Pipelining Graph Construction and Agent-based Computation over Distributed Memory

Yan Hong and Munehiro Fukuda
Computing and Software Systems
University of Washington Bothell
Bothell, WA 98011
{yanh2020, mfukuda}@uw.edu

Abstract*—Graph streaming has received substantial attention for the past 10+ years to cope with large-scale graph computation. Two major approaches, one using conventional data-streaming tools and the other accessing graph databases, facilitate continuous analysis of endlessly flowing graphs and query-based incremental construction of huge graphs, respectively. However, some scientific graphs including biological networks need to stay in memory for repetitive but various analyses. Although a cluster system, thus distributed memory can entirely handle a big graph in memory, a challenge is substantial overhead incurred by loading graphs into memory. A solution is hiding such graph-loading and construction overheads with graph computation in a pipelined fashion. We adapted this pipelining approach for agent-based graph computing where thousands of agents traverse a graph for finding its attributes and shape. We used the multi-agent spatial simulation (MASS) library to implement the concept. A huge graph is incrementally constructed in batches, each spawning and walking agents over the corresponding subgraph, and thus all eventually completing a given computation. We coded and ran two MASS benchmark programs: triangle counting and connected components, with which we evaluated our pipelined graph processing. The best performance was obtained once the batch size shrunk enough to fit cache memory, regardless of the number of cluster nodes. For a single node execution of connected components over a 140MB graph, our graph-pipelining implementation performed 7.7 times faster than non-pipelining execution. Its parallel execution with 24 cluster nodes achieved 8.3 times speed-up as compared to the pipelined single-node execution.

***Index Terms*—graph streaming, multi-agent systems, parallel graph computing, distributed memory, pipelined parallelization**

I. Introduction

Data streaming with Spark, Storm, and Flink[1] is the most popular solutions to big-data computing when it comes to handling plain text-oriented datasets, (e.g., tweet messages), which are continuously produced, thus demanding spatial scalability. However, it cannot smoothly support all scientific-computing domains that prefer processing graphs such as biological networks and social networks.

We observe two approaches to facilitating spatial scalability of big graphs. One is borrowing the concept of conventional data streaming, which is called graph streaming [1]. The other is incremental graph construction with graph databases [2] and run-time computation with their graph queries [3]. The former benefits recommender systems [4] and fraud detection [5] as their entire graph views have no need to stay in memory or disk. This corresponds to the Lambda architecture's speed layer. On the other hand, some scientific graphs such as biological networks need to be incrementally updated in databases as more research findings are published and will be re-analyzed for their attributes and shapes [6]. This approach corresponds to batch and serving layers in the Lambda.

However, big graphs in scientific computing may not always fit to either of these two approaches. Consider biological networks as an example. They have been academically shared through public databases such as bioDBnet [7]. While these databases accept basic network queries on vertices and edges, (e.g., proteins and their reactions), topological and clustering analyses of a given network must be performed by downloading the entire dataset to different graph analyzers such as Cytoscape [8], most of which however run on a single machine, thus difficult to scale up the graph size. If biologists use graph streaming, (e.g, Spark micro-batch streaming) as a speed-layer solution, they may have to address the so-called watermark problem [9] for properly counting subgraphs laid over batch boundaries. Since they tend to conduct different analyses such as network motif searches with various degrees, the same dataset must go through the speed layer repetitively. On the other hand, if scientists use graph databases as a batch/serving-layer solution, their graph dataset must be converted into a given query language, mostly in openCypher [3]. Although graph databases facilitate incremental and scalable graph construction, their disk-based implementations slow down graph analysis.

Previously, we proposed agent-based graph computing where a large graph was incrementally constructed in distributed memory and was analyzed in parallel by propagating agents over the graph [10]. Our approach thus served as a solution to speed up graph computing in batch/serving layers. However, most graph datasets are described in unique formats such as HIPPIE [11] and MATSim XML [12], almost non-parallelizable but supposed to be read sequentially from top to down.

To address this drawback, we added a graph-streaming concept to our implementation: pipelining graph construction

[1]https://{spark,storm,flink}.apache.org

and agent-based computation[2] while maintaining a graph in distributed memory and later logging it into disks. Our contribution to the speed-up of graph computation is three-fold: (1) overlapping file read and graph construction with actual agent-based graph computation; (2) preventing a graph search from traversing too deep in each streaming batch, which eventually lowers the number of agents involved in computation; and (3) effectively using cache memory by adjusting the batch to the cache size, particularly for graph processing with 1-4 cluster-computing nodes.

The rest of the paper is organized as follows: section 2 compares our approach with related work; section 3 details our implementation techniques for pipelining graph construction and computation in distributed memory; section 4 evaluates our strategy with two graph applications, (i.e., triangle counting and connected components); and section 5 concludes the research outcomes and plans on our future tasks.

II. Related Work

This section looks at related work from the following three perspectives: (1) the background of our pipelined graph construction and agent-based computation, (2) the position of our approach in graph processing and streaming algorithms, and (3) the differentiation from parallel graph-processing systems.

A. Background of Agent-based Graph Computing

In contrast to streaming text data to conventional big-data tools, we feel that many scientific datasets, (e.g., NetCDF [13] in climatology and HIPPIE [11] in bioinformatics), would be better handled by maintaining their structure in distributed memory and injecting software agents into them for repetitive analyses. For this purpose, we applied our Multi-Agent Spatial Simulation (MASS) library [14] to graph computing [10]. However, our biggest challenge was reading a big graph data into distributed memory. For instance, the world largest 69GB biological network [15] needed 70 minutes just to be read and distributed over 24 computing nodes [16]. It is our motivation to mitigate graph construction overheads, using graph streaming - reading a next batch of graph data while processing the current batch. This strategy works easily to batch-local computation such as degree centrality search. On the other hand, stateful computation, (e.g., topological or clustering analysis) needs to consider a so-called watermarking technique that identifies every single subgraph lying over multiple batches. However, batch streaming with watermarking does not work perfectly to all graphs whose size is unknown and whose topology may include cycles. Therefore, rather than depend on conventional data streaming such as Spark and Kafka, we implement our own graph streaming feature with the following two strategies: (1) streaming a graph in small batches and incrementally constructing the entire graph over distributed memory and (2) processing the current batch of graph data in a pipelining fashion and revisiting the previous batches if necessary.

