Faster Stochastic Block Partition Using Aggressive Initial Merging, Compressed Representation, and Parallelism Control

Ahsen J. Uppal[1], Jaeseok Choi[1], Thomas B. Rolinger[2], and H. Howie Huang[1]

[1] The George Washington University [2]The University of Maryland, College Park

2021 IEEE High Performance Extreme Computing Conference (HPEC) | 978-1-6654-2369-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/HPEC49654.2021.9622836

***Abstract*—The community detection problem continues to be challenging, particularly for large graph data. Although optimal graph partitioning is NP-hard, stochastic methods, such as in the IEEE HPEC GraphChallenge, can provide good approximate solutions in reasonable time. But the scalability with increasing graph size of such solutions remains a challenge. In this work, we describe three new techniques to speed up the stochastic block partition algorithm. The first technique relies on reducing the initial number of communities via aggressive agglomerative merging (a portion of the algorithm with high parallel scalability) to quickly reduce the amount of data that must be processed, resulting in an independent speedup of 1.85x for a 200k node graph. Our second technique uses a novel compressed data structure to store the main bookkeeping information of the algorithm. Our compressed representation allows the processing of large graphs that would otherwise be impossible due to memory constraints, and has a speedup of up to 1.19x over our uncompressed baseline representation. The third technique carefully manages the amount of parallelism during different phases of the algorithm. Compared to our best baseline configuration using a fixed number of threads, this technique yields an independent speedup of 2.26x for a 200k node graph. Combined together, our techniques result in speedups of 3.78x for a 50k node graph, 4.71x for a 100k node graph, and 5.13x for a 200k node graph over our previous best parallel algorithm.**

***Index Terms*–community detection, GraphChallenge, stochastic block partitioning.**

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 and activities. But performance and scalability remain challenging issues because the computational complexity of many traditional algorithms, including graph partitioning (also known as community detection) is NP-hard. For graph partitioning, the development of approximation algorithms [1], [2] has been critical in finding feasible solutions for real-world datasets. Unfortunately, there are major scalability challenges even for approximation algorithms. These challenges are made even more difficult for graph partitioning when the number of communities is not known a priori and there are no a priori cues about the membership of some of the nodes.

Previous IEEE HPEC GraphChallenge[1] works have used a shared memory Louvain implementation [3] and spectral clustering methods [4] to achieve substantial speedups in partitioning performance over the baseline stochastic block partition algorithm. In this work, we focus on improving the algorithmic performance of stochastic block partition – not the absolute best performance of community detection. To this end, we describe three techniques to address different portions of the algorithm that yield individual speedups of up to 1.85x, 1.19x, and 2.26x, respectively, and a combined speedup up to 5.13x over our previous best parallel implementation [5].

The contributions of our work are as follows:

- The design and implementation of three optimization techniques for stochastic block partitioning in Python that (1) reduce the initial number of communities via aggressive agglomerative merging, (2) utilize a novel compressed data structure that maintains the bookkeeping of the algorithm, and (3) carefully manage the amount of parallelism during different phases of the algorithm.
- A performance study that evaluates the performance of the three optimization techniques against a parallel baseline implementation. Our results show that speedups as large as 1.85x, 1.19x, and 2.26x are possible for each optimization independently. When all optimizations are combined together, we achieve a total speedup of 5.13x for a 200k node graph.

The rest of the paper is outlined as follows. Section II provides an overview of the stochastic block partition algorithm. Details of our three optimization techniques are provided in Section III. Section IV presents a performance evaluation of the optimization techniques. Finally, concluding remarks and future work are provided in Section V.

II. Background

The stochastic block partition algorithm, which forms the GraphChallenge baseline, uses a generative statistical model based on work by Peixoto [6], [7], [8] and builds on the work from Karrer and Newman [9]. The algorithm has two challenges: the optimal number of blocks is not known a priori, and neither is the assignment of each node to a block. To overcome these challenges, the static algorithm uses an entropy measurement function which measures the quality of a partition for a given number of blocks. Once it finds the optimal partitioning at a particular number of target blocks and the associated entropy, it can compare that entropy against future partitions at a different number of target blocks.

