Streaming data analytics via message passing with application to graph algorithms

Steven J. Plimpton and Tim Shead
Sandia National Laboratories
Albuquerque, NM
sjplimp@sandia.gov

Keywords: streaming data, graph algorithms, message passing, MPI, sockets, MapReduce

Abstract

The need to process streaming data, which arrives continuously at high-volume in real-time, arises in a variety of contexts including data produced by experiments, collections of environmental or network sensors, and running simulations. Streaming data can also be formulated as queries or transactions which operate on a large dynamic data store, e.g. a distributed database.

We describe a lightweight, portable framework named PHISH which enables a set of independent processes to compute on a stream of data in a distributed-memory parallel manner. Datums are routed between processes in patterns defined by the application. PHISH can run on top of either message-passing via MPI or sockets via ZMQ. The former means streaming computations can be run on any parallel machine which supports MPI; the latter allows them to run on a heterogeneous, geographically dispersed network of machines.

We illustrate how PHISH can support streaming MapReduce operations, and describe streaming versions of three algorithms for large, sparse graph analytics: triangle enumeration, subgraph isomorphism matching, and connected component finding. We also provide benchmark timings for MPI versus socket performance of several kernel operations useful in streaming algorithms.

25 Oct 2012 version
submitted for publication to J Parallel & Distributed Computing

1 Introduction

Streaming data is produced continuously, in real-time. Processing it is often different from more familiar numerical or informatics computations or from physical simulations, which typically read/write archived data from/to disk. The stream may be infinite, and the computations are often resource-constrained by the stream rate or by available memory [17, 14]. For example, it may not be possible for a computation to "see" a datum in the stream more than once. A calculation on a datum may need to finish before the next datum arrives. While attributes of previously seen datums can be stored, such "state" information may need to fit in memory (for speed of access), and be of finite size, even for an infinite stream of data. The latter constraint can require a mechanism for "aging" or "expiring" state information.

There are at least two motivations for computing on streaming data in parallel: (1) to enable processing of higher stream rates and (2) to store more state information about the stream across the aggregate memory of many processors. Our focus in this paper is on a distributed-memory parallel approach to stream processing, since commodity clusters are cheap and ubiquitous, and we wish to go beyond the memory limits of a single shared-memory node.

A natural paradigm for processing streaming data is to view the data as a sequence of individual datums which "flow" through a collection of programs. Each program performs a specific computation on each datum it receives. It may choose to retrieve state information for previous datums, store state for the current datum, and/or pass the datum along as-is or in altered form to the next program in the stream. By connecting a collection of programs together in a specified topology (which may be a linear chain or a branching network with loops), an algorithm is defined which performs an overall computation on the stream.

On a shared-memory machine, each program could be a thread which shares state with other threads via shared memory. For distributed-memory platforms, each program is an independent process with its own private memory, and datums are exchanged between processes via some form of message passing. This incurs overhead, which may limit the stream rates that can be processed, since datums must be copied from the memory of one process into the memory of another via a message, in contrast to shared memory where just the pointer to a datum can be exchanged. However, distributed memory parallelism has other advantages as discussed above.

We are aware of several software packages that work with streaming data in this manner. Some are extensions to the MapReduce model [12] with add-ons to Hadoop [1] to enable streaming data to be mapped and reduced incrementally, with intermediate results made available continuously [11, 16]. Other packages implement their own streaming framework, such as the S4 platform [18] and the Storm package [2], recently released by Twitter. The former distributes the stream to processing elements based on key hashing, similar to a MapReduce computation, and was developed for data mining and machine learning applications. The latter is advertised as a real-time Hadoop, for performing parallel computations continuously on high-rate streaming data, including but not limited to streaming MapReduce operations.

There is also commercial software that provides stream processing capability. IBM has a system, called InfoSphere Streams [3], designed to integrate data from 1000s of real-time sources and perform analyses on it. SQLstream [4] is a company devoted to processing real-time data and sells software that enables SQL queries to be made on streaming data in a time-windowed fashion.

We also note that the dataflow model [15], where data flows through a directed graph of processes, is not unique to informatics data. The open-source Titan toolkit [23], built on top of VTK [5], is a visualization and data analytics engine, which allows the user to build (through a GUI) a network of interconnected computational kernels, which is then invoked to process simulation or other data that is pipelined through the network. Often this is archived data, but some streaming capabilities

have recently been added to Titan.

In this paper we describe a new software package, called PHISH, detailed in the next section. PHISH is a lightweight, portable framework written to make the design and development of parallel streaming algorithms easier, particularly for the kinds of graph algorithms described in Section 3. We also wanted a means of running streaming algorithms on a wide variety of parallel machines. PHISH can be linked with either of two message-passing libraries, the ubiquitous message-passing interface (MPI) library [13], and the socket-based open-source ϕMQ library [6], (pronounced zero-MQ and hereafter referred to as ZMQ). The former means that a PHISH program can be run on virtually any monolithic parallel machine; the latter means it can also run on a distributed network of heterogeneous machines, including ones that are geographically dispersed.

Of the packages discussed above, PHISH is most similar to Storm, which also uses ZMQ for socket-based communications. Though we began working on PHISH before Storm was released, the model the two packages use for connecting computational processes together in different communication patterns to enable parallelism is also similar. Storm has additional features for fault tolerance and for guaranteeing that each datum in the stream is processed (and not dropped), which PHISH does not provide. On the MPI-based parallel machines we most often run on, these issues are not as important.

In the next section we give a brief description of the PHISH framework and its features. We illustrate with an example of how both a traditional and streaming MapReduce computation can be performed in parallel. In section 3, we outline three graph algorithms, implemented using PHISH, that operate on edges arriving as a stream of data. Finally, in section 4 we provide benchmark timings for prototypical stream operations, running on a traditional parallel HPC platform, using both the MPI and socket options in PHISH. The results are a measure of the maximum stream rates that PHISH can process.

2 PHISH Pheatures

PHISH, which stands for Parallel Harness for Informatic Stream Hashing, is a lightweight framework, written in a few 1000 lines of C++ and Python code. Aside from the acronym, we chose the name because "phish" swim in a stream (of data in this case).

The framework has two parts. The first is a library, with a C interface, which can be called from programs written in any language (C, C++, Fortran, Python, etc). Such a program typically performs a specific operation on individual datums in the stream. We call such simple programs "minnows", though they can be more complex (sharks, or even whales). Writing a minnow typically only requires defining one or more callback functions which will be invoked when a datum arrives. Additional library functions can be called to unpack a datum into its constituent fields, and pack new datums to send downstream to other minnows. The PHISH library handles the actual communication of a datum from one minnow to another, via the MPI or ZMQ libraries it was linked to.

The second part of the framework is a pre-processing tool called "bait" which enables a computation using one or more minnows to be specified. The bait tool reads a simple input script with 3 kinds of commands. "Minnow" commands specify what executable to launch, and any command-line arguments it needs. "School" commands specify how many minnows of each type to launch and, optionally, how to assign them to physical processors. "Hook" commands define a communication pattern used to route datums between minnows in schools. Examples of such patterns are "paired" (each sending minnow sends to one receiver), "roundrobin" (each sender sends to a different receiver each time), "hashed" (send to a specific receiver based on a hash operation), and

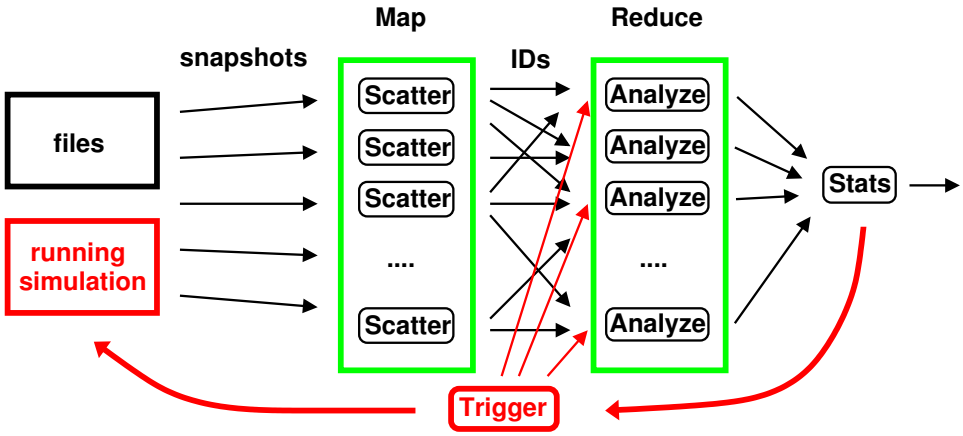

Figure 1: *A PHISH net (in black and green) for performing a traditional or streaming MapReduce on snapshots produced by a particle simulation. The school of scatter minnows perform a "map" operation in parallel; the analyze minnows perform a "reduce". The minnow in red can be added to enable real-time interaction with a running simulation.*

"bcast" (send to all receivers). The "bait" tool converts the input script into a file suitable for launching all the minnow executables in parallel. The user can choose to have the bait tool invoke the parallel job as well, which it does via an "mpirun" command (if using MPI) or "ssh" commands (if using ZMQ). In the latter case, the bait tool also initiates and synchronizes all the necessary socket connections between the various minnows.

The commands in the input script define a PHISH "net" which specifies the sequence of computations to perform on a stream of datums and the topology of how datums flow between minnows. When a PHISH net is launched, it becomes a distributed-memory parallel program, a MPMD model (multiple program, multiple data) in parallel taxonomy. Each minnow runs as its own process with its own private memory, not as a thread.[1] Within the PHISH library, datums are sent from minnow to minnow using standard message-passing techniques (MPI, ZMQ).