[2]We use "graph pipelining" with conjunction of computation, for the purpose of differentiation from continuous or micro-batch streaming.

B. Graph Processing and Streaming Algorithms

Besta et.al. distinguished streaming graph processing and graph streaming theory in [1]. Below we compare our graph pipelining approach with their categorization.

Streaming graph processing can consider the following four processing styles:

1) *Batch or stream analytics* incrementally runs graph analytics from scratch by streaming graph data to memory (in batches). IncRDD [17] and IndexRDD [18] facilitate this analytics by utilizing Spark streaming and by specializing Spark (immutable) RDD in variable memory storage with Cuckoo Hashing [19] and Persistent Adaptive Radix Tree [20] respectively. Our graph pipelining is implemented on top of the MASS library to overlap graph streaming and processing. It focuses on only incremental vertex/edge insertion during a pipelining stage but later allows any graph modification once an entire graph is mapped over distributed memory.
2) *Graph databases and NoSQL stores* maintain large graphs in disks or memory and update their shapes through consecutive database queries. Batch or stream analytics are made available by linking a data-streaming tool to query inputs of a given graph database, (e.g., Kafka to Neo4j [21] and Spark to RedisGraph [22]). In contrast, our approach maintains a graph in distributed memory, which can also save the current graph into disks, using the MASS checkpointing feature [23].
3) *Streaming processing of static graphs* divides a static graph into batches and streams each batch into memory for per-batch computation. An example is a combination of Spark Stream and GraphX [24]. Our graph pipelining is similar to this approach in reading a static graph. However, once an entire graph is constructed in memory, MASS can accept any modifications on it.
4) *Historical graph processing* stores all past graph data to be able to query the graph at any point in the past, which in turn enables reverse graph streaming. An example is Tegra [25] that stores an evolving graph in distributed snapshots over time. The MASS library allows users to take at most one copy of snapshot in the past but automatically memorizes a history of library calls, from which it reconstructs any past graph between the last snapshot and the current computation [23].

On the other hand, *graph streaming theory* can consider the following three algorithms:

1) *Streaming graph algorithms* slide a window from top to bottom of a given graph-update scenario [26]. They carry out edge insertions and deletions only within the current window at once, from which the algorithms optimize the memory usage and the response time as well as pursue the accuracy of various graph queries/computations. Our approach applies a similar sliding window model to batched pipelining but focuses on only vertex/edge insertion and single graph computation until the entire graph is mapped over distributed memory.

2) *Dynamic graph streams* still bound the number of graph updates per window but redo graph streaming over multiple passes for the purpose of improving the computational accuracy [27]. Our graph pipelining can look beyond the current batch and thus cover a graph under construction so far. This is a similar effect of sliding a window back and forth.
3) *Parallel dynamic graph algorithms* accelerate modification of a given graph by parallelizing updates per window, which results in cost reduction per update [28]. Our approach reads a graph dataset sequentially but overlaps the graph construction and computation in a pipelining fashion. Furthermore, graph computation is carried out with many agents in parallel.

C. Graph Steaming Systems

Below we narrow down the above-mentioned *streaming graph processing* into three well-known systems, for each of which we identify the mismatches between their features and our needs.

1) *Spark and GraphX* [24] is the best fit to streaming a static graph in its micro-batches if it needs to cover only batch-local computation. However, incremental graph construction creates a new RDD repetitively, which deteriorates the execution performance and wastes memory space [10], [18]. Furthermore, stateful computation requires to address watermarking over multiple batches.
2) *Spark and RedisGraph* [18] streams an input graph into RedisGraph, an in-memory graph database. The graph can be dynamically modified with queries in Cyphers. The main drawback is scalability as it runs on a single machine, thus not supporting distributed memory. Furthermore, Abughofa et.al. in [18] reports that Redis performs much slower in graph construction than in retrieval.
3) *Kafka and Neo4j*'s streaming strategy [21] is similar to but different from Spark and RedisGraph in running Neo4j over a cluster system and thus using distributed disks. This configuration supports spatial scalability. However, the nature of disk accesses makes this Kafka-Neo4j combination slower than all the other streaming systems.

In summary, we believe that our graph-pipelining approach, implemented on top of the MASS library, can complement to what the other streaming systems are missing: speed-up of incremental construction and analysis of a huge graph.

III. Pipelined Graph Processing in MASS

Below we show our implementation of pipelined graph construction and agent-based computation in MASS, followed by a user-level code framework.

A. Design Overview

The MASS library is the platform of our agent-based graph computing. It distinguishes two classes, *Places* and *Agents*. The former constructs a multi-dimensional array of elements, each called *a place*, over a cluster system. The latter populates mobile objects, each named *an agent* that autonomously navigates from *place* to *place*, thus over the cluster. Their *callAll*, *exchangeAll*, and *manageAll* functions invoke parallel method calls, message exchanges, and agent status changes, (e.g., cloning, termination, and migration).

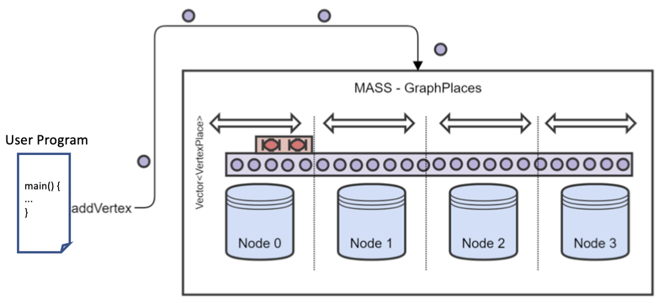


Fig. 1. Incremental graph construction in MASS

Derived from *Places*, *GraphPlaces* supports agent-based graph computing where each vertex and its emanating edges are managed with a *place* and its internal *neighbors* data member [10]. During an instantiation, *GraphPlaces* reads an input file and constructs the corresponding graph over a cluster system. Thereafter, it can dynamically add a new vertex and its edges with *addVertex() / addEdge()* or delete an existing vertex and its edges with *deleteVertex() / deleteEdge()*.