[1]https://graphchallenge.mit.edu/

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

In particular, the block phases of the algorithm work as follows, where N refers to the number of nodes in the graph. It first finds optimal partitions and entropies for partition sizes $\frac{N}{2}$, $\frac{N}{4}$, $\frac{N}{8}$, ..., until a valley is found where further reductions in block size no longer produce decreases in entropy. When this minimum entropy is bracketed between two partition sizes, the algorithm switches to a Golden section search [10] to find the final partition and entropy. To find the optimal partition for a given number of blocks, the algorithm goes from a larger partition size to a smaller one. It does this using two distinct program operations – *agglomerative merging* and *nodal movements*.

During an agglomerative merge, two existing candidate blocks are found by 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. Once the algorithm reaches the target number of blocks, it switches to performing 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. The overall algorithm proceeds in this manner, alternating between agglomerative merges and nodal movements at each block phase. When the overall change in entropy stabilizes for a particular number of blocks, the algorithm stops movements for that partition size and goes to the next target block number. This technique is based on a Markov Chain Monte Carlo (MCMC) method that uses the Metropolis-Hastings algorithm [11], [12]. The overall complexity of the baseline algorithm is $O(E \log^2 E)$ [13].

One advantage of stochastic approaches is that more complex rules that govern the goodness of communities in different application domains can be easily adapted to a stochastic method. Other recent work shows the advantage of using sampling compared to modularity minimization [14]. Our work is complementary to this approach since the core data structure and algorithmic operations remain the same.

III. Optimizations

In our previous work, we focused on initial efforts on parallelization, scalability, and efficient streaming of the stochastic block partition algorithm [15], [5]. In this work, we extend our previous approach to adapt it for efficiently processing much larger graph data. We do this via aggressive initial merging, use of a novel compressed data structure, and careful parallelism control. In the following sections, we describe each of these optimization techniques in detail. We implement each of these optimizations in Python, which is the language of the baseline GraphChallenge implementation.

We will refer to graphs as described in Table I when presenting results that motivate our optimizations. These graphs are the baseline datasets from the GraphChallenge, supplemented by larger graphs we synthesized using the generator in the GraphChallenge repository. Any performance results mentioned in the following sections are from executions on a single-node workstation with an Intel Xeon Gold 6126 CPU at 2.60 GHz with 24 cores, 48 threads, and 1480 GB of RAM.

TABLE I. Graph Datasets

$\|V\|$	$\|E\|$	Density	Blocks	Baseline Runtime(s)
500	9,384	3.8e−2	8	2.42
1k	20,135	2e−2	11	4.94
5k	101,973	4.1e−3	19	38.79
20k	408,778	1e−3	32	565.4
50k	1,018,039	4.1e−4	44	3,575
100k	2,037,415	2e−4	56	15,498
200k	4,064,602	1e−4	71	70,970

A. Aggressive Initial Merging

The agglomerative merge portion of the referenced stochastic algorithm decomposes cleanly into a parallel implementation. Each worker picks a subset of blocks and finds the best merge candidate for every block in that subset. The main thread then picks the top-k merge candidates and carries out those merges. In our previous work [15], we found that the vast majority of the algorithm runtime is spent performing nodal updates. Because of the low proportion of time spent in merging, we reasoned that the gains from parallelization are limited by Amdahl's law. We therefore previously focused our efforts on optimizing nodal updates in parallel, with low latency, and at a large scale. But that development was done with relatively small graph sizes and did not consider the interplay between merging and nodal movements. Optimizations to the merge phases of the algorithm are *not* independent and affect the performance of the subsequent nodal movement phase.

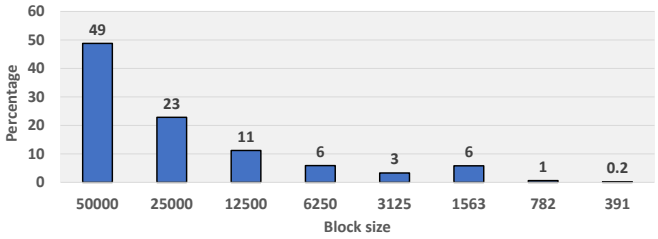

Fig. 1. Percentage of the entire program's runtime spent in each block phase as the program advances and block size decreases. Input is a 50k node graph. Early phases dominate the runtime of the program.

