch manner avoids sending a piece of message for each update, and consequently saves network bandwidth in a distributed setting. We will show that centralized systems GraphChi and XStream inherently adopt the batch manner to implement the gather/scatter abstraction. Due to the significant ly difference between the distributed and centralized settings, the batch manner could lead to suboptimal performance in centralized systems, which will be verifie in our experimental results.

(*ii*) **Relatively slower SSD-based operations**: Given limited memory of a single machine, the centralized systems cannot load an entire graph into memory, and frequently use external storage, e.g., SSD. Since sequential access to SSD can offer much faster I/O throughput than random access, GraphChi and XStream *sequentially* read graphs from SSDs and *sequentially* write computation results back to SSDs. The SSDs become an intermediate media to propagate the update of computation results from a vertex to an-

HotPlanet'15, September 7, 2015, Paris, France.
© 2015 ACM. ISBN 978-1-4503-3534-8/15/09 ...$15.00.
DOI: http://dx.doi.org/10.1145/2798087.2808077.

other. Sequential write to SSD, though faster than random write, is still slower than memory-based read and write. We believe that memory-based operations provide a new opportunity to offer higher computation efficiency, instead of SSD-based sequential write.

To address the above challenges, in this paper, we design a general graph analytical system, namely L-Graph, on SSD-based single machines, with the following contributions.

- We propose a new *continuous* graph computation abstraction. The novelty of the abstraction is to continuously execute the UDFs, incrementally update vertex states (a.k.a computation results, such as PageRank of a vertex), and fi nally acquire the correct computation result when an entire iteration of graph processing is completed. The continuous style can help to quickly propagate the updated vertex states for faster convergence.
- For the graph execution model, L-Graph advocates the maintenance of vertex states inside memory. The updates of vertex states can be immediately seen even inside the same computation iteration. Furthermore, L-Graph adopts memory-based updates, instead of sequential write onto SSD, to quickly propagate updated vertex state. Finally, we design an efficien memory layout scheme to minimize the overall I/O cost, when main memory can maintain a subset of vertex states only.
- For demonstration, we implement four example graph analytic applications atop L-Graph: Weakly Connected Components (WCC), PageRank, graph traversals and Single Source Shortest Path (SSSP). The implementation optionally uses a user define iterator (UDI) to improve graph traversals and SSSP for faster running time.
- We develop a proof of concept prototype and perform extensive evaluations on both real and synthetic graphs. The experimental results verifie that the proposed continuous model greatly outperforms the widely used batch model and L-Graph can achieve much faster running time than the state of the art GraphChi.

The rest of this paper is organized as follows. Section 2 reviews the preliminaries and related works. Section 3 introduces the continuous computation model, and Section 4 gives the execution engine. After that, Section 5 evaluates L-Graph. Section 6 finall concludes this paper.

2. PRELIMINARIES AND RELATED WORK

In this section, we briefl review the widely used gather/scatter graph computation model and give a quick overview of the state of art GraphChi and recent work XStream.

2.1 Graph Computation Abstraction

Given the diverse graph analytic applications (including graph traversals, PageRank and shortest path), general graph analytic systems frequently require a graph computation abstraction and expose application programming interfaces (APIs). Application programmers next use such APIs to comfortably implement various graph applications with several lines of codes. The vertex centric gather/scatter model in Alg. 1 has been widely used by the state of art systems Pregel, GraphLab, PowerGraph, GraphChi and XStream.

The vertex centric gather/scatter model maintains a state of computation result in each vertex. We denote such a state to be a *vertex state*, e.g., the computed PageRank of a vertex. The main loop (lines 1-3) alternatively runs through an iterator over vertices that need to scatter vertex states by a *scatter* function and another iterator over those that need to gather states by a *gather* function. The *gather* function (lines 4-8) takes the updates on an input vertex's incoming edges to update the state of the vertex. The *scatter* function (lines 9-13) takes an input parameter vertex to generate updates to be propagated along outgoing edges.