The original *GraphPlaces* implementation was a collection of *Places*, (i.e., a list of arrays), each incrementally instantiated in batches if more vertices are requested at run time. However, repetitive operations of vertex insertion and deletion will unbalance dynamic graph mapping over a cluster system. Therefore, as shown in Figure 1, the current implementation uses a collection of vector queues, each pair of which are allocated to a different computing node: one for maintaining active vertices (named *VertexPlace*) and the other for recycling zombie vertices [16].

Figure 2 illustrates a workflow of reading each batch from an input file into MASS. A user can determine the batch size as the number of vertices to be read at once. Once MASS completes the construction of $batch_i$ from a given file, it starts graph computing for i in parallel, while advancing to $batch_{i+1}$'s construction. To cover large and/or cyclic subgraphs laid over multiple batches, computation in $batch_i$ may access $batch_0$ through to $batch_{i-1}$, all maintained in distributed memory. In agent-based graph computing with MASS, $batch_i$ instantiates agents within its range. Many of them may complete their traverse or computation in the current batch, whereas some may need to migrate to $batch_{i+1}$. In that case they are suspended until $batch_{i+1}$ is made available.

B. File Preprocessing

File formats for graph descriptions depend on scientific-computing domains. We pick up HIPPIE and MATSim XML, each intended to describe biological networks and traffic networks respectively.

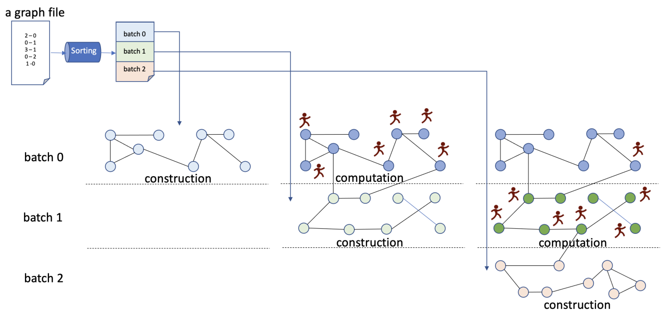


Fig. 2. Workflow of graph pipelining in MASS

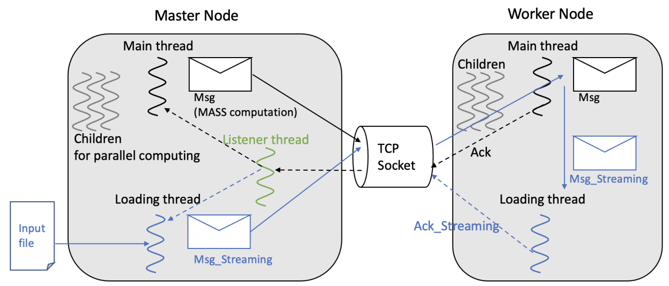
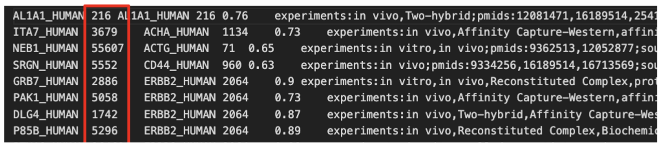


Fig. 5. Graph construction with an additional thread in MASS



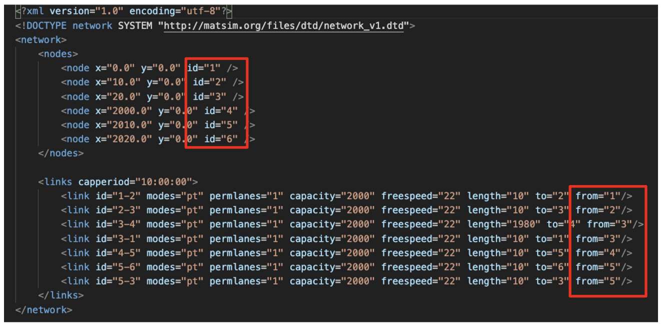
```
AL1A1_HUMAN 216 AL1A1_HUMAN 216 0.76    experiments:in vivo,Two-hybrid;pmids:12081471,16189514,2541
ITA7_HUMAN  3679    ACHA_HUMAN  1134    0.73    experiments:in vivo,Affinity Capture-Western,affini
NEB1_HUMAN  55607   ACTG_HUMAN  71  0.65    experiments:in vitro,in vivo;pmids:9362513,12052877;sou
SRGN_HUMAN  5552    CD44_HUMAN  960 0.63    experiments:in vivo;pmids:9334256,16189514,16713569;sou
GRB7_HUMAN  2886    ERBB2_HUMAN 2064    0.9 experiments:in vitro,in vivo,Reconstituted Complex,prot
PAK1_HUMAN  5058    ERBB2_HUMAN 2064    0.73    experiments:in vivo,Affinity Capture-Western,affini
DLG4_HUMAN  1742    ERBB2_HUMAN 2064    0.87    experiments:in vivo,Two-hybrid,Affinity Capture-Wes
P85B_HUMAN  5296    ERBB2_HUMAN 2064    0.89    experiments:in vivo,Reconstituted Complex,Biochemic
```

Fig. 3. A snippet of a HIPPIE file

In the HIPPIE format, each line of a file represents a protein-to-protein reaction, thus an edge from one vertex to another. These edges are enumerated randomly (as shown in Figure 3). If a file is simply partitioned into batches from top to down, an edge insertion in $batch_k$ may have to refer back to $batch_i$ and $batch_j$ where $i < k$ and $j < k$. This in turn means that $batch_i$, while it is being computed, may re-invoke graph computation over its former batches.