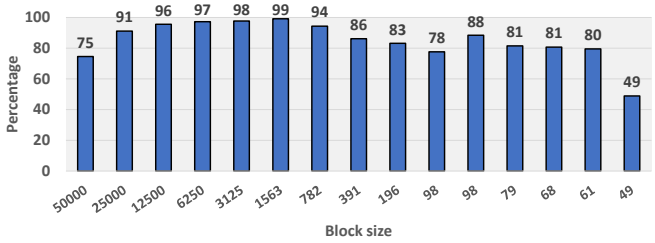

Fig. 2. Percentage of time spent in nodal movement during each block phase as the program advances and block size decreases. Input is a 50k node graph. Nodal movement dominates the runtime compared to agglomerative merge.

While nodal movements take the largest fraction of time in our parallel implementation, merging can help because merging communities greatly reduces the amount of work that has to be done during subsequent nodal movement phases. In our previous work on parallelization, we found that using 12 worker threads for parallel nodal movements and parallel agglomerative merges yielded the best speedups over the serial

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

case on our test machine.

We profiled our baseline implementation using this 12 fixed thread configuration on a 50k node graph, and found that the time spent during the very first round – an agglomerative merge block reduction from 50k blocks to 25k blocks, followed by nodal movements – constitutes 49% of the total program runtime. In a similar vein, the percentage of time spent during the next rounds is 23% at 25k blocks, and 11% at 12.5k blocks. As shown in Figure 1, these early block phases dominate the runtime of the program.

We also found that the time spent in nodal movements dominates compared to agglomerative merges. The percentage of time spent in nodal movements compared to agglomerative merges during each block phase is shown in Figure 2. This percentage is high, and stays high throughout the lifetime of the program, with a maximum of 99%. When added up over all block phases, nodal movements constitute 84.3% of the program runtime. Finally, nodal movements are more expensive during the early block phases, taking 3.4x longer at 50k blocks compared to 12.5k blocks. Combining these observations, we reasoned that the cumulative time spent in nodal movements can be reduced by using more aggressive *merging* operations that reduce the initial size of the problem set.

Using the merge phase of the program to produce a deeper block reduction than the default halving behavior is fairly straightforward. We modified the merge loop stopping behavior to increase the number of blocks to be merged during the first merging operation. Every subsequent merge after the first continues to use the default 0.5x reduction behavior. There is a limit to how aggressively merging can be done without affecting the accuracy of the final partition. In our evaluation in Section IV, we measure both the total partition time, as well as the accuracy of the final partition, using different initial block reduction amounts. We experiment with initial block reduction rates of 0.75x and 0.90x.

B. Compressed Representation

From profiling our previous implementation of stochastic block partitioning, we observe an N^2 increase in memory consumption as the number of vertices in the input graph increases. This is due to the central data structure of the stochastic block partition algorithm – the *interblock edge count matrix*. This matrix keeps track of the count of in- and out-edges from every community to every other community. Every vertex is in its own community in its initial state, and its density is extremely low. For example, the initial density for the 50k and 100k node graphs in Table I is $4.1e{-4}$ and $2e{-4}$, respectively. Existing compressed representations for sparse arrays such as compressed sparse row (CSR), compressed sparse column (CSC), and dictionary of keys (DOK), are not well-suited for the stochastic block partition algorithm because the algorithm needs to extract and make temporary modified copies of row and column slices. The CSR and CSC formats compress along only one dimension, and thus make extraction along the other axis slow.

We devised an alternative representation that uses an array of hashtables across both rows and columns. The use of hashtables permits fast updates of arbitrary individual entries, quick extraction of non-zero entries, and reasonably fast updates of rows and columns. The cost compared to other compressed representations is duplicate entries, as each logical entry has two actual entries to allow for both row and column indexing. More critically, our implementation saves memory resources so more of the data can be served from CPU cache, rather than DRAM. One other consideration is that the entropy computations do not require any particular computation order for the entries in the requested row or column. We can support this fast retrieval of rows and columns, as long as order is not important, by returning just the non-zero entries in arbitrary order based on traversing the hashtable for the specified index and axis.

```
def init(dim):
  # Initialize row and column arrays
  # of hashtables.
  nr,nc = dim[0],dim[1]
  rows=[nonzerodict() for i in range(nr)]
  cols=[nonzerodict() for i in range(nc)]
def __getitem__(ix):
  i,j = ix
  return rows[i][j]
def __setitem__(ix, val):
  i,j = ix
  rows[i][j] = val
  cols[j][i] = val
def set_axis(ix, axis, dict_new):
  # Updates along one axis require
  # deletions along the other.
  # Find these deletion locations using
  # set differences.
  if axis == 0:
    for k in (rows[ix].keys - d_new.keys):
      del cols[k][ix]
    for k,v in d_new.items():
      cols[k][ix] = v
    rows[ix] = d_new
  else:
    for k in (cols[ix].keys - d_new.keys):
      del rows[k][ix]
    for k,v in d_new.items():
      rows[k][ix] = v
    cols[ix] = d_new
def take(ix, axis):
  # Take and return nonzero key-value
  # entries along the specified axis.
  if axis == 0:
    return (rows[ix].keys,rows[ix].vals)
  elif axis == 1:
    return (cols[ix].keys,cols[ix].vals)
```

