Large-scale Neural Modeling in MapReduce and Giraph

Shuo Yang
Graduate Programs in Software
University of St. Thomas
yang9136@stthomas.edu

Nicholas D. Spielman
Neuroscience Program
University of St. Thomas
spie6388@stthomas.edu

Jadin C. Jackson
Department of Biology
University of St. Thomas
jadincjackson@stthomas.edu

Brad S. Rubin
Graduate Programs in Software
University of St. Thomas
bsrubin@stthomas.edu

***Abstract*—One of the most crucial challenges in scientific computing is scalability. Hadoop, an open-source implementation of the MapReduce parallel programming model developed by Google, has emerged as a powerful platform for performing large-scale scientific computing at very low costs. In this paper, we explore the use of Hadoop to model large-scale neural networks. A neural network is most naturally modeled by a graph structure with iterative processing. In this paper, we first present an improved graph algorithm design pattern in MapReduce called Mapper-side Schimmy. Experiments show that the application of our design pattern, combined with the current best practices, can reduce the running time of the neural network simulation on a neural network with 100,000 neurons and 2.3 billion edges by 64%. MapReduce, however, is inherently not efficient for iterative graph processing. To address the limitation of the MapReduce model, we then explore the use of Giraph, an open source large-scale graph processing framework that sits on top of Hadoop to implement graph algorithms with a vertex-centric approach. We show that our Giraph implementation boosted performance by 91% compared to a basic MapReduce implementation and by 60% compared to our improved Mapper-side Schimmy algorithm.**

I. Introduction

There is a growing need for computational scientists to perform their scientific computing at a large-scale which is well beyond the capabilities of an individual machine or workstation. Traditionally, supercomputers are used to achieve this, but access to supercomputers is limited and expensive. Fortunately, with the emergence of Apache Hadoop[1], an open source implementation of the MapReduce parallel programming model [1], these needs may be met using commodity hardware at very low cost. MapReduce is a processing architecture for large-scale data processing developed by Google. In this paper, we investigate using Hadoop to model a large-scale neural network. Unlike well-known problems in areas like social networking and web connectivities, the use of Hadoop in scientific computing for fields like computational neuroscience has not gained much traction. We hope this paper encourages further exploration into using Hadoop to tackle large-scale computing problems in these fields.

The graph model is an intuitive way of modeling many real-world problems, such as the PageRank problem [5]. We can use the MapReduce model to process large-scale graphs, such as neural networks. In [2], Lin and Schatz proposed several enhanced design patterns for MapReduce graph algorithms. These represent the current best practices for large-scale graph processing in MapReduce.

Based on these current best practices, we propose an enhanced design pattern that can be used in a large class of graph algorithms based on message passing. While these improvements proved to be effective, they could not address some of the fundamental limitations of the MapReduce model for graph processing. After briefly discussing these limitations, we turn to another approach: a vertex-centric abstraction based on the bulk-synchronous parallel (BSP) model [3], which fits well for iterative graph processing. We use an open source graph processing framework Giraph[2] to implement the vertex-centric graph algorithm. Finally, we summarize our work and give future directions.

II. Neuron Model

Neurons are information-processing cells that connect through synapses with other neurons to form networks whose coordinated activity is responsible for computational processes such as cognition, sensory processing, decision-making, action selection, and reflexes in humans and other animals. A single neuron's functional state can be largely characterized by the electrical potential of its membrane, which changes in response to synaptic inputs from other neurons. These synaptic connections from other neurons can be either excitatory, providing positive current, or inhibitory, providing negative current. The strength of the current depends on the strength, or weight, of the synapse that was activated, and the direction of the current, positive or negative, will raise or lower, respectively, the membrane potential. When the membrane potential reaches a critical, threshold value, an all-or-nothing electrical event called an action potential or "spike" is produced by the neuron. The spike causes the neuron to activate the synapses of neurons to which it sends connections, thereby activating positive or negative currents in these downstream neurons depending on the type of synapse. Following an action potential, a neuron's state is reset, returning the neuron's membrane potential to baseline.

