Maintaining connected components for infinite graph streams

Jonathan Berry
Sandia National Laboratories*

Matthew Oster†
Rutgers University

Cynthia A. Phillips
Sandia National Laboratories*

Steven Plimpton
Sandia National Laboratories*

Timothy M. Shead
Sandia National Laboratories*

ABSTRACT

We present an algorithm to maintain the connected components of a graph that arrives as an infinite stream of edges. We formalize the algorithm on X-stream, a new parallel theoretical computational model for infinite streams. Connectivity-related queries, including component spanning trees, are supported with some latency, returning the state of the graph at the time of the query. Because an infinite stream may eventually exceed the storage limits of any number of finite-memory processors, we assume an aging command or daemon where "uninteresting" edges are removed when the system nears capacity. Following an aging command the system will block queries until its data structures are repaired, but edges will continue to be accepted from the stream, never dropped. The algorithm will not fail unless a model-specific constant fraction of the aggregate memory across all processors is full. In normal operation, it will not fail unless aggregate memory is completely full.

Unlike previous theoretical streaming models designed for finite graphs that assume a single shared memory machine or require arbitrary-size intemediate files, X-stream distributes a graph over a ring network of finite-memory processors. Though the model is synchronous and reminiscent of systolic algorithms, our implementation uses an asynchronous message-passing system. We argue the correctness of our X-stream connected components algorithm, and give preliminary experimental results on synthetic and real graph streams.

*Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

†Supported by the U. S. Department of Homeland Security under grant award number 2008-ST-104-000016. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U. S. Department of Homeland Security.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
BigMine '13, August 11-14, 2013, Chicago, IL, USA
Copyright is held by the owner/author(s).
Publication rights licensed to ACM.
ACM 978-1-4503-2324-6/13/08 ...$15.00.

Keywords: Streaming graph algorithms, streaming models, parallel distributed computing

1. INTRODUCTION

In classic (finite) streaming applications, data are generated piece by piece from external processes such as sensors. The computing system does not have space to store the entire stream, but we still must compute global information about it. Streaming problems are similar to online algorithms, where data arrives piece by piece and the algorithm must make irrevocable decisions now without knowledge of future inputs. In streaming algorithms, one must decide not only what to compute, but also what to store in limited memory.

Our primary application is cybersecurity. Streaming cyber data can represent relationships between entities that are often modeled as graphs. Analysts could infer interesting things from such graphs, but generally cannot keep up with today's streams. Further, institutions will likely have difficulty hiring enough analysts to keep up with tomorrow's streams. We wish to develop practical fundamental streaming graph algorithms to partially automate the analyst's task.

We present a real-time graph mining methodology for answering queries about connected components in a streaming graph. Our algorithm is based on a new theoretical streaming model that stores the whole graph in a distributed system with periodic bulk deletions. Connected components are fundamental topological structures involving global information, but other structures could be mined, such as component spanning trees and path queries.

In the classic streaming model, the input is a finite series of edges, each represented by the pair of vertex endpoints. For a finite graph, this series is an arbitrary permutation of its edges, possibly with repetitions. The output is a (vertex, label) pair for each vertex such that two vertices have the same label if and only if they are in the same connected component.

An algorithm for the classical streaming model [5, 6, 8] has parameters s, the amount of space available to the algorithm, p, the number of times the algorithm can see the same input stream (called the number of passes), and sometimes the amount of computational time required per stream element, which should be small. For a finite stream, a single-pass algorithm cannot output two vertices, say u and v, with different labels until the entire stream has passed. Otherwise the last edge could join the previously disjoint components containing u and v, and the algorithm will have made an irreparable mistake. In fact, the algorithm cannot forget any vertex name, since if the algorithm outputs and forgets vertex u, the last

edge could be (u, v) for some new vertex v, which would then be mislabeled. Thus a one-pass algorithm must remember the names of all nodes, which is $\Omega(n)$ names or $\Omega(n \log n)$ bits in the worst case. In 2009, Demetrescu et al. reported having found no published streaming graph algorithms with both sublinear space and passes. [3]. We are not aware of any such algorithms in the intervening years. Henzinger et al. proved that any p-pass streaming algorithm for connected components for a graph requires $\Omega(n/p)$ space [5]. So any algorithm that uses a constant amount of space requires $\Omega(n)$ passes.

Some theoretical models have relaxed the strict classical streaming model to allow more than a constant amount of space, that is, allowing space that is a function of the graph size, or allowing the algorithm to alter the stream for subsequent passes.

Demetrescu et al. [3] introduced the *W-Stream* model, where "W" stands for "write." In this model the algorithm is allowed to write a new stream as it processes the finite stream. Thus in each pass of the algorithm, it reads the altered stream it produced in the previous pass. This model assumes an external file system large enough to store the intermediate file the stream is reading and the next intermediate file it is writing. W-Stream is a more restricted and realistic version of the StreamSort model that Aggarwal et al. [1] introduced in 2004. The latter allows external memory sorting, which W-Stream does not. Demetrescu et al. developed an elegant W-Stream algorithm that gives a nice tradeoff between space and number of passes: $O((n \log n)/s)$ passes for any given amount of space s. They also prove an $\Omega(n/s)$ lower bound in the W-Stream model, so their algorithm is within a $\log n$ factor of optimal. Theirs was the first non-sorting-stream-based algorithm to work with $o(n \log n)$ space. Our algorithm is based on this W-Stream algorithm. We describe the Demetrescu et al. algorithm in some detail in Section 2.