Listing 1. Compressed Matrix Operations

Putting these ideas together, the pseudo-code for our compressed matrix operations are shown in Listing 1 in Python-like code. The hashtable implementation used is called a *nonzerodict* and is built around a Python `dict` object, with the additional feature that missing entries default to zero. Besides changes to the data representation of the interblock edge count matrix, small additional changes were needed to find move proposals and compute entropy changes. These changes are straight-forward since only non-zero entries in the interblock

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

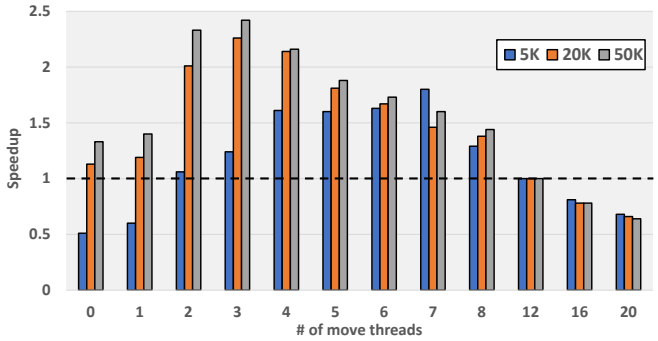

Fig. 3. Overall program speedup for different graph sizes, with a constant number of 12 merge threads, and a varying number of nodal move threads. Speedup is relative to 12 move threads. No one number of move threads is optimal for different graph sizes. The horizontal dotted line represents no speedup.

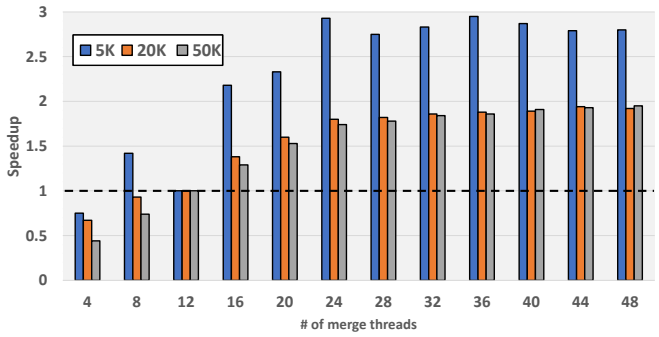

Fig. 4. Overall program speedup for different graph sizes, with a constant number of 12 nodal move threads, and a varying number of agglomerative merge threads. Speedup is relative to 12 merge threads. A larger number of merge threads is optimal, and yields less variation, compared to results for varying move threads. The horizontal dotted line represents no speedup.

edge count matrix need to be iterated over.

During our testing, we found that performing nodal updates on the compressed array is 2x–3x slower than nodal updates on the uncompressed matrix. We believe this is due to slow iteration operations in Python when writing to a row or column in the compressed matrix. Nodal movements tend to write more often than agglomerative merges, since merges typically move many vertices as an entire block, instead of one at a time. As a temporary workaround, we uncompress the matrix into a shared-memory buffer to pass to the workers during parallel nodal updates. We intend to explore optimizing this in future work.