Neuroscientists routinely use computational models to explore how the brain functions. In this paper we use the hybrid neural model described by Izhikevich [4],

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \qquad (1)$$

[1]Apache Hadoop. http://hadoop.apache.org/

[2]Apache Giraph. http://giraph.apache.org/

$$\frac{du}{dt} = a(bv - u) \tag{2}$$

with the auxiliary after-spike resetting,

$$\text{if } v \geq 30mV, \text{then } \begin{cases} v = c \\ u = u + d \end{cases} \tag{3}$$

This represents a simple spiking neural model that is as biologically plausible as the Hodgkin-Huxley model, yet as computationally efficient as the integrate-and-fire model [4]. Here *v* represents the membrane potential of the neuron and *u* represents the membrane recovery variable that provides negative feedback to *v*. After the spike reaches its apex (+30mV), the membrane potential and the recovery variable are reset according to the equation (3). Synaptic currents or injected DC-currents are delivered via the variable *I*.

Note that in this paper we do not focus on evaluating the model itself, instead, we take it as an application case to explore how open source big data technology like Hadoop can be used to compute the model at a large scale.

From a computer science perspective, the neuron, itself, is the vertex, and the edges represent directional synaptic connections between neurons. Next, we will describe in more detail how we modeled the neural network with MapReduce and Giraph.

III. Graph Algorithms in MapReduce

A. Overview of MapReduce

MapReduce is a parallel programming model for large-scale data processing developed by Google. Inspired by the map and reduce functions commonly used in functional programming, it abstracts away messy details in distributed programming (such as synchronization, scheduling and fault tolerance) and simply leaves programmers with two main abstractions: Mapper and Reducer. Mappers take key-value pairs as processing primitives and produces intermediate key-value pairs, which are then automatically sorted, shuffled and passed to Reducers by the MapReduce framework, such that all the values associated with the same key are grouped in a list and fed to a Reducer. Each Reducer then aggregates these inputs and generates arbitrary key-value pairs as output. Both Mappers and Reducers operate in parallel, and can thus process large-scale datasets efficiently.

B. The general pattern of graph processing in MapReduce

MapReduce provides an enabling technology for large-scale graph processing. In MapReduce, we can represent a graph by key-value pairs. Keys denote vertices with an identifier and some associated metadata while values comprise local graph structure. The types of the key and value can be a primitive type (e.g. IntWritable, FloatWritable in Hadoop) or a complex object (e.g. custom writable in Hadoop).

Many graph algorithms are by nature iterative, including PageRank and the neural model we just introduced. At each iteration, the processing can be broken down into three steps [2]:

1) computations occur at every vertex as a function of the vertex's internal state and its local graph structure;
2) partial results in the form of arbitrary messages are passed via directed edges to each vertex's neighbors;
3) computations occur at every vertex based on incoming partial results, potentially altering the vertex's internal state.

Applying these three steps into MapReduce, each iteration is a single MapReduce job. Step-1 is naturally implemented in a Mapper while step-3 is implemented in a Reducer. A programmer does not have to worry about step-2 since it is taken care of by the MapReduce framework itself. More specifically, for the neural model mentioned in section 2, consider a sample neural network where three neurons N1, N2 and N3 are connected with each other in Figure 1-A. Figure 1-B illustrates the three-step processing for one iteration.