These algorithms all assume finite streams, but cyber data streams for active systems have no obvious end. So they are better modeled by infinite streams. In an ideal world, edges would stream forever into an infinite store, analysts would query it, and the query answers would pop out instantly.

In cyber analysis, infinite edge streams can gradually expose ever growing, but instantaneously finite graphs such as portions of the Internet or World-Wide Web. Such streams may have many duplicate edges, limiting the grah growth rate. However, eventually graph storage requirements might exceed any system's memory. So we model periodic bulk deletions of edges, most easily specified by an aging operation.

The only parallel dynamic connected components algorithm for infinite graph edge streams that we are aware of is the work of Ediger et al. [4]. They have a shared-memory algorithm for power-law social networks where there can be explicit edge deletion, such as "unfriending". The experiments in [4] indicate that their algorithm can handle 240,000 updates per second on 32 processors of a shared-memory Cray XMT when most actions are insertions, updates are processed in batches, and the user can tolerate some error between full recomputations. The observed graph must fit entirely into the available shared memory at any given point, even though the stream can be infinite. The experiments in [4] process edge insertions and deletions in batches of one thousand (throughput $11,800$) to one million (throughput $240,000$). All updates in a batch are processed in parallel. Thus, queries can only happen between batches, at granularities of thousands to millions of updates.

R-MAT graphs, used in experiments in [4], are a standard input set for the GRAPH 500 benchmark [7]. Given appropriate parameters, R-MAT graphs have a scale-free degree distribution, but they do not necessarily have other properties common in social networks such as high clustering coefficient [12]. Probably graphs with more local triangles would allow the triangle-finding heuristic in [4] to work even better. More recent unpublished results demonstrate the algorithm processing over one million updates per second [2] in some cases.

We introduce X-Stream, a streaming model motivated by the cyber analyst's ideal, but having realistic limitations. We give an X-Stream connected components algorithm that also maintains a spanning forest. Generally it is difficult to build an efficient arbitrarily-large shared-memory system. We consider distributed-memory architectures where we can theoretically use an arbitrary number of processors, and hence an arbitrary (though finite) amount of memory to store the graph. An algorithm for our model must not fail for space reasons unless the entire distributed memory is asymptotically full. Queries have some latency, depending on type, but are correct for the graph at the time of the query. The operator or a daemon must monitor storage availability. When storage levels become dangerously low, the operator must explicitly remove a sufficient number of relatively less interesting edges. The model supports arbitrary predicates to determine which edges to delete, provided the test is quick. For this discussion, we assume the operator *ages* edges older than a time t (s)he chooses, on the assumption newer edges are more interesting. Daemons can choose t based on a set of rules. If the system doesn't age soon enough or aggressively enough, then the system might fill, and thus fail. In the monotonic setting where edges never leave, like the finite-stream static setting, we could throw away extra edges that connect vertices known to be in the same component. But we now may need them to repair components after edge removal.So X-Stream connected components requires distributed storage of the whole graph. During aging, we disallow queries while we rebuild our data structures, but we continue to accept and correctly handle new edges.

Because the system may be out for an indeterminate amount of time during aging, it is best that this happens during a time when the operator can best tolerate the query outage. This might be at night when system usage is lower. The daemon should periodically warn the operator of impending automatic aging, which will likely be days ahead of the event, so the operator can manually induce an aging at a better time.

2. THE W-STREAM CONNECTED COMPONENTS ALGORITHM

Our X-Stream algorithm – particularly the graph representation – is based on that in [3], which we call W-Stream for short. Our algorithm has additional complexity to handle aging and infinite streams.

In the W-Stream algorithm, each pass contracts a set of disjoint connected subgraphs into single nodes until each connected component is represented by a single node. The stream in each pass has two contiguous parts. The first part represents the current partially contracted graph, and the second part stores the nodes that are inside the contracted pieces. In the input stream, the first part is the whole uncontracted graph and the second part is empty. In the output from the last pass, the first part is empty and the second part gives the final com-

ponent labels for each node.

During the first pass, the algorithm computes connected components using union-find data structures until the memory of size s is full. In the union-find data structure, each partial component found so far has a root node, which is the leader of the current component. At this point, the processor starts to emit the second stream, outputing the graph with the current components contracted into a single node represented by the leader. When edge (u, v) arrives, the processor looks to see if the endpoints have been seen yet. The algorithm finds the leaders for the node(s) that have been seen before. If both nodes have been seen and the leaders are the same, the edge is discarded. Otherwise the edge spans two current components and must be output to the next stream. The algorithm replaces the node name of a previously seen node with its component leader name. If the node is new, the name is unchanged. This is called *relabeling* an edge. The algorithm relabels all the remaining edges in the stream, then emits a dividing marker, indicating the beginning of the second part of the stream. It then outputs a (node, leader) pair for each node hidden in a contracted component.