In the MATSim XML, a file includes two sections: the first section declares all road intersections as vertices, whereas the second section enumerates all road segments from one intersection to another, thus listing the corresponding edges. Figure 4 partially captures sorted vertices and edges. However, the MATSim format does not always guarantee such data sorting and thus has the same problem as HIPPIE when being streamed in batches.

To avoid repetitive computation over batches that have been previously read into memory, we need to sort graph edges in the order of their source vertex IDs or to cluster all vertices, in advance of graph pipelining. As file preprocessing tools, we use the GNU parallel and GNU sort tools [29]. The former executes jobs in parallel. On the other hand, the latter sorts a file by line numerically or alphabetically, using the merge sort algorithm. The command to achieve this file sorting is *parsort*. Since such file preprocessing incurs additional overheads to graph pipelining, we will estimate its performance impact and propose potential solutions later in Section IV.



```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE network SYSTEM "http://matsim.org/files/dtd/network_v1.dtd">
<network>
    <nodes>
        <node x="0.0" y="0.0" id="1" />
        <node x="10.0" y="0.0" id="2" />
        <node x="20.0" y="0.0" id="3" />
        <node x="2000.0" y="0.0" id="4" />
        <node x="2010.0" y="0.0" id="5" />
        <node x="2020.0" y="0.0" id="6" />
    </nodes>

    <links capperiod="10:00:00">
        <link id="1-2" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="2" from="1"/>
        <link id="2-3" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="3" from="2"/>
        <link id="3-4" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="1980" to="4" from="3"/>
        <link id="3-1" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="1" from="3"/>
        <link id="4-5" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="5" from="4"/>
        <link id="5-6" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="6" from="5"/>
        <link id="5-3" modes="pt" permlanes="1" capacity="2000" freespeed="22" length="10" to="3" from="5"/>
    </links>
</network>
```

Fig. 4. A snippet of a MATSim file

C. Pipelined Graph Construction

The MASS library's programming framework is based on the master-worker model in that the master computing node runs a user's *main()* function and initiates *Places / Agents* parallel computation, using *callAll()*, *exchangeAll()*, and *manageAll()*, whereas all worker nodes remotely respond to these parallel function calls, each acknowledged back to the master node with barrier synchronization. This is the same as Spark's execution model that runs a user program on the master and invokes its RDD transformation or action remotely.

To overlap graph construction with the above-mentioned computation, we instantiate an additional thread called a *loading* thread. As sketched in Figure 5, the loading thread on the master node reads a given graph file in batches. It repeats either creating new vertices and edges locally or sending their information with a *msg_streaming* tag to remote nodes. To reduce the number of messages sent to remote nodes, the loading thread locally caches vertex/edge information per remote node and later sends it when the cache becomes full.

On each remote node, its main thread needs to distinguish those tagged with *msg_streaming* from the other computation-initiating messages. Upon receiving a *msg_streaming* message, the main thread spawns a *loading* thread that takes charge of graph construction. The main and loading threads share the same TCP socket to send back their acknowledgments, each marked with *ack* and *ack_streaming* respectively, to the master node.

Back on the master node, we implement another thread called a *listener* thread that relays *ack_streaming* messages to the loading thread. Collecting *ack* or *ack_streaming* from all remote nodes, the main and loading thread advance to a next batch of graph computation and construction process.

For each batch of graph construction, the loading thread on the master node reads from a graph file as many lines as the number of their source vertices reaches a user-specified size. By exchanging *msg_streaming* and *ack_streaming*, the master and worker computing nodes instantiate new vertices and all edges emanating from each vertex. These edges are connected to their destination vertex if they are located within the range up to the current $batch_i$. Otherwise, the destination is temporarily created as an "incomplete" vertex. The loading thread also provides a user with the smallest and largest vertex IDs of the current $batch_i$ as its lower and upper boundary.

Figure 6 exemplifies a graph file that has been sorted in the order of the source vertices. Given a batch size with four, the entire file will be streamed in three batches, each including vertices 0-3, 4-7, and 8-9 respectively. During the graph construction, vertex 8 is marked "incomplete" in $batch_0$ and still maintained as is in $batch_1$; vertex 9 is marked "incomplete" in $batch_1$, too; but both vertices eventually become "complete" in $batch_2$.

These attributes are used to control agents. The current batch allows agents to be spawned only in its range between lower and upper boundaries. Each agent may complete its travel and computation in the same batch, or may suspend itself on an incomplete vertex. For instance, $batch_1$ will suspend two agents, one on vertex 8 and the other on vertex 9.

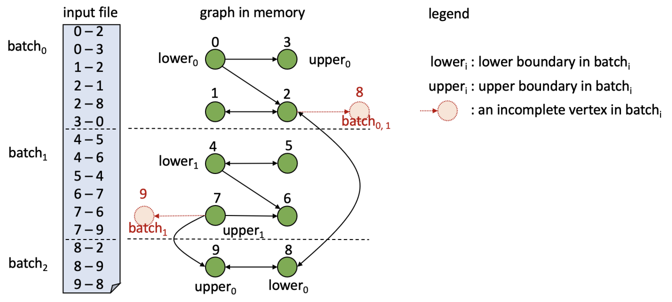


Fig. 6. An example of graph under construction

D. Pipelined Graph Computation

We customized the MASS-original *Agent* class into *GraphStreamingAgent* that supports the new agent behaviors in pipelined graph computation: agent creation in a current batch, agent computation below its upper boundary, and agent suspension on incomplete vertices. For this purpose, this class not only maintains the current lower and upper boundary information but also implements the following two methods:

- *currentPlaceBeyondBoundary()*: informs the calling agent if it is on an incomplete vertex.
- *suspendComputation()*: suspends the calling agent's execution, deposits its state on the current place, (i.e., the current vertex), and kills the agent. It is called upon *if (currentPlaceBeyondBoundary()==true)*.

