Fast Stochastic Block Partition for Streaming Graphs

Ahsen J. Uppal, and H. Howie Huang

The George Washington University

***Abstract*—The graph partition problem continues to be challenging, particularly for streaming graph data. Although optimal graph partitioning is NP-hard, stochastic methods can provide approximate solutions in reasonable time. However, such methods are optimized for static, not dynamic graph data. In this paper, we describe a new efficient algorithm we have developed for stochastic block partitioning on time-varying, streaming graph data. Our algorithm is a refinement of the baseline algorithm of the IEEE HPEC Graph Challenge [1]. Our incremental algorithm efficiently updates its previous internal state as new pieces are streamed in, and generates a complete partition at every time step.**

Compared to the naive baseline which performs a complete partitioning from scratch at every time step, our algorithm offers speedups between 1.96x for N=500 and 3.56x for N=20k overall, for a graph streamed over 10 parts, with similar accuracy. At the margin, the speedup in processing time for additional streaming pieces over the baseline is between 7.1x for N=500 to 25.1x for N=20k.

I. Introduction

Graph data has come to play a critical role in a diverse array of applications, particularly when looking for hidden or complex relationship structures. Many theoretical algorithms assume that graph data is fixed and known at one time. But in real-world applications, data sizes are often too large to store in their entirety, and relationships must be identified with low latency to permit rapid reaction. These stringent constraints motivate the development of efficient stream-oriented algorithms for graph processing, which can efficiently generate incremental results as fresh data is streamed in.

The goal of graph partitioning (also called community detection or graph clustering) is to discover the higher-order relationship community structure of nodes in a graph. Informally, nodes that form dense connections with each other can be considered as part of the same community (or block). But the computational complexity of many graph optimization algorithms, including optimal graph partitioning is NP-hard. Fortunately, the development of approximation algorithms, including stochastic block partition has been critical for efficiently solving the graph partitioning problem for static graphs.

But using stochastic block partition is a challenging case for streaming data, because the number of communities is not only unknown ahead of time, but may also vary over time. The arrival of fresh streaming data may upset previous results. Furthermore, there are no a priori clues about the membership of the nodes. Finally, the *type* of streamed data also varies. A dynamic graph may have edges added (or deleted), or even new nodes added or deleted. Edge weights may also change.

A streaming partitioning algorithm should efficiently merge fresh data into its existing algorithmic state and generate up-to-date results given the current state of the graph that are as accurate as performing a full partitioning from scratch. Our approach accomplishes both of these goals.

Note that our focus in this work is on improving the algorithmic performance of stochastic block partition – not the absolute best performance of community detection. As such, our high-level Python streaming implementation is compared to a baseline which uses our static implementation to fully partition a static graph after every subgraph piece is streamed in.

II. Background

The static algorithm for stochastic block partition which forms the GraphChallenge baseline uses a generative statistical models based on work by Peixoto [2][3][4] building on Karrer and Newman[5].

The algorithm has two challenges: the optimal number of blocks is not known a priori, and the assignment of each node to a block is not known. The static algorithm uses an entropy measurement which measures the goodness of a partition, to address these challenges. It proceeds by picking a number of target blocks, finding an optimal partitioning for that number, and computing the resulting entropy. It can then make a comparison against the partitioning for a different target number of blocks.

In particular, the baseline algorithm first proceeds to find optimal partitions and entropies for partition sizes B of $\frac{N}{2}$, $\frac{N}{4}$, $\frac{N}{8}$, ..., until the partition with the minimum entropy is bracketed between two partition sizes. It then switches to a Golden section search[6] to find the final partition and entropy.

To target a given number of blocks, at each iteration the static algorithm proceeds down from a larger partition size to a smaller one. It does each such iteration in two distinct program phases. First, it repeatedly does an agglomerative merge of two existing blocks based on greedily picking the merge with the lowest resulting entropy. Each agglomerative merge takes two communities and puts them together as one, resulting in one fewer number of blocks. When the target number of blocks is reached, the algorithm performs nodal moves that can reassign a vertex from one block to another. These nodal moves are accepted or rejected with a probability proportional to the resulting change in entropy. Movements that result in large decreases in entropy are likely to be accepted and movements that do not are unlikely to be accepted. When the overall change in entropy stabilizes for a particular partition size, the algorithm stops movements for that partition size, and proceeds to the next target block number. These Markov Chain Monte Carlo (MCMC) nodal movements are done in the manner of Metropolis-Hastings[7][8].

