EXPLORING APPLICATIONS OF RANDOM WALKS ON SPIKING NEURAL ALGORITHMS

LEAH E. REEDER*, AARON J. HILL †, JAMES B. AIMONE ‡, AND WILLIAM M. SEVERA§

Abstract. Neuromorphic computing has many promises in the future of computing due to its energy efficient and scalable implementation. Here we extend a neural algorithm that is able to solve the diffusion equation PDE by implementing random walks on neuromorphic hardware. Additionally, we introduce four random walk applications that use this spiking neural algorithm. The four applications currently implemented are: generating a random walk to replicate an image, finding a path between two nodes, finding triangles in a graph, and partitioning a graph into two sections. We then made these four applications available to be implemented on software using a graphical user interface (GUI).

1. Introduction.

There is growing interest in computing fields to utilize neural approaches; the importance of neuromorphic computing is growing, especially with its potential in beyond-Moore's Law techniques [8, 12]. Neuromorphic computing connects computing systems to neural systems in two ways: by using hardware to emulate the way the brain behaves and by using new architectures and paradigms that are brain inspired. The primary motivator for neuromorphic computing is to achieve high energy and volume efficiency, similar to human brains [4, 10].

In terms of power consumption and energy usage, nontraditional hardware is at a serious advantage in comparison to other traditional computing platforms [5, 9]. For random processes specifically, it can be seen that in some applications GPUs are more desirable than CPUs [17]. However, a recent paper showed that GPUs fall short on random simulations that are highly task-parallel rather than highly data-parallel [2]. Neuromorphic architectures can excell where GPUs fall short as they have similar parallelization capabilities but outperform energy-wise for random processes [6].

IBM's TrueNorth chip and the University of Manchester's SpiNNaker chip are two of the most popular neural-inspired hardwares. TrueNorth has a full custom application specific integrated circuit (ASIC) design that is highly optimized for a fixed spiking neuron model. Conversely, SpiNNaker is optimized for communication of the spike-based network and is flexible regarding the neuron model used. SpiNNaker, however, uses more energy than TrueNorth but still significantly less than traditional von Neumann architectures [10].

On the other side of neuromorphic computing are neural-inspired algorithms. Spiking neural algorithms for a specific random process, markov chain random walks, have been developed previously [1, 11]. It has been shown that the small neural circuits created from these algorithms are able to handle random walk simulations both efficiently and scalably. These algorithms were designed specifically to be used on neuromorphic architecture (both TrueNorth and SpiNNaker) in order to perform energy efficient simulations.

We present in this paper an extension to one of the algorithms previously developed as well as several important random walk applications utilizing the algorithm. Once these simulations are running on neuromorphic architecture, there will be sig-

*Colorado School of Mines, lreeder@mines.edu
†Sandia National Laboratories, ajhill@sandia.gov
‡Sandia National Laboratories, jbaimon@sandia.gov
§Sandia National Laboratories, wmsever@sandia.gov

nificant power consumption advantages. Additionally, we will describe resulting visualizations of each application on an easy to use graphical interface that utilizes the spiking neural algorithm. Lastly, we will discuss hopes for further developments of the applications.

2. Background.

2.1. Introduction to Random Walks. Random walks are heavily studied stochastic processes and are well defined models of the motion a particle takes in a diffusion scheme. A 1-dimensional random walk without interactions is defined as a function $\tau : \mathbb{N} \to \mathbb{Z}$ where

$$\mathbb{P}(\tau(n+1) = x+1 | \tau(n) = x) := p \tag{2.1}$$

and

$$\mathbb{P}(\tau(n+1) = x-1 | \tau(n) = x) = 1 - p := q \tag{2.2}$$

Essentially, a particle at position x moves to the right $(x+1)$ with probability p and to the left $(x-1)$ with probability $1-p = q$. After n timesteps, the position of the particle is defined to be at some distance $m(t)$ from the origin. The 1-dimensional random walk generalizes easily to higher dimensions and ultimately to walks on general graphs [3, 15].