Subsequent passes are processed in much the same way. Some of the nodes in the data structure now represent contracted subgraphs, but the algorithm treats the leaders like any other node. In the second part of the stream, for each (node, label) pair, the algorithm determines if the label has been pulled into a new (higher-level) component with a new leader. If so, it relabels the node. Thus the nodes in the second part of the stream are always represented by the most up-to-date leaders.

3. THE X-STREAM MODEL

X-Stream is a follow-on model to W-Stream, hence uses the next letter in the alphabet. X-Stream does not use intermediate files. Instead the graph is distributed in a ring of processors as shown in Figure 1.

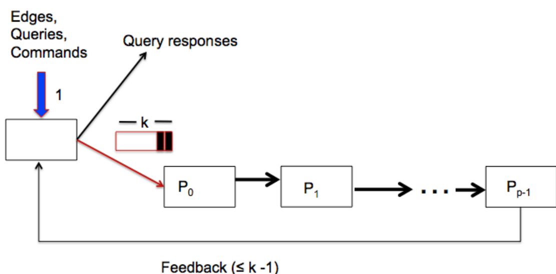

Figure 1: The X-Stream architecture

The system has p processors plus an I/O processor. The I/O processor sends the input stream to the first processor and relays information, such as query responses, to the operator from the last processor. Processor p_i has s_i bytes of local memory, though for this paper, we simplify, assuming all processors have the same memory size s. We denote the total memory across all p processors as S. We assume $p \ll s$ since even on massively-parallel machines, each processor has at least megabytes and usually gigabytes or more of local memory. Each processor has a *rank* determined by its place in the ring. The processor that receives the stream input from the I/O processor has rank 0, and is called the *head*. The last processor has rank $p - 1$, and is called the *tail*. If a processor has rank r, then the processors with ranks $0, 1, \ldots, r - 1$ are *upstream* with respect to processor r and processors with ranks $r + 1, \ldots, p - 1$ are *downstream*. Processors have unique IDs and addresses. When we refer to processor P_i, we mean the processor with rank i. Processors can reconfigure the ring, changing rank if necessary. This is necessary for our connected components algorithm. The input stream and the operator always interface to the I/O processor, which does not change roles.

As Figure 1 illustrates, X-Stream looks like an "unrolling" of the W-Stream model, with each pass represented by a processor. Indeed, the W-Stream algorithm for a finite stream will simply "fill-and-spill" from one processor to the next, with the last processor finally emitting the component labels. There is a link for feedback communication from the last processor to the I/O processor, which is necessary for maintaining data structures for an infinite stream with bulk deletions.

We model the communication as synchronous. It's possible to implement X-Stream algorithms in an asynchronous environment, and in fact we do. That is also likely to be more efficient. However, it is simpler to think of all processors communicating to the processor immediately downstream and receiving information from the processor immediately upstream at the same time. We assume messages hold a constant number of elements, such as a constant number of node names. Messages need not all be the same size, but they must obey a fixed upper bound. As a helpful mental image, we think of each message as fitting into a "basket." In an asynchronous implementation, messages or baskets will travel at various rates, but we always process them in order so the data structures and answers match those of a synchronous implementation. We have found the synchronous simplification useful for designing algorithms.

The input stream must not fully consume the bandwidth between processors. The processors must communicate to maintain the graph data structure and answer queries. We assume that the stream input rate (edges and queries) is only $1/k$th the maximum bandwidth around the ring, where k is the *bandwidth expansion parameter*. As shown in Figure 1, a block of k baskets moves between processors on the ring each tick. Two *black* baskets are reserved for the input stream (see Section 4.1 for further explanation). Algorithms can use the remaining $k - 2$ *white* baskets for maintaining data structures and answering queries with non-constant-sized output. We require $k \geq 3$, since the algorithm requires at least one basket for processors to communicate with each other outside the stream. The I/O processor is the only processor with multiple input and output channels. Its only job is to take at most $k-1$ baskets of information on the feedback link from the tail, output any information meant for the operator, and bundle the remaining baskets of feedback information plus the 1 basket of input from the stream into a size-k block it sends to the head.

There are several theoretical ways to measure performance of an X-Stream algorithm: 1) k, the bandwidth expansion parameter. 2) The time required to process a message. 3) The amount of space used per node or edge, for a graph algorithm. 4) The amount of memory guaranteed in use when an algorithm fails for lack of storage anywhere in the system. 5) Time to stabilize after a bulk edge removal. 6) Query reponse latency, which must always be at least p time steps. A good

algorithm should not fail unless $\Omega(S)$ space is holding relevant data. In this case, the constant in the Ω term can also be a relevant measure.

All of these measures can also be evaluated experimentally and in practice. Practitioners can also compare algorithms based on *streaming rate*, that is, how many stream elements the algorithms can process per second.

4. ALGORITHM OVERVIEW

This section contains a high-level description of our X-Stream algorithm for maintaining the connected components of a streaming infinite graph. There are numerous details necessary for correctness which we must omit due to space restrictions. We introduce notation and terms as needed since many would be difficult to understand out of context. We use the terms "node" and "vertex" interchangeably.