Additionally, the *Agents* class, (i.e., a collection of *GraphStreamingAgents*) revises its constructor and implements the *resumeAgent()* method as follows:

- *Agents(... long lowerBoundary, long upperBoundary)* populates new agents in the range between lowerBoundary and upperBoundary as well as calls *resumeAgent()*.
- *resumeAgent()*: salvages agent states deposited on each of all incomplete vertices and resumes as many agents as necessary from these states.

At the beginning of each batched computing, the *Agents* class populates a new agent on each vertex within the range and additionally checks if the vertex marked with incomplete. If so, *resumeAgent()* counts the number of agent states deposited on this vertex, instantiates the same number of agents, reinitializes them with the the states salvaged, and resumes their computation. As illustrated in Figure 7, if there are four different agent states on a given vertex, four agents will be created with each of the deposited states in addition to a brand-new agent.

The original specification of *Places.callAll()* intends to invoke a given function on all the vertices of a graph. To align with the design of pipelined graph computing that processes an available portion of the graph, a new *callAll()* method receives the lower boundary and upper boundary of a current batch, which calls only the vertices in the current batch.

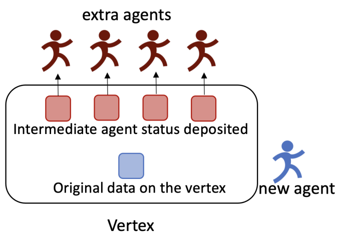

extra agents
Intermediate agent status deposited
Original data on the vertex
new agent
Vertex

Fig. 7. Agents creation and suspension on an incomplete vertex

E. User-Level Code Framework

The latest MASS implementation can handle HIPPIE and MATSim XML formats to read a data file into memory. For this purpose, *HippieStreaming* and *MATsimStreaming* classes are extended from *GraphPlaces*. They have the following methods:

- *HippieStreaming()/MATSimStreaming()*: opens a file.
- *hasNext()*: checks if there are more batches to read.
- *next()*: reads a next batch.
- *get_lowerBoundary()*: returns the current lower boundary.
- *get_upperBoundary()*: returns the current upper boundary.
- *syncOneCycle()*: advances to a next batch.
- *close()*: closes the graph file.

Using these methods, a graph application will be programmed in the MASS library's code framework as shown in Listing 1. After starting a MASS daemon on each computing node (line 1), this program instantiates an HippieStreaming object that opens a given file (line 2). Thereafter, it falls into a *while* loop where each iteration checks if more batches are available to read (line 3), and reads the next batch (line 4). Without waiting for this graph reading, the user program can passes the range of the current subgraph to the *Agents* constructor (lines 5-7). The *agents* object refers to not only new agents but also those resumed from the suspension. The user program then codes agent logic to analyze the subgraph (lines 9-14). Listing 1 counts the number of triangles in the graph by walking agents along an edge three times and by adding up those who returned back to their source vertex. At the end of each batch computing, the user program needs to confirm the completion of the current batch streaming (line 16). The last two lines in the program close the file and terminate all the MASS daemons.

Listing 1. Pipelined graph computation

```
 1 MASS.init( ); // start MASS daemons
 2 HippieStreaming hstreaming = new HippieStreaming( ... );
 3 while ( hstreaming.hasNext( ) ) { // read a next batch
 4   Hippie hippie = hstreaming.next( );
 5   lowerBoundary = hstreaming.get_lowerBoundary( );
 6   upperBoundary = hstreaming.get_upperBoundary( );
 7   Agents agents = new Agents( lowerBoundary, upperBoundary );
 8   // a user logic begins: triangle counting
 9     for ( int i = 0; i < 3; i++ ) { // traverse on 3 edges
10       agents.callAll( onArrival, i );
11       agents.callAll( departure, i );
12       agents.manageAll( );
13     }
14     numTriangles += agents.nAgent( ); // sum up #active agents
15   // end of the logic
16   hstreaming.syncOneCycle( ); // advance to a next batch
17 }
18 hstreaming.close( );
19 MASS.finish( );
```

IV. Performance Evaluation

To verify the correctness of and to evaluate the execution performance of our pipelined graph construction and computation, we used two graph applications: triangle counting and connected components. Both were previously coded by former MASS developers [30], [31].

We used a cluster of 24 computing nodes, all connected to 1Gbps LAN and available at University of Washington Bothell. Their machine specifications are summarized in Table I.

Two input graphs, each with 290MB and 140MB, were generated randomly with our own software tool (called GraphGen.java) and were saved in HIPPIE files. We determined these sizes in order to complete both benchmark programs in 20 minutes when running them on a single computing node. They includes 40K and 25K vertices, respectively.

TABLE I
Cluster-computing environments for performance evaluation

#machines	#cores	model	memory	cache
3	4	Xeon 5150 @ 2.66GHz	16GB	4MB
4	8	Xeon E5410 @ 2.33GHz	16GB	12MB
5	4	Xeon Gold 5220R @ 2.20GHz	16GB	35.75MB
12	4	Xeon Gold 6130 @ 2.10GHz	16GB	22MB

A. Triangle Counting

Triangle counting is a well-known social network benchmark that counts the number of triangle relationships among network users as an indicator of their friendship degrees. As shown in Listing 1, MASS walks an agent along a series of three edges connected from one to another. Starting from each vertex, for the first two steps, an agent propagates itself to all the neighboring vertices with a smaller ID than the current vertex. It gets terminated if there are no such neighbors. For step 3, all remaining agents attempt to return along an edge to their source vertex. A successful return results in discovering a triangle.