2.2. Diffusion Equation and Random Walks. The motivation for the random walk algorithms mentioned earlier was to solve partial differential equations (PDEs). As random walks model the motion of diffusion, they inherently solve the well known diffusion PDE. A derivation of the 1-dimensional diffusion equation as the limit of a random walk is outlined in Chapter 5 of [14]. We will summarize those steps here.

Assume a particle begins at the origin and has a step length of Δx and a timestep of Δt. Furthermore, assume that p is the probability that the particle moves Δx to the right and $q = 1 - p$ is the probability that it moves Δx to the left. Let the probability event that the particle is at position $x = m(\Delta x)$ with m steps to the right at time $t = n(\Delta t)$ after n time periods be denoted as:

$$p(m, n) = \text{probability of } r \text{ steps right} = \binom{n}{r} p^r q^{n-r}, \text{ where } r = \tfrac{1}{2}(n-m).$$

This is the binomial distribution, and can be approximated by a normal distribution using the Central Limit Theorem.

Let the function $u(x,t)$ represent the probability that the particle lies in an interval centered at x at time t with width $2\Delta x$. Then, we can denote $u(x,t)$ as the space- and time-scaled probability function, appropriately discretized as

$$u(x,t) = \frac{p(\frac{x}{\Delta x}, \frac{t}{\Delta t})}{2(\Delta x)}. \tag{2.3}$$

It can be shown that this simplifies nicely to the Gaussian distribution,

$$u(x,t) = \frac{e^{-\frac{x^2}{4Dt}}}{\sqrt{4\pi D t}} \tag{2.4}$$

where D is the diffusion coefficient and is defined as

$$D = \frac{\Delta x^2}{2(\Delta t)}. \tag{2.5}$$

The function $u(x,t)$ satisfies the following PDE (the diffusion equation)

$$\frac{\partial u}{\partial t} = D\frac{\partial^2 u}{\partial x^2}. \tag{2.6}$$

Thus, the diffusion equation can be derived from a 1-dimensional random walk. As stated previously, this case is easily generalizable to higher dimensions, which is critical for generating random walks on arbitrary graphs.

2.3. Introduction to Spiking Algorithm. The spiking neural algorithm discussed here was first presented in [11]. This paper presents two spiking algorithms for random walks, the particle method and the density method. The goal of these algorithms is to solve the diffusion equation PDE stochastically using random walks without interaction on neuromporhic architecture. We will focus on the density method only.

In the density method, rather than tracking each individual walker's position at any given point in time, we can instead keep track of the nodes on the graph and count how many walkers are at each node at any given point in time. When we are running the spiking simulation, we embed a spiking circuit that is located at each node in the graph. An outline of a frame of the random walk simulation utilizing the density method is described below.

1. An injected current starts initial walkers at a given node in graph by sending signals to walker generator neurons and walker counter neurons
2. Walkers are distributed throughout the circuit at that node through excitatory signals from the walker generator neurons
3. When a signal is received by output gate neurons, their potentials are modified
4. If the output gate neuron's thresholds are met, the output gate neurons determine whether or not to spike based off of a certain probability (discussed in next section)
5. If the neurons spike, a signal is sent from the output gate neurons in the current circuit to the determined walker generator neurons and walker counter neurons at the neighboring node's circuit

This process is followed generally in each timestep until there are walkers spread out over all circuits throughout the graph. Originally, the code only supported up to 2 (1-dimensional graphs) or 4 (2-dimensional graphs) output neurons per circuit, which corresponds to only 2 or 4 neighbors on a graph. However, this has been extended to support arbitary graphs with any number of neighbors. This extension is outlined in the next section.

The density method is implemented on TrueNorth. It currently supports only 4 neighbors, but we are working on extending it to support arbitrary graphs. This method implemented on TrueNorth has been shown to have significant power advantages compared to random walk codes on traditional architectures [1].

3. Extending the Current Algorithm.

In order to make the code generalizable to more than just 1D and 2D graphs, more neighbors needed to be supported. This is useful for applications such as social network graphs, where each node has significantly more connections than just 4.