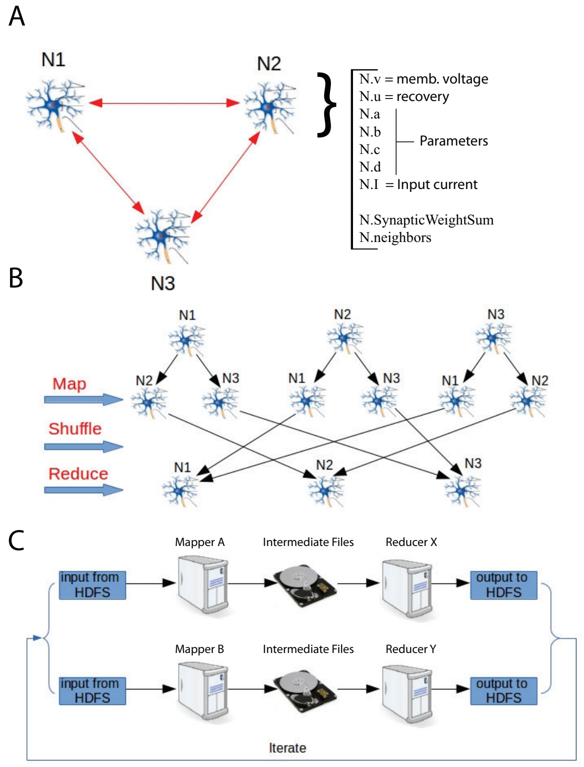

Fig. 1: A sample neural network simulated in MapReduce **(A)** an example of three interconnected neurons, with neuron N1 sending ouput connections to N2 and N3, and receiving input synaptic connections from N2 and N3. The parameters for each neuron can be encapsulated in a single structure with elements for membrane voltage, a recovery variable, dynamic parameters, and connectivity. **(B)** each simulation iteration is composed of three distinct steps: the Map step where the state of each neuron is updated and messages (i.e. spikes) are generated to be sent to its neighbors depending on the state of each neuron's membrane voltage variable; the Shuffle-sort step, which combines messages to the same neighbor and partitions the messages to be sent to the reducers; and, Reduce step which applies the incoming messages from a neuron's neigbors to that neuron's list of active inputs. **(C)** the hardware-level I/O involved in each processing step in (B) means that there is one read and one write to disk for each Map step and one read and one write to disk for each Reduce step.

To fully implement this neural network as a graph algorithm in MapReduce, we need to chain the same MapReduce job together until a certain criterion is reached; in our case, the computation stops when the maximum number of iterations, or time steps has been reached. The implications of this iterative pattern are diagrammed in the Figure 1-C.

C. Basic graph algorithm

Sahai and Sahai [6] gave a detailed illustration on how to implement the basic graph algorithm in MapReduce for the neural model. We restate it in Algorithm 1.

ALGORITHM 1: Basic Graph Algorithm in MapReduce

```
class Mapper
    method map (id n, neuron N)
        GenerateThalamicInput (N)
        UpdateInternalState (N)
        if N.v ≥ 30 then
            foreach edge ∈ N.neighbors() do
                Emit (edge.id, edge.weight)
            N.v ← N.c
            N.u ← N.u + N.d
        Emit (id n, neuron N)
class Reducer
    method reduce (id n, [w1, w2, . . .])
        sum ← 0
        neuron N ← ∅
        foreach w ∈ [w1, w2, . . .] do
            if IsNeuron(w) then
                N ← w
            else
                sum ← sum + w
        N.SynapticWeightSum ← sum
        Emit (id n, neuron N)
function GenerateThalamicInput
    if N.type = 'Excitatory' then
        N.I ← 5 × Random([-1, 1])
    else
        N.I ← 2 × Random([-1, 1])
function UpdateInternalState(neuron N)
    N.I ← N.I + N.SynapticWeightSum
    N.v ← N.v+0.5×(0.04×N.v$^2$+5×N.v+140-N.u+N.I)
    N.v ← N.v+0.5×(0.04×N.v$^2$+5×N.v+140-N.u+N.I)
    N.u ← N.u + N.a×(N.b×N.v-N.u) N.SynapticWeightSum ← 0
    N.time ← N.time + 1
```