X-stream has three major modes: **Normal:** Not in one of the other two modes. **Aging:** between the start of an aging operation and the restoration of data structures, when the system emits a token re-enabling queries. **Emptying:** A single processor must delegate responsibility for (that is, send downstream) all of its data, so it can become the new tail. This is necessary in theory for our algorithm to maintain components on an infinite stream with a finite number of processors, but we haven't needed it in practice.

In our algorithm, each edge is stored in exactly one processor[1], and processors know only the nodes that are endpoints of the edges they store. Each edge is stored in a constant number of data structures. Thus it is easier to think of each processor as defining storage notions such as "full" based on number of edges rather than bytes [2].

The graph is stored in a nested manner from rank 0 to p, similar to passes in the W-stream algorithm. Each processor holds *local components*, which appear as if contracted to a single node from the perspective of the downstream processors. When an edge arrives at a processor, we call its endpoint nodes (whether real nodes or contracted subgraphs that look like nodes) *building blocks* (BB). A building block containing exactly one node is called *primitive*.

A processor joins building blocks into a local component representing a higher-level contraction with respect to the full graph. The contracted view of a local component becomes a building block for a downstream processor. From the point of view of a single vertex, this looks like "nesting dolls," where the connected subgraph to which it belongs is progressively included in larger connected subgraphs. For example, in Figure 2, the primitive vertices gathered into LC X in an upstream processor are included in LC Z for processor P_2, which is in turn part of LC C in processor P_3.

The distinction between a building block and a local component is largely one of perspective. Each processor accepts building blocks as input. It creates new local components (LCs) from building blocks and connecting edges. Each such subgraph is created once as a local component, sent as output, and then consumed as input by exactly one processor, where the same (sub)component, now contracted, is considered a BB.

[1]An edge can be transiently stored by two processors during aging, but that is only for at most p timesteps

[2]We assume storage for node names is bounded. Technically, this requires preprocessing the stream, maintaining node names in a hash table, perhaps over many processors.

LCs (or BBs depending on context) are named with the processor that created them and a serial number.

When a local component is contracted to form a BB for a downstream processor, its substructure is largely hidden from the processor P that consumes it. The only vertices a processor *knows* are the endpoints of the edges it stores, since knowlege of the endpoints is part of the knowledge about the edge. For example, in Figure 2, vertex v_1 is inside BB A for P_3, but since P_3 has no edge with endpoint v_1, it does not know about it. Other than the edge endpoints in a BB, the only information P has about the BB is a count of the total number of primitive vertices in the subgraph represented by the BB. This is fundamental to the model of a set of distributed, fixed-sized memories. If the BB carried the names of all the vertices in the subgraph it represents, without holding edges against which to amortize their storage, we would no longer have a constant space bound per edge. Furthermore, for a connected graph, the processor that creates the component that represents the whole graph would have to know all the nodes. The node (BB) information also couldn't be sent in a single message.

4.1 Processing edges in normal mode

When the algorithm begins, processor P_0 accepts edges which arrive as pairs of nodes, each in a primitive building block, and maintains connected components with a union-find data structure. Each edge that starts a new component or joins two previously-separate components is a *spanning tree edge*. That is, these edges form a spanning tree of each component found so far. A new edge that falls within a component (both endpoints in the same component) would be dropped by the W-Stream algorithm. However, the X-stream algorithm must store it, because any edge may be critical for correct connectivity after aging. We call such edges *non-tree* edges. If an edge is a duplicate (already seen), the processor updates the edge timestamp.

Processor P_0 becomes full when it can store no more edges, leaving a minimal working space for answering queries, processing new edges, etc. When processor P_0 becomes full, it "seals off." This is equivalent to the point in the W-Stream algorithm when the processor starts generating the next stream. At this point, P_0's local components (LCs) are determined. They may change due to aging or emptying, but not in normal operation. Processor P_1 now becomes the *filling processor*. There is always one filling processor, the only processor that computes components with new edges that arrive on its input. All processors upstream of the filling processor are *sealed*.

Suppose a new edge $([u, B_u], [v, B_v])$ arrives at sealed P_0, where u and v are the vertices and B_u and B_v are (one-node) primitive building blocks. Processor P_0 relabels the edge endpoints, changing only the building block name. For example, in processing the endpoint u, P_0 checks if it is storing any edges with endpoint u (that is, if it knows u). If so, it replaces B_u with the name of the local component that contains u. Otherwise, the building block name is unchanged.