In a streaming context, a PHISH program can exploit parallelism in the two ways mentioned in Section 1, via the school and hook commands described above. The stream of incoming data can be split, so that individual minnows compute on portions of the stream. Second, state information on datums in the stream can be stored across the aggregate memory of many minnows. For example, as discussed in Section 3, the edges of a large graph can be distributed in this manner, enabling analysis of a larger graph than fits in one node's memory.

To make this concrete, consider the PHISH net illustrated in Fig 1. The black and green portions represent a MapReduce operation [12], which can be performed in a traditional batch-oriented manner on archival data (i.e. similar to Hadoop [1]), or in streaming mode.

The calculation performed by this MapReduce is motivated by an example given in [22]. Imagine a large-scale particle simulation has generated an archive of snapshot files. A post-processing calculation reconstructs the trajectory of each particle and performs an analysis on it for statistical purposes. E.g. how many times did some atom approach a binding site on a protein, or how many times did particles traverse a porous membrane. Doing the analysis for an individual trajectory is easy, however reconstructing each trajectory requires data (for a specific particle) from every snapshot. The MapReduce is a means to "transpose" the data from a per-snapshot to a per-trajectory representation.

[1] A minnow, meant to run on a multi-core node, could be written so that it spawns threads, which in turn use the node's memory to store shared state across threads.

The left green box is a school of "scatter" minnows; the right box is a school of "analyze" minnows. Each can contain as many minnows as desired. An individual scatter minnow reads one or more snapshots from disk. It parses the snapshot into datums, one per particle. A datum contains the particle ID, its coordinates, and a timestamp. It sends the datum to a unique analyze minnow, determined by hashing on the particle ID. This ensures the same analyze minnow will receive all datums for that particle, from all scatter minnows. This mode of hashed all-to-all communication is indicated in the figure by the overlapping arrows between the scatter and analyze schools. When all snapshots have been read, the scatter minnows signal the analyze minnows, and the analyze minnows can begin their analysis. For each particle they own, they sort the received datums by timestamp to reconstruct a trajectory. Then they analyze the trajectory and send an appropriate result as a datum to the "stats" minnow. When the analyze minnows are done, they signal the stats minnow and it outputs aggregate statistics.

Similar to a MapReduce with Hadoop, the user need only write the scatter, analyze, and stats functions; the PHISH framework provides the rest of the functionality, including movement of datums between the minnows. Unlike Hadoop, PHISH does not run on top of a parallel file system or provide fault-tolerance or out-of-core capabilities (unless the user writes minnows that archive data to disk and later retrieve it). Also note that unlike Hadoop which moves data between processors in big blocks (e.g. during a shuffle operation), the communication here is fine-grained and continuous; a datum for one particle is a few bytes.

With minor modifications (in red), the PHISH net of Fig 1 can also perform its MapReduce in streaming mode. If it runs at the same time as the simulation, the scatter minnows can read snapshots from disk as they appear, or directly from the processors running the simulation. The analyze minnows can store particle datums in sorted order as they arrive. An additional "trigger" minnow could be added which responds to user input, either from the keyboard or a created file. The user requests a status report on the running simulation, which the trigger minnow relays to the analyze minnows. They examine the trajectories they have thus far and send results to the stats minnow. The results can be perused by the user, or relayed back to the running simulation to alter its parameters, i.e. in a computational steering scenario.

Additional features of the PHISH framework, useful for implementing streaming algorithms, are as follows:

- Minnows can define multiple input and output "ports" for receiving and sending datums of different kinds, and minnows are free to assign semantic meanings to ports in any way they choose. This allows flexible combinations of minnows to form complex PHISH nets. For example, the analyze minnows of Figure 1 are using 2 input ports, one for datums from scatter minnows, and one for datums from the trigger minnow.

- Minnows typically wait to perform work until a datum arrives. PHISH allows a callback function to be specified which is invoked by the library when the input queue of datums is empty, so that the minnow can perform additional computations when they would otherwise be idle.

- Minnows in a PHISH net may send datums to themselves, as in Figs 2 and 5. The library treats such messages efficiently, without using MPI or ZMQ. The library also allows received datums to be queued for delayed processing.

- A PHISH net can be setup to run continuously (until the user kills it) on an infinite stream of data. Or it can be run on a finite stream, e.g. by reading data from disk. In the latter case, special termination datums can be sent which enable each minnow to exit gracefully.

- As data flows through a PHISH net, one or more minnows can be bottlenecks, e.g. due to the expense of the computations they perform. This can cause overflows in communication buffers. The ZMQ socket library handles this transparently; the stream of data is effectively throttled to flow at the rate allowed by the slowest minnow. Some MPI implementations will do this as well, but some can generate errors. PHISH has a setting to invoke synchronized MPI send calls for every Nth datum, which handshake between the sender and receiver, avoiding this problem.

- PHISH formats a datum as a collection of one or more fields, each of which is a primitive data type (4-byte integer, 8-byte unsigned integer, 8-byte double, character string, vector of 4-byte integers, etc). Because the fields have precise types (4-byte integer, not just an integer), datums can be exchanged between minnows written in different languages or running on different machines. Minnows can also build on the low-level format to implement their own custom datum format, e.g. a list of labeled key/value pairs.

- A PHISH wrapper (pun intended) for Python is provided, so that minnows can be written in Python. This is a convenient way to develop and debug a streaming algorithm, since Python code is concise and Python and C/C++ minnows can be used interchangeably (see the previous bullet). Callbacks from the PHISH library to a Python minnow (e.g. when a datum is received) require an extra layer of function calls, which introduces some overhead. The performance difference is benchmarked in Section 4.

- PHISH programs are very portable. As with any MPI or socket program, a PHISH net can be run on a single processor, a multi-core desktop machine, or a shared-memory or distributed-memory cluster, so long as the platform supports MPI or socket communication (via ZMQ). This includes running with more minnows than physical processors, though performance may suffer. Using ZMQ, a PHISH net can even be run on a geographically dispersed set of heterogeneous machines that support socket communication.

- In many cases, minnows can be written generically, to accept multiple input datum types (e.g. integers or floating-point values or strings). This makes it easier to re-use minnows in multiple PHISH nets.

- A minnow can look for incoming datums on a socket port. It can likewise export data to a socket port. This means that two or more PHISH nets can be launched independently and exchange data. This is a mechanism for adding/deleting processes to a calculation on the fly.

- There are minnows included in the PHISH distribution which wrap non-PHISH applications that read from stdin and/or write to stdout. This allows such an application to be used in a PHISH net and exchange datums with other minnows.

3 Graph Algorithms

In this section we outline three graph algorithms we have devised that operate in a streaming context, where edges of a graph arrive continuously. All have been implemented in PHISH, as described below. The first is for enumerating triangles, the second for identifying sub-graph isomorphism matches, and the third for finding connected components. A nice survey of graph algorithms for streaming data is given in [19], where various extensions to the fundamental one-pass streaming model are considered.

These particular algorithms, whether used in streaming mode or to analyze an archived graph, are useful for characterizing the structure of the graph or finding patterns within it, e.g. in a data

mining sense. The streaming versions of the algorithms exploit parallelism, both by storing the graph in the aggregate distributed memory of a set of processors, and by spreading computations across processors. The algorithms are described for undirected graphs, but could be modified to work with directed edges as well.

We note that these graph algorithms do not meet all the criteria of resource-constrained streaming algorithms described in Section 1. In particular, they store all the edges of the graph (in a distributed manner), at least for some time window, and are thus not sub-linear in their memory cost. But as the benchmark results in Section 4 indicate, they can operate in real-time for fast stream rates. For the first two algorithms this is true, so long as the graph structure is sufficiently sparse that relatively few triangles or sub-graph matches exist. Otherwise the arrival of a new edge may trigger substantial communication and computation which could slow the rate at which new edges are processed. (Although any source of edges will typically allow for some buffering.) The third algorithm, for connected components, processes edges in constant $O(1)$ time, so that it is guaranteed to keep up with streams, up to some maximum rate.

3.1 Triangle enumeration

A triangle in a graph is simply 3 vertices IJK connected by 3 edges (I, J), (J, K), (I, K). The computational task is to enumerate (or just count if desired) all such triangles in the graph. A MapReduce algorithm for this operation was described in [10], and its implementation in a MapReduce framework built on top of MPI was detailed and benchmarked in [21]. The MapReduce algorithm enables triangles to be found in parallel in an out-of-core fashion on huge graphs that cannot be stored in memory.

In streaming mode, as each edge arrives, we will identify all triangles induced by that edge. At any point in time, all triangles for the current graph will thus have been found. If a final edge arrives (for a finite graph), all its triangles will have been found.

A PHISH net for performing this operation is shown in Fig 2. There is a source of edges, which could be one or more files, or a synthetic edge generator such as for a randomized sparse R-MAT matrix [9] (useful for testing), or a real-time data stream. The edge source sends each edge (I, J) to the specific triangle minnow that "owns" vertex I within the school of minnows in the green box. This is done via the "hashed" communication mechanism described in Section 2, by using the hash of the vertex ID to both route the datum and assign ownership.

The data structure maintained by each triangle minnow is a list of edges for each vertex that it owns. Each edge (I, J) is thus stored twice, once by the minnow that owns vertex I and once by the minnow that owns J. When edge (I, J) is added to the graph, the triangles that contain it are those that include a vertex K that is a neighbor vertex (edge) of both I and J. The list of all such K vertices can be identified if the two minnows that own I and J exchange neighbor information. These are the minnows highlighted in red and blue in Figure 2; the details of the algorithm are outlined in Figure 3.