Note that the above pseudocode is just a high-level description of the algorithm. Also, note that we abstract the thalamic input generation and internal neuron state processing into two helper functions and we will not mention them in the upcoming improved MapReduce graph algorithm. The algorithm corresponds to the three steps mentioned before. The network graph is constructed as key-values pairs in which the key is a unique id to identify a neuron and the value represents the internal state of a neuron and its local structure. These pairs are sent to each mapper as input. The mapper performs computations in step-1 where each individual neuron is processed and its internal state is updated. The entire neuron structure is passed to the reducers and if the neuron fires, it will also send synaptic weight messages to all its neighbors. This corresponds to step-2, where message shuffling and sorting occurs. The MapReduce framework takes care of routing of messages to ensure values associated with the same key are grouped together and delivered to the same reducer. In the reducer, all weight messages that are destined to the same neuron arrive together and are summed, and also include the neuron structure itself. We need a function *IsNeuron()* to distinguish between weight messages and neuron structure. The reducer finally updates the synaptic summation field of the neuron with the total weights emitted to a neuron, and writes the neuron data to the underlying HDFS, making it available as input for the next iteration's mapper. This is step-3. Each MapReduce job corresponds to one iteration of the algorithm.

D. Current best practices of graph algorithm in MapReduce

In [2], Lin and Schatz proposed two improved design pattern of graph algorithms in MapReduce which are In-Mapper Combining (IMC) and Schimmy. These are the current best practices for larges-scale graph processing in MapReduce. In this section we will give a brief review of both.

1) In-Mapper Combining (IMC): A large class of graph algorithms in MapReduce share a simple feature: mappers perform some computations on each vertex and emit messages to some of its neighbors. Reducers then group those messages in some fashion and then update the vertex's internal state. However, local aggregation can be done on messages before sending them to reducers in order to reduce the network traffic between map and reduce.

Algorithm 2 shows the improved graph algorithm using IMC.

ALGORITHM 2: Graph Algorithm using IMC

```
class Mapper
    method setup ()
        H ← AssociativeArray
    method map (id n, neuron N)
        GenerateThalamicInput (N)
        UpdateInternalState (N)
        if N.v ≥ 30 then
            foreach edge ∈ N.neighbors() do
                H{edge.id} ← H{edge.id}+edge.weight
            N.v ← N.c
            N.u ← N.u + N.d
        Emit (id n, neuron N)
    method cleanup ()
        foreach id n ∈ H do
            Emit (id n, value H{n})
```

Prior to processing any key-value pairs, the mapper first calls the *setup* method to initialize an instance of an associative array (e.g. a HashMap in Java) which maps a set of keys to a set of values. This associative array is updated whenever a neuron fires. However, emitting the messages is deferred to the cleanup phase where all messages with the same destination neuron have been aggregated, thus only a single message is emitted for each neuron. The downside of the IMC is the extensive usage of the local memory, which can cause swapping and become a performance bottleneck. Note that in the IMC pattern, the reducer code remains unchanged.

2) Schimmy: In the basic graph algorithm or IMC, the graph structure will be passed and shuffled across the network. This is very undesirable for the large graph datasets because the graph structure is usually much larger than the messages passed along the graph edges. Schimmy is designed for avoiding the shuffling of the graph structure. It is inspired by a well-known join-technique in the relational database field called a *parallel merge join* [7]. Suppose two relations, S and T, were both partitioned (in the same manner by the join key) into ten files and in each file, the tuples were sorted by the join key. In this case, we simply need to join the first file of S with the first file of T, etc. These merge joins can happen in parallel. Schimmy applies this idea in MapReduce graph processing by partitioning the graph into n parts, such that a graph $G = G_1 \cup G_2 \cup \ldots \cup G_n$, and within each part, vertices are sorted by vertex id. Hadoop provides a Partitioner interface for users to write their custom partitioner, therefore as long as we use the same partition function for partitioning the graph in the MapReduce graph algorithm, and set the number of reducers equal to the number of input partitions, Hadoop's MapReduce runtime system guarantees that the intermediate keys (vertex ids) processed by the reducer R_1 are exactly the same as vertex ids in G_1 and sorted in the same order; the same for R_2 and G_2 until R_n and G_n. Further, the intermediate keys in R_n represent messages passed to each vertex, and G_n key-value pairs comprise the graph structure. Therefore, a parallel merge join between R_n and G_n can merge the results of computations based on message passed to a vertex and the vertex's local structure, thus enabling the update of the vertex's internal state. In doing this we no longer need to shuffle the graph structure across the network. Algorithm 3 illustrates the MapReduce algorithm with Schimmy implemented for the neural model.

