Performance Effects of Dynamic Graph Data Structures in Community Detection Algorithms

Rohit Varkey Thankachan[1], Brian P. Swenson[2], and James P. Fairbanks[2]

***Abstract*— Community detection algorithms create a dynamic graph as an internal data structure for tracking agglomerative merges. This community (block) graph is modified heavily through operations derived from moving vertices between candidate communities. We study the problem of choosing the optimal graph representation for this data structure and analyze the performance implications theoretically and empirically. These costs are analyzed in the context of Peixoto's Markov Chain Monte Carlo algorithm for stochastic block model inference, but apply to agglomerative hierarchical community detection algorithms more broadly. This cost model allows for evaluating data structures for implementing this algorithm and we identify inherent properties of the algorithm that exclude certain optimizations.**

I. Introduction

Representation and storage of graphs in computer memory is a significant factor affecting performance of algorithms operating on graphs. Each representation has advantages and disadvantages with respect to memory and performance considerations for different kinds of graphs and for different graph algorithms. All real world networks change over time, and dynamic graph analysis studies the behavior of these graphs as they are modified with insertions, updates and deletions. Community detection algorithms often use an internal data structure to represent the connections between communities. Viewing these internal data structures as a dynamic graph enables novel analysis of such community detection algorithms. Dynamic graph analysis adds further performance considerations for graph representations because memory characteristics are determined by both storage efficiency and the performance of modifications.

Community detection is an important problem in analzing complex networks [1] with diverse applications in areas such as social networks, financial networks [2] and biological networks [3], [4]. Given a graph $g = (V, E)$, global community detection algorithms assign each vertex of the input network to communities (or blocks), where each community consists of vertices with more intra-community interactions and fewer inter-community interactions [5], [6]. Community detection algorithms infer and extract useful structure and information from complex networks. Community detection algorithms exists for both high accuracy applications where getting the optimal communities is important and high performance approximations for larger networks where finding optimal partitions is impractical. These algorithms are computationally expensive and require high performance computing techniques for effective computation.

Several community detection algorithms are based on hierarchical agglomerative clustering and create a block multigraph from the given graph using the block assignments and modify this block graph to improve block assignments [7]–[9]. In order to study community formation and detection, stochastic block models, which sample graphs from distributions where each vertex has a latent block label and the probability of an edge between two vertices depends only on the labels of the endpoints, have been developed and analyzed [10]. Tiago Peixoto's community detection algorithm [11]–[13] is based on the degree corrected stochastic block models [14]. It also uses a block graph during the computation of the community assignments with a complexity of $O(nln^2n)$, where n is the size of V. The 2018 GraphChallenge competition uses this $O(nln^2n)$ algorithm as a baseline algorithm for its streaming stochastic block model partitioning challenge [15]. Community detection algorithms that use block graphs generally use an inter-block edge count matrix to represent block graph as a dense or sparse matrix. Several community detection algorithms use a weighted adjacency matrix of the block graph, i.e the inter-block edge count matrix, as the core backing data structure.

The Peixoto community detection algorithm uses a Monte Carlo Markov Chain (MCMC) approach to sample community assignments and proposes a new assignment of blocks until the target partition is found [11]–[13]. The $O(nln^2n)$ algorithm proceeds by alternating between an agglomerative merge phase and a nodal update phase. The agglomerative merge phase is used to create an initial block assignment from a previous block assignment with a greater number of blocks. The best merges for the vertices are found and carried out until a block assignment with the required number of blocks is obtained. The nodal update phase is performed then, where a new block is proposed for every vertex in the graph using the block assignments of neighbors of the vertex. The change in entropy by carrying out these proposals are evaluated and accepted or rejected based on a uniform random draw. The inter-block edge count matrix is updated with the new rows and columns of the accepted proposal. Updates to the rows and columns affects the sparsity of the inter-block edge count matrix. The nodal update iterations continue until a specified maximum iterations is reached or the convergence criteria based on the fraction of change of entropy is obtained. A new inter-block edge count matrix whose size depends on the number of blocks is created

[1]School of Computer Science, Georgia Institute of Technology, Atlanta, GA, USA rohitvarkey@gatech.edu
[2]Georgia Tech Research Institute, Atlanta, GA, USA brian.swenson,james.fairbanks@gtri.gatech.edu

during each agglomerative phase. The interblock edge count matrix is dynamically modified in the nodal phase with insertions, deletions and updates occurring to the data structure.