Algorithm 1: Vertex centric gather/scatter computation model

```
1  while not done do
2      for all vertices v in GatterIterator do gather(v)
3      for all vertices v in ScatterIterator do scatter(v)
4  Function gather(v) begin
5      for all edges e into v do
6          v = User_Defined_Gather v.e.update)
7          if v.need_scatter then scatterIterator.add(v)
8  end
9  Function scatter(v) begin
10     for all edges e out of v do
11         e.update=User_defined_scatter v,e)
12         if e.update.need_gather then GatherIterator.add(e.dest)
13 end
```

A fairly large number of graph computation abstractions can be mapped to the vertex-centric gather/scatter model,including Bulk Synchronous Parallel (BSP) abstraction in Pregel [6], Asynchronous Iterative Computation (AIC) abstraction in GraphLab, and a hybrid of BSP and AIC in PowerGraph. The centralized system GraphChi supports the asynchronous vertex-centric scatter/gather model, and the recent one XStream is an asynchronous *edge*-centric gather/scatter model, though not vertex-centric any more.

2.2 GraphChi and XStream

GraphChi [2] was the firs system to explore the idea of using sequential operations to external storage (SSD) for high I/O throughput. The key of GraphChi is specially designed shards (or partitions) of edges on SSDs. A shard, associated with an interval of vertices, stores all the edges that have destinations inside the interval. The edges in such a shard are sorted by their source. Based on this design, GraphChi processes the shards one by one. When fully loads the subgraph of a shard *p* to memory via sequential read, GraphChi invokes a UDF to update vertex states of destinations. Due to the sorted edges ordered by source in all shards, the out-edges for the vertices in *p* are stored in *consecutive* chunks in all other shards. This indicates that the updates of the vertex states in *p* are all sequentially written to the other shards. Such updates are then seen during the processing of the remaining shards. In this way, with the help of sequential read and write, GraphChi uses the SSD as medium to propagate the updates of vertex states, avoiding random access to external storage.

By using streaming, XStream [7] proposed an edge centric graph computation model. Nevertheless, it still follows a gather/scatter model to process the edges. For example, edges are all processed before the updates are all gathered. Due to streaming partitions, XStream removes the necessity for pre-processing and builds an index that causes random access into the set of edges. However, as reported in [7], a severe problem of XStream is the wasted I/O cost of streaming the entire edge list but contributing little to graph processing when an input graphs is associated with a high diameter.

3. CONTINUOUS MODEL

In this section, we describe the continuous computation model. Section 3.1 motivates the model, and Section 3.2 gives the detail.

3.1 Motivation

By a motivation example, we will show that the batch-based implementation of the gather/scather models, adopted by many distributed and centralized systems, could delay the prorogation of vertex states and lead to slow convergence.

Algorithm 2: WCC_UDF (int u, int v, floa w, int $s(u)$, int $s(v)$)

```
if $(s(u) > s(v))$ then return $s(u)$;
else return null;
```

Consider that the label propagation algorithm [10] is used to fin Weakly Connected Components (WCC) in an input graph. Each vertex is initiated with its vertex ID (i.e., the label) as the initial vertex state. In the following iterations to process an edge $\langle u, v \rangle$, the algorithm executes the UDF define in Alg. 2. The input parameters of the UDF include the endpoints of an edge $\langle u, v \rangle$, the edge weight w (note that WCC does not use the weight w, yet w could be used by other applications such as PageRank), and the associated vertex states $s(u)$ and $s(v)$. If the label $s(v)$ of destination v is smaller than the one $s(u)$ of source u, the UDF returns the larger one $s(u)$, and otherwise null. Depending on the returned result, WCC then updates the vertex state $s(v)$ by the non-null result. Thus, a larger label is propagated from source u to destination v.

Next, we defin the convergence of WCC as follows.

Definitio 1 ***Convergence of WCC****: The vertex state of v, denoted by $s(v)$, becomes convergent, if the vertex v is updated by the largest label L_i among all source vertices pointing to the vertex v. If and only if all vertex states in a graph are convergent, we then say that WCC becomes convergent.*

Based on the above UDF and convergence, we show that the synchronous and asynchronous computation abstractions (which are used by Pregel and PowerGraph, respectively) inherently follow a batch style to execute WCC.