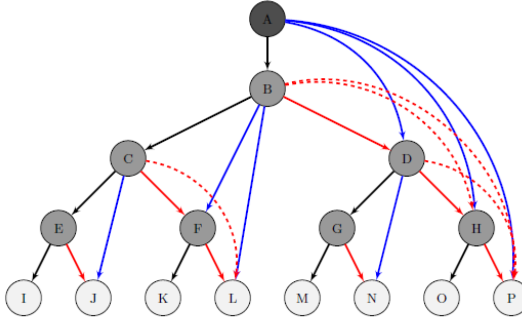

Fig. 3.1. *A probability tree with 8 output gate neurons (8 directions). The walker generator neuron is depicted in dark grey and labeled as A, the random gate neurons are depicted in grey and are labeled as B-H, and the output gate neurons are depicted in light grey and are labeled as I-P. The connections are colored as follows: excitatory connections with delay 1 are black (blue with higher delays), inhibitory connections with delay 1 are red (dashed red with higher delays)*

We were able to develop a stochastic algorithm that supports any number of neighbors. The direction that a neuron is going to take is determined stochastically through a probability tree. Once a certain type of neuron is signalled to the root node in the tree, walkers are subsequently propagated down through the tree. This probability tree works by always receiving a spike through a walker generator neuron. The generator neuron then has connections with stochastic neurons that randomly spike. If a stochastic neuron spikes, it sends excitatory signals to the left half of the tree and inhibitory signals to the right half. These types of signals propagate throughout the tree until they reach output signals at the bottom of the tree.

Each stochastic neuron in the probability tree has two outputs (excitatory and inhibitory), meaning the algorithm currently best supports base 2 number of output directions. That is, graphs with base 2 number of neighbors (2, 4, 8, etc.) are optimal. If the graph has a differing number of neighbors, there will always be more output neurons than needed. This has potential memory cost limitations as there often will be non-optimal numbers of neighbors, but has relatively low impact on the rest of the simulation.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Direction
1	1	1	0	1	0	-	0	1	0	-	0	-	-	-	0	Direction 1
1	1	1	0	*	0	-	0	-	1	-	0	-	-	-	0	Direction 2
1	1	*	0	-	1	-	0	-	-	1	0	-	-	-	0	Direction 3
1	1	*	0	-	*	-	0	-	-	-	1	-	-	-	0	Direction 4
1	0	-	1	-	-	1	0	-	-	-	-	1	0	-	0	Direction 5
1	0	-	1	-	-	*	0	-	-	-	-	-	1	-	0	Direction 6
1	0	-	*	-	-	-	1	-	-	-	-	-	-	1	0	Direction 7
1	0	-	*	-	-	-	*	-	-	-	-	-	-	-	1	Direction 8

Table 3.1

A schematic of the probability tree with 8 directions. The walker generator neuron is labeled as A, the random gates are labeled as B-H and the output gate neurons are I-P. Each output gate neuron corresponds to a different direction. If a neuron recieves an excitatory signal and spikes, it is denoted by a '1'. If a neuron recieves an inhibitory signal and does not spike, it is denoted by a '0'. If a neuron recieves an excitatory signal but does not spike, it is denoted by a ''. Nodes that do not recieve a signal or spike are denoted by a '-'.*

An example of a probability tree with 8 output gate neurons can be seen in Figure 3.1 and a supplementary schematic can be seen in Table 3.1. This is especially useful to see what combination of random neurons are needed in order for each output neuron to spike.

4. Applications of Random Walks. After running a random walk simulation, the resulting walk can be used in applications such as social network analysis and graph theory. Data analytics is difficult on large graphs. It can be shown that some of the more traditional graph algorithms do not scale well on larger graphs due to being sequential in nature [7]. We expect that it could be beneficial to analyze the results of the random walk simulation rather than the actual graph. By generating these applicatons on a random walk, we can instead implement these algorithms in a nontraditional approach. Here we present four applications that we have implemented using the spiking neural algorithm introduced earlier, instead of using the graph dataset.