ALGORITHM 3: Graph Algorithm with Schimmy

```
class Mapper
    method map (id n, neuron N)
        GenerateThalamicInput (N)
        UpdateInternalState (N)
        if N.v ≥ 30 then
            foreach edge ∈ N.neighbors() do
                Emit (edge.id, edge.weight)
            N.v ← N.c
            N.u ← N.u + N.d
class Reducer
    method setup ()
        G.OpenGraphPartition()
    method reduce (id m, [$w_1$, $w_2$, ...])
        sum ← 0
        repeat
            (id n, neuron N) ← G.Read()
            if n ≠ m then
                Emit (id n, neuron N)
        until n = m;
        foreach w ∈ [$w_1$, $w_2$, ...] do
            sum ← sum + w
        N.SynapticWeightSum ← sum
        Emit (id n, neuron N)
```

Note that the mapper code remains nearly unchanged except for the deletion of the line emitting the graph structure. The reducer first opens the corresponding graph partition in the setup method. Then it advances through the graph structure until the corresponding vertex's structure is found. After jumping out of the loop, the vertex's internal state is updated and written back to the file system. The partitioner ensures consistent partitioning of the graph structure from iteration to iteration.

Although Schimmy eliminates the graph structure shuffling, it introduces remote file reading, as files containing graph partitions reside on the underlying distributed file system (HDFS for Hadoop) and the MapReduce runtime system arbitrarily assigns reducers to cluster nodes. This is potentially a performance bottleneck. The Schimmy pattern works well for the PageRank problem as discussed by Lin and Schatz in [2], but as we will see later, in our case, it does boost the performance.

3) Improved graph algorithm design pattern - Mapper-side-Schimmy: Inspired by the Schimmy pattern, we found that not only shuffling the graph structure is unnecessary, but in many algorithms, the graph structure itself is read-only. In Schimmy, the reducer remotely reads the graph structure and writes it back to the distributed file system after updating the vertex's internal state. To avoid the often unnecessary step of writing out the graph structure, we propose an improved design pattern: Mapper-side Schimmy, where we move the Schimmy to the mapper side and separate the vertex's internal state with the read-only graph structure such that the graph structure is read only once by the mapper and the reducer's tasks are simplified. This idea is illustrated as pseudo-code in the Algorithm 4.

ALGORITHM 4: Mapper-side Schimmy

```
class Mapper
    method setup ()
        G.OpenGraphPartition()
    method map (id n, NeuronState N)
        GenerateThalamicInput (N)
        UpdateInternalState (N)
        if N.v ≥ 30 then
            repeat
                (id m, neuron M) ← G.Read()
            until n = m;
            foreach edge ∈ M.neighbors() do
                Emit (edge.id, edge.weight)
            N.v ← N.c
            N.u ← N.u + N.d
        Emit (id n, NeuronState N)
class Reducer
    method reduce (id n, [$w_1$, $w_2$, ...])
        sum ← 0
        foreach w ∈ [$w_1$, $w_2$, ...] do
            if IsNeuronState(w) then
                N ← w
            else
                sum ← sum + w
        N.SynapticWeightSum ← sum
        Emit (id n, neuron N)
```

As we can see, in the Mapper-side Schimmy design pattern, a mapper no longer takes the entire graph structure as the value, instead, the value only contains a neuron's internal state ($NeuronState$). It remotely reads the graph partition corresponding to the input key-value pairs. Note that, just like Schimmy, we must use the same partition function for partitioning the graph as the partitioner in the MapReduce graph algorithm, and set the number of reducers equal to the number of input partitions. In the reducer, we still need a function ($IsNeuronState$) to distinguish between the synaptic weight and the neuron's internal state. By moving the Schimmy to the mapper side and separating the read-only graph structure with the vertex's internal state, we eliminate the need of writing the entire graph structure back to the distributed file system. The graph structure is, therefore, only read once with this pattern.