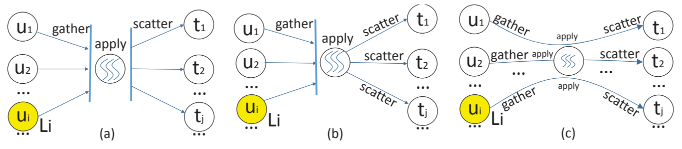

Figure 1: Three computation abstractions: (a) Synchronous, (b) Asynchronous, (c) Continuous

First in Alg. 1 (a), in a vertex v, the synchronous BSP model (used by Pregel) gathers messages from neighbors $u_1, u_2 ... u_i ...$ (see the *gather* function), processes a bunch of received messages as a batch to execute the *apply* function (which next invokes the user define functions UDFs), propagates the updated vertex states by sending a bunch of outgoing messages to outgoing neighbors $t_1, t_2 ... t_i ...$ (see the *scatter* function). The batch manner avoids processing each piece of the update and sending too many messages, and helps optimizing the distributed solutions for less network traffi and faster processing time. In Pregel, the batch style is implicitly implemented by using the barrier between supersteps.

In Alg. 1 (b), the asynchronous model has been used by PowerGraph, GraphLab and GraphChi. We take PowerGraph as an example. Besides BSP, PowerGraph also supports asynchronous execution which does not execute the *gather*, *apply* and *scatter* phases in strict order. Instead, the asynchronous style immediately commits the updates made to vertex states during the *apply* and *scatter* functions to the graph, and such updates are visible to subsequent computation on neighboring vertices [1]. However, the *gather* function still adopts the batch style to gather a bunch of received messages.

We show that the above batch manner could delay the convergence of WCC. Recall that both synchronous and asynchronous models require the batch manner to gather received messages from the sources. Thus, if and only if the *gather* function is executed to gather all such labels from the sources, the largest label, say L_i, among all sources to v can be found, i.e., the vertex state $s(v)$ becomes convergent (see Def. 1).

Now we consider the case that the message containing the label L_i from source S_i is received to v in a very early moment. At such a moment, if the WCC_UDF is executed, the state $s(v)$ then becomes convergent, even though the labels from other sources are still not received and processed. Next, if the update to $s(v)$ is prorogated from v to its neighbors (no matter whether labels from the sources to v are received to process or not), we have chance to greatly speedup the prorogation throughout graphs. Intuitively, the prorogation of the largest labels can be treated as a broadcasted throughout the graph. It significantl differs from the batch manner, and could greatly speedup the convergence of WCC. Instead, the batch-based manner needs to receive all labels from sources (before the convergence of s_i) and delays the convergence.

3.2 Continuous Computation Model

In this section, we propose a continuous model to make vertex states quicker convergent than the batch model does.

Algorithm 3: Continuous Computation Model

```
while not done do
    for all edges e in User_Defined_Ite ator do
        returned_val = Process(e);
        if returned_val ≠ null then e.dst = returned_val;

Function Process(e) begin
    if e.src has_updates then return UDF(e.src, e.dst, e.wgt)
end
```

Definitio 2 ***Continuous Computation Model****: Whenever an edge $\langle u, v \rangle$ and the current states $s(u)$ and $s(v)$ are gathered to be available in memory, the vertex state $s(v)$ can be updated immediately by the invoke of UDF.*

Based on the above definition when those edges with v as destination are continuously gathered to be available in memory (unnecessarily processed as a batch), the vertex state $s(v)$ is also continuously updated. We describe how the continuous model can be used for the graph processing. In Alg. 3, there exist a user define function (UDF) and a user define iterator (UDI). Each graph application is required to explicitly implement a specifi UDF, and optionally implement a UDI if necessary and otherwise a default UDI is used. The UDI can schedule the invoke of a UDF. The key point of Alg. 3 is that the UDF over a vertex v is invoked to update the vertex state $s(v)$ as soon as the edge $\langle u, v \rangle$ is loaded to be available in memory. Thus, for all edges e, Alg. 3 continuously invokes the UDF to update the vertex states of the destination of edges e.