The algorithm starts with n blocks and reduce the number of blocks by a factor of two until the entropy increases. Once the partition entropy increases, a golden section search is used to find the optimal number of blocks [16]. The nodal update phase of the Peixoto MCMC algorithm removes and inserts vertices from communities allowing for higher quality partitions. The Peixoto algorithm aims to minimize overall entropy [13] as the evaluation metric as opposed to maximizing modularity based metrics used in other community detection algorithms such as the Louvain algorithm [17].

Adjacency matrix representations along with external auxiliary data structures like max heaps [8], [18], [19], dense modularity matrices [17], and eigenvectors to optimize performance are seen in literature [20]. These static representations of the inter-block edge count matrix, commonly used in purely agglomerative algorithms, are not designed to perform well for dynamic operations on the representation. The nodal update phase tends to dominate the runtime leading to poor performance of the overall algorithm with static representations of the inter-block edge count matrix.

While this paper deals with the Peixoto MCMC algorithm, much of our analysis applies to any community detection algorithm that uses random sampling of neighborhoods, modifies an inter-block community matrix, and allows pooling of many small updates into parallel phases.

II. Methodology

A. Inter-block Edge Count Data Structure

The inter-block edge count structure stores the representation of the block graph. It is a matrix M, where M_{ij} represents the number of edges $i \to j$ between vertices in blocks i and j. The updates to the counts in the algorithm are updated in the structure by setting M_{ij} to the required value. We represent the inter-block edgecount matrix, M, as a directed weighted graph, M_G, where the weights represent the edge counts between the blocks in the original graph. We can further break down updates on the inter-block edge count matrix, M, to the following operations on the graph, M_G.

1) Insertion: M_{ij} changes from 0 to a positive integer, w_{ij} corresponds to adding an edge $i \to j$ to M_G with weight w_{ij}.
2) Deletion: M_{ij} changes from a positive integer, w_{ij} to 0 corresponds to removing an edge $i \to j$ from M_G.
3) Updates: M_{ij} changes from a positive integer, w_{ij} to another positive integer, w'_{ij} corresponds to updating the weight of the edge $i \to j$ in M_G with weight w'_{ij}.

Algorithms that assign vertices to communities only once do not need to consider deletions from this data structure. Therefore, they use static structures which are usually faster. We analytically compare different graph representations as backing stores for this graph and experimentally probe the performance effects of the data structures on the algorithm.

B. Graph Representations

Memory access patterns have a significant impact on performance of graph algorithms. For typical graph algorithms like traversal using BFS, graphs have poor spatial and temporal locality making memory access patterns difficult to optimize [21]. However, several steps in the proposed algorithm are able to make use of memory access patterns as we are dealing with one vertex or one block at a time. Using data structures that provide us with good locality for these kinds of memory access is advantageous for the performance of the algorithm. Adjacency matrix based and adjacency list based graph representations are used as data structures to store the block graph. The block graph is continuously modified through the course of the algorithm with insertions, updates and deletions of edges in the block graph. The streaming graph behavior exhibited by the block graph requires choosing a data structure that is optimized for streaming graphs to obtain good performance.

1) Dense Matrix: The standard inter-block edge count matrix used in the Peixoto algorithm. $M[i, j]$ denotes the number of edges from block i to block j. The memory complexity required to store graphs with this structure is $O(V^2)$. Memory accesses and updates are quick. However, the memory complexity renders dense matrices unable to store large graphs within the memory limitations of even large computers.

2) Sparse Matrix: The sparse variant of the matrix specified above. The block graph generally exhibits sparse characteristics. The sparsity offered by the graph can be exploited to store the graph in a sparse matrix. Sparse matrices are fast for read operations, but are slow for insertions, updates, and deletes. Sparse matrices are stored in Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) formats which are the most rigid as compared to the other formats.

3) Hash-map based structures: Hash maps allow for fast look ups of specific edges, $i \to j$. These data structures are fast for insertions, deletions and updates. Hash maps are the most flexible format. However, they can result in wasted memory due to the use of several near empty hash maps, especially with sparse vertices. A hash map of hash maps (nested dictionaries) structure with the outer hash map containing the source vertex, i, and the inner hash map containing the destination vertex j was used for the experiments.