If, after relabeling, the two endpoints are in different LCs, then P_0 sends the edge downstream where it will connect components later. If the two endpoints have the same LC, then P_0 keeps the edge as a non-tree edge if it has room. Before the first aging operation, processor P_0 will not have extra space, since it was sealed for lack of space and no edges have aged. However, later on, after aging, it may have gained some space. If there is no space for this edge, then the processor labels the

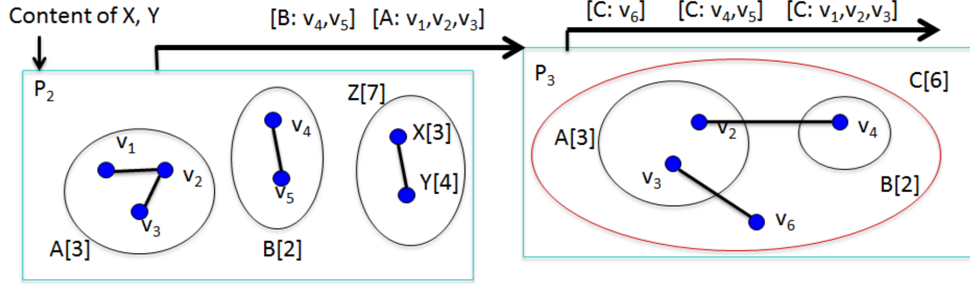

Figure 2: An example of the nested data structure.

edge as a *storage edge*, marked as stored for P_0 (using P_0's ID). The edge is stored in a special data structure by the first downstream processor that has room.

The filling processor, the first non-sealed processor in rank order, creates connected components as described above, though generally the building blocks may be non-primitive, representing multiple nodes. If an edge comes in labeled as storage, and the processor is not full, it will store it, at least temporarily. Suppose the filling processor is full, but some of the edges it holds are storage for upstream processors. If another storage edges comes in, it sends it downstream to the next processor that has room. If the filling processor receives a real edge that is either a spanning tree edge (linking two current LCs) or a non-tree edge within one of its LCs, it keeps that new edge and sends an arbitrary storage edge that it was holding downstream. We can think of storage edges as filler. They can be stored anywhere. So whenever a processor has a chance to take an edge it would like to store (spanning tree, non-tree), then it keeps the better edge and jettisons a storage edge. The filling processor seals only when full of real (non-storage) edges.

In general, after aging as the system is settling into a normal (post-start-up) state, a storage edge will go downstream, perhaps cycling around to P_0 and continuing downstream until it finds a processor that is not full. If a processor sends out an edge for storage and that edge goes all the way around without finding a place to stay, then the system is totally full. This will not happen with a daemon, but could with insufficiently attentive operators in a manual system.

We reserved two (black) baskets out of the K for the input stream. If there were no such restriction and all k buckets were full in a single tick arriving at the tail, that would either starve the input stream or lead to unbounded back up on the tail processor. We require two black baskets reserved for the input stream because each stream edge may have to go all the way around once, passing back through the I/O processor, before settling into a processor as storage. In more detail, when a stream edge e arrives, the I/O processor puts it into a black basket. If the edge goes into storage it may cycle past the head. In this case, it remains in the black basket as it travels through the I/O processor and a new stream edge takes the second black basket. Edge e will settle into a processor before it reaches the processor that sent it into storage. Otherwise, the system will fail from being totally full. Therefore, the black basket e is using will be emptied by the second time through.

4.2 Queries

The system can handle a variety of connectivity-related queries including: 1) Are vertices u and v in the same connected component? 2) Return a component name and one vertex for all the components with size less than ℓ (small components). 3) Return all the vertices in small components (components with at most ℓ vertices). 4) What are the neighbors of node u? 5) Return a spanning tree of the component that contains vertex u. The system can also handle maintainence queries such as how many more edges the system can hold, or, equivalently, percentage full, how many edges currently have a timestamp older than t, etc.

In the list above, the first is an example of a constant-size query. This query is answered with guaranteed latency, the time to move from the head to the tail. When this query arrives, each processor relabels the endpoints. If the endpoints ever acquire the same LC name, then the answer is yes. Otherwise, if the filling processor finds they have different LCs after relabeling there, it returns no. The answer passes down to the tail, which communicates the answer to the I/O processor.

Other queries, such as listing all the nodes in the small components have non-constant-sized answers. Each component in the answer will have no more than ℓ vertices, but ℓ may be a function of the graph size and there could be an unbounded number of such components. Here is how we answer this query, illustrated in Figure 2 for $\ell = 6$. The query passes down the line of processors, so processors can prepare to answer. Sealed processors have stable components, but the filling processor must remember small components so the query is correct for the time of asking.

The head processor uses white baskets to send the information about all its small components. If a processor has a small component with non-primitive building blocks, it relabels the BBs as the information goes by. For example, in Figure 2, processor P_3 has small LC C. It relabels the nodes in BBs A and B as they stream by. If a processor has LCs that are not small, but contain small BBs, then it removes the small BB information as it goes by. For example, in Figure 2, the processor(s) that create(s) BBs X and Y send their contents downstream, since it is possible they are small components. Processor P_2, however, knows that BBs X and Y merge into a component with too many nodes. So processor P_2, does not relabel or propagate the messages for the nodes in X or Y.

When the head processor has sent all its small-component information, it passes a (query fulfillment) token to the next processor, which sends out all the primitive BBs it holds that are in small components, and so on. When the filling processor receives the token and has sent out all the information involving primitive vertices it holds, then the filling processor issues

an "all done" token to the operator.