In the first stage (receive a datum on port 0), the triangle minnow owning vertex I stores edge (I, J), assuming the same edge has not been seen before and is not a self edge with $I = J$. It then sends the edge as (J, I) to the owner of J (on port 1). Note that this could be itself if the same minnow owns both I and J.

In stage 2, the owner of J receives the edge and stores it, then sends the edge plus the current degree D_j of J, back to the owner of I. The purpose of exchanging degree information is to determine which of the 2 vertices has smaller degree, so that the shorter of the two neighbor lists can be sent, to minimize communication. A similar idea is exploited in the MapReduce triangle enumeration algorithm [10, 21] at an intermediate stage when low-degree vertices are selected to

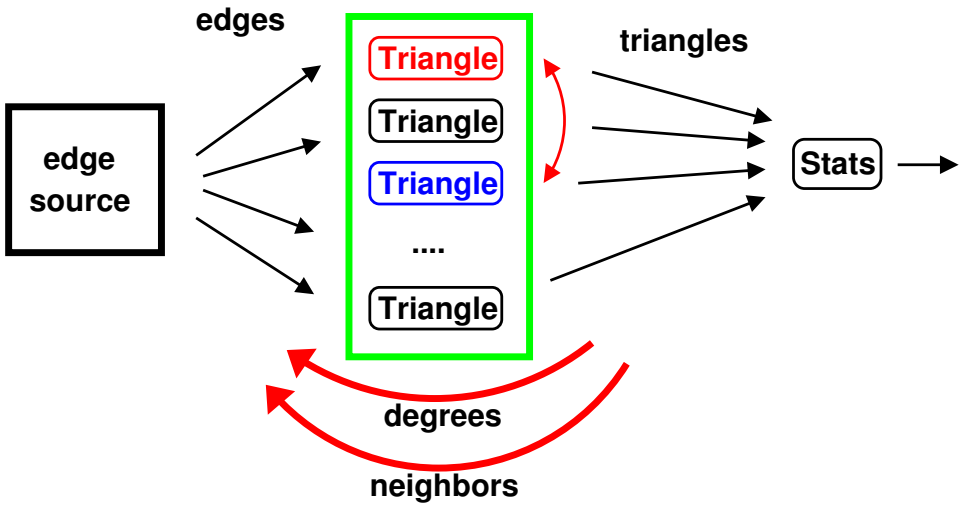

Figure 2: *PHISH net for enumerating triangles in a stream of incoming graph edges. Each triangle minnow stores a fraction of the edges. Exchange of edge degree and edge neighbor datums occur between the pair of triangle minnows (blue,red) owning each vertex of a new edge that arrives. Found triangles are sent to the stats minnow.*

Input	Method	Output
port 0: (I, J)	Store edge with I; send edge to owner of J	port 1: (J, I)
port 1: (J, I)	Store edge with J; send degree of J to owner of I	port 2: (I, J, D_j)
port 2: (I, J, D_j)	Compare degree of I to D_j; if $D_i > D_j$: request neighbors of J else: send neighbor list of I to J	port 3: (J, I, D_j, D_i) port 4: (J, I, D_j, N_i)
port 3: (J, I, D_j, D_i)	Send neighbor list of J back to I	port 4: (I, J, D_i, N_j)
port 4: (I, J, D_i, N_j)	Find common neighbors K of I and J; send triangle IJK to *stats* minnow	port 0: (I, J, K)

Figure 3: *Triangle enumeration algorithm performed by the "triangle" minnows in Figure 2, whenever an edge (I, J) is received. Except for the input edge (port 0) and output triangles (port 0), all other datums are exchanged between the two minnows owning vertex I and J, as indicated in Fig 2.*

generate "wedges" (triangles without a 3rd side) so as to minimize the number of wedges generated in order to find triangles.

In stage 3, the owner of I receives D_j and compares it to D_i. If D_j is smaller, it sends a request back to the owner of J for its neighbor list. Otherwise it sends the current neighbor list of I to the owner of J, i.e. the list of vertices that share an edge with I. Stage 4 is only invoked if $D_j < D_i$; the owner of J sends its neighbor list back to the owner of I.

The final stage 5 (receive a datum on port 4) is where the two neighbor lists of I and J are compared to find all the K neighbors they have in common. Note that this comparison may be performed by either the owner of I or J depending on which neighbor list was shorter (whether the received datum was sent from stage 3 or 4). Each common K neighbor represents a triangle IJK that now exists due to the arrival of edge (I, J); the triangle is sent to the "stats" minnow, as in Figure 2.

Note that each minnow in the triangle school performs all the stages of this algorithm, depending on which of the 5 input ports an incoming datum was received on. In a real-time sense, the graph edges are streaming into the school of triangle minnows and being continuously stored. Each new edge (I, J) triggers a back-and-forth exchange of datums between two minnows in the school that own I and J as they swap degree and neighbor information. In aggregate, these additional datums are sent from the school of triangle minnows to itself (on other ports), via hashed all-to-all communication. This is indicated in Figure 2 via the reverse arrows for "degrees" and "neighbors". Thus an individual triangle minnow receives a mixture of incoming edges and degree/neighbor datums from other triangle minnows.

In a serial or MapReduce algorithm for triangle enumeration, it is easy to insure each triangle is found exactly once. In streaming mode, it is more difficult, due to unpredictable delays between arrivals of datums. The algorithm of Figure 3 does not miss any triangles, but can find a triangle more than once. Duplicates are reduced by two features of the algorithm. The first is waiting to trigger degree and neighbor exchanges until a new edge (I, J) is stored the second time by vertex J. If this is not done, a second edge (J, I) in the stream may trigger a duplicate triangle search before the minnow owing J knows that (I, J) was also received by the minnow owning I. The second feature is to limit the size of communicated neighbor lists to D_i or D_j at the time their degree was first computed, not their current degree at the time they are sent. The latter will be larger if new edges have since arrived, leading to extra communication, computation, and duplicate triangles.

Finally, we note that sending the smaller of the neighbor lists N_i or N_j does not mean the communication and associated computation is always small. If two high-degree vertices are finally linked by an incoming edge (I, J), then a large number of triangles will likely be found and emitted by stage 5 of the algorithm.

3.2 Sub-graph isomorphism matching

Before presenting the streaming algorithm for sub-graph isomophism (SGI) matching, we define what an SGI match is. A semantic graph is one whose vertices and edges have labels, as represented by colors in Figure 4. A small "target" graph is defined, e.g. the 5-vertex, 6-edge graph at the left of the figure. The computational task is to find all isomorphic matches to the target graph in the full graph. A matching sub-graph is one with the same topology and same coloring of vertices and edges as the target graph.

A shared-memory parallel algorithm for this operation was described in [8], and a distributed-memory parallel MapReduce algorithm in [20]. The latter allows SGI matches to be found in parallel within huge graphs that cannot be stored in memory. In streaming mode, as with the triangle algorithm, as each edge arrives, we will identify all SGI matches induced by that edge.

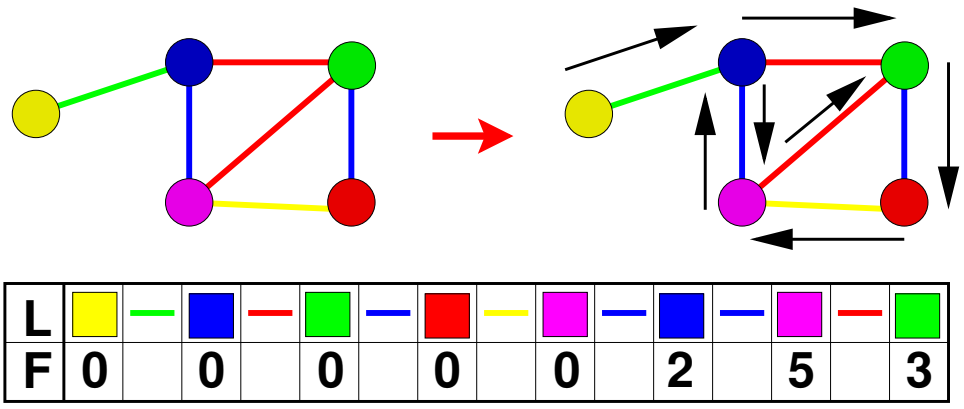

Figure 4: *Conversion of a small semantic (colored) graph (left) to an ordered path (right and bottom) that touches all vertices and edges. The topology of the graph is encoded in the vertex flags (F at bottom), which indicate a vertex is new (0) or previously seen as the N^{th} vertex.*

At any point in time, all matches for the current graph will thus have been found. If a final edge arrives (for a finite graph), all SGI matches will have been found.

The shared-memory and MapReduce algorithms for this operation both employ a representation of the target graph called an ordered "path", which the streaming algorithm will also use. It is illustrated in Figure 2. The target graph is traversed such that each vertex and edge is visited at least once, as with the arrowed graph at the right of the figure. In this case the traversal is a 7-step path through the target graph. Note that the path is not unique; other traversals also yield valid paths. Computationally, the shorter the path, the more efficient the matching algorithm will be. The colored vertices and edges encountered along the path are tallied as "L" at the bottom of the figure. The topology of the target graph is encoded by the "F" flags. A zero value indicates the vertex is visited for the first time; a value of $M > 0$ indicates the visited vertex is also the Mth vertex in the path. E.g. the blue vertex with $F = 2$ is the same as the 2nd vertex (also blue). For reasons explained below, in the streaming algorithm, N paths are generated as a pre-processing step, one that begins with each of the N edges in the target graph.