4) Dynamic Graphs: STINGER is a data structure that is designed for high performance analysis on dynamic graphs similar to the dynamic graph problem created by the block graph [22], [23]. STINGER preallocates memory for edges and vertices and edges are added as part of edge blocks. Edge blocks are a small contiguous amount of memory that allows for edges to be added in until it is full, at which it is linked to another edge block. STINGER acts as a linked list of edge blocks. STINGER adds back edges on insertion of a directed edge to make quick lookups to in neighbors. STINGER is an efficient design hybrid with some flexibility and efficient access patterns. Parallel community detection on STINGER is well studied [24]–[26].

5) Relational Databases: We explored storing the block graph in databases like SQLite and PostgreSQL. Databases have efficient indices for Create, Read, Update and Delete (CRUD) operations which map directly to the Insert, Read, Update, and Delete operations on the block graph [27]. A table storing rows with the source vertex, destination vertex, edgecount, and number of blocks was used along with indices over the source, destination and number of block columns for quick lookups. Preliminary experiments revealed that these options are two orders of magnitude slower than native data structure options and do not lend themselves to easy parallelism. The overheads involved with serialization of data and communication between the database and the program overshadow any benefits obtained from efficient indexing. We omit these from the experiments due to these factors. Deeper integration of traditional database internals which avoid overheads would provide higher performance for future work.

C. Parallel implementation

The MCMC based Peixoto algorithm described in Section I performs updates to the inter-block edge count matrix after every accepted proposal in nodal update phase. Efficient parallelism is hard to obtain with interleaved writes occurring to a data structure shared among all the threads. The addition of a lock, which is required to ensure data consistency by eliminating race conditions, creates a bottleneck for writes to the data structure. The inter-block edge count matrix from the previous iteration of the MCMC sweep is used to generate the proposals for each vertex among all the threads. Keeping track of all the accepted proposals among the threads eliminates this bottleneck. The approximate nature of the MCMC algorithm allows for this relaxation while retaining approximate correctness [15]. The key to parallelism is the separation of read operations and write operations between the phases in the algorithm. The parallel phase is read only and a batched write is done after the parallel phase to perform the required insertions, updates and deletes to reconcile the inter-block edge count data structure before the next read only parallel phase. We can exploit this property to use a read optimized data structure for the read only parallel phase and a write optimized data structure for the update phase.

D. Algorithm Cost Analysis

In introducing and analyzing the Peixoto algorithm, overall cost analysis is shown to be $O(nln^2n)$ [11]. For HPC applications it is important to understand the various components of an overall runtime bound because the different operations take different amounts of time on modern computers. We decompose this runtime analysis into read and write components based on operations in the algorithm that access data from the inter-block edge count matrix, and operations that modify this matrix.

$Read$ operations are performed for every proposal examined and $write$ operations that modify the inter-block edge count matrix are only performed for proposals that are accepted. Let the number of proposals per vertex be denoted by N_p and the number of proposals accepted per vertex be denoted by N_e. Let the cost of a read operation be α and the cost of a write operation be β. Cost is measured according to the time or cycles used per operation. The runtime formula is given by

$$\alpha N_p V + \beta N_e V \tag{1}$$

This analysis allows us to study the affect of different datastructures on the runtime of the overall algorithm. And our implementation uses generic code to allow for readily changing out these data structures.

E. Hybrid data structure approach

While separating the computation into read and write phases is known for the purposes of parallel computation, we study how the introduction of a hybrid data structure can be used to improve performance.

Let us assume we are using a read optimized data structure, R and a write optimized data structure, W. Let the cost of a read be denoted by α_R and α_W and cost of writes be β_R and β_W for the read and write optimized data structures respectively. Here, $\alpha_R << \alpha_W$ and $\beta_W << \beta_R$. The hybrid approach requires conversions between R and W during the phases of the algorithm. Let the cost of conversion be γ and number of conversions be N_c.

The cost of using only the read optimized data structure will be $\alpha_R N_p V + \beta_R N_e V$. The cost of using only the write optimized data structure will be $\alpha_W N_p V + \beta_W N_e V$. The cost of using a hybrid data structure will be $\alpha_R N_p V + \beta_W N_e V + \gamma N_c$.