4.1. Using an image as underlying graph structure. The first application we implemented was using an image as the mesh. Instead of running the simulation on a graph with arbitrary probabilities, the probabilities are determined by the color of each pixel in the image. A noise constant can be used that serves to decrease the variation in each probability without changing the underlying structure.

One application that this method is particularly useful for is defining energy landscapes. For example, if the different colors in the image correspond to an energy gradient, then the directional probabilities assigned to each pixel in the graph would define the resulting energy landscape. When the random walk is generated, there is a higher number of walkers at the ideal energy levels. The pseudocode is listed in Algorithm 1 and an example of the output is depicted in Figure 5.4.

Algorithm 1 Image to Walk

```
1:  read in image file
2:  specify initial walker starting points
3:  specify noise constant
4:  convert file into matrix (each pixel in image is converted to a number based on
    color)
5:  for each entry in matrix do
6:      determine 4 neighboring entry locations (up, down, left, right)
7:      determine 4 neighboring entry values (number corresponding to color)
8:      calculate probabilities for all 4 neighboring entries
9:      the probability is determined by the value of the neighboring entries
10:     (the darker the pixel, the more likely the walker will go to that pixel)
11: end for
12: generate random walk
```

4.2. Path finding on network graphs. The second application that is implemented is the classic path finding algorithm. Given two specified nodes, this algorithm can determine if there is a path between them using the random walk simulation data.

Finding paths in graphs is commonly used in road networks to find directions from point A to point B. It can also be used for flight path connections between different airports. Overall, this application is best used on arbitrary network graphs.

The pseudocode is listed in Algorithm 2 and an example of the output is shown in Figure 5.6.

Algorithm 2 Path Finding

```
1: read in network data
2: initialize graph from data
3: specify source and target nodes
4: for each node in graph do
5:     determine the number of neighbors that node has
6:     the probability is proportional to the number of neighbors that node has (as-
   suming all edges have equal weights)
7:     assign probability to each neighboring node
8: end for
9: generate random walk with initial walkers at source node
10: for all timesteps in simulation do
11:     if neurons at target node spiked at time then
12:         there is a path from source to target
13:     else
14:         there is not a path from source to target
15:     end if
16: end for
```

4.3. Triangle Finding on Network Graphs. The third application that is implemented is an algorithm that finds triangles on an arbitrary graph. Given a specific dataset, a random walk is generated for each node in the graph. Using the data from the random walk simulation, it can determine whether or not there was a triangle at each node.

Triangle finding can be used to find close-knit communities with obvious links between them. Many social network sites can use found triangles to suggest 'friends' or 'connections'. Many traditional triangle finding or counting algorithms utilize the square of adjacency or incidence matrices [16]. With arbitrary graphs especially, these matrices can get very large and costly quickly. Utilizing random walks to find triangles can help in this regard as this nontraditional triangle finding algorithm does not deal with adjacency or incidence matrices. The pseudocode is listed in Algorithm 3 and an example of the output is shown in Figure 5.9.

Algorithm 3 Triangle Finding

```
1: read in network data
2: initialize graph from data
3: for each node in graph do
4:     for each node2 in graph do
5:         determine the number of neighbors that node2 has
6:         the probability is proportional to the number of neighbors that node2 has
   (assuming all edges have equal weights)
7:         assign probability to each neighbor of node2
8:     end for
9:     generate random walk with initial walkers at node
10:     if neurons at node spiked at time=0 and time=3 then
```

11: node is in a triangle
12: **end if**
13: to find other nodes in same triangle, look through spike log to see which neurons sent and recieved the spikes starting at time=0 through time=3
14: **end for**

4.4. Graph Partitioning on Network Graphs. The fourth application that is implemented is an algorithm that partitions a graph into two parts. Given a specific dataset, a random walk is generated that starts at the first node in the set. Using the data from the random walk simulation, the time at which neurons at each node spiked can be determined. If it took walkers a 'long' time (above a certain threshold) to get to a certain section of the graph, then the nodes in that section become their own partition.