If an aging command arrives while a non-constant query is still unfinished, the system does not finish answering the query. The operator has the option to wait for the query before aging if the system is not in a critical state. We assume there is at most one active non-constant query in the system at a time.

4.3 Aging

When an aging command arrives, the message travels downstream to all processors. Each processor deletes edges that have aged out, either by removing a few edges in each time step or by multithreading deletions in a threadsafe way.

Connectivity queries are not allowed during aging. We will discuss how to handle new incoming edges during aging at the end of this section. If there has been any emptying since the last aging, there is an initial storage repair step at the start of aging (see Section 4.4).

As with the query fullfillment token, there is a token that passes from the head to the tail during aging. We call it the "snowplow," since it leaves clean processors behind it. The processor with the snowplow completes the removal of any edges it has not removed while waiting for the snowplow to arrive. It then goes through five stages to clean up after aging: 1) **Resolve**, 2) **Recompute**, 3) **Recycle** storage, 4) **Release** unattached building blocks, and 5) **Split** broken BBs if necessary. These phases are designed to repair the data structures and restore the properties listed in Section 5.

The *Resolve* stage removes uncertainty caused by upstream splits. Specifically, suppose a processor built an LC with a BB that splits into q pieces due to aging. The processor knows about several vertices inside the original BB: those with edges that connected that BB into the LC. It does not know which piece each vertex belongs to. For example, suppose in Figure 2, that BB A in P_3 had more than 3 vertices inside and that after aging, A split into two pieces. P_3 does not know which piece vertices v_2 and v_3 belong to. Processors that originally created the broken BBs generally cannot tell the downstream processors how to place vertices into the subpieces because it may not know about all the vertices in the BB. For example, in Figure 2, if LC C created in P_3 were to split, it could not tell a downstream processor where to place vertex v_1. We say that the split creates a *purgatory*. This is a data structure that represents uncertainty about which child BB a vertex belongs to. The splitting BB is relabeled to be a purgatory and vertices are moved to new BBs (one BB per new piece) as the processor learns where to put them. We call such moves the *resolution* of vertices. Some vertices may already be resolved when the snowplow arrives. For example, when a new edge arrives during aging that involves a vertex in purgatory, the relabeling process tells the processor which BB the vertex belongs to. Thus relabeling any vertex from the head resolves all purgatories for that vertex along the chain of processors.

In this first stage of aging, the processor sends out resolution requests for all the vertices in purgatory, one by one, or in small batches. The request goes downstream to the tail, and then goes back to the head along the feedback link. The head then puts the vertex in an initial BB, either a primitive one if the processor doesn't know it, or the appropriate LC from the head if it does know it. The request passes downstream with each processor relabeling the component name as appropriate until it arrives at the processor that requested the information, resolving the vertex.

The processor then *recomputes* its connected components. This is done in a threadsafe manner while handling the input stream. We will describe the handling of new edges below.

After the processor has recomputed its connected components, it asks for all its storage edges back. These storage edges might reconnect pieces that have fallen apart or may now be able to fit on that processor. This is also done with a *recycle token*. The processor passes this token (labeled with its ID) to its downstream neighbor. That processor then uses white baskets to send out all the storage edges it's holding for the snowplow processor. Each recycling edge will go back around to the head and arrive at the requesting processor. Correct implementation involves details such as edge replacement if the snowplow becomes full and handshaking between processors to maintain correct edge timestamps.

After all storage edges have recycled, the processor with the snowplow may have building blocks it originally picked up from upstream that are isolated. That is, no remaining edges connect them to other BBs in the processor. It then *releases* these BBs. To release a primitive BB (single vertex), the processor sends a release message, saying which LC the vertex used to belong to. There is at most one processor downstream that consumed that LC as a BB. This processor recognizes the LC name. If it knows the vertex (has an edge with it as endpoint), it creates a new primitive BB, removing the vertex from its original BB. If it does not know the vertex, it reduces the vertex count in the BB by one, relabels the BB, and sends the release message downstream. If the release message goes all the way past the filling processor, then the vertex is forgotten; there are no edges adjacent to it left in the system. We must omit a discussion of releasing non-primitive BBs and how releasing interacts with storage due to space restrictions.

Finally, if an LC has broken into multiple pieces, the processor announces a *split* of the LC. It tells the processor that consumed this LC how many pieces it has split into. This is the mechanism that creates the purgatories described above. Each of these pieces is labeled with a temporary piece name acquired during recomputation. Whenever an edge touches one of these pieces during relabeling (e.g for a new edge or a recycled edge), we give the piece a real LC name, which it uses consistently from then on. Any piece that is not named by the end of aging is a terminal LC, the last LC for the current component.

After splitting, the processor passes the snowplow downstream. The filling processor buffers new edges during its time as the snowplow processor, starting with the recomputation phase. When the filling processor finishes, including processing any buffered edges, the snowplow token becomes an "end of aging" message to the operator. This signals it is now safe to ask queries.