A read optimized data structure and a write optimized data structure such that

$$\begin{aligned} \alpha_R N_p V + \beta_W N_e V + \gamma N_c < min(&\alpha_R N_p V + \beta_R N_e V, \\ &\alpha_W N_p V + \beta_W N_e V) \end{aligned} \tag{2}$$

would make the hybrid version faster than either of the approaches using just the native data structures. We can write (2) as

$$\frac{2\gamma}{V}\frac{N_c}{N_p} < \frac{N_e}{N_p}(\beta_R - \beta_W) + (\alpha_W - \alpha_R) \tag{3}$$

The amount of data stored affects the cost of reads and writes in certain data structures. It could be constant, linear or logarithmic depending on the size of the graph, V. Analysis of these equations shows that the reads α_R and the writes β_W must be constant for arbitrarily large graphs for an optimal choice of data structure.

F. Sparse Matrix Hybrid

We propose the following hybrid data structure based on a sparse matrix to be used as the inter-block edge count data structure in community detection algorithms. Let A be the adjacency matrix of the input graph, $G(V, E)$, be represented as a sparse matrix and C ($C_{ij} = 1$ if v_i in block j, else 0) the block assignment matrix of vertices. C can be implemented

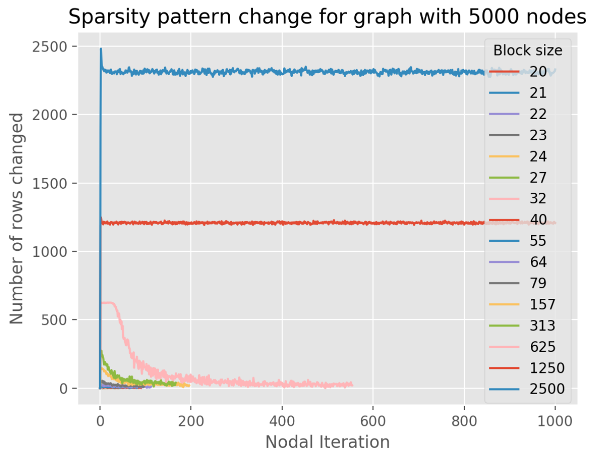

Fig. 1. Number of rows changed with nodal iterations in the nodal phase for a 5000 node graph. Sparsity changes significantly for iterations of sizes 2500 and 1250 with almost all rows touched. As the existing partition becomes more modular, the sparsity changes due to a nodal move become smaller. Each series is the initial number of vertices n.

as an integer array of size V. Let M be the inter-block edge count matrix. M can be computed from C and A as

$$M = C'AC \tag{4}$$

Let Δ be a sparse matrix representing updates to C, such that $C_{new} = C + \Delta$. Therefore, substituting in (4),

$$\begin{aligned} M_{new} &= (C+\Delta)'A(C+\Delta) && (5) \\ &= C'AC + \Delta'AC + C'A\Delta + \Delta'A\Delta && (6) \end{aligned}$$

We can batch updates up to the point where time to compute $(C+\Delta)'A(C+\Delta)$ is less than the time to compute $C'AC + \Delta'AC + C'A\Delta + \Delta'A\Delta$. Reads are performed from the read optimized data structure, the sparse matrix M_{new} computed after the parallel phase using (6). The write optimized data structure, the vector, C, is updated after each parallel phase during the write phase and M is recomputed. Furthermore, symmetry can be exploited in the case of undirected graphs to obtain faster computation of (6). M can be recomputed for streaming graphs by substituting A with $A+D$, where D is sparse matrix representing the updates to the graph $G(V,E)$ to obtain $G'(V',E')$. We use this hybrid data structure in our benchmarks to compare performance with the other data structures presented in Section II-B.

III. Experiments

A. The Julia Programming Language

The baseline algorithm was implemented in the Julia programming language [28]. We chose Julia for the following reasons:

1) The Julia language solves the "two language problem" by offering high performance in a high level language.
2) It supports generic programming and the JIT compiler along with multiple dispatch generates specialized machine code for each set of arguments a method dispatches on. High performance generic programming is achieved in this manner. We are able to write a generic implementation of the partitioning algorithm and specialize on the parts of the algorithm that require specialized operations based on the data structure. This saved development time while yielding high performance code.
3) A mature graph library LightGraphs.jl [29].
4) A high performance high level wrapper to the STINGER library is available in Julia. We have previously shown that this library obtains performance comparable to the STINGER C library [30].
5) Julia has easy to use parallelism constructs which we can utilize to parallelize the algorithm.