We previously found that the vast majority of the algorithm is spent performing nodal updates, and focused on performing these nodal updates in parallel, with low latency, and at large scale.

III. Approach

In our previous work[9] we focused on parallelization and scalability of the stochastic block partition algorithm. In this work we extend our previous approach to adapt it to streaming graph data.

We adapted our fast parallel Python implementation to efficiently process streaming data. Since our goal here is an efficient streaming algorithm, we focused on high-level algorithmic changes. This means comparing a new streaming algorithm with a similar static algorithm, instead of maximizing raw partitioning performance on static graphs. The rationale here is that system implementation performance gains would roughly equally benefit both the naive streaming algorithm, as well as our new streaming algorithm. Thus we did not focus on using the largest possible parallel implementation, most efficient data layout, or leveraging hardware accelerators. In addition, we did not use a divide-and-conquer style algorithm to speed up the computations for large graphs.

A. Streaming Graph Data

The streaming graph data provided in the GraphChallenge comes in two variants: emerging edges, and snowball sampling. In emerging edge streaming, the number of nodes is fixed and edges are randomly sampled and added to the graph over time. For our current work, we focused on emerging edge streaming.

B. Streaming Baseline

The streaming baseline we used for comparison uses our fast static parallel algorithm, run from scratch to completion after every streamed piece updates the graph. This approach is somewhat naive, but is simple and easy to understand. It is particularly wasteful because it discards all of the work done during the early iterations of the algorithm when the number of target blocks is repeatedly cut in half.

The naive algorithm is shown in Listing 1 in Python-like code.

```
def naive_stream():
  for part in range(parts):
    dG=load_graph_part(part)
    G=update_graph(G, dG)
     run_static_partition(G)
```

Listing 1. Naive Streaming Algorithm

C. Saving and Re-using Algorithm State

Our starting intuition for an efficient incremental streaming algorithm is that a good partition found during a previous time serves as a good starting point for partitioning at a the next time interval. It is also desirable to output a complete partitioning as each piece is streamed in.

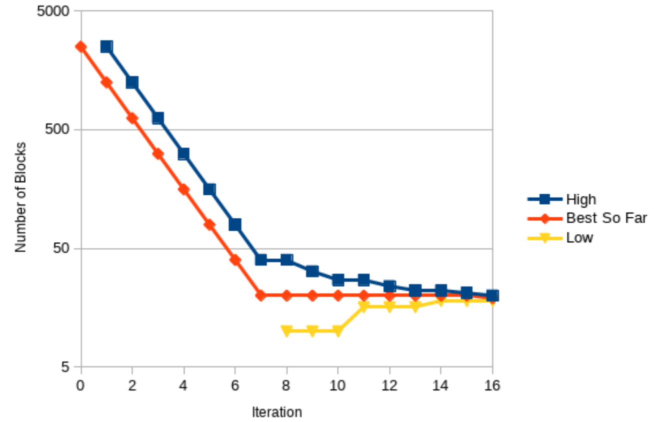

Fig. 1. Bracketing of the optimal number of blocks during each iteration of static partition. For a naive implementation of streaming partition, this full bracket search occurs for every time increment of the dynamic graph.

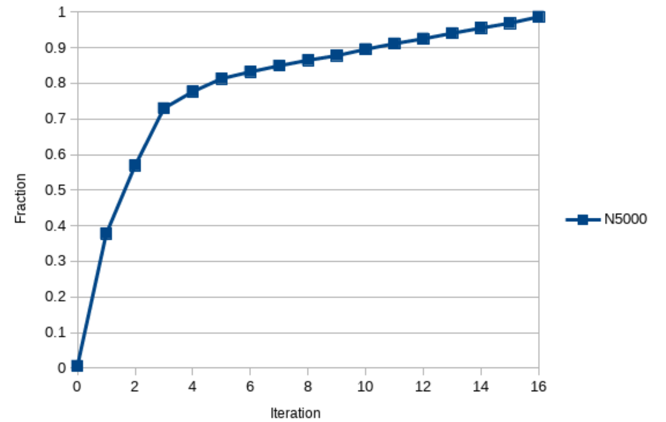

Fig. 2. Cumulative runtime of each iteration of static partitioning as a fraction of the whole compute time. Early iterations take the majority of the time.