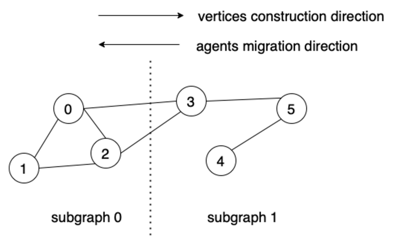


Fig. 8. Agent-based triangle counting in graph pipelining

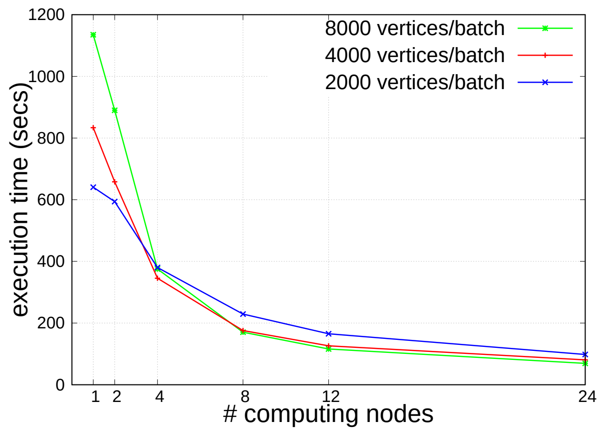


Fig. 9. Execution performance of triangle counting in graph pipelining

As illustrated in Figure 8, we found that agent-based triangle counting took advantage of our graph pipelining. Graph construction grew to vertices with larger IDs, whereas agent migration flowed down to those with smaller IDs, thus all in distributed memory. Our performance evaluation took a 290MB HIPPIE file with 40K vertices where the average degree of each vertex was 50. Our measurements distinguished 2K, 4K, and 8K vertices as its batch size. For each batch size, the evaluation averaged three measurements of time elapsed to complete triangle counting.

Figure 9 demonstrates effective graph-pipelining performance of MASS-based triangle counting with up to 24 computing nodes. The smaller size a batch has, the faster computation it achieves with single or two-way parallelization, whereas the batch size does not matter with four or more ways. This is because the graph with 40K vertices fit into the entire distributed memory supported by 4+ cluster nodes.

Obviously, Figure 9 itself cannot say if this performance improvement is brought by graph pipelining or by restricted

TABLE II
EXECUTION PERFORMANCE OF TRIANGLE COUNTING WITHOUT GRAPH PIPELINING

# Computing nodes	Execution time (sec)
1	Out of memory
2	Out of memory
4	890.422

graph growth. Therefore, performance measurements of non-pipelining computation (by setting the batch size in 40K vertices) are needed. Table II summarizes the non-pipelining execution with 1, 2, and 4 cluster nodes. The memory overflow with one or two cluster nodes results from the rapid growth of agent population. This is because our graph pipelining can ultimately construct the entire graph in memory with any of 2K-8K batch sizes. On the other hand, four cluster nodes allow non-pipelining computation to complete in success. However its performance is 2.3 times slower than the pipelining computation with the batch size of 8K, (i.e., 890.422 versus 380.083 seconds). We believe that this is a positive evidence of graph-pipelining effect. Below, we scrutinize it through our measurement of "connected components".

B. Connected Components

This graph algorithm is used frequently to identify groups in a biological network and a social network. Our agent-based solution is similar to BFS that picks up each unexplored vertex and searches for its neighbors and all the descendants. However, to prevent any divergence of agent population, we allow agents to keep traveling to only vertices whose IDs are higher than where the agents got started. Therefore, each group of connected vertices is eventually ruled by only one agent that started from the vertex with the lowest ID.

Figure 10 shows an example of three connected components, each at last colored by agents 0, 3, and 6. The figure also describes how the graph construction and computation is pipelined:

- *batch-0 computation*: populates 5 agents, each at vertices 0-4 upon completing the subgraph construction. They move toward vertices with a higher ID. Agent 3 rules a group of vertices 3-4, whereas agents 0 and 1 are suspended at vertices 5 and 9 respectively as they are incomplete.
- *batch-1 computation*: populates another set of 5 agents, each at vertices 5-9, and resumes agents 0 and 1. The computation identifies agents 0 and 6 as winners: the former beats out agent 1 and the latter takes over vertices 6-8.
- *counting components*: distinguishes three groups, each colored with agents 0, 3, and 6 when all agents are gone. We will collect these colors with *HippieStreaming.callAll()*.

Figure 11 shows MASS parallelization of connected components in a 140MB HIPPIE graph file that includes 25K vertices. MASS demonstrates its CPU scalability with up to

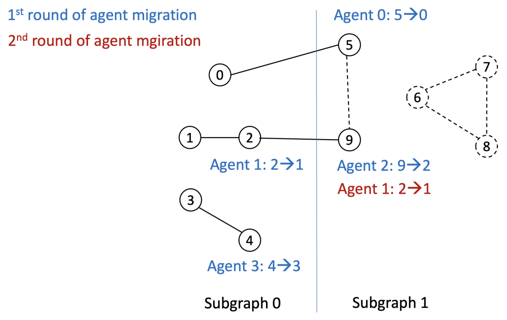


Fig. 10. Agent-based connected components in graph pipelining

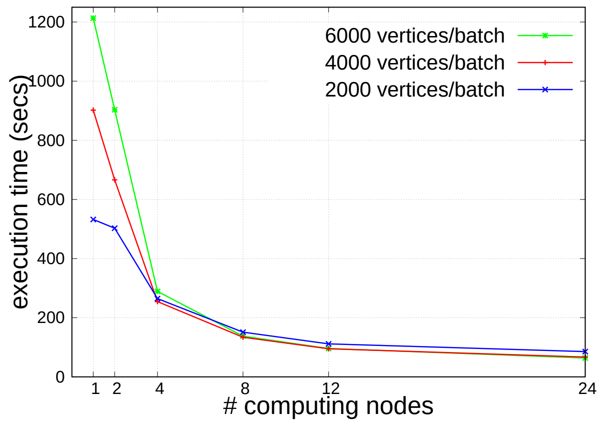


Fig. 11. Execution performance of connected components in graph pipelining

24 computing nodes. Its trend in batch sizes and execution time is the same as triangle counting: the smaller batch, the better performance.