C. Parallelism Control

Unlike agglomerative merges, nodal movements affect one node at a time rather than entire communities. Such behavior makes them write-intensive and difficult to parallelize. Furthermore, the quality of nodal updates for one worker depends on the timely incorporation of updates from other workers. Our baseline implementation avoids expensive global updates to the shared interblock edge count matrix by having each worker thread maintain its own copy of the data structure. We previously designed our implementation to carefully manage the granularity of messages, both from the workers to the main aggregator and back to each worker, to minimize the amount of data transferred. This parallel nodal movement design uses a pool of worker processes, each responsible for a group of vertices, which finds nodal movement proposals for that group, computes the associated acceptance probability, and ultimately accepts or rejects the proposals. Each worker reports the changes in the state back to the main aggregator for efficient distribution to the other workers.

We designed our baseline to minimize the amount of copying each worker must do by tagging updates and avoiding heavy-weight synchronization mechanisms. But despite our efforts to make nodal updates efficient, we found that the vast majority of the time is still spent on nodal updates. This leads us to theorize that the write-intensive nature of nodal updates limits the amount of useful parallelism. We further theorized that the scalability of nodal updates is also affected by the number of blocks at each block phase (i.e., the optimal number of nodal movement threads varies during the life of the program).

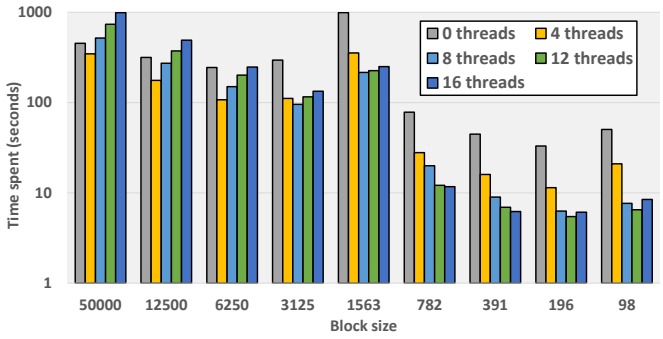

Fig. 5. Total time spent in nodal movements per block phase for a varying number of move threads, merge threads held constant at 12. Input is a 50k node graph. The optimal number of move threads varies by program phase.

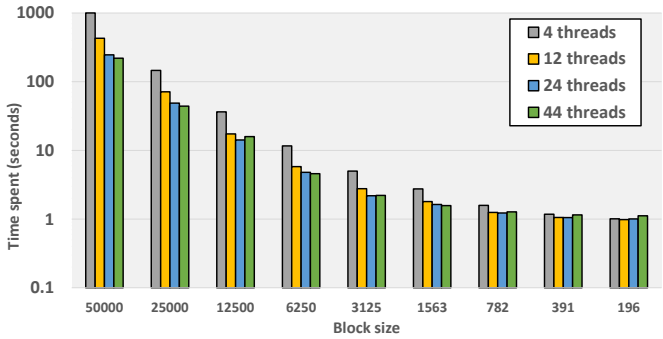

Fig. 6. Total time spent in merge movements per block phase for a varying number of move threads, move threads held constant at 12. Input is a 50k node graph. The optimal number of merge threads does not vary significantly by program phase.

We modified our program to allow for a configurable number of move threads and merge threads and varied one while holding the other constant at 12 threads. Using 12 threads was the best-performing configuration for our parallel baseline when the number of threads for each type of operation was the same. The results of these tests are shown in Figures 3 and 4. We found that no one number of nodal movement threads is optimal for different input graph sizes. We also found that the scalability of agglomerative merging is much better compared to nodal movements. A larger number of merge threads leads to faster runtimes, almost up to the limit of cores on the system. However, no one number of move threads is

optimal for different graph sizes.

We also observed a consistent pattern in the optimal number of move threads across the *phases* of a single program run. This is shown in Figure 5. For the 50k node input from Table I, we found that more move threads is better when there are fewer communities. This is similar to the behavior seen with different graph sizes. In contrast, we found that the optimal number of merge threads does not vary significantly by program phase, as seen in Figure 6. Based on our observations from varying the number of nodal movement threads while holding the number of merge threads constant, we established a heuristic to set the number of move threads based on the current block size. This mapping uses either 2, 3, 7, or 12 threads conditionally based on the number of blocks at > 20k, > 5k, > 2.5k, and < 5k, respectively.