A common application of graph partitioning is image segmentation [13]. To segment an image, often there are clear sections that correspond to different parts of the image. Once these segmentations are partitioned, the image (or graph) is segmented. This is useful to identify certain parts of the image as being different from others. The pseudocode is listed in Algorithm 4 and an example of the output is shown in Figure 5.11.

Algorithm 4 Graph Partitioning

1: read in network data
2: initialize graph from data
3: specify threshold
4: **for** each node in graph **do**
5: determine the number of neighbors that node has
6: the probability is proportional to the number of neighbors that node has (assuming all edges have equal weights)
7: assign probabilities to each neighboring node
8: **end for**
9: generate random walk with initial walkers at initial node
10: **for** all neurons at each node **do**
11: determine the time at which each neuron first spiked (if at all)
12: **if** time at which they spiked is greater than the threshold **then**
13: node is put into second partition
14: **end if**
15: **end for**

5. A Graphical User Interface for Applications of Random Walks. Here we present a Graphical User Interface (GUI) for users to be able to use a plethora of applications of random walks on our spiking neural framework. This is a nice, clear way to see the useful applications of random walks.

The visualizations shown in three of the GUI applications (Path Finding, Triangle Finding, and Graph Partitioning) depict the user-specified graphs. The nodes are sized and colored based on howmany connections they have (unless the color is otherwise specified on the legend).

The main menu, with the four random walk applications can be seen below.

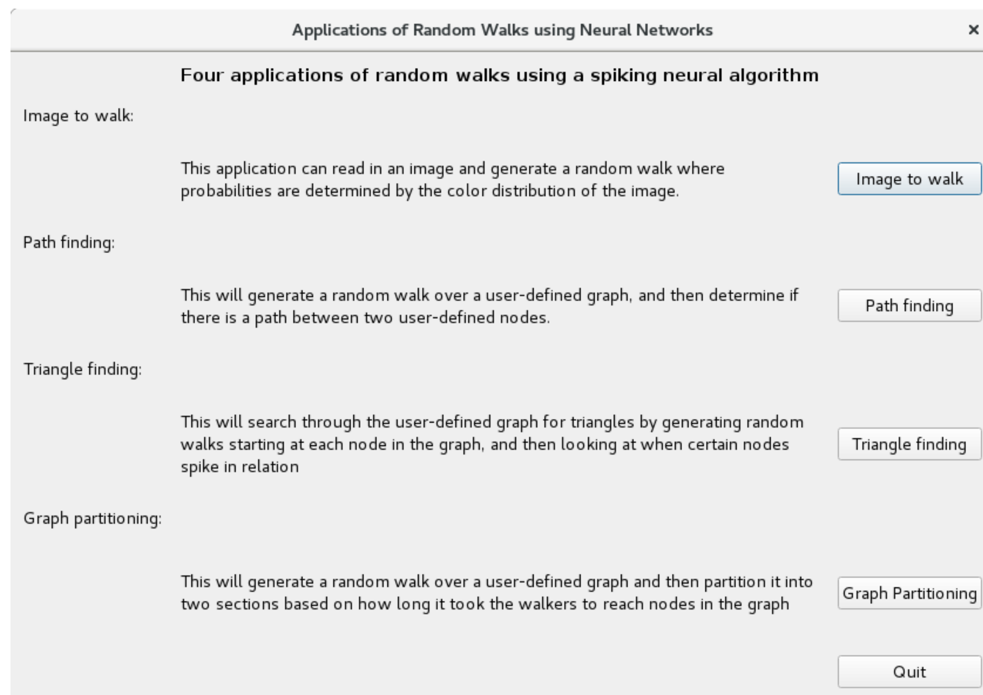

Fig. 5.1. *Main menu: Gives four application options, with summaries of each*

5.1. Image to Walk. The first application, Image to Walk can be seen below. The user is asked to upload the image in array form. This is because there are many different ways to convert an image into an array with various condensing techniques. By letting the user convert the image themselves, they can control what kind of condensing they will get. A pop-up message indicates when the simulation is complete that specifies the name of the resulting file to view the results. This file is stored in the same directory as the GUI files.