A PHISH net for identifying SGI matches is shown in Fig 5. As with the triangle net, there is a source of edges, which could be from files, a synthetic edge generator, or a real-time data stream. The edge source sends each edge (I, J) to the specific SGI minnow that "owns" vertex I within the school of minnows in the green box, via "hashed" communication, as described in Section 2.

As with the triangle algorithm, the data structure maintained by each SGI minnow is a list of edges for each vertex that it owns, and each edge (I, J) is stored twice. When edge (I, J) is added to the graph, any new matches to the target graph induced by that edge will be described by one (or more) of the N pre-computed paths that begin with a matching edge. Each match initiates a "walk" through the path that can be extended, one edge at a time, by the minnow that owns the last vertex in the walk. For example, if the minnow highlighted in blue finds 2 neighbor vertices of the last vertex that extend the walk, it sends the extended walk to each of the minnows highlighted in red that own those 2 vertices. If the process can be repeated until all edges in the target graph path are matched, then an SGI match has been found. The details of the algorithm are outlined in Fig 6.

In the first stage (receive a datum on port 0), the SGI minnow owning vertex I receives edge (I, J), along with labels (colors) L_i and L_j on the two vertices and the edge L_{ij}. It stores the edge and labels assuming the same edge has not been seen before and is not a self edge with $I = J$. It then sends the edge and labels as (J, I, L_j, L_i, L_{ij}) to the owner of J (on port 1).

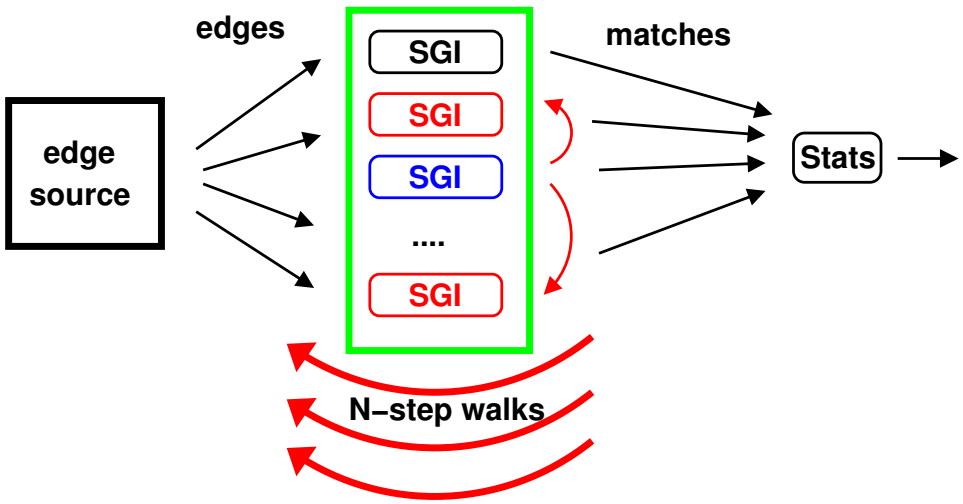

Figure 5: *PHISH net for finding sub-graph isomorphism (SGI) matches to a target graph in a stream of graph edges. Each SGI minnow stores a fraction of the edges. New edges that match an edge in the target graph initiate walks along the path of Fig 4. A blue owner of the last vertex in a walk extends the walk by matching the next edge and sends it to red owners of the new last vertices. After N repetitions complete the path, SGI matches are sent to the stats minnow.*

Input	Method	Output
port 0: (I, J, L_i, L_j, L_{ij})	Store edge with I; send edge to owner of J	port 1: (J, I, L_j, L_i, L_{ij})
port 1: (J, I, L_j, L_i, L_{ij})	If edge matches path P_n; initiate walk by sending to owner of I or J	port 2: $(P_n, [I, J])$
port 2: $(P_n, [I, \cdots, J])$	Find neighbors K of J that extend walk; send extended walk to owner of K; if complete: send SGI match to stats minnow	port 2: $(P_n, [I, \cdots, J, K])$ port 0: $([I, \cdots, J])$

Figure 6: *Sub-graph isomorphism matching algorithm performed by the "SGI" minnows in Figure 5, whenever an edge (I, J) is received with vertex and edge labels (L_i, L_j, L_{ij}). The algorithm generates walks through the ordered path of Fig 4, matching the path one edge at a time via datums sent and received on port 2.*

In stage 2, the owner of J receives the edge and labels and stores them. It then checks if the edge is a match to the first edge in any of the N pre-computed paths that represent the target graph. A "match" means that the two vertex labels L_i and L_j and edge label L_{ij} of the incoming edge all match the path edge. For an asymmetric edge ($L_i \neq L_j$) the match can be for (I, J) or (J, I). If a match is found, (I, J) is an initial one-edge walk along path P_n, where n ranges from 1 to N. Each match is sent to the owner of the final vertex in the walk as $(P_n, [I, J])$ (on port 2). Note the final vertex can be either I or J.

Stage 3 (receive a datum on port 2) is repeated as many times as necessary to extend a walk into a match to the full path. The edges of the last vertex in the walk are scanned to find matches to the next edge in path P_n. As before, a match means the vertex and edge labels match. It also means that the constraint flag F is satisfied, i.e. the new edge vertex does not appear in the walk if $F = 0$ or it appears as the Mth vertex in the walk if $F = M > 0$. For each matching edge, the walk is extended by the neighbor vertex K and sent to the owner of K as $(P_n, [I, \cdots, J, K])$. Note that if N matching edges are found, a single incoming walk is output as N extended walks. If no matches are found, the walk vanishes, since it is not communicated further. This process repeats until the walk is a full-length match to the path P_n, at which point it is sent as an SGI match on port 0 to the "stats" minnow, as in Fig 5.

Each minnow in the SGI school performs all the stages of this algorithm, including multiple repetitions of stage 3. Graph edges are streaming into the school of SGI minnows and being continuously stored. Each new edge (I, J) that matches some edge in the target graph triggers a cascade of walk datums sent within the "neighborhood" of minnows owning I and J, which includes the owners of the vertices that are "close" to I and J, i.e. neighbors of neighbors of neighbors out to some range that spans the target graph diameter. In aggregate, these additional walk datums are sent from the school of SGI minnows to itself (on port 2), via hashed all-to-all communication, indicated in Figure 5 via the reverse arrows for the "N-step walks".

As with the triangle enumeration algorithm, this algorithm does not miss any SGI matches, but can find duplicate matches, depending on the time of arrival of incoming edges and random delays in subsequent messages they induce between SGI minnows. For example, if two edges in the same SGI match arrive at nearly the same time, they may both initiate walks that complete because the other edge arrives before the growing walk reaches it.

Finally, we note some differences between the streaming version of this algorithm and its shared-memory [8] and MapReduce [20] counterparts. Because the latter operate on a complete graph after it has been stored, they need only represent the target graph with a single path. If the character of the full graph is known *a priori*, the path can be ordered so that vertex and edge labels that appear less frequently come early in the path. This can greatly reduce the number of walks that are generated. The streaming algorithm cannot take advantage of either of these ideas. N paths, each starting with a different edge, are needed to represent the target graph, because an incoming edge may match any target graph edge, and the SGI matches it induces cannot be found until it arrives. This also means that walks that begin with edges that appear with high frequency are all initiated and followed to completion or termination. In practice, this means the streaming SGI algorithm is more likely to generate large numbers of walks that do not end in matches than its non-streaming cousins. As with the triangle algorithm, the addition of a single edge to the graph can trigger a large number of new datums.

3.3 Connected component finding

A connected component (CC) of a graph is a set of vertices whose edges can be traversed to reach any other vertex in the set, but no vertices not in the set. Sparse graphs may have one or many

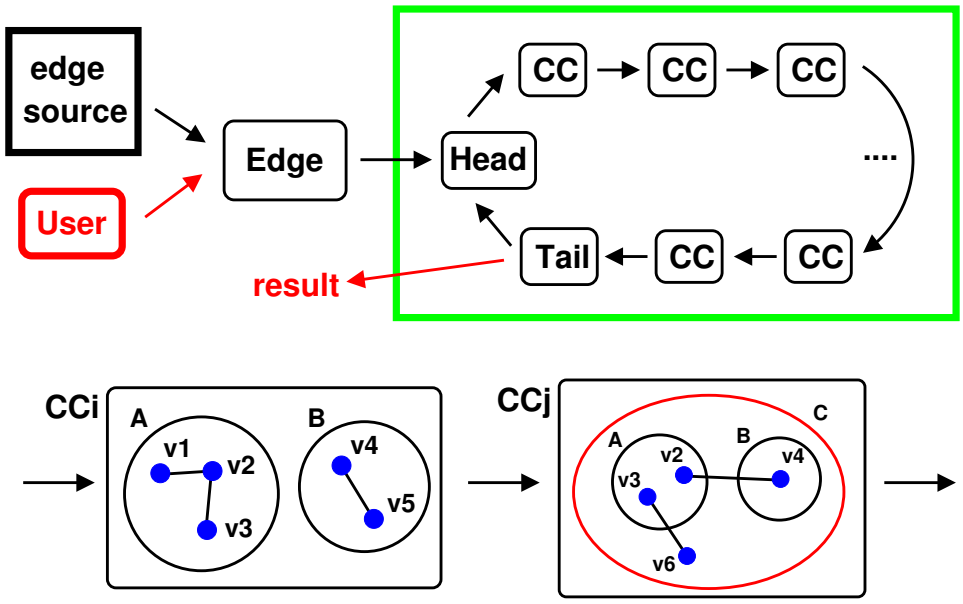

Figure 7: *PHISH net for finding connected components (CC) in a stream of graph edges which circulate around a ring of CC minnows from head to tail. The edges are stored in a hierarchical data structure across the CC minnows, as illustrated at the bottom for two of the CC minnows, which encodes the component structure. User queries can be interleaved into the stream and results output by the tail minnow.*

such components.