IV. Graph Algorithm in Giraph

Although MapReduce is an enabling technology for large-scale graph processing, it is far from ideal. No matter how we improve the design pattern for graph algorithms, MapReduce is not a suitable model for iterative graph processing per se. First, it is unnecessarily slow because each iteration is a single MapReduce job with lots of overhead, including scheduling, reading the graph structure from disk, and writing the intermediate results to the distributed file system. Second, the MapReduce abstraction is not intuitive for expressing graph algorithms as programmers have to "think in key-value pairs" in designing graph algorithms, thus making algorithms hard to implement and code hard to read and understand. Finally as we found in the Schimmy pattern and Mapper-side Schimmy pattern, the best strategy is data dependent and, therefore, hard to generalize.

As an alternative to the MapReduce model, we find that the vertex-centric approach is more powerful than MapReduce in expressing iterative graph algorithms. It employs a "think-like-a-vertex" programming model to support iterative graph computation. Each vertex is a single computation unit, which contains its internal state and all outgoing edges. Thus, the abstraction is made on a vertex-centric level, which is more intuitive for graph algorithms. The computation for a vertex involves receiving messages from its incoming edges, updating its internal state and sending the messages to other vertices along its outgoing edges.

Apache Giraph is an open source software framework for large-scale graph processing. It is a loose implementation of Google's Pregel [8]. Both Pregel and Giraph employ a vertex-centric programming model. Because Giraph is open source, we chose Giraph to implement the neural model. Giraph uses Bulk synchronous parallel (BSP) as its execution model. A BSP computation consists of a series of global supersteps[3]. Each superstep consists of three components: 1) concurrent computation: each processor is assigned a number of vertices and processes in parallel; 2) communication: the processors exchange messages between each other; 3) barrier synchronization: when a processor reaches the barrier, it waits until all other processors finish their communications. The BSP model is illstrated in the Figure 2-A. Based on vertex-centric model and BSP, we can express the processing for the small neural network in the Figure 2-B.

Algorithm 5 shows pseudo-code for the graph algorithm as a vertex-centric model. We can see that the algorithm expressed by a vertex-centric BSP model is more intuitive, because we treat each vertex as a class and a single computation unit. We limit the simulation to *40* iterations. In each iteration, the $compute$ method is called, it sums a list of messages sent from the previous iteration and updates the neuron's state, sending outgoing messages if the neuron fired. The computation halts if the maximum number of iterations has been reached.

Note that methods like $getSuperstep$, $getVertexValue$, $getEdges$, $setVertexValue$ and $voteToHalt$ are provided by the frameworks like Pregel or Giraph, with different naming conventions.

[3]Bulk_synchronous_parallel.
http://en.wikipedia.org/wiki/Bulk_synchronous_parallel