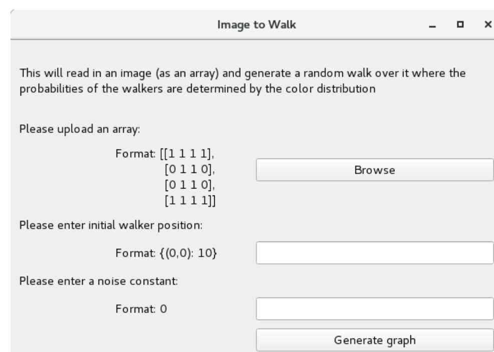

Fig. 5.2. *Image to walk menu: allows user to specify array, initial walker locations/size, noise constant*

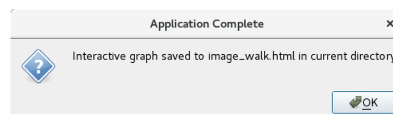

Fig. 5.3. *Pop-up message that indicates when simulation is complete/ where output file is stored*

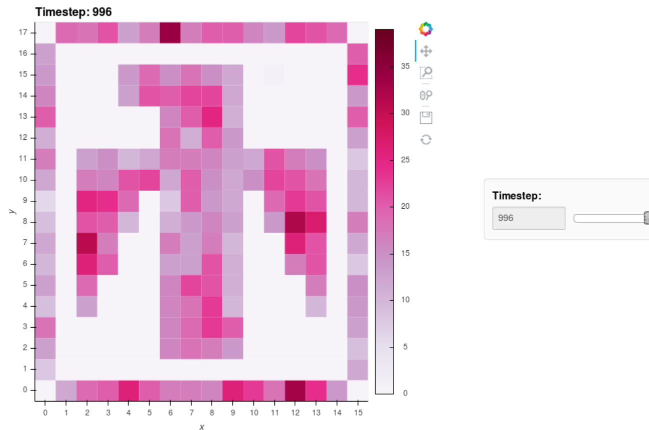

FIG. 5.4. *Resulting image from Image to Walk application at last timestep*

5.2. Path Finding. The second application, Path Finding, can be seen below. The user is asked to choose the file that the graph data set is in, specify the source node, and specify the target node. Once the simulation is done running, if there is a path from the source node to the target node, a visualization of the data set showing the path from the source to the target node pops up onto the screen.

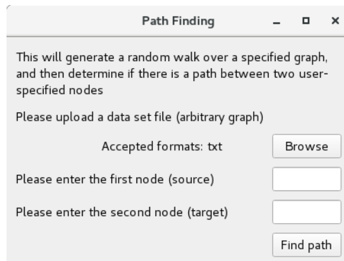

FIG. 5.5. *Path finding menu: allows user to choose dataset file and specify source/target nodes*

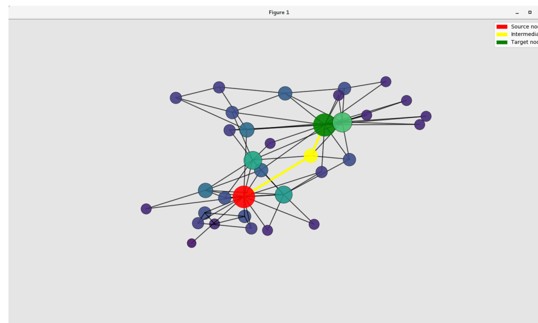

FIG. 5.6. *An example of a visualization of a path. This dataset is Zachary's Karate Club [18]*

5.3. Triangle Finding. The third application, Triangle Finding, can be seen below. The user is asked to choose the file that the graph data set is in. Once the simulation is complete, a visualization of the data set and all of the triangles in it pops up onto the screen. Note that since the triangle finding algorithm is done on the random walk, it often does not find all triangles in the dataset, just some. This is because walkers will not go in all directions every time the simulation is run. A limitation with this application is that sometimes the nodes are in multiple triangles and the visualization does not support that case. In this case, that node only shows one of the triangles that it is in, not both. As this application generates a random walk for each node in the graph, it takes a while to run. There is a popup message before the application starts running to warn the user that it will take a while.