IV. Performance Evaluation

A. Experimental Setup

For the evaluation of the techniques described in Section III, we start with our fast parallel stochastic block partition implementation written in Python. This implementation is based on the GraphChallenge stochastic block partition algorithm, with heavy modifications to support higher performance. Our implementation relies heavily on NumPy[16] for low-level array operations, and on Python multiprocessing parallel operation. We used Python 3.9.2 and NumPy 1.21.0 with an OpenBLAS backend on a CentOS Stream system. Our partitioning experiments use the baseline datasets from GraphChallenge, as described in Table I. Our single-node test machine is an Intel Xeon Gold 6126 CPU at 2.60 GHz with 24 cores, 48 threads, and 1480 GB of RAM. We ran experiments on static graphs with the number of vertices varying from 500 to 200k.

One straightforward measure of algorithm 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 also measured the move time and merge time taken during each block phase of the program. For each optimization, we first measured the independent incremental speedup on merge time, move time, and overall partition time compared to our default baseline implementation. We then measure the overall speedup when using all of the optimizations together.

B. Results

1) Aggressive Initial Merging: The independent impact of a more aggressive initial block reduction rate of 0.75 is shown in Table II. We chose 0.75 as a reasonable rate that did not introduce too many additional errors in the final partitioning, even though we have observed reasonable precision and recall scores with a block reduction rate of 0.90. The performance impact is significant, and improves both the overall merge time and nodal move time throughout the lifetime of the program. We also measured the correctness of our aggressive initial block reduction approach using pairwise precision and recall metrics [17]. The mean precision and recall scores due to different amounts of initial merging over 5 runs are shown in Table III and Table IV, respectively. In our combined testing, we again chose the reduction rate of 0.75 (see Section IV-B4) to balance performance and accuracy. In the future, we plan to measure the performance and accuracy tradeoffs involved with this technique more precisely.

TABLE II. Speedups due to Aggressive Initial Merging (0.75 reduction)

$\|V\|$	Merging	Moving	Overall
500	1.20	1.09	1.12
1k	1.20	1.10	1.12
5k	1.19	1.45	1.40
20k	1.14	1.59	1.49
50k	1.19	1.79	1.66
100k	1.18	2.01	1.82
200k	**1.19**	**2.04**	**1.85**

TABLE III. Accuracy – Mean Precision due to Aggressive Initial Merging (average of 5 runs)

$\|V\|$	Default Initial Reduction 0.5	0.75 Initial Reduction	0.90 Initial Reduction
500	1	1	0.5324
1k	1	1	1
5k	1	1	1
20k	1	1	1
50k	0.9937	1	1
100k	0.8708	0.9205	0.8569
200k	0.4763	0.5509	0.6114

TABLE IV. Accuracy – Mean Recall due to Aggressive Initial Merging (average of 5 runs)

$\|V\|$	Default Initial Reduction 0.5	0.75 Initial Reduction	0.90 Initial Reduction
500	1	1	0.9528
1k	1	1	1
5k	0.9743	0.9738	1
20k	1	1	0.9543
50k	1	1	0.9169
100k	0.9816	0.9129	0.6849
200k	0.9317	0.9054	0.8284

TABLE V. Speedups due to Data Compression

$\|V\|$	Merging	Moving	Overall
500	0.99	1.03	1.00
1k	0.99	0.95	0.95
5k	1.23	**1.07**	1.09
20k	3.16	1.02	1.15
50k	4.00	1.03	1.16
100k	4.20	1.06	**1.19**
200k	**4.65**	1.02	1.14

2) Compressed Representation: The independent impact of using the compressed representation for the interblock edge count matrix is shown in Table V. As expected, the compressed representation has a major impact on agglomerative merging operations, and no significant impact on nodal moves due to implementation details described in Section III-B. This optimization by itself has limited impact, yielding a 1.19x overall improvement in the best case. But when combined with the aggressive initial merging feature, the improvement is more substantial. We discuss this in more detail in Section IV-B4.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

TABLE VI. SPEEDUPS DUE TO PARALLELISM CONTROL

$\|V\|$	Merging	Moving	Overall
500	1.02	1.07	1.06
1k	1.04	1.02	1.03
5k	1.01	1.11	1.10
20k	1.01	1.63	1.48
50k	1.01	2.03	1.75
100k	1.89	2.27	2.19
200k	**2.02**	**2.34**	**2.26**

3) Parallelism Control: The results for the parallelism control optimization are shown in Table VI. As can be seen, the speedups achieved increase as the size of the graph increases. By using the heuristic described in Section III-C, which uses a different number of threads depending on the current number of blocks, the maximum overall speedup is 2.26x, with an average of 1.55x across all graph sizes.