A PHISH net for finding CCs is shown in Figure 7. As before, edges arrive from some external source (file, real-time data stream). The "edge" minnow sends them along to a school of CC minnows (green box) configured as a ring. The edge minnow also polls for occasional queries from a "user" minnow which are converted to specially formatted datums and interleaved between the graph edges. Sample queries might be "which CC is vertex I a part of?" or "how many small components of size $< N$ are there?". The CC minnows store the graph in a distributed data structure that allows them to answer the queries.

All communication between the CC minnows is either graph edges or additional datums interleaved between the edges that circulate around in the ring in one direction. An individual CC minnow thus inputs and outputs a single stream of datums. It may store edges, grouped by component, as in the figure for minnow CC_i storing components A and B. It also has a memory limit for the number of edges it can store. When it exceeds the limit, it sends edges along to the next CC minnow, but may alter the edge vertices to reflect the component a vertex is in. Thus the next minnow CC_j sees components A and B as pseudo-vertices, without knowing their internal structure of edges and vertices. A new edge such as (V_2, V_4) received by CC_j may connect components A and B into one component C. The graph is thus stored in an hierarchical fashion, akin to a set of nested Russian dolls, across the CC minnows. Over time, CC minnows in the ring fill up with edges one by one, until the "tail" starts to fill. At that point, the tail signals the head that "old" edges need to be deleted and memory freed up. The CC minnows jettison edges based on a stored timestamp, and adapt the component data structures on the fly as new edges continue to arrive. Adapting the data structures requires communication of additional datums, which are again interleaved into the continuous stream of incoming graph edges.

A key attribute of this streaming CC algorithm is that all of its data structures can be searched and updated in constant $O(1)$ time as each edge arrives or as aging is invoked, for an infinite

stream of edges. This means the algorithm is guaranteed to "keep up", for some maximum allowed stream rate of input edges. The maximum depends on the ratio of communication to computational cost. The details of how the algorithm does this are beyond the scope of this paper. There are complex issues associated with aging (deletion) of edges, and handling certain edges when a CC minnow's memory is already full. Details of the entire algorithm are provided in [7]. Here we have simply highlighted what PHISH provides to enable the ring-based algorithm to process edges and queries. An additional PHISH feature, which the algorithm uses, is the ability to reconfigure the communication pattern between minnows on the fly. When the tail minnow fills up and aging occurs, the ring topology may rotate one position, with the second CC minnow becoming the head, the old head becoming the new tail, and the edge minnow sending edges to a new head. Or the ordering of the ring may be permuted with a new minnow becoming the tail. The PHISH library has functions which allow for these reconfigurations, which the CC minnows can invoke.

Finally, we note that the triangle enumeration and SGI matching algorithms discussed earlier in this section could also be adapted to infinite edge streams, like the CC algorithm, with minor modification. If a time stamp is stored with each edge, then when memory fills up, each triangle or SGI minnow could jettison old edges as needed. Future new edges would then only generate triangles or sub-graph matches in the time-windowed graph that includes recently stored edges.

4 Performance Results

In this section we use PHISH to benchmark three simple operations, which are communication kernels in many streaming algorithms, including those of the preceding section. We also benchmark a fourth operation, that communicates graph edges in a hashed manner and stores them by vertex in the aggregate memory of a set of minnows. This is a also a common operation in graph algorithms like those of the preceding section, and is a test of both datum communication and the use of a prototypical graph data structure.

The first kernel exchanges a datum, back and forth between 2 minnows. The second sends datums in one direction down a chain of minnows. The third uses one school of minnows to send datums in a hashed manner to another school. These 3 tests were run across a range of datum sizes and minnow counts. They were also performed in 4 PHISH modes: using MPI or ZMQ (sockets) as the backend, and using minnows written in C++ or Python. The latter uses a Python wrapper on the PHISH library. Performance in a fifth mode was also tested, using traditional SPMD (single-program multiple-data) MPI-only implementations of the 3 kernels, which allows PHISH overhead to be measured. The fourth kernel generates graph edges, and sends them in a hashed manner to minnows that store them in a simple data structure, as a list of edges per vertex.

All of the tests were run on a large production Linux cluster at Sandia. It has 2.93 GHz dual quad-core Nehalem X5570 processors (8 cores per node) and a QDR InfiniBand interconnect configured as a 3d torus. Tests with MPI used OpenMPI 1.4.3; tests with ZMQ used version 2.2.0. Runs were made on up to 32 nodes of the machine (256 cores); the hash benchmark was also run on 1024 nodes. To insure the measured stream rates came to equilibrium, each test was run for at least 30 seconds and repeated twice, averaging the results.

We also tested how the mapping of minnows (processes) to cores of the multi-core machine affected performance. For the chain and hash tests, 3 mapping modes were tested. The first two modes use all cores of as many nodes as required, based on the total process count. The first mode assigns minnows first by core, then by node. The second mode assigns minnows first by node, then by core. For example, if 16 minnows are used in the PHISH net, both the first and second mode use all the cores of 2 nodes (8 cores per node). The first mode runs minnows 1-8 run on the first node, and 9-16 on the second node. The second mode runs odd-numbered minnows on the first

node, and even-numbered minnows on the second node.

The third mode uses only a subset of cores on each node if possible. As in the second mode, minnows are assigned first by node, then by core. However all the nodes allocated for the batch job are used (32 in our case, or 256 cores). Thus for a 16-minnow test, each minnow is assigned to a single core of a different node. The remaining cores on those nodes (as well as the entire other 16 nodes) are idle. As seen below, runs in these various modes perform differently. This is because they stress intra-node communication (between cores in a node) versus inter-node communication capabilities differently.

It is worth noting that all these tests perform message passing in a somewhat unnatural mode for a distributed-parallel memory machine, whether via MPI or ZMQ. For optimal communication performance, traditional MPI programs often bundle data together into a few, large messages, and each processor typically communicates with a handful of neighbor processors, e.g. by spatially decomposing a physical simulation domain. By contrast, in these tests, millions of tiny messages are sent in various patterns, including an all-to-all pattern for the hash test, and there is little computation to offset the communication cost, even when the kernels are used in a graph algorithm. Yet, as we shall see, MPI in particular supports this fine-grained style of communication fairly well.

4.1 Pingpong test for latency

The first test sends a datum back and forth between 2 minnows, in a pingpong fashion. Its performance, measured in millions of datums per CPU second, is shown in Figure 8, for datum sizes from 0 to 16K bytes (empty 0-byte datums in PHISH are actually a few bytes due to datum type and port information included in the message). For a 0-byte datum this is effectively a latency test, with latency being the inverse of the rate. The two minnows were run either on two cores of a single node (circles in the plot, (c) in the legend), or on a core of two different nodes (X's in the plot). Using both cores within a single node was faster in all cases, since latencies for intra-node communication are smaller than for inter-node.

For small datums, a rate of over 1 million datums/sec is achieved (for the intra-node layout) by native MPI as well as by PHISH running on top of MPI, which adds little overhead in this case, as well as for the other benchmarks discussed below. Running minnows written in Python degrades the performance by a factor of 2-3x. For the ZMQ (sockets) backend, the small-datum rate is about 30x slower than PHISH with MPI. This is because sockets are not optimized for this style of back-and-forth communication, whereas MPI is.

For larger datums, communication bandwidth becomes the bottleneck rather than latency, and the rate degrades, though PHISH with MPI can still exchange over 100K datums/sec for 16K byte datums. ZMQ now performs better relative to MPI.

4.2 Chain test for throughput

The second test configures a school of P minnows as a 1d chain. The first minnow sends datums to the second as rapidly as it can. The second minnow receives each datum and sends it to the third, and so on down the chain. The final minnow in the chain receives and discards all the datums, and signals the first minnow when it is done. Since the communication is one-way, this is effectively a test of throughput.

Results of this test for varying minnow count (chain length) are shown in Figure 9 for 0-byte datums. Ideal performance would be a constant rate, independent of chain length, meaning the stream of data flows unimpeded through an arbitrary number of minnows. Most of the curves flatten out once the chain length is a few minnows, though the rate degrades slowly as chain length