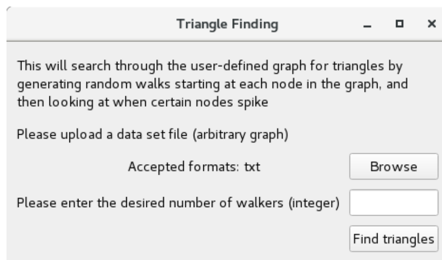

FIG. 5.7. *Triangle finding menu: allows user to choose dataset file*

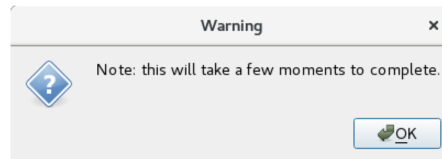

FIG. 5.8. *Pop up message before application starts to run, warning user of long runtime*

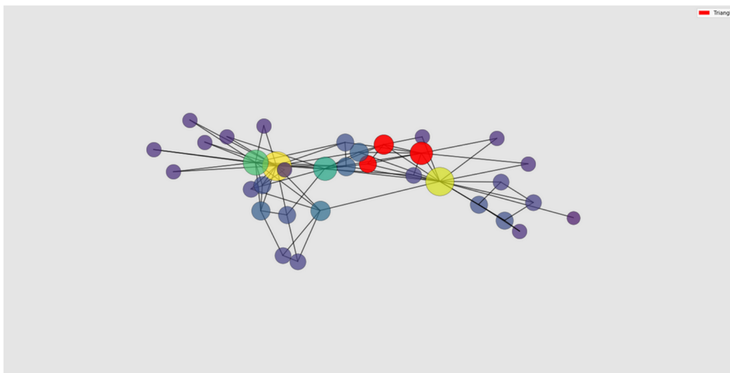

FIG. 5.9. *An example of a visualization of one triangle in a dataset. This dataset is Zachary's Karate Club [18]*

5.4. Graph Partitioning. The last application, Graph Partitioning, can be seen below. The user is asked to choose the file that the graph dataset is in. Once the simulation is complete, a visualization of the two partitions pops up onto the screen. As the partition algorithm runs on the random walk output on an undirected and unweighted graph, the graph gets partitioned differently every time. A limitation of this is that occasionally it would be better for the graph to be partitioned into three or four graphs, and the algorithm only splits it into two partitions.

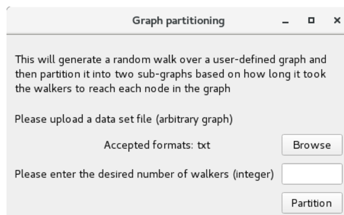

FIG. 5.10. *Graph partitioning menu: allows user to choose data set file*

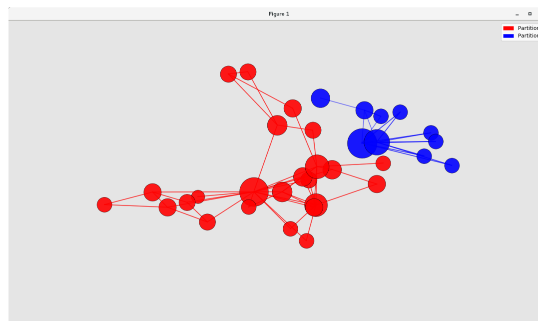

FIG. 5.11. *An example of a visualization of a partitioned dataset. This dataset is Zachary's Karate Club [18]*

6. Future Work. In the future, it would be advantageous to generate even more random walk applications to use with the spiking neural algorithm. One specific application that would be useful is image segmentation. Another important next step is getting all of the applications up and running on TrueNorth, a neuromorphic chip. Although the applications work on the software by themselves, we can really see vast performance advantages on neuromorphic hardware [6]. If we could get these applications running on the hardware, we could potentially have the user specify on the GUI whether they would like to run the random walk application on the spiking software, or the spiking hardware available (TrueNorth, SpiNNaker, Loihi, etc). This would be particularly useful to compare the performance of the different platforms.