B. Input Graph Data Sets

The input graphs used for this work were taken from the sample graphs that were provided by GraphChallenge. These are graphs generated using the stochastic block model for a given number of nodes. The truth partitions for these graphs are available that help us calculate several correctness metrics mentioned in the GraphChallenge benchmark specifications. We use graphs with 50, 100, 500, 1000, 5000 and 20000 nodes to evaluate the partitioning algorithm. We run the algorithm on the static graph with all vertices and edges.

C. Hardware Used

We used the PACE shared university cluster to run the experiments [31]. 64 core machines with 160GB of memory and a runtime limit of 12 hours were available for the experiments.

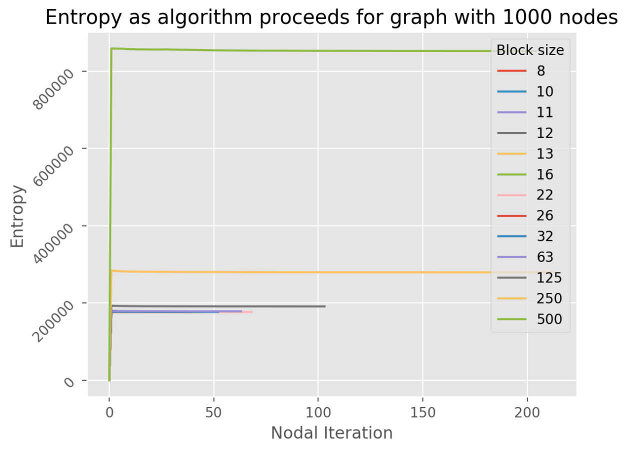

Fig. 2. Entropy of nodal iterations in the nodal phase for a 1000 node graph. The nodal phase does not result in significant changes in entropy. Entropy is shown as overall description length [15].

IV. Results

A. Change in Sparsity Pattern

Since many approaches to high performance algorithms rely on exploiting sparsity, it is important to quantify this sparsity. We measured the change in sparsity for each iteration of the algorithm. This measures the change in the data represented independent of the data structure used to represent it and quantifies how much *churn* the data structure will experience. The maximum number of nodal iterations was capped at 1000. Fig 1 shows the number of rows in which at least one element changed from a zero to a non-zero or a non-zero to a zero (inserts or deletes) for the input graph with 5000 nodes. We see that for the initial larger versions of the agglomerative phase with number of blocks of 2500 and 1250, the nodal update phase did not converge but continues until the maximum number of nodal iterations. We can also notice that for these 2 iterations, the number of rows that change are almost constant and approximate the total number of rows in the block graph. For larger graphs, such phases where almost every row of the sparse matrix changes with every iteration will dominate the runtime. These updates to almost every row of the sparse matrix structure are expensive. This result implies that thus any optimization which relies on touching only modified rows will fail to deliver meaningful speedup for this algorithm.

B. *Change in Entropy*

We measured the entropy of the partition for each iteration of the algorithm as shown in Fig 2. The changes in entropy effected by nodal update iterations are quite insignificant as compared to the changes that occur due to the agglomerative updates. However, reducing the number of nodal iterations runs the risk of not obtaining a good partition. It is hard to find a relationship between the change in entropy and the stopping condition for the MCMC. This requires further study to develop an optimal stopping criteria [32].

C. *Performance*

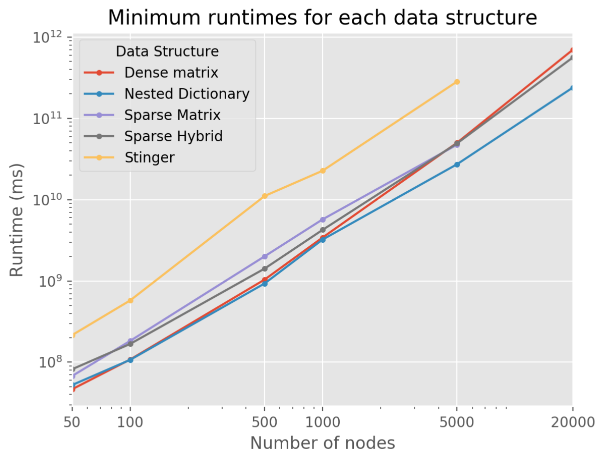

Fig. 3. Run time of each data structure as a function of graph size n. The hybrid data structure is faster than the sparse matrix structure after the crossover point at $n \approx 5000$