For the similar purpose as triangle counting, (i.e., to check pipelining effects), we compared non-pipelining and pipelining executions of connected components (see Table III). Non-pipelining execution over 25K vertices fitted into a single computing node's memory. However, It performed 3.37 times slower than the pipelining computation with the batch size of 6K vertices. (i.e., 4085.260 versus 1213.205 seconds). We also observed CPU and memory usages for both cases to identify any suspicious overhead incurred by disk thrashing. The resource usage summarized in Table IV did not find any substantial difference between non-pipelining and pipelining executions.

From our preliminary measurements on *GraphPlaces* [16], 50MB HIPPIE file reading and graph construction (thus without performing any computation) takes only 2.5 seconds, which in turn means that 140MB HIPPIE file would need only up to 10 seconds. Therefore, it is inferred that our graph pipelining on connected components mainly contributes to the reduction of graph-computation time even on a single computing node. We attribute this performance improvement

TABLE III
EXECUTION PERFORMANCE OF CONNECTED COMPONENTS WITHOUT AND WITH GRAPH PIPELINING

# Computing nodes	Batch size	Execution time (sec)
1	Non-pipelining	4085.260
1	6K vertices	1213.205
1	4K vertices	902.164
1	2K vertices	532.133

TABLE IV
RESOURCE USAGE WHEN COMPUTING CONNECTED COMPONENTS

Pipelining	VM	Physical Mem	%Memory	%CPU
No	8.9GB	4.7G	30.6%	372.4%
Yes	8.8GB	4.7G	30.4%	331.9%

to cache memory usage. Consider the batch size of 2-6K vertices respectively. This corresponds to 11.2MB-33.6MB of the 290MB HIPPIE data and can be halfway or even fully covered by the cache memory of most cluster nodes we used (refer to Table I). The subgraph constructed in $batch_i$ remains in cache and is ready for computation by agents in $batch_{i+1}$.

For 4-way or more parallelization, the same effect by cache memory usage keeps maintained regardless of batch sizes. This is because all batches in 11.2MB-33.6MB, divided by 4 through to 24 cluster nodes, completely fit into the cache memory.

C. Other Performance Considerations

Our implementation of graph pipelining assumes that a graph dataset is sorted in the order of vertices. Our performance evaluation does not consider sorting overheads of 290MB and 250MB HIPPIE files. This is because both are completed in less than 20 seconds and therefore negligible as compared to the entire pipelining execution. However, the larger file we have, the more sorting overheads are incurred in the order of sorting complexity: $\mathcal{O}(n \log n)$.

As mentioned in Section III-B, our graph pipelining is based on *parsort* for sorting input files in advance. Its execution performance on larger files is summarized in Table V. From the 40GB-file sorting performance, we can infer that sorting the world largest 69GB biological network file would need 327 minutes in total by substituting $87min \div 40GB \times 69GB$ for n in $\mathcal{O}(n \log n)$. This is 4.7 times slower than the 69GB (non-pipelined) graph construction in 70 minutes.

Obviously, sorting overheads can be removed by streaming an unsorted graph file directly to our pipelining mechanism, which however extends the range of edge insertions from the current $batch_i$ all the way back to $batch_0$ rather than limits such insertions within $batch_i$. To address this extended construction, our agent-based computation must instantiate new agents not only in $batch_i$ but also on vertices in former batches, all reachable through the newly inserted edges. This re-invokes graph computation over these former batches. The higher degree (= the more edges) each vertex has, the more former batches agents need to recompute. As more and more batches cannot fit into cache memory, its memory usage is increasingly deteriorated toward the end of computation.

TABLE V
FILE SORTING PERFORMANCE OF THE PARSORT COMMAND

File size	#lines	Execution time (minutes)
2.5GB	120 million	1 min
26GB	1400 million	34 mins
40GB	2100 million	87 mins

Needless to say, the degree distribution depends on application domains. Considering biological networks in HIPPIE and traffic networks in MATSim, their degree range would be 3-7. In general, biological networks are scale-free. According to [32], scale-free networks' average degree is 7 and 75% of their nodes has a degree of 3 or less, while a few nodes have a degree above 500. Most traffic network files including MATSim describe an intersection as a vertex. We assume that most vertices are 4-way intersections, (i.e., a degree of 4). On the other hand, social networks have a much higher degree. For instance, Ugander et.al. counted 99 as the median friendships for Facebook global users [33]. Therefore, for HIPPIE and MATSim graphs, we expect that our graph pipelining implementation, even to be revised for streaming unsorted graphs, will be able to run with acceptable graph re-computation overheads.

Finally, we would like to mention about our former comparison work between MASS and Spark in triangle counting [30] and connected components [31], both evaluated with non-pipelining execution. MASS performed triangle counting 6.6-11.2 times faster than Spark with 1-8 cluster nodes, whereas its computation of connected components with 2K vertices was 3.8 times slower than Spark even with 7 nodes (49.298 versus 12.923 seconds). For the latter case, assuming that Spark's computing time grows in proportional to the number of vertices, its time for 25K vertices would go up to 160 seconds as compared to 137.591 seconds with the latest MASS version, which makes our graph-pipelining competitive to Spark.

V. CONCLUSION

This research implemented graph pipelining in the MASS library for the purpose of better supporting agent-based graph computing with batched graph construction. Our achievements are five-fold: (1) supporting biological networks in HIPPIE and traffic networks in MATSim XML as input graphs; (2) running and synchronizing a user program and a graph-constructing thread in a stop-n-go fashion; (3) controlling agent instantiations, suspensions, and terminations as shifting computation from one batch to another; (4) demonstrating CPU and spatial scalability of our graph pipelining implementation through triangle counting and connected components; and (5) tracing the best performing condition that happens when the size of a batch or a subgraph distributed to a different cluster node is small enough to fit into cache memory.

Our future work includes the following three items: (1) supporting more different graph formats including the world largest 69GB biological network file; (2) measuring MASS graph execution with this 69GB file; (3) interfacing MASS *GraphPlaces* to major graph databases through *openCypher*. Finally, for more detail and trial uses, please visit our website at `http://depts.washington.edu/dslab/MASS`.