Again we use WCC as the running example to show the continuous model's benefits

- *Faster convergence*: When the edge $\langle u_i, v \rangle$ associated with the largest label L_i is gathered to process in the earliest moment, the vertex state $s(v)$ can be updated by L_i and WCC has chance to become fastest convergent.

- *Low memory consumption*: The batch model requires that all edges pointing to v are gathered to be available in memory. This indicates high memory consumption to buffer such edges. Real graphs frequently exhibit power-law degree distribution, and the vertices with very high degrees require high memory to buffer such edges. Instead, when an edge is loaded into memory, Alg. 3 immediately invokes UDF without such buffering overhead.

- *Trivial preprocessing overhead*: Recall that GraphChi requires nontrivial preprocessing overhead to organize special graph shards by sorting edges by source vertices (and also by the associated destination vertices), such that all edges pointing to a vertex v can be loaded to process together as a batch. Instead, the continuous model does not require any sorting of the edges.

4. EXECUTION ENGINE

This section describes the centralized engine to implement the continuous model on a SSD-based single machine. We firs briefl introduce the steps to process an input graph (Section 4.1), and next highlight the challenges and then gives a solution (Section 4.2).

4.1 Processing Steps

For a graph $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, the edges $\mathcal{E}$ are rather voluminous. It is common that the memory of a machine cannot fully load the edges $\mathcal{E}$. L-Graph internally organizes $\mathcal{E}$ into edge blocks, and an edge is uniquely assigned to an edge block. A compression algorithm over the blocks, such as Zip, can be used to save the I/O cost of loading the blocks.

Next, L-Graph similarly organizes vertices $\mathcal{V}$ into blocks. Here we firs make an assumption that all vertex states can be maintained in nowadays main memory (Next section will give a solution that main memory cannot fi all vertex states). Based on the assumption, L-Graph directly maintains an array of vertex states in memory. Still using WCC as the motivation example, we show the steps of L-Graph to execute WCC_UDF define in Alg. 2. (*1*) *Loading Edges*: By sequential access to SSD, L-Graph loads edge blocks to main memory. Using multitheading requests can help achieving maximal I/O throughput. (*2*) *Invoking WCC_UDF*: When an edge $\langle u, v\rangle$ is available in memory, L-Graph invokes WCC_UDF with the vertex states $s(u)$ and $s(v)$ and weight w as input parameters. Here, the states $s(u)$ and $s(v)$ associated with the two endpoints u and v can be directly read from memory (i.e., the labels of u and v). (*3*) *Convergence of WCC*: When all edges have been loaded to process, L-Graph finishe one iteration of the graph processing. One iteration can ensure that at least a subset of the vertex states becomes convergent. By multiple iterations of graph processing, L-Graph checks whether or not the vertex states are still needed to be updated. If none of the vertex states is needed to be updated, WCC becomes convergent.

4.2 Challenge and Solution

When only a subset of $\mathcal{V}$ is inside memory, i.e., *vertices out of core*, the invoke of UDF over an edge $\langle u, v\rangle$ may request a vertex state out of memory. We say that such a request is *missed*. The request is otherwise *hit* if the requested vertex state is just inside memory. We defin the hit ratio to be the one between the hit requests and all requests to the needed vertex states during the processing of a graph. Vertices out of core indicate that the hit ratio is always smaller than 1.0.

For the hit request, the processing is exactly the same as Section 4.1. However, the missed state could lead to slower graph processing that is illustrated by Fig. 2. To access the missed vertex state $s(u)$, a simple approach is to lookup the vertex block on SSD for the missed $s(u)$. That is, we have to seek for the missed $s(u)$ on SSD by the sequential scan of vertex states in such a block until the missed $s(u)$ is found. In the worst case, all memory space is occupied by the current vertex states and no more free memory space can be allocated for the missed $s(u)$ that should be loaded from SSD to memory. In this case, we have to dump a currently available vertex state, say $s(u')$, from memory back to SSD and then release the associated memory space for $s(u)$. Consequently, *two* operations are used: (*i*) *write* of the memory-resident $s(u')$ and (*ii*) *read* of the missed $s(u)$. For an edge, the associated cost of the two operations might be trivial. However, if not properly designed, each of all E edges incurs two of such operations (E is the total number of edges), leading to high I/O cost with the scale of $O(E)$.