Figure 4 shows that the hybrid data structure based on sparse matrices proposed in Section II-F performs better than the sparse matrix implementation and even better than the dense matrix implementation as the the size of the input graph grows. The hybrid data structure exhibits better parallel scaling metrics than the dense matrix or the sparse matrix. The sparse matrix and the STINGER implementations did not complete in the runtime limit of 12 hours to obtain results for the largest graph of 20000 nodes. We can see a crossover point occurring in Figure 3 with the sparse matrix and the hybrid implementations. The hybrid data structure is faster than the sparse matrix implementation, particularly for larger graphs as predicted by equation 2.

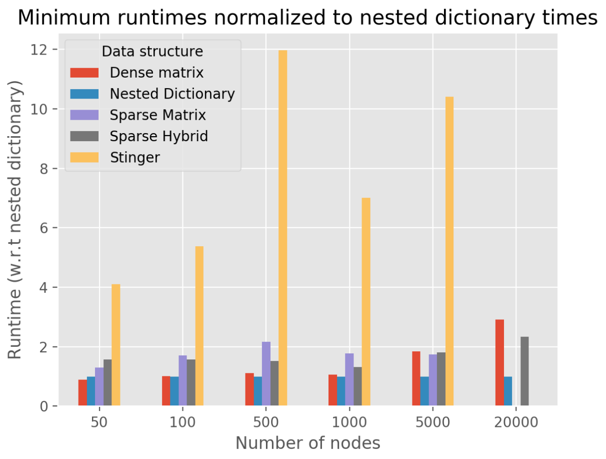

Fig. 4. Run time normalized to nested dictionary performance for each graph size n. Nested dictionary is faster in most cases. Performance of sparse hybrid data structure is better than sparse matrix, as predicted by equation 2

The dictionary based structure outperforms all the other data structures consistently for large graphs. It also exhibits good parallel scaling as seen from Figure 5. STINGER is significantly slower than all the other graph representations used. Having been designed for algorithms such as breadth first search, STINGER is sacrifices single edge access time for faster neighborhood traversals. STINGER saves memory by only storing the edge weight with one direction of an edge thus, the frequent edge weight lookups required by the algorithm lead to a performance penalty with STINGER. The algorithm run time depends mostly on the time for fast single access edge weight lookups subject to a constraint that insertions, deletions and updates are fast enough. This result implies that optimizing for simple, neighborhood oriented algorithms like BFS can lead to data structures with very poor performance for complicated, single edge oriented algorithms like the Peixoto MCMC algorithm.

D. *Memory Usage*

Table I shows that the dense matrix uses more memory when compared to the other data structures as expected. The hybrid implementation uses approximately the same amount of memory as the sparse matrix implementation but is much faster. We see similar numbers for the dense matrix, hybrid and dictionary data structures for the largest graph of 20000 nodes. Dictionary based structures minimize both time and memory costs.

TABLE I

AVERAGE MEMORY ALLOCATED (NORMALIZED TO DENSE MATRIX ALLOCATION) FOR 5000 NODES

Name	Memory Allocated (GB)	Normalized Memory
Dense matrix	1996.7	1
Nested Dictionary	311.704	0.156
Sparse Matrix	662.199	0.332
Hybrid	665.545	0.333
Stinger	1225.696	0.614

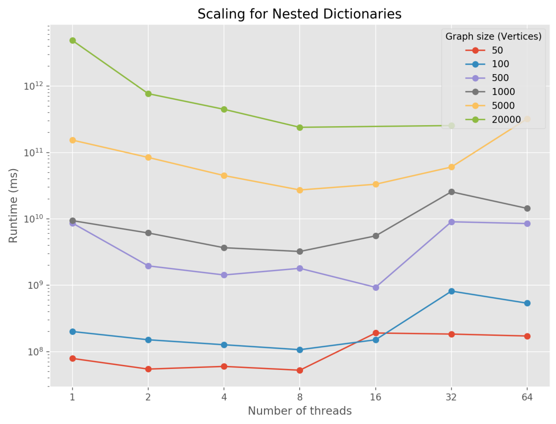

Fig. 5. Strong Scaling: Run time as a function of thread count. Scaling is better for larger values of n where there is more work to be done. Also, hyperthreading (16−64 threads) is not substantially helpful for this problem.