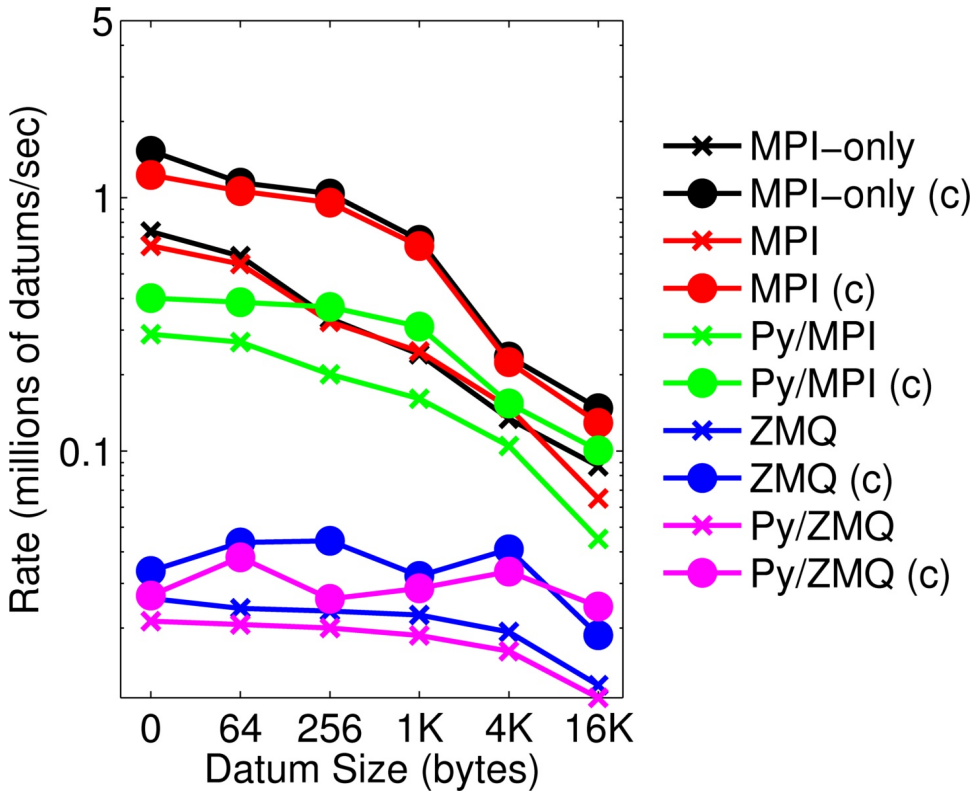

Figure 8: *Rate of datum exchange between 2 minnows in a pingpong test for varying datum sizes. All but the MPI-only curves are for PHISH, using its MPI or ZMQ backend, with minnows written in C++ or Python. The test was run on 2 cores of the same node (circles) or on 1 core each of 2 different nodes (X's).*

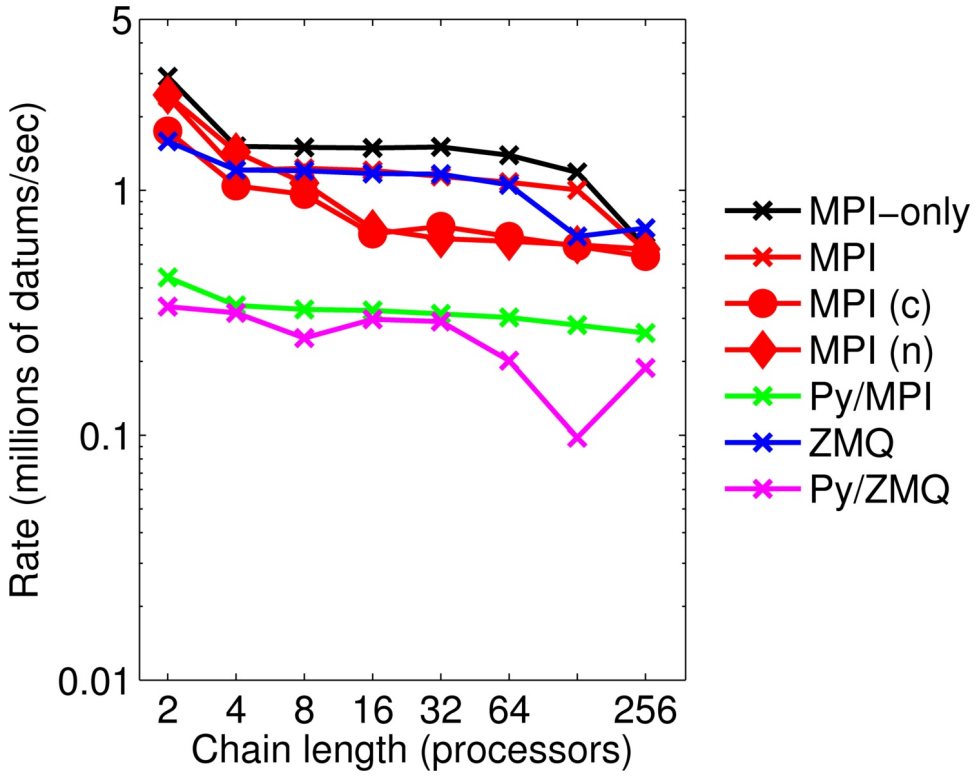

Figure 9: *Rate of sending 0-byte datums in one direction along a chain of minnows of varying length. All but the MPI-only curves are for PHISH, using its MPI or ZMQ backend, with minnows written in C++ or Python. Curves with X's may use less than all 8 cores per node. Two of the MPI tests (red) use all cores on each node, ordering minnows by core (circles) or by node (diamonds), as discussed in the text.*

increases. Note that any minnow in the chain that runs a little slowly, due either to variations in its CPU clock or communication links, will act as a bottleneck for the entire chain.

In contrast to the pingpong benchmark, the ZMQ performance in this test is nearly identical to the MPI performance. This is because sockets are optimized for throughput, i.e. one-way communication, which is also a good match to the streaming paradigm.

As discussed above, the chain test was run with 3 different modes of mapping the P chain minnows to the multi-core nodes. The majority of the curves (X's) used the third mapping mode where minnows were assigned to all 32 allocated nodes, first by node, then by core. Runs in this mode thus used less than 8 cores/node (except on 256 processors), e.g. a chain-length of 16 uses one core on each of 16 nodes. This gave the fastest performance for this benchmark because it provides more inter-node communication capability on a per-core basis. For PHISH running on MPI (red curves), performance for the other 2 mapping modes is also shown. In these modes all 8 cores of the minimum number of required nodes were used, e.g. a chain-length of 16 used only 2 nodes. The curve with circles assigned minnows by core, then by node; the curve with diamonds assigned minnows by node, then by core. Both of these modes ran at a rate about 2x slower than the third mode. Similar trends for the 3 modes were seen for the other PHISH options (ZMQ, Python) as well as the MPI-only case.

Figure 10 shows the rate at which datums of varying sizes from 0 to 16K bytes can be sent down a chain of 64 minnows. For small datums the rate is fairly constant; above 256 bytes the rate degrades due to communication bandwidth limitations. However, 16K-byte datums can still

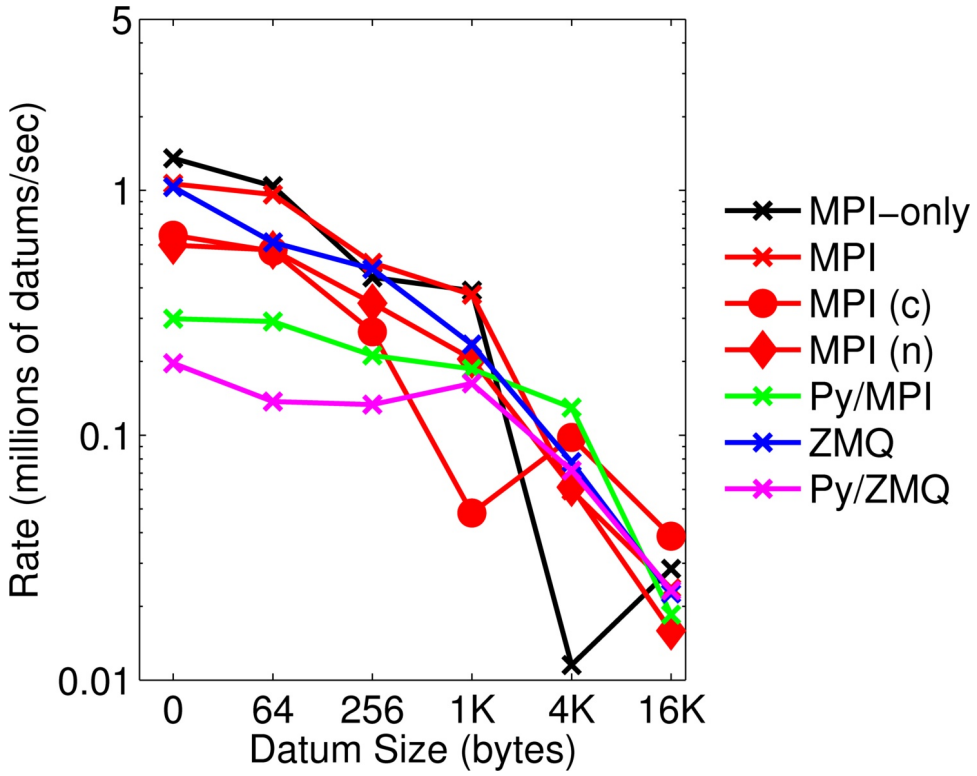

Figure 10: *Rate of sending datums of varying sizes in one direction along a chain of 64 minnows. All but the MPI-only curves are for PHISH, using its MPI or ZMQ backend, with minnows written in C++ or Python. The meaning of curves with X's, circles, and diamonds is the same as in Figure 9.*

be streamed at a rate of tens of thousands per second. The performance differences between the 3 modes of mapping minnows to multi-core nodes are similar to that of Figure 9.

4.3 Hash test for scalability

The third test configures two schools, a source and sink, each with $P/2$ minnows. They are connected via a "hashed" communication pattern. Each source minnow sends datums continuously, using a random number as the value to hash on, so that each datum is sent to a random sink minnow. The $P/2$ source minnows are thus sending in an all-to-all pattern to $P/2$ sink minnows. Ideally, the larger P is, the greater the aggregate datum rate should be, meaning that parallelism enables processing of higher stream rates. Results of this test for varying total minnow count P are shown in Figure 11 for 0-byte datums.