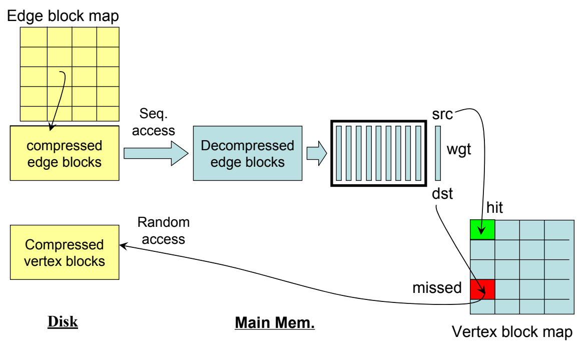

Figure 2: Flow of Edge Processing

To tackle the above challenge, we propose a memory layout in order to maximize the hit ratio. In more detail, we design a simple and yet efficient memory layout by allocating main memory into two parts:

(*1*) *Static memory* (SM) for the vertices with the top highest degree. Based on the input graph, such vertices can be easily found during the preprocessing phase. Selecting the vertices with highest degree is based on the following observation. For an iteration of processing an entire graph $\mathcal{G}$, L-Graph loads every edge only *one time* to read and write the state of each endpoint, say u. Since the vertex u can be the source or destination of multiple edges, L-Graph reads and writes the vertex state $s(u)$ by *multiple times*. It is not hard to fin that the frequency of reading and writing $s(u)$ is equal to the vertex degree of u. Thus, the vertices with highest degree will help maximizing the hit ratio.

(*2*) *Dynamic memory* (DM) for runtime needed vertices. Such vertices are those need by missed requests. The key is to treat DM as a *caching system*, and the read and write of vertex states between SSD and DM as the *replacement* of the existing cached items by newly incoming items. The classic caching replacement policy, such as least recently used (LRU), has been shown to be practically useful to achieve high hit ratio. L-Graph can use such a replacement policy for DM to replace the currently available vertex states and write them from DM to SSD.

5. EVALUATION

Section 5.1 firs briefl describes L-Graph prototype, and Section 5.2 reports the performance result by preliminary experiments.

5.1 Prototype

We implement a L-Graph prototype by Java 1.70 with the following components:

(*i*) *VertexManager* manages vertex states on SSD and main memory. It maintains vertex states as many as possible in memory with SSD as external memory.

(*ii*) *EdgeManager* maintains read-only edge blocks (compressed by Java built-in Zip algorithm) on SSD and in main memory. When loading edge blocks from SSD, the manager allocates a subset of memory to buffer the loaded blocks. The size of the buffered edge blocks depends upon the available memory size (say S_m) and space cost (say S_v) of maintained vertex states. Given $S_m > S_v$, we set the memory size allocated for edge blocks to be $(S_m - S_v)$. Otherwise, the manager allocates the majority of main memory, e.g., $0.8 * S_m$, to maintain a subset of vertex states. Thus vertex states are maintained on both SSD and memory, and the remaining $0.2 * S_m$ memory buffers loaded edge blocks. We set the block size $B = 40$ Megabytes by default. Based on such a size, we compute the total number, say B, of blocks to maintain all edges. After that, we map an edge $\langle u, v, w\rangle$ to a specifi block by a hash function, e.g., the modular operation % , over the vertex ID of source u. In this way, all edges are assigned to the B blocks.

(*iii*) *L-GraphEngine* manages the overall runtime engine of L-Graph. In detail, it schedules the loading of edge blocks based on UDI if any, manages vertex states, and invokes UDF.