When a new edge arrives during aging, processors still relabel. If the snowplow has passed, the processor treats new edges as it would in normal mode. If the snowplow has not arrived yet, the processor optimistically assumes its LCs are intact and labels and acts according to the old structure. These will be repaired, if necessary, as the snowplow advances. If a new edge arrives while the processor has the snowplow, the action depends upon the stage of cleaning. Before recomputation is complete, it acts as if the snowplow hadn't arrived. This includes keeping any edges that might be non-tree edges in the old structure. After recomputation but before splitting,

if the edge is going downstream, it still relabels according to the old LCs, even if the LC may split. Any edge that "belongs" to it (with endpoints inside an old LC) is treated as if it were being recycled. That is, it can reconnect components that fell apart, etc. After a split is announced, new edges are relabeled according to the split structure, with the processor naming the pieces as necessary. If a new edge comes in that involves a component set to split but the split has not been announced, the processor can prepend the split information in front of the edge information in the same basket. The split information travels only to the processor that consumed the old LC.

4.4 Emptying

It's possible after aging that the system has reasonable room for more edges, but the last open processor has started to fill. For the algorithm to continue, there has to be another empty processor that the filling processor can spill into when it no longer has free space to accept any tree or non-tree edge that arrives. If the head processor holds few enough edges (where this is defined below) it can empty, sending its edges to other processors, and become the new tail. If the head processsor is not empty enough, we find one that is, empty it, and splice it into the ring as the new tail. If there is no such processor, then the algorithm is using a sufficiently large constant fraction of all memory and fails due to inadequate application of aging. The emptying process requires arbitrary network connectivity between the processors and is more complicated, but does not affect the correctness of the connected components algorithm.

The full emptying procedure is beyond the scope of this paper, but we now give the basic idea. Suppose the emptying processor is the head, which will be the case as long as it has sufficiently few non-storage edges. Each LC is composed completely of primitive vertices. The processor will send the contents of its LCs one by one downstream. The processor that consumed each LC as a BB will accept responsibility for its edges. If this is a terminal LC, the filling processor accepts it. The BB, originally a black box, is *refined*, giving it additional internal structure. This is the only way a processor can gain a tree edge when it is not filling and not reconnecting during aging. Refinements are usually simple, but in rare cases can lead to significant complication.

The emptying procedure guarantees that the empyting processor will finish emptying before the filling processor fills unless the memory is asymptotically full. Specifically, it will succeed unless at least $((k-2)/(k-1))(S-\epsilon)$ memory is full, for a small constant $\epsilon > 0$ (k is the X-stream bandwidth expansion parameter). For the minimum value $k = 3$, this mean just under half the memory is full. For larger values of k, the usage fraction improves. As we mentioned before, if the algorithm fails for memory outside of emptying, then the memory is *completely* full.

THEOREM 1. *If the connected components algorithm fails for space during emptying because the emptying processor does not empty before the filling processor fills, then at least* $\frac{k-2}{k-1}(S-\epsilon)$ *of the memory is full, where* S *is the total memory across all processors, ps.*

PROOF. Omitted for space.

Emptying and aging can co-exist. Aging-related edge feedbacks halt temporarily, freeing white baskets for the emptying processor. Since there is no required time to recover from aging, this delay is acceptable.

If the system is not aging, it must handle queries during emptying. The main complication is a primitive building block traveling downstream during a refinement when a query arrives. But since the query is following the traveling node, the node will have settled in some processor before the query catches it.

Because there is some reorganization of responsibility among the processors, some of the edges in storage may now be incorrectly labeled. That is, the processor that placed it in storage may now no longer have an LC structure for which that edge is a non-tree edge. This is true for all storage for the emptying processor. It could also be true for for some LCs in other processors, if the refinement cascaded downstream. Note this can only happen if the receiving processor is completely full of spanning tree edges. An easy strategy is to consider storage suspect for all processors that sent refinements. There are heuristic ways to reduce the number of suspect storage edges such as marking storage with LC and internal BB marks for the endpoint to better identify only those that must be repaired.

We repair storage edges by recycling them. When the storage edge goes back to the head and pretends to be new, it is relabeled and eventually handled by the correct downstream processor. If there has been any emptying since the last aging and there is a new aging command, the system must repair all storage before starting the snowplow protocol.

5. CORRECTNESS

We sketch an argument that the connected components algorithm is correct. We sketch correctness for the query "Are vertices u and v connected?" Similar arguments prove the correctness of other related queries. The following properties of the distributed data structure hold when queries are enabled:

Property 1 : Each LC is a connected set of at least 2 BBs.

Property 2 : Every vertex in the system appears in exactly one primitive building block.

Property 3 : Each non-primitive building block is consumed by at most one LC.

Property 4 : Storage edges do not affect the connectivity structure.

These properties suffice for correct query responses. Proofs are omitted for space.