The trend for all of the tests (MPI-only, PHISH on MPI or ZMQ or with Python) is that the rate rises linearly with minnow count until a saturation level is reached. The level depends on the mode of mapping minnows to multi-core nodes. For PHISH on MPI, curves for all 3 mapping modes are shown in red. The performance in the circle curve (mapping by core, then by node) saturates first, at 8 cores, which means 4 source and 4 sink minnows all assigned to one node. The performance in the diamond curve (mapping by node, then by core), saturates at a higher rate on 16 cores. In this case, the 8 source and 8 sink minnows are each split evenly across 2 nodes (4 source minnows on each node). This outperforms the circle curve mapping, which puts 8 source minnows on one node, and 8 sink minnows on the other, by a factor of 4x. This is because communication is one-way from source to sink minnows, and the outbound inter-node communication capability of both nodes is

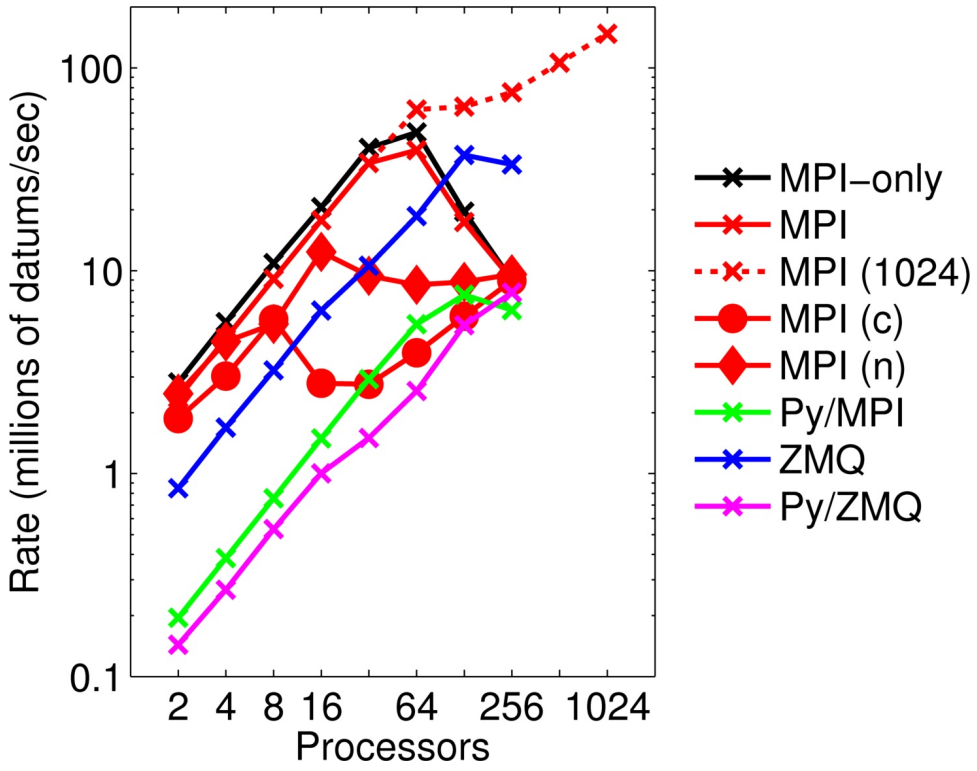

Figure 11: *Rate of sending 0-byte datums in a hashed mode from $P/2$ source minnows to $P/2$ sink minnows, as a function of total minnow count P. All but the MPI-only curves are for PHISH, using its MPI or ZMQ backend, with minnows written in C++ or Python. As in Figure 9, the X's, circles, and diamonds indicate 3 different modes of mapping minnows to multi-core nodes. The solid lines were runs on 32 nodes; the dotted line was a run on 1024 nodes.*

thus being used.

The performance of the third mode of mapping (by allocated node, then by core) is best of all, as shown by the X curves. As with the previous benchmarks, this is because fewer cores per node are used and thus more inter-node communication capability is available on a per-minnow basis. For runs on 32 nodes of the parallel machine (solid lines in Figure 11, this mapping mode does not saturate until 64 processors, at an aggregate rate of ∼40 million datums/sec for PHISH on MPI (or MPI-only). With a 32-node allocation, the 64-processor runs are using 1 core per node as the source, and 1 core per node as the sink. The 128- and 256-core runs use more cores per node, resulting in less inter-node communication capacity on a per-core basis, and performance falls off. Similar trends for the 3 modes were seen for the other PHISH options (ZMQ, Python) as well as the MPI-only case.

We also ran the same test on a 1024-node allocation (dotted line in Figure 11). The aggregate rate now continues to increase to 147 million datums/sec on 1024 processors (1 core per node), with a similar roll-off in performance if more cores per node are used. The rate fell to 49 million datums/sec on 2048 processors (2 cores per node).

The ZMQ performance difference versus MPI for this test is about 2x, which is worse than for the chain test, but considerably better than the pingpong test. Again, datums are streaming in one direction through this PHISH net, which is more compatible with socket throughput capabilities.

Figure 12 shows the rate at which datums of varying sizes can be communicated in an all-to-all hashed mode from 32 source minnows to 32 sink minnows (64 total). As in the chain test, for small datums the rate is fairly constant. Above 1K bytes the rate degrades due to communication bandwidth limitations. 16K-byte datums can still be streamed at a aggregate rate of 1 million per second by the 32 source minnows. The performance differences between the 3 modes of mapping minnows to multi-core nodes are similar to that of the the 64-processor data points in Figure 11.

4.4 Edge test for graph algorithms

Finally, we test two schools of minnows, communicating in a hashed all-to-all manner, similar to the hash test of the previous sub-section. However, the first school of minnows now generate graph edges from an R-MAT distribution [9], as non-zeroes sampled from a uniform sparse matrix, hashing each edge (I, J) on its first vertex I. While random entries in a uniform sparse matrix (non-zero $A_{ij} \Longleftrightarrow$ edge (I, J)) can be generated more quickly by other means, the R-MAT algorithm is more general, since edges from a graph with a highly skewed degree distribution (e.g. a power-law graph) can be generated via the same procedure.

Each receiving graph minnow owns a subset of the vertices (due to the result of the hash function) and stores each received edge in a list of neighbors J associated with I. These operations are essentially the first stages of both the triangle enumeration and sub-graph isomorphism matching algorithms of Section 3. In a benchmark sense, these results indicate the maximum stream rate for incoming edges that a graph algorithm built on top of PHISH can accept, assuming there is an overhead cost to looking up and storing edges in a data structure keyed by vertex ID, before they are processed further.

Figure 13 shows performance results for the benchmark. All the results are for PHISH using its MPI backend, with R-MAT and graph minnows written in C++. Only two cores of each 8-core node were used, one for an R-MAT minnow, one for a graph minnow. Thus when running on 16 processors, the PHISH net had 8 minnows in each school and was run on 8 nodes of the Linux cluster.

The total edge count and size of the associated R-MAT matrix scaled with the number of processors used. The R-MAT matrix was NxN where $N = 2^{22+P/2}$ and the average edge count per vertex

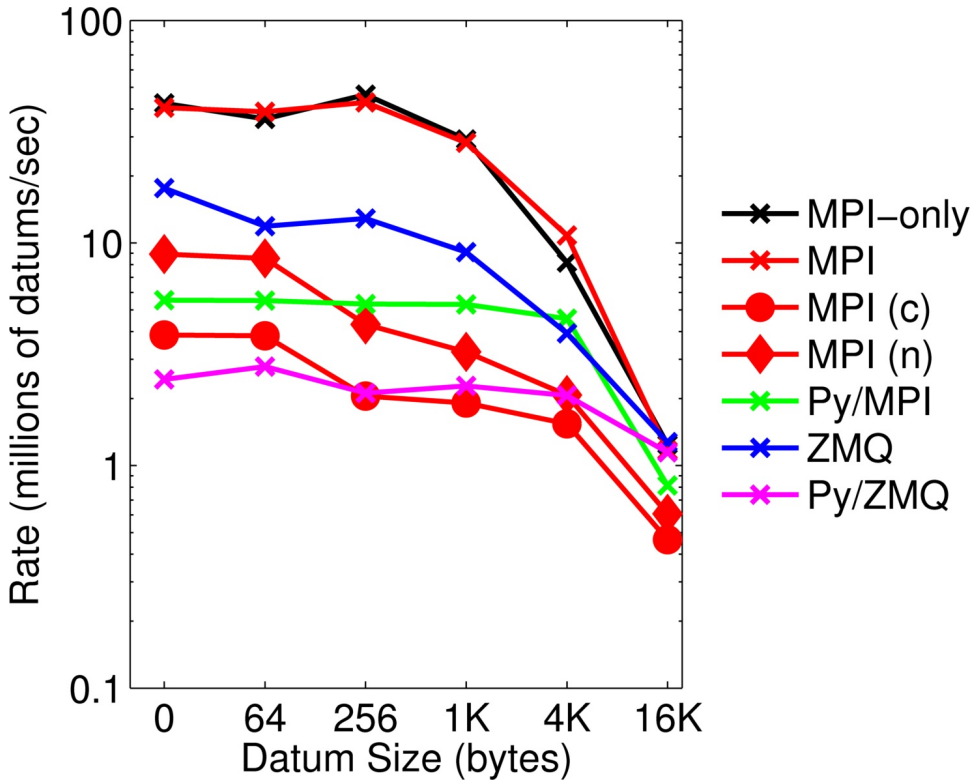

Figure 12: *Rate of sending datums of varying sizes in a hashed mode from 32 source minnows to 32 sink minnows. All but the MPI-only curves are for PHISH, using its MPI or ZMQ backend, with minnows written in C++ or Python. The meaning of curves with X's, circles, and diamonds is the same as in Figures 9 and 11.*