E. Parallel Performance: Strong Scaling

Figure 5 shows the results of strong scaling experiments on shared memory nodes of the PACE cluster. For the larger graphs with 5000 and 20000 nodes, there is strong scaling up to 8 or 16 cores. For smaller graphs there is not enough parallel work to be split among the threads so there is relatively little parallel speedup. The increase in the amount of work to be done will allow the algorithm to take advantage of the parallelism better with increasing graph sizes.

TABLE II

AVERAGE DETECTION QUALITY

Name	Accuracy	Pairwise precision	Pairwise recall
Dense matrix	0.94	1	0.95
Nested Dictionary	0.93	0.99	0.94
Sparse Matrix	0.96	1	0.97
Sparse Hybrid	0.93	1	0.94
Stinger	0.97	1	0.97

F. Community Detection Quality

In order to verify the correctness of the algorithm one can examine the quality metrics of the final block assignments are evaluated using the true partitions of the stochastic block model. We found that the number of nodal iterations and fraction of change threshold are two important parameters for convergence. The parameters used in the experiments gave good assignments as evidenced by the high average accuracy, pairwise precision and pairwise recall scores shown in Table II. There were a few runs where the algorithm did not converge to a good solution due to the randomized nature of the algorithm, but overall it returned very good partitions. Due to the randomized nature of the algorithm, convergence is not guaranteed and we excluded runs that failed to converge to good solutions. There is an inherent trade-off between speed and quality and we studied algorithmic performance when operating at high quality. Future work should rigorously study this trade-off.

V. CONCLUSION

This paper studies Peixoto's algorithm for stochastic block model inference where algorithm performance with various data structures representing the inter-block edge count matrix allows us to draw conclusions about the algorithm. We provide a novel analysis of the algorithm with respect to inter-block edge count matrix modifications, general method for evaluating such data structures, and some limitations on potential optimizations for this algorithm.

The cost analysis shows that using a read optimized data structure with constant per edge read times and nearly constant per edge write times, such as nested dictionaries, is the best solution for this algorithm due to the high ratio of read operations to write operations. The faster performance of dictionary implementations over complex solutions such as the hybrid data structure and dynamic graph proves that simple data structures that satisfy the read and write access time constraints can prove to be better than complex data structures. Hybrid data structures are well suited to provide accelerated performance by utilizing read optimized and write optimized forms of the inter-block edge count matrix. Our theoretical analysis of hybrid data structures provides a formula to evaluate the acceleration provided by a hybrid data structure and accurately identifies the observed acceleration of compressed sparse matrix performance when used in a hybrid pair.

Inherent to Peixoto's algorithm are several limitations on high performance data structures. Reassigning vertices between communities causes significant changes in the sparsity pattern of the inter-block edge count matrix. Optimizations based on sparsity from compressed sparse matrices or dynamic graph data structures will not accelerate this algorithm due to the churn of stored graph. Empirical evidence shows that entropy decrease is neither a necessary nor sufficient condition for convergence and more analysis is needed to find a robust stopping criterion for nodal iterations. Without reducing the number of nodal moves in the early phases, dynamic sparse graph data structures cannot provide significant speed up.

Dynamic graph data structures focus on optimizing insertion performance at the expense of individual edge access performance. Transitioning from a vertex oriented computation to a n edge oriented computation would allow for optimizations from dynamic graph data structures to have more impact.

VI. Acknowledgments

The authors would like to thank Geoffrey Sanders, David Bader, Eric Hein, and Edward Kao for their feedback on drafts of this work, along with the anonymous reviewers. We would also like to acknowledge the research computing infrastructure resources and services provided by the Partnership for an Advanced Computing Environment (PACE) at the Georgia Institute of Technology.

References

[1] Mark EJ Newman. Detecting community structure in networks. *The European Physical Journal B*, 38(2):321–330, 2004.