Thus our basic approach is to save the state of the algorithm at some point, during its partitioning of each piece to completion. When the next streamed piece arrives, that previous saved state is updated based on the information contained in the new piece. The algorithm is then resumed from this updated state and once again run to completion, with the state of the algorithm once again being saved for the next piece.

The initial iterations of the partitioning algorithm to reduce the block counts dominate the runtime of the partitioning. By re-using the expensive portions of the algorithm state, we aim to greatly reduce the marginal cost of streaming processing.

The cost of these expensive initial iterations is illustrated in Figure 1 and Figure 2 for a graph size of N=5000. The bracketing search for the optimal block count number is shown in Figure 1. Initially the target number of blocks is cut in half until the minimum entropy block count is bracketed in iteration 8 and the fine-grained Golden section search begins. However the cumulative fraction of the runtime in the first iteration (which reduces the block count to B=2500) consumes 37.7% of the overall partitioning runtime, and by iteration 8, the cumulative runtime represents 86.5% of the whole.

But there are complications with this save and reuse approach. The first tricky parts is to know when precisely the algorithm state should be saved. When just a small fraction of the graph has been streamed in, the accuracy of the ultimate resulting communities may be poor simply because there is not enough graph data yet. If the state is saved at a late iteration of the bracketing search for this graph piece, then the quality of the communities at *subsequent* iterations will also be adversely impacted.

We devised a heuristic that works well to determine save points during graph partitioning. We save off a snapshot when either the Golden ratio bracket is established *or* the number of blocks drops below a minimum threshold. This minimum threshold is cut in half after every piece is streamed in, and mostly ensures that enough graph data is present to form stable communities as the first few pieces are streamed in.

D. Updating Algorithm State

The other tricky part is to how to efficiently update previously-saved algorithm state with new streamed data. When a new piece of a dynamic graph has been streamed in, the saved internal state variables of the algorithm will be out-of-date and must be adjusted for proper operation. If the previous saved state was during the Golden section search, three copies of partition state (corresponding to the low, best, and high block numbers) must be updated.

The particularly algorithm state that must be updated with new graph data is, of course, the overall entropy computation. But all of the block degree counters, and the interblock edge count matrix must be adjusted since these track cumulative edge counts between blocks during partitioning.

If the structure of communities in a streaming graph changes radically (i.e. the number of communities increases or decreases rapidly), then the Golden section search will no longer bracket the optimal entropy. We handle these cases by re-computing the overall entropy for each number of blocks in the saved bracket, and re-ordering the brackets a. If the Golden section criterion is no longer satisfies, the static algorithm will still proceed from the highest block number in the saved bracket. This method can handle some increases in the number of communities, but only up to the largest number in the saved block state. We plan to address this limitation in future work by restarting partitioning from a higher number when this condition is detected.

Putting these ideas together, the pseudo-code for our algorithm is shown in Listing 2 in Python-like code.

```
def efficient_stream():
  G=None
  st=None
  min_num_blocks=N/2
  for part in range(parts):
    dG=load_graph_part(part)
    G=update_graph(G, dG)
    st=update_state(st, dG)
    st=run_static_partition(G, st,
               stop=1, min_num_blocks)
    # Note: no saved state second time
    run_static_partition(G, st, stop=0)
    min_num_blocks /= 2
  return

def update_state(st, dG):
  for from_idx,to_idx in dG.edges:
    # There may be up to three entries
    # to adjust due to the Golden
    # bracket search.
    for j in [0,1,2]:
      if st.partition[j]:
        b = st.partition[j]
        M = st.interblock_edge_count[j]
        b_out = st.block_degrees_out[j]
        b_in = st.block_degrees_in[j]
        b_all = st.block_degrees[j]

        M[b[from_idx], b[to_idx]] += 1
        b_out[b[from_idx]] += 1
        b_in[b[to_idx]] += 1
        b_all[b[from_idx]] += 1
        b_all[b[to_idx]] += 1

  for j in [0,1,2]:
    if st.partition[j]:
      # Assign M, b_out, b_in as before.
      st.overall_entropy[j] =
        compute_entropy(M, b_out, b_in,
                  num_blocks, N, E)

  # If the updated entropies no longer
  # bracket the minimum then reorder entries.
  return
```