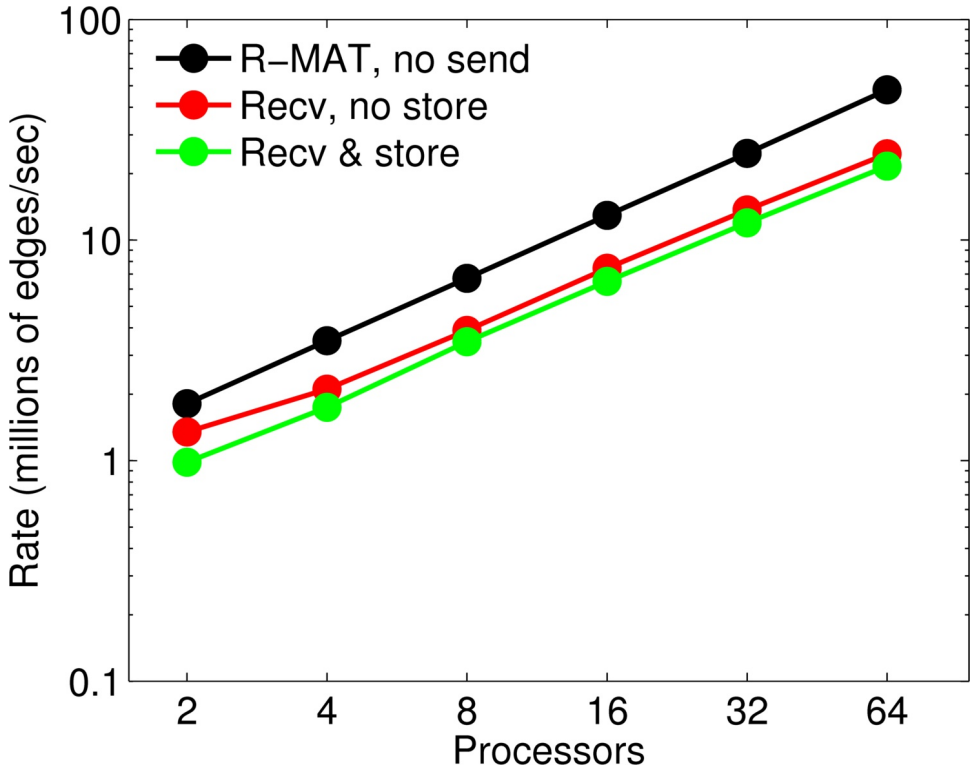

Figure 13: *Rate of generating, communicating, and storing graph edges on $P/2$ R-MAT minnows and $P/2$ graph minnows, as a function of total minnow count P. (Black) R-MAT minnows generate random edges, but do not send them. (Red) Graph minnows also receive edges, but do not store them. (Green) Graph minnows also store edges in a hash table of hash tables, as described in the text.*

was 8. Thus when running on 2 processors (one R-MAT minnow, one graph minnow), $2^{26} \simeq 67.1$ million edges from an order $N = 2^{23} \simeq 8.4$ million matrix were generated and sent by the R-MAT minnow, and were received and stored by the graph minnow. On 64 processors, $2^{31} \simeq 2.1$ billion edges were generated and stored from a sparse matrix with $N = 2^{28} \simeq 268$ million rows and columns.

The figure has 3 sets of results. For the black curve, R-MAT minnows generate edges but do not send them, so the graph minnows are idle. This gives a baseline rate for generating random edges from a large, sparse graph. The R-MAT algorithm generates each edge in a recursive manner, where the number of recursion levels is $\log_2(N)$, with N = 23-28 on P = 2-64 processors in our case. Thus the black curve scales essentially perfectly, if the slow increase in recursion levels is accounted for.

For the red curve, R-MAT minnows send the generated edges to the graph minnows, which receive them, but discard them. The added communication cost drops the effective edge rate by about a third on 2 processors, and in half on 64 processors. Since the graph minnows are operating independent of the R-MAT minnows, the communication between the two schools is now the rate-limiting step; even if the R-MAT minnows generated edges more quickly (e.g. read them from a file or high-bandwidth source), it would not significantly reduce the run time.

For the green curve, graph minnows store the (I, J) edges they receive by looking up vertex I in a primary hash table. The edges J of I are stored in a secondary hash table. The reason to use hash tables is that the cost to lookup or add an entry is $O(1)$, independent of the number of entries

N in the table.[2] This is important for streaming graph algorithms, since retrieving, modifying, or iterating over the edges of a vertex are common operations. Ideally their cost should not depend on the size of the graph, so that the algorithm can keep up with a constant stream rate of incoming edges. In this benchmark, the primary hash table on each processor (keyed by vertex I) is large; each graph minnow stores ∼8.4 million vertices by the end of the run. The secondary hash tables (8.4 million of them, each keyed by edge J) are tiny; on average each has only 8 entries. The figure shows that the additional cost of storing the received edges is small compared to the communication cost (green curve versus red).

The bottom line for Figure 13 is that a single graph minnow can receive and store graph edges at a rate of roughly 1M edges/sec. 32 graph minnows can receive edges, from any of 32 R-MAT sources, communicated in an all-to-all manner, at an aggregate rate of 21.6M edges/sec, for a parallel efficiency of 68%.

5 Conclusions

We have described a freely available software framework, called PHISH, which can be used to develop and deploy algorithms that process streaming or other kinds of big data.

Our chief goal in developing PHISH was to make it easier to process streaming data in parallel on distributed-memory platforms. Because PHISH uses either MPI or the ZMQ sockets library to send stream datums from one process to another, PHISH programs can run on nearly any kind of parallel machine, including networked, heterogeneous, and geographically dispersed machines, so long as they allow socket-based communication.

The benchmark results of Section 4 indicate that on a prototypical HPC platform using MPI, single processors can send and receive small datums via PHISH at rates around a million datums/sec. If a stream is split, so that it is communicated and processed by multiple processes, e.g. by hashing on values in the stream, streams with rates of many tens of millions of datums/sec can be processed.

A secondary goal was to provide a framework that enables rapid development, debugging, and benchmarking of streaming algorithms themselves. As an example, the triangle enumeration and sub-graph isomorphism matching algorithms of Section 3 required "triangle" and "SGI" minnows that consisted of 90 and 140 lines of Python respectively (without comments). We think this is a relatively small amount of code to find triangles and SGI matches in parallel, in a continuous stream of graph edges.

Our own interest is in graph algorithms, but various kinds of statistical, data mining, machine learning, anomaly detection, and even numerical algorithms can be formulated for streaming data. We hope that PHISH can be a useful tool in those contexts as well.

6 Acknowledgments

The PHISH library is open-source software, which can be downloaded from `http://www.sandia.gov/~sjplimp/phish.html`. It is freely available under the terms of the BSD license. PHISH minnows that implement the graph algorithms of Section 3 and the benchmarks of Section 4 are included in the distribution.

[2] The commonly-used std::map container class provided by the C++ Standard Library is not a true hash table, but a binary tree with $O(logN)$ cost for lookup or insertion. We used the std::tr1::unordered_map class for the primary container, and std::tr1::unordered_set class for the secondary; both are hash tables with average-case $O(1)$ cost, and are now part of the C++11 standard.

We thank Jon Berry and Cindy Phillips (Sandia) for permission to discuss their as-yet unpublished streaming connected-component finding algorithm in the context of PHISH. We also thank Karl Anderson (DoD) for many insightful discussions of streaming algorithms and related ideas.

This work was supported by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy, under contract DE-AC04-94AL85000.

References

[1] Hadoop web site: `http://hadoop.apache.org`.

[2] Twitter Storm web site: http://storm-project.net.

[3] IBM InfoSphere Streams web site: http://www-01.ibm.com/software/data/infosphere/stream-computing.

[4] SQLstream web site: http://www.sqlstream.com.

[5] Visualization Toolkit (VTK) web site: http://www.vtk.org.

[6] ϕMQ or ZMQ web site: http://www.zeromq.org.

[7] J. Berry, M. Oster, C. Phillips, and S. J. Plimpton. Maintaining connected components for infinite graph streams. Draft manuscript, 2012.

[8] J. W. Berry. Practical heuristics for inexact subgraph isomorphism. Draft manuscript, 2011.

[9] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In *SIAM Data Mining*, 2004.

[10] J. Cohen. Graph twiddling in a MapReduce world. *Computing in Science and Engineering*, 11:29–41, 2009.

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce online. In *NSDI'10 Proceedings of the 7th USENIX conference on Networked systems design and implementation*, page 21, 2010.

[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[14] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms: DIMACS Workshop on External Memory and Visualization*, pages 107–118. AMS, 1999.

[15] W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36:1–34, 2004.

[16] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy. A platform for scalable one-pass analytics using MapReduce. In *SIGMOD '11 Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 985–996, 2011.

[17] S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science Series. Now Publishers Inc., 2005.

[18] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings 2010 10th IEEE International Conference on Data Mining Workshops*, 2010.

[19] T. C. O'Connell. A survey of graph algorithms in extensions to the streaming model of computation. In S. S. Ravi and S. K. Shukla, editors, *Fundamental Problems in Computing: Essays in Honor of Professor Daniel J. Rosenkrantz*, pages 455–476. Springer, 2009.

[20] T. Plantenga. Inexact subgraph isomorphism in MapReduce. *J Parallel & Distributed Computing*, 2012. To appear.

[21] S. J. Plimpton and K. D. Devine. MapReduce in MPI for large-scale graph algorithms. *Parallel Computing*, 37:610–632, 2011. MapReduce-MPI web site: `http://mapreduce.sandia.gov`.

[22] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[23] B. Wylie and J. Baumes. A unified toolkit for information and scientific visualization. volume 7243, page 72430H. IS&T/SPIE Electronic Imaging 2009, Visual Data Analytics (VDA 2009), 01 2009. Titan Informatics Toolkit web site: http://titan.sandia.gov.