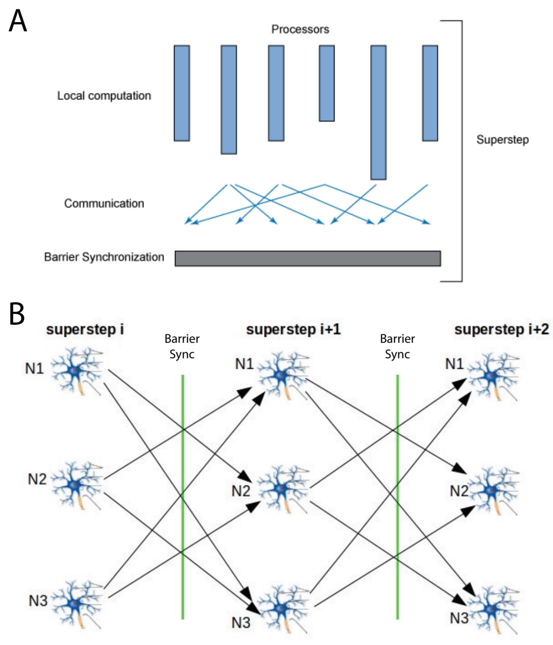

Fig. 2: BSP Model applied to Graph processing with a vertex-centric approach
(A) the barrier synchronization processing (BSP) model waits for processing for different computations within an iteration to complete and send out their messages before beginning processing the next iteration, this forced wait comprises a *barrier synchronization* and segregates processing within each iteration into a single *Superstep*. **(B)** this BSP model lends itself well to a vertex-centric approach where all of the computations for a single neuron are performed on a single processor and outgoing messages about that neuron's activity are determined and distributed to other neurons before the next iteration's superstep begins.

V. Results

For MapReduce graph algorithms, we implemented five versions: 1) basic implementation; 2) IMC; 3) Schimmy; 4) Mapper-side Schimmy; 5) the combination of Mapper-side Schimmy and IMC. We also have implemented a vertex-centric graph algorithm with Giraph. We tested our implementations on a Hadoop cluster with 16 worker nodes, 192 map task capacity and 96 reduce task capacity, running Hadoop-2.0. A neural network with 100,000 neurons and 2.3 Billion edges was implemented with an approximate size of 24GB (depending on the different implementations). First, we compared the running time for a 40 ms simulation (40 iterations) among six implementations (See Figure 3-A).

Although Schimmy has been proven to be effective in PageRank applications, it did not boost performance for our neural model. However, the Mapper-side Schimmy pattern improved performance by 11%. IMC, despite its simplicity, improved performance by 48%. We found that the combination of Mapper-side Schimmy and IMC produced the best result among these five MapReduce implementations and improved the performance by 64%. The Giraph implementation showed the best performance boost, which was 91% compared to the

ALGORITHM 5: Graph Algorithm in vertex-centric model

```
class NeuronVertex
    MaxSuperStep ← 40
    method compute (messages [$m_1, m_2, \ldots$])
        if getSuperstep() ≤ MaxSuperStep then
            sum ← 0
            foreach $w \in [m_1, m_2, \ldots]$ do
                sum ← sum + m
            neuron ← getVertexValue()
            neuron.SynapticWeight ← sum
            GenerateThalamicInput (N)
            UpdateInternalState (N)
            if neuron.v ≥ 30mV then
                foreach edge ∈ getEdges() do
                    sendMessage (edge.getVertexID(),
                    edge.getValue())
                neuron.v ← neuron.c
                neuron.u ← neuron.u + neuron.d
            setVertexValue (neuron)
        else
            voteToHalt()
```

basic MapReduce implementation. Compared to the Mapper-side Schimmy + IMC implementation, Giraph had a 60% performance improvement.

During experiments, we also recorded the running time of each iteration in order to see how each implementation performed as the network activity evolved (Figure 3-C). Usually, after approximately the 13^{th} iteration, neurons started firing intensively, which led to huge amount of network traffic. The Basic, Schimmy and Mapper-side Schimmy implementations all suffered from the increasing network traffic, while Basic+IMC, Mapper-side Schimmy+IMC and Giraph implementations all benefited from in-memory computation, with Giraph showing superior performance.

VI. Conclusion

Compared to Schimmy, this work presents an improved MapReduce graph algorithm design pattern which has been shown to be effective in the neural model we implemented. It can be applied to a large class of graph algorithms based on message passing. One limitation of the Mapper Side Schimmy implementation, is that it assumes there are no changes in the synaptic weights between neurons, an important feature of associative learning models. However, we addressed many inherent limitations of applying the MapReduce model to graph algorithms by exploring another approach, a vertex-centric BSP model, implemented with the open source large-scale graph processing framework Giraph. Using Giraph not only led to better performance, but also reduced the complexity of implementation. Our results suggest that application of our Mapper-side Schimmy design pattern and Giraph to other graph processing problems, such as PageRank, could potentially boost their performance.