Listing 2. Our Streaming Algorithm

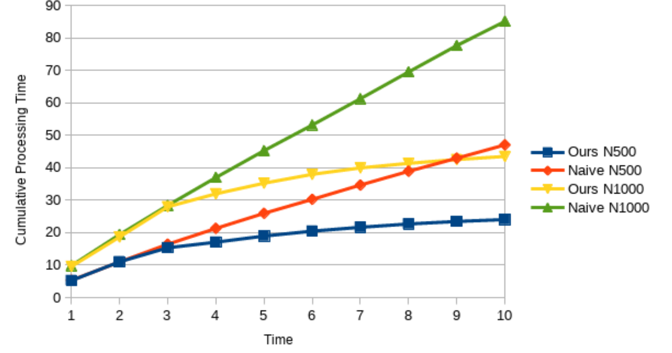

Fig. 3. Cumulative processing time of ours and naive baseline for small graphs.

E. Implementation

For our implementation, we started with our fast parallel stochastic block partition implementation written in Python. Our baseline for comparison is our implementation run to completion after every piece of a graph is streamed in. We modified this so that the algorithm can stop as it processes a graph to completion. It stops, stores a snapshot of the state, then resumes and finishes processing the current graph (but not further modifying the saved off state).

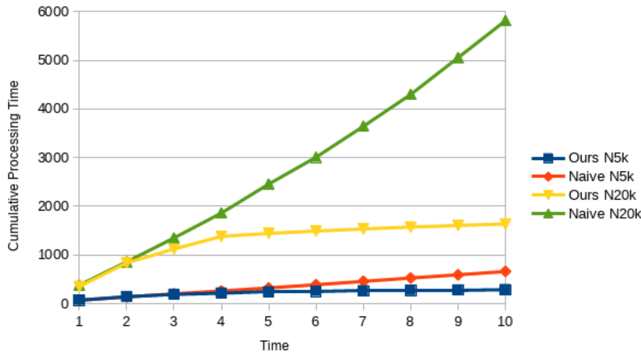

Fig. 4. Cumulative processing time of ours and naive baseline for medium graphs.

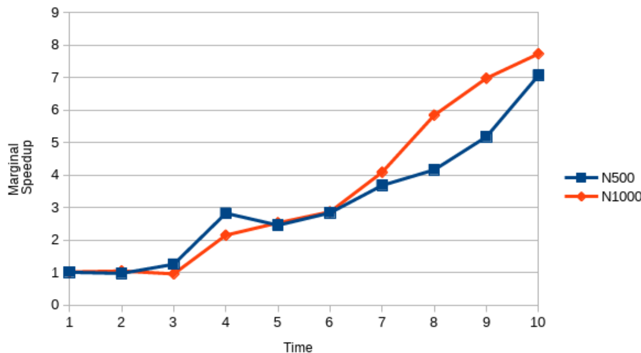

Fig. 5. Marginal speedup over time of ours against naive baseline for small graphs.

One major advantage of algorithmic development in Python is the ease of modification, particularly for high-level operations relative to C++. Internally the implementation uses NumPy[10] for low-level array operations, and we rely on Python multiprocessing with shared memory for multi-core operation.

IV. Evaluation

Our streaming partitioning experiments use the baseline datasets from GraphChallenge. Our single-node test machine is an Intel Xeon E5-2683 server with 56 cores and 512GB RAM.

We ran experiments on streaming data at graph sizes of N=(500, 1k, 5k, 20k) and evaluated both the performance and accuracy of our method. The GraphChallenge streaming datasets take a graph and divide it into 10 streamed pieces.