7. Conclusion. With these applications implemented on a neural algorithm, they have the potential to be more energy-efficient and have less power-consumption compared to their traditional counterparts. Once implemented on neural hardware, there will be significant performance advantages, even without being seriously optimized. It is important to consider these non-neural applications that utilize neural frameworks in order to promote more use of nontraditional architectures.

8. Acknowledgement. This research was funded by the Laboratory Directed Research and Development program at Sandia National Laboratories. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract de-na0003525.

REFERENCES

[1] J. B. Aimone, A. Hill, R. Lehoucq, O. Parekh, and W. Severa, *Neuromorphic hardware implementation of spiking algorithms for markov random walks*, in Proceedings of the 2018 International Conference on Neuromorphic Systems (ICONS), 2018. To appear.

[2] R. M. Bergmann, K. L. Rowland, N. Radnović, R. N. Slaybaugh, and J. L. Vujić, *Performance and accuracy of criticality calculations performed using warp – a framework for continuous energy monte carlo neutron transport in general 3d geometries on gpus*, Annals of Nuclear Energy, 103 (2017), pp. 334 – 349.

[3] P. C. Bressloff, *Stochastic processes in cell biology*, vol. 41, Springer, 2014.

[4] A. Calimera, E. Macii, and M. Poncino, *The human brain project and neuromorphic computing*, Functional neurology, 28 (2013), p. 191.

[5] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, *A performance study of general-purpose applications on graphics processors using cuda*, Journal of parallel and distributed computing, 68 (2008), pp. 1370–1380.

[6] A. J. Hill, J. W. Donaldson, F. H. Rothganger, C. M. Vineyard, D. R. Follett, P. L. Follett, M. R. Smith, S. J. Verzi, W. Severa, F. Wang, et al., *A spike-timing neuromorphic architecture*, in Rebooting Computing (ICRC), 2017 IEEE International Conference on, IEEE, 2017, pp. 1–8.

[7] J. A. Miller, L. Ramaswamy, K. J. Kochut, and A. Fard, *Research directions for big data graph analytics*, in 2015 IEEE International Congress on Big Data, June 2015, pp. 785–794.

[8] D. Monroe, *Neuromorphic computing gets ready for the (really) big time*, Commun. ACM, 57 (2014), pp. 13–15.

[9] D. Ristè, M. P. da Silva, C. A. Ryan, A. W. Cross, A. D. Córcoles, J. A. Smolin, J. M. Gambetta, J. M. Chow, and B. R. Johnson, *Demonstration of quantum advantage in machine learning*, npj Quantum Information, 3 (2017), p. 16.

[10] C. D. Schuman, T. E. Potok, R. M. Patton, J. D. Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank, *A survey of neuromorphic computing and neural networks in hardware*, arXiv preprint arXiv:1705.06963, (2017).

[11] W. Severa, R. Lehoucq, O. Parekh, and J. B. Aimone, *Spiking neural algorithms for markov process random walk*, arXiv preprint arXiv:1805.00509, (2018).

[12] W. Severa, O. Parekh, K. D. Carlson, C. D. James, and J. B. Aimone, *Spiking network algorithms for scientific computing*, in Rebooting Computing (ICRC), IEEE International Conference on, IEEE, 2016, pp. 1–8.

[13] J. Shi and J. Malik, *Normalized cuts and image segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 22 (2000), pp. 888–905.

[14] R. W. Shonkwiler and F. Mendivil, *Explorations in Monte Carlo Methods*, Springer Science & Business Media, 2009.

[15] L. Sjogren, *Chapter 2 : Random walks [pdf]*.

[16] C. E. Tsourakakis, *Counting triangles in real-world networks using projections*, Knowledge and Information Systems, 26 (2011), pp. 501–520.

[17] D. Van Antwerpen, *Improving simd efficiency for parallel monte carlo light transport on the gpu*, in Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, ACM, 2011, pp. 41–50.

[18] W. Zachary, *An information flow model for conflict and fission in small groups*, Journal of anthropological research, 33 (1976).