(*iv*) *ProgramRunner* executes graph analytic applications. Each application is required to implement an interface `Program`. The interface define an iterator `iter` for UDI and four abstract functions: `udf`() to implement the user define function, `stop`() to set the stopping condition, `get`() and `put`() to defin the put and get operations over the UDI. By default, the iterator `iter` is scheduled to sequentially load edge blocks with no explicit implementation. Based on the `Program` interface, we have implemented four example applications PageRank, BFS, DFS and SSSP.

5.2 Preliminary Result

We measure the running time of four implemented applications: WCC (2 iterations), PageRank (2 iterations), BFS and SSSP. For each application, we repeat each test f ve times and compute the average running time. We follow a test environment similar to the GraphChi and XStream on two personal computers: (i) a HP EliteBook laptop (4 cores, 3GB memory and a 160GB Intel 320 SSD) installed with a 32-bit Windows 7; and (ii) a Dell Desktop (4 cores, 16GB memory, and 180GB Intel 520 SSD) with 64-bit Ubuntu (Linux version 3.2.0). The maximal Java virtual memory is set to 1024MB on the HP laptop, and 6144MB on the Dell desktop. We conduct the following preliminary experiments. The preliminary experiment is used to verify the advantages of L-Graph over two previous works, and the sensitivity experiment shows how L-Graph performs well by varying some key parameters.

In the experiment, we compare L-Graph with GraphChi and XStream as follows. (*i*) GrapChi provided two implementation versions including Java and C++[1]. Both versions offered the example applications PageRank and WCC, but without traversals and SSSP. Therefore, we use the exposed APIs to implement traversals and SSSP. We test Java and C++ versions on the HP Laptop and Dell Desktop, respectively. Following the evaluation section of GraphChi [2], we use 16 shards and a block size of 4MB (such a configuratio can achieve good enough performance). (*ii*) XStream provided only the C++ version and have implemented all the four graph applications. By the configuratio of out-of-core XStream with a thread-private buffer of size 256 MB, we test XStream on the Dell Desktop.

We use two graphs: (*i*) a generated Erdos Renyi (ER) random graph with 13 million vertices and 103 million edges, and (*ii*) a real Twitter graph with 52 million vertices and 1,963 million edges.

[1]https://github.com/GraphChi

The data size of the raw data to maintain the the two graphs is 1.59GB and 28GB, respectively. In addition, the file contain only graph structure with no edge weights, so we generate such weights randomly between the range (0.0, 1.0] for the SSSP application. We test L-Graph, GraphChi and XStream on the HP laptop with the generated graph, and the Dell desktop with the Twitter graph. By default, L-Graph uses 4 threads to load edge blocks and execute UDF in order to optimize the throughput.

	HP Laptop	Dell Desktop
WCC	16	459
PageRank	57	1032
BFS	36	683
SSSP	173	4592

Table 1: Experiment Result (running time: seconds)

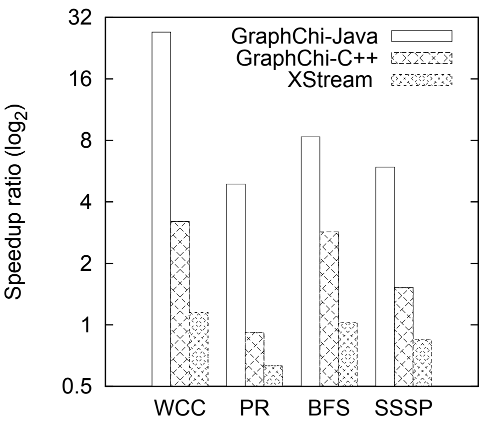

Figure 3: Speedup over GraphChi and XStream

Table 1 shows the running time of 4 applications implemented atop L-Graph, and Fig. 3 gives the runtime speedup of L-Graph over GraphChi and XStream. The speedup ratio of GraphChi-Java is the one between the running time by GraphChi Java version over the L-Graph, both of which are tested on HP Laptop, the ratio of GraphChi-C++ is the one between GraphChi C++ version over the L-Graph, both tested on Dell Desktop, and finall the ratio of XStream is the one between XStream C++ version over the L-Graph on Dell Desktop. A larger ratio indicates that L-Graph uses less running time compared with the counterpart and vice versa.