A. Performance

One straightforward measure of streaming data performance is the amount of time used to perform partitioning. We instrumented our code to measure just the time spent during computation, ignoring overheads such as graph load time from disk. We measured the overall cumulative times as seen in Figure 3 and Figure 4. Here the overall compute times and the growth rates are clearly visible. The naive approach has a growth in compute time that is linear, with each part of

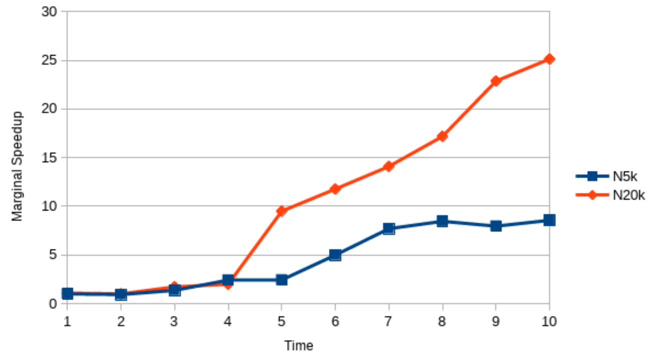

Fig. 6. Marginal speedup over time of ours against naive baseline for small graphs.

TABLE I. Speedups

N Vertices	Speedup Marginal	Speedup Overall
500	7.075	1.96
1000	7.736	1.96
5000	8.553	2.31
20000	25.14	3.56

the streaming graph takes nearly the same amount of time to run. In fact, the growth is slightly super-linear as seen in the N=20k growth because there are more edges in the graph over time. In contrast, our fast incremental algorithm grows sub-linearly in overall compute time. By re-using program state from approximately where the Golden ratio bracketing starts, our algorithm can run much faster overall.

It is also useful to look at the marginal speedup of our algorithm compared to the baseline in Figure 5 and Figure 6. This compares the compute time taken to process each additional piece of the graph stream. We obtained marginal speedups of 1.96x to 25.14x as summarized in Table 1. The speedups obtained improve with increasing graph size indicating good scalability to our approach.

B. Accuracy

We also measured the correctness of our approach using pairwise precision and recall metrics [11] over time. These are shown in Figure 7, Figure 9, Figure 8, and Figure 10. We see that the precision quickly converges to 1.0 over time for our algorithm, just as for the naive baseline. This indicates that the algorithm rapidly performs correct partitioning once enough graph data has been streamed in. We see a similar effect for the recall over time.

V. Related Work

Previous GraphChallenge works haves used a shared memory Louvain implementation [12] and spectral clustering methods [13] to achieve large speedups in partitioning performance that obtain large speedups over the baseline stochastic block partition algorithm. Our work differs in that our focus is narrowly on improving the *streaming* performance of the stochastic graph partition baseline algorithm. One advantage of the stochastic approach is that more complex rules that govern the goodness of communities in different application domains can be easily adapted to a stochastic method.

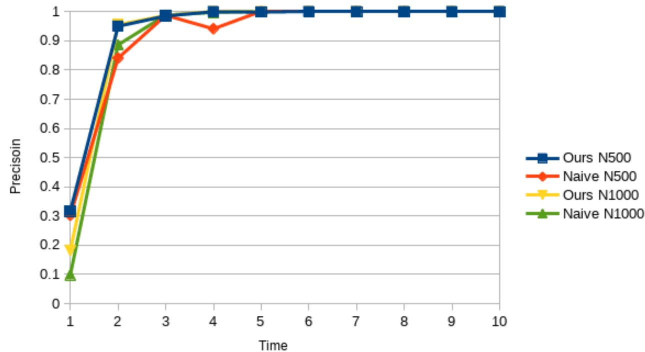

Fig. 7. Precision over time for ours and naive baseline. Convergence rates are very similar.

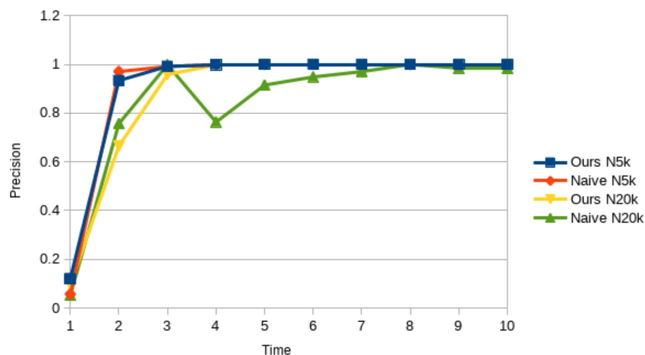

Fig. 8. Precision over time for ours and baseline. Convergence rates are very similar.