4) Combined Techniques: In the results thus far, we have demonstrated the individual impact of each optimization. The results in Table VII show the speedups when all three optimizations are applied together. We kept the number of merge threads at 44 and used the heuristic from III-C for varying the number of move threads. We used 0.75 for the initial block reduction rate, which was the largest one where we did not see a major increase in the error rate. The combination of optimizations shows benefits starting at the 1k node graph and increases as the input graph size grows. We achieve a 5.13x speedup over the baseline at 200k nodes.

Finally, we conducted one test with a large 500k node graph (with 10.2M edges) as a proof-of-concept. Our baseline algorithm exhausts memory and fails to handle a graph of this size. But with all three of our optimizations enabled, we processed this graph in a total runtime of 99,811 seconds (compared to an average of 13,834 seconds for a 200k node graph), with the precision score of 0.3035 and the recall score of 0.9834. We will further explore processing larger graph sizes in future work.

TABLE VII. COMBINED SPEEDUPS – ALL THREE TECHNIQUES

$\|V\|$	Merging	Moving	Overall
500	0.62	1.04	0.88
1k	0.79	1.09	1.01
5k	1.67	1.51	1.53
20k	5.75	2.11	2.34
50k	10.73	3.38	3.78
100k	15.41	4.23	4.71
200k	**21.26**	**4.56**	**5.13**

V. CONCLUSIONS AND FUTURE WORK

We have described three new techniques to speed up stochastic block partition: aggressive initial merging, compressed data structures, and parallelism control. We have developed a prototype that shows excellent performance gains over our parallel baseline. These speedups over the baseline increase with increasing graph size, achieving speedups as large as 1.85x for aggressive initial merging, 1.19x for compressed data structures and 2.26x for parallelism control. Furthermore, when all three techniques are used together, the overall speedups are as high as 5.13x.

In the future, we would like to further develop and enhance our algorithm, particularly to further improve computational performance on even larger graphs. We believe that our current compressed representation of the interblock edge count matrix can be implemented in a faster language, such as C/C++, to achieve even greater speedups. Since the code that implements the uncompressed data structures at the heart of NumPy is compiled to native code, a compiled version of our compressed implementation prototype would make a better comparison. In the same vein, the entire codebase can be migrated to a compiled version to obtain better performance. Since our initial block reduction values are conservative, we plan to measure tradeoff between aggressive merging and accuracy more precisely. This aggressive merging can even be applied to later program phases for more improvements. Furthermore, the parallelization of nodal movements can be improved further and the current shared-memory communication between threads can also be optimized with low-level code. Finally, the compressed data structure and optimized algorithm allow us to perform community detection on much larger-sized graphs that did not previously fit into our system's memory. We plan to explore additional algorithmic optimizations, handle more difficult partitioning problems that exhibit high block overlap and varying block sizes, apply our techniques to streaming graphs, and tackle even larger graph sizes.

VI. ACKNOWLEDGMENT

The authors thank the anonymous reviewers for their constructive suggestions. This work was supported in part by Laboratory for Physical Sciences, and the National Science Foundation grants 1618706 and 1717774.

REFERENCES

[1] Y. Jin and J. F. Jaja, "A high performance implementation of spectral clustering on cpu-gpu platforms," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 825–834.

[2] H. Kanezashi and T. Suzumura, "An incremental local-first community detection method for dynamic graphs," in *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 2016, pp. 3318–3325.

[3] 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.

[4] 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.

[5] A. J. Uppal and H. H. Huang, "Fast stochastic block partition for streaming graphs," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–6.

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

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

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

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

978-1-6654-2369-4/21/$31.00 ©2021 IEEE

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

[11] 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.

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

[13] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song, and et al., "Streaming graph challenge: Stochastic block partition," *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2017. [Online]. Available: http://dx.doi.org/10.1109/HPEC.2017.8091040

[14] F. Wanye, V. Gleyzer, and W.-c. Feng, "Fast stochastic block partitioning via sampling," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019, pp. 1–7.

[15] 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.

[16] 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.

[17] 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.

978-1-6654-2369-4/21/$31.00 ©2021 IEEE