Decontentioned Stochastic Block Partition

Ahsen J. Uppal*, Thomas B. Rolinger†, and H. Howie Huang*

* The George Washington University, Washington, DC USA
† Laboratory for Physical Sciences, College Park, MD USA

auppal@gwu.edu, tbrolin@lps.umd.edu, howie@gwu.edu

***Abstract*—Stochastic block partitioning (SBP) is an important community detection algorithm that can achieve good accuracy, even on graphs with irregular structure. But SBP is difficult to parallelize because updates to its internal state are interdependent. This inherently serial nature limits its scalabilty and applicability to large, real-world graph data. In this work we address this challenge by introducing a Decontentioned approach to reduce the write contention on its internal state data. We apply a lock-free compressed data structure to handle writes, and split the parallel nodal movement procedure into a read phase for generating proposals, and a write phase for updating shared state. Finally, we find an optimal batch size to balance parallelism in worker threads and the overhead and convergence rate of the algorithm. Compared to our previous approach that buffers and combines updates to shared state, Decontentioned scales to the maximum number of CPU cores, where it yields a speedup of up to 5.38x on a 100k node input graph, and allows parallel processing of larger-sized graphs due to its more efficient memory usage.**

***Keywords*-community detection, GraphChallenge, stochastic block partitioning**

I. Introduction

Many relationships between real-world entities can be modelled as graph data, and hidden relationships between those entities can be identified by graph processing. Community detection is one such application that plays a vital role in finding complex or hidden structural relationships. But the performance and scalability of graph processing applications remains a challenging issue because many algorithms, including community detection are NP-hard.

Stochastic block partition (SBP) [1], is a probabilistic algorithm for community detection, presented as part of IEEE HPEC GraphChallenge. In our prior work [2]–[4], we developed several techniques to improve the performance of SBP. Our prior work, though effective, still struggled to achieve high performance due to challenges in parallelizing critical parts of the SBP algorithm. In this paper, we further improve the computational performance of SBP by improving the internal data structure and parallelism model.

The contributions of our work are as follows:

- The design and implementation of optimization techniques for parallel SBP in C and Python that include (1) the use of a novel lock-free shared-memory compressed data structure for the internal bookkeeping of the algorithm, (2) a method for separating read and write phases during nodal movements, and (3) the use of an efficient batch size to balance parallelism in each worker against the overhead and convergence rate of the algorithm.
- A study that evaluates the performance of the above SBP optimization techniques against the private copy, buffered update parallelism model used in our prior work. Our results show a speedup of 5.38x on a 100k node graph compared to our previous approach. Our implementation also achieves better scalabilty with a large number workers, and a much smaller memory footprint.

The rest of this paper is organized as follows. Section II presents background information pertaining to the stochastic block partition algorithm. We describe our approach to optimizing stochastic block partition 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 (SBP) algorithm is a probabilistic algorithm for performing community detection which uses a generative statistical model based based on work by Peixoto [5]–[7] and builds on the work from Karrer and Newman [8]. The algorithm uses a Markov Chain Monte Carlo (MCMC) method in the spirit of the Metropolis-Hastings algorithm [9], [10].

The number of blocks and assignment of each node to a block is not known ahead of time. The algorithm finds both by using an entropy measurement function to assess the quality of partitioning at each number of blocks. It compares the entropies at different numbers of blocks, ultimately bracketing the optimal number.

To actually find the optimal partitioning at a given number of blocks, SBP performs an agglomerative merge where two existing blocks of vertices are merged together, repeatedly and greedily, until the target number of blocks is reached. Once the target number of blocks is reached, the algorithm switches to a MCMC nodal movement phase. Here each vertex is proposed to be moved from its current community to another community with some probability based on the resulting change in entropy. Our previous work has shown that this nodal movement phase dominates the runtime of the algorithm and is difficult to parallelize [2], [3].