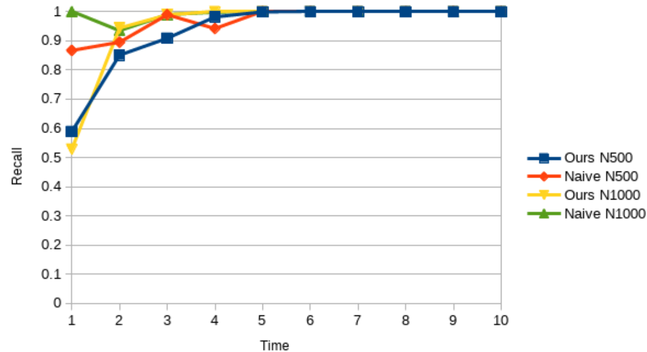

Fig. 9. Recall over time for ours and naive baseline.

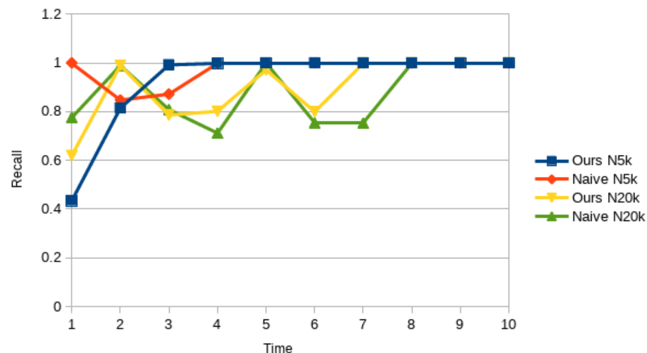

Fig. 10. Recall over time for ours and baseline.

VI. Future Work

Our preliminary work in this area suggests several potentially fruitful areas for future work. Most prominently, our current multi-process solution can be combined with our previous distributed static graph partitioning implementation. With some adaptions, our streaming solution can also be extended to support snowball streaming. Handling edge deletions is more challenging, possibly requiring splitting existing blocks and increasing the number of blocks over time. We also plan to handle radical changes in community structure by storing more entropy and partition state, and backtracking further when the optimal number of blocks increases suddenly. Finally, we plan to investigate leveraging the performance benefits of the Louvain and spectral clustering approaches to seed the stochastic block partition.

VII. Conclusion

We have described our new stochastic block partition for streaming graphs. We have developed a prototype that shows excellent performance gains over the baseline, especially at the margin.

In the future, we would like to further develop and enhance our algorithm, particularly to improve computational performance, handle rapidly changing communities, and extend to a distributed streaming solution.

VIII. Acknowledgment

This work was supported in part by The National Science Foundation CAREER award 1350766 and grants 1618706 and 1717774.

References

[1] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, D. Staheli, and S. Smith, "Streaming graph challenge: Stochastic block partition," 2017.

[2] T. P. Peixoto, "Entropy of stochastic blockmodel ensembles," *Physical Review E*, vol. 85, no. 5, p. 056122, 2012.

[3] ——, "Parsimonious module inference in large networks," *Physical review letters*, vol. 110, no. 14, p. 148701, 2013.

[4] ——, "Efficient monte carlo and greedy heuristic for the inference of stochastic block models," *Physical Review E*, vol. 89, no. 1, p. 012804, 2014.

[5] B. Karrer and M. E. Newman, "Stochastic blockmodels and community structure in networks," *Physical review E*, vol. 83, no. 1, p. 016107, 2011.

[6] J. Kiefer, "Sequential minimax search for a maximum," *Proceedings of the American mathematical society*, vol. 4, no. 3, pp. 502–506, 1953.

[7] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[8] W. K. Hastings, "Monte carlo sampling methods using markov chains and their applications," *Biometrika*, vol. 57, no. 1, pp. 97–109, 1970.

[9] A. J. Uppal, G. Swope, and H. H. Huang, "Scalable stochastic block partition," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–5.

[10] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[11] A. Banerjee, C. Krumpelman, J. Ghosh, S. Basu, and R. J. Mooney, "Model-based overlapping clustering," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. ACM, 2005, pp. 532–537.

[12] M. Halappanavar, H. Lu, A. Kalyanaraman, and A. Tumeo, "Scalable static and dynamic community detection using grappolo," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–6.

[13] D. Zhuzhunashvili and A. Knyazev, "Preconditioned spectral clustering for stochastic block partition streaming graph challenge (preliminary version at arxiv.)," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*. IEEE, 2017, pp. 1–6.