[2] Stefano Battiston, James B Glattfelder, Diego Garlaschelli, Fabrizio Lillo, and Guido Caldarelli. The structure of financial networks. In *Network Science*, pages 131–163. Springer, 2010.

[3] Michelle Girvan and Mark EJ Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 99(12):7821–7826, 2002.

[4] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[5] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[6] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: a comparative analysis. *Physical review E*, 80(5):056117, 2009.

[7] Martin Rosvall and Carl T Bergstrom. Multilevel compression of random walks on networks reveals hierarchical organization in large integrated systems. *PloS one*, 6(4):e18209, 2011.

[8] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[9] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[10] Emmanuel Abbe. Community detection and stochastic block models: recent developments. *arXiv preprint arXiv:1703.10146*, 2017.

[11] Tiago P Peixoto. Efficient monte carlo and greedy heuristic for the inference of stochastic block models. *Physical Review E*, 89(1):012804, 2014.

[12] Tiago P Peixoto. Parsimonious module inference in large networks. *Physical review letters*, 110(14):148701, 2013.

[13] Tiago P Peixoto. Entropy of stochastic blockmodel ensembles. *Physical Review E*, 85(5):056122, 2012.

[14] Brian Karrer and Mark EJ Newman. Stochastic blockmodels and community structure in networks. *Physical Review E*, 83(1):016107, 2011.

[15] Edward Kao, Vijay Gadepally, Michael Hurley, Michael Jones, Jeremy Kepner, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Siddharth Samsi, William Song, et al. Streaming graph challenge: Stochastic block partition. *arXiv preprint arXiv:1708.07883*, 2017.

[16] E Isaacson, William H Press, Brian P Flannery, Saul A Teukolsky, and William T Vetterling. Numerical Recipes: The Art of Scientific Computing. *SIAM Review*, 30(2):331–332, jun 1988.

[17] Mark EJ Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[18] Jordi Duch and Alex Arenas. Community detection in complex networks using extremal optimization. *Physical review E*, 72(2):027104, 2005.

[19] Zhongying Zhao, Shengzhong Feng, Qiang Wang, Joshua Zhexue Huang, Graham J Williams, and Jianping Fan. Topic oriented community detection through social objects and link analysis in social networks. *Knowledge-Based Systems*, 26:164–173, 2012.

[20] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. Community detection in social media. *Data Mining and Knowledge Discovery*, 24(3):515–554, 2012.

[21] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[22] David A Bader, Jonathan Berry, Adam Amos-Binks, Daniel Chavarría-Miranda, Charles Hastings, Kamesh Madduri, and Steven C Poulos. STINGER: Spatio-temporal interaction networks and graphs (sting) extensible representation. *Georgia Institute of Technology, Tech. Rep*, 2009.

[23] David Ediger, Robert McColl, Jason Riedy, and David A Bader. STINGER: High performance data structure for streaming graphs. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–5. IEEE, 2012.

[24] E Jason Riedy, Henning Meyerhenke, David Ediger, and David A Bader. Parallel community detection for massive graphs. In *International Conference on Parallel Processing and Applied Mathematics (PPAM)*, pages 286–296. Springer, 2011.

[25] Jason Riedy, David A Bader, and Henning Meyerhenke. Scalable multi-threaded community detection in social networks. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1619–1628. IEEE, 2012.

[26] Jason Riedy and David A Bader. Multithreaded community monitoring for massive streaming graph data. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1646–1655. IEEE, 2013.

[27] Ramez Elmasri and Shamkant Navathe. *Fundamentals of database systems*. Addison-Wesley Publishing Company, 2010.

[28] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[29] Seth Bromberger, James Fairbanks, and other contributors. JuliaGraphs/LightGraphs.jl: LightGraphs v0.13.1, Sep 2017.

[30] Rohit Varkey Thankachan, Eric R Hein, Brian P Swenson, and James P Fairbanks. Integrating productivity-oriented programming languages with high-performance data structures. In *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pages 1–8. IEEE, 2017.

[31] PACE. *Partnership for an Advanced Computing Environment (PACE)*, 2017.

[32] Eisha Nathan, Geoffrey Sanders, James P Fairbanks, David A Bader, et al. Graph ranking guarantees for numerical approximations to katz centrality. *Procedia Computer Science*, 108:68–78, 2017.