References

[1] M. Besta, M. Fischer, V. Kalavri, M. Kapralov, and T. Hoefler, "Practice of Streaming and Dynamic Graphs: Concepts, Models, and Systems," 2019.

[2] neo4j, "Accessed on: August 10, 2022. [Online]. Available: https://neo4j.com."

[3] openCypher, "Accessed on: August 10, 2022. [Online]. Available: https://opencypher.org."

[4] A. Sharma, J. Jiang, P. Bommannavar, B. Larson, and J. Lin, "Graphjet: real-time content redommendations at twitter," *Proceedings of the VLDB Endowment*, no. 13, pp. 1281–1292, September 2016.

[5] X. Qiu, W. Cen, Z. Qian, Y. Peng, Y. Zhang, X. Lin, and J. Zhou, "Real-time constrained cycle detection in large dynamic graphs," *Proceedings of the VLDB Endowment*, no. 12, pp. 1876–1888, August 2018.

[6] D. Miljkovic, V. Podpečan, T. Stare, I. Mozetič, K. Gruden, and N. Lavrač, "Incremental Construction of Biological Networks by Relation Extraction from Literature," *Current Bioinformatics*, no. 2, pp. 177–190, April 2015.

[7] biological DataBase network, "Accessed on: August 10, 2022. [Online]. Available: "https://biodbnet-abcc.ncifcrf.gov/."

[8] Cytoscape: An Open Source Platform for Complex Network Analysis and Visualization, "Accessed on: August 10, 2022. [online]. available: http://cytoscape.org."

[9] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, pp. 154 300–154 316, October 2019.

[10] J. Gilroy, S. Paronyan, J. Acoltzi, and M. Fukuda, "Agent-Navigable Dynamic Graph Construction and Visualization over Distributed Memory," in *Proc. of Seventh Int'l Workshop on Big Graphs at IEEE BigData'20*, December 2020, pp. 2957–2966.

[11] HIPPIE, "Accessed on: August 10, 2022. [online]. available: http://cbdm-01.zdv.uni-mainz.de/∼mschaefer/hippie/index.php."

[12] MATSIM Multi-Agent Transport Simulation, "Accessed on: August 10, 2022. [Online]. Available: https://www.matsim.org/."

[13] Unidata — NetCDF, "Accessed on: August 10, 2022. [online]. available: https://www.unidata.ucar.edu/software/netcdf/."

[14] MASS: A Parallel Library for Multi-Agent Spatial Simulation, "Accessed on: August 10, 2022. [online]. available: http://depts.washington.edu/dslab/mass/."

[15] J. Leskovec, "Stanford Large Network Dataset Collection," Accessed on: August 10, 2020. [Online]. Available: https://snap.stanford.edu/biodata/datasets/10028/10028-PP-Miner.html.

[16] B. Luger, "Distributed Data Structures," University of Washington Bothell, MS Indepeden Research Term Report. Accessed on: August 10, 2022. [Online]. Available: https://depts.washington.edu/dslab/MASS/reports/BrianLuger_su21.pdf, 2021.

[17] P. D. Prakash, "IncRDD: Incremental Updates for RDD in Apache Spark," Master's thesis, The University of Texas at Dallas, May 2017.

[18] T. Abughofa and F. Zulkernine, "Towards online graph processing with spark streaming," in *Proc. of IEEE Internaitonal Conference on Big Data*, Boston, MA, December 2017, pp. 2787–2894.

[19] R. Pagh and F. F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, p. 122–144, May 2004.

[20] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *Proc. of - International Conference on Data Engineering*, Brisbane, Australia, April 2013, p. 38–49.

[21] D. Allen, "Streaming Graphs: Combining Kafka and Neo4j," Neo4j Blog, Access on: August 15, 2022 [Online]. Available: https://neo4j.com/blog/streaming-graphs-combining-kafka-neo4j/, october 29, 2019.

[22] RedisGraph, "Accessed on: August 10, 2022. [Online]. Available: https://redis.io/docs/stack/graph/."

[23] D. Blashaw and M. Fukuda, "An interactive environment to support agent-based graph programming," in *Proc. of the 14th International Conference on Agents and Artificial Intelligence - Volume 1*, February 2022, pp. 148–155.

[24] Spark GraphX, "Accessed on: August 10, 2022. [Online]. Available: https://spark.apache.org/graphx/."

[25] A. P. Iyer, Q. Pu, K. Patel, J. E. Gonzalez, and I. Stoica, "Tegra: Efficient ad-hoc analytics on evolving graphs," in *Proc. of the 18th USENIX Symposium on Networked Systems Design and Implementation*, April 2021, p. 337–355.

[26] A. N. Zakrzewska, "Graph analysis of streaming relational data," Ph.D. dissertation, Georgia Institute of Technology, May 2018.

[27] S. Assadi, G. Kol, R. R. Saxena, and H. Yu, "Multi-pass graph streaming lower bounds for cycle counting, max-cut, matching size, and other problems," in *Proc. of 2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*, Durham, NC, November 2020, pp. 354–364.

[28] U. A. Acar, D. Anderson, G. E. Blelloch, and L. Dhulipala, "Parallel batch-dynamic graph connectivity," in *Proc. of ACM Symposium on Parallelsim in Algorithms and Architectures*, Phoenix, AZ, June 2019, p. 381–392.

[29] O. Tange, "GNU Parallel - The Command-Line Power Tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, February 2011.

[30] M. Fukuda, C. Gordon, U. Mert, and M. Sell, "Agent-Based Computational Framework for Distributed Analysis," *IEEE Computer*, vol. 53, no. 3, pp. 16–25, 2020.

[31] C. Liu, "Development of Application Programs Oriented to Agent-Based Data Analysis," University of Washington Bothell, Tech. Rep., March 2020.

[32] Math Insight, "Scale-free networks, Access on: August 15, 2022 [Online]. Available: https://mathinsight.org/scale_free_network."

[33] J. Ugander, B. Karrer, L. Backstrom, and C. Marlow, "The anatomy of the facebook social graph," 2011.