Based on Fig. 3, we have the following results:

- First, for all four graph applications, L-Graph greatly outperforms GraphChi Java version, with the speedup ratios of 27.1×, 4.89×, 8.33× and 5.92× on the WCC, PageRank, BFS and SSSP, respectively. These results indicate the obvious advantage of the continuous model over the batch model used by GraphChi.

- Second, though GraphChi-C++ version is much faster than its Java version, L-Graph still outperforms GraphChi-C++ version on WCC, BFS and SSSP (except PageRank with the speedup ratio 0.92, indicating that the GraphChi-C++ version achieves only slightly faster running time on PageRank than L-Graph).

- Third, compared with XStream, L-Graph is slower on PageRank and SSSP with the speedup ratios 0.63 and 0.85 respectively, but is still slightly faster on WCC and BFS. The slower running time of L-Graph is caused by the implementation of different programming languages. In addition, our result is consistent with the one reported by XStream [7]: the running time of GraphChi C++ version is slower than XStream on all four applications by around 2-3×.
- Finally, by comparing the running time of different applications, we fin that the absolute running time of WCC implemented by two GraphChi versions and XStream is larger than the one of PageRank, consistent with the result in GraphChi [2] and XStream [7]. Instead, the time of WCC implemented by L-Graph is much faster than the one of PageRank. We believe that it is caused by the continuous model in L-Graph, which allows WCC to quickly propagate (or broadcast) the (largest) labels throughout graphs, leading to the fastest convergence and thus the largest speedup ratio among all the four applications. Instead, the computation of PageRank requires all edges to invoke UDF.

As a summary, the above preliminary experiment has successfully verifie that L-Graph is much faster than GraphChi on all four applications, when both implemented by the same programming language Java. Even with the C++ implementation of GraphChi and the very recent XStream, L-Graph is faster than GraphChi on WCC, BFS and SSSP, only except PageRank, and still faster than XStream on WCC and BFS. These results are enough to demonstrate the advantage of the continuous model over the batch model.

6. CONCLUSION

In this paper, we have presented a general graph analytic system L-Graph on single machines. First, by the proposed continuous computation model, L-Graph ensures faster propagation of vertex states. Second, based on the execution model, L-Graph avoids writes of voluminous onto SSD edges and instead advocates memory-based operations to update vertex states. Finally, L-Graph designs an adaptive memory layout to minimize the overall I/O processing overhead. Our preliminary experiment indicates that that L-Graph greatly outperforms GraphChi Java version by tends of faster running time, and achieves comparable result as GraphChi and XStream C++ versions.

As the future work, we continue the research of L-Graph on the following directions. i) In terms of the computation model, we are particularly interested in how our proposed computation model can be integrated with asynchronous models, such as GRACE [9], for even faster convergence, and fin potential graph applications that can benefi from the computation model. ii) We plan to extend L-Graph for a distributed setting, where each machine can perform the local graph processing based on L-Graph. iii) Given the dynamical graph evolution, incremental operations over L-Graph can avoid the entire graph re-processing.

Acknowledgment: Part of this work was done when W. Rao was at University of Cambridge. This work is partially supported in part by Science and Technology Commission of Shanghai Municipality (Grant No. 15ZR1443000). We also would like to thank the anonymous ACM MobiCom 2015 HotPlanet workshop reviewers for their valuable comments that helped improve this paper.

7. REFERENCES

[1] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 17–30. USENIX Association, 2012.

[2] A. Kyrola and G. Blelloch. Graphchi: Large-scale graph computation on just a PC. In *Proceedings of the 10th conference on Symposium on Opearting Systems Design & Implementation (OSDI)*. USENIX Association, 2012.

[3] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[4] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

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

[6] 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 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[7] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488, New York, NY, USA, 2013. ACM.

[8] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 505–516, New York, NY, USA, 2013. ACM.

[9] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR*, 2013.

[10] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002.