LEMMA 2. *Properties 1-4 hold whenever querying is enabled.*

THEOREM 3. *There exists a processor P that gives vertices* u *and* v *the same label if and only if the vertices are in the same connected component.*

THEOREM 4. *The amortized bound for union-find can be made deterministic for this algorithm.*

6. EXPERIMENTS

We have developed a prototype implementation of our algorithm in C++ using the *phish* streaming library [10, 11]. We handle normal operation as described in Section 4.1 and aging as abstracted in Section 4.3. We have not yet had a need to implement emptying. Although phish can be run on huge

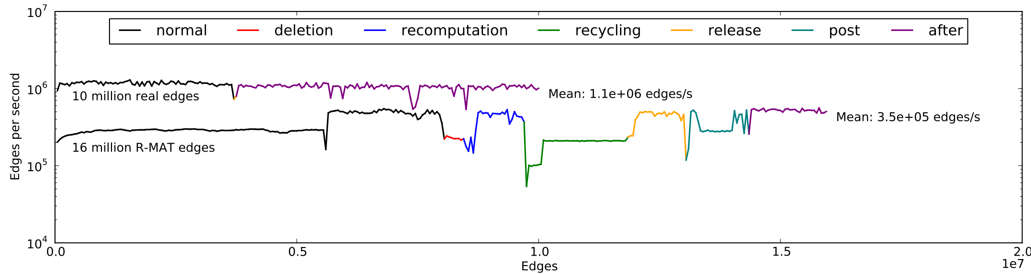

Figure 3: X-stream prototype results: a real stream of 10 million edges, and a stream of 16 million R-MAT edges

distributed-memory supercomputers, we ran our preliminary experiments on a single node shared memory Linux workstation with 50 GB of RAM and 16 Intel Xeon cores clocked at 3.47GHz.

Plimpton et al. [9] give phish benchmarks indicating the maximum sustainable communications throughput we can expect on this platform is roughly 400 million bytes per second.

We use $k = 3$ for our X-stream expansion factor. We package 20 of these 3-tuples into a message bundle that has size roughly 2K. However, we note that the algorithm will process these edges in sequence, so queries are correctly answered at the granularity of a single edge.

Figure 3 depicts X-stream runs on two streams: (a) ten million edges generated from a real stream of Sandia data, and (b) the 16 million edges of an R-MAT graph with 2^{21} vertices, edge factor 8, and SSCA-2 parameters (0.45, 0.15, 0.15, 0.25). The real stream features a significant proportion of repeated edges, which speeds the computation and limits the amount of recycling necessary. In both cases, every tenth vertex pair is interpreted as a connectivity query instead of a new edge.

Stream (a) consists of 30 million baskets coming from the I/O processor. An aging event is triggered at basket 11,000,000, but only roughly 8000 edges need to be recycled. Furthermore, the graph induced by these edges has about 189,000 unique edges. In aggregate the 10 million edges are processed at an aggregate rate of roughly 1.1 million edges per second. This translates to a throughput of roughly 100 million bytes per second. The figure shows the instantaneous edge processing rates throughout the computation.

Stream (b) contains no repeated edges and requires the recycling of almost 4 million edges during aging. The aggregate edge processing rate of roughly 350,000 edges per second.

The code will soon be available in the phish library. Download phish and view documentation at `http://www.sandia.gov/~sjplimp/phish.html`.

7. CONCLUSION

The system will generally work with heterogeneous systems. Most operations are effectively constant (e.g. using hash tables), or can be done lazily using multiple threads. For lazy data structure updates or aging computations, the amount of work can be chosen to balance the amount of work for each message across all processors. So processors that have larger data structures may do effectively slightly slower lazy work, requiring more time during aging. Theorem 1 holds when S is the sum of non-uniform space amounts over all processors.

8. REFERENCES

[1] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2004)*, 2004.

[2] D. Bader. Personal communication, 2012.

[3] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. *ACM Transactions on Algorithms*, 6(1):6:1–6:17, Dec. 2009.

[4] D. Ediger, E. J. Riedy, D. A. Bader, and H. Meyerhenke. Tracking structure of streaming social networks. In *5th Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2011.

[5] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In J. M. Abello and J. Vitter, editors, *External Memory Algorithms*, pages 107–118. DIMACS series in Discrete Mathematics and Theoretical Computer Science, Volume 50, 1999.

[6] I. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[7] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. In *Cray Users' Group (CUG)*, May 2010. Current Graph 500 information at `http://www.graph500.org/`.

[8] S. Muthukrishnan. Data streams: Algorithms and applications. `http://www.cs.rutgers.edu/~muthu/stream-1-1.ps`, 2012. [Online survey apparently started in 2004; accessed October 26, 2012].

[9] S. Plimpton, B. Benner, J. Berry, K. Chiang, J. Doak, J. Ingram, P. Kegelmeyer, S. Mitchell, C. Phillips, T. M. Shead, and B. Wylie. Streaming data analysis for cybersecurity. Technical Report SAND2013-0366, Sandia National Laboratories, Albuquerque, NM, 2013.

[10] S. Plimpton and T. Shead. Phish library. `http://www.sandia.gov/~sjplimp/phish.html`, 2013. [accessed February 13, 2013].

[11] S. Plimpton and T. Shead. Streaming data analytics via message passing with application to graph algorithms, 2013. submitted.

[12] C. Seshadhri, A. Pinar, and T. Kolda. An in-depth study of stochastic Kronecker graphs. In *ICDM'11: Proceedings of the 2011 IEEE International Conference on Data Mining*, Dec. 2011. To appear in JACM.