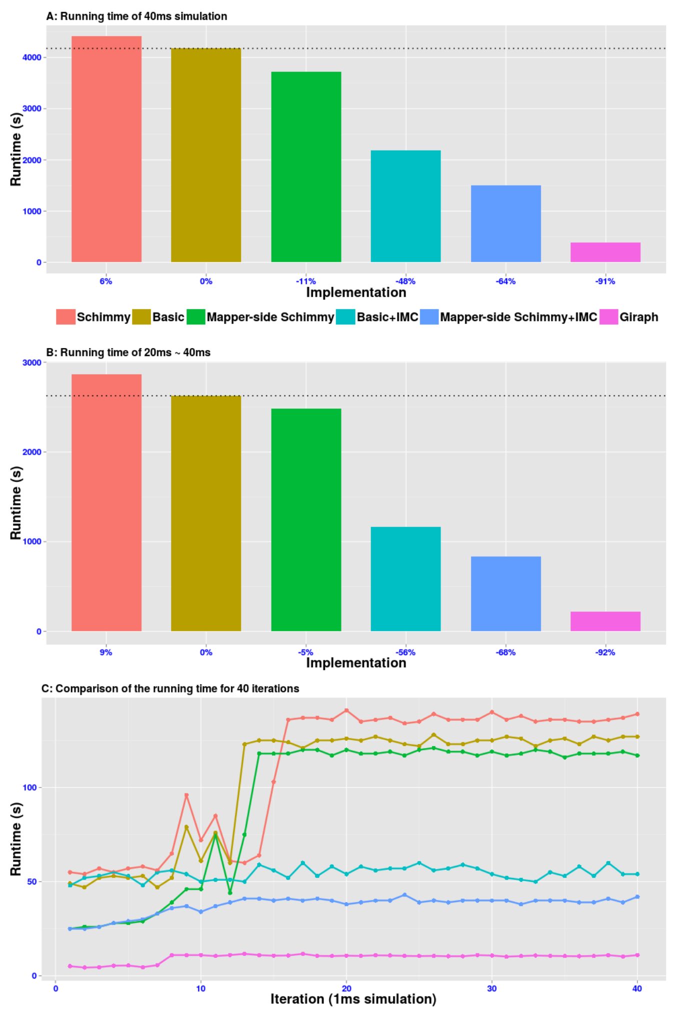

Fig. 3: Comparison of running times for different graph processing methods
(A) comparison of running times of 40 iterations, we use the "Basic" implementation as a benchmark for measuring the performance. **(B)** comparion of running times from 20 ms to 40 ms for steady-state network activity. Note that the ranking of computational savings is the same when considering only steady-state firing.**(C)** shows how performances vary for each iteration. After approximately 13 msec, the networks enter full-firing mode, where many more neurons are active in the network.

References

[1] J. Dean and S. Ghemawat, *MapReduce: Simplified data processing on large clusters*, In Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004). pages 137-150, San Francisco, California, 2004.

[2] Jimmy Lin and Michael Schatz, *Design Patterns for Efficient Graph Algorithms in MapReduce*, MLG 10. Washington, DC USA, 2010.

[3] Leslie G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, Volume 33 Issue 8, Aug. 1990.

[4] E. M. Izhikevich, *Simple model of spiking neurons*, IEEE Transactions on Neural Networks, 14(6):1569-1572, 2003.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd, *The PageRank citation ranking: Bringing order to the web*, Technical Report 1999-66, Stanford InfoLab, November 1999.

[6] Esha Sahai and Tuhin Sahai, *Mapping and Reducing the Brain on the Cloud*, arXiv:1207.4978, August 14, 2012.

[7] D. A. Schneider and D. J. DeWitt, *A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment*, In Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, pages 110-121, Portland, Oregon, 1989.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, *Pregel: a system for large-scale graph processing*, In SIGMOD10, pages 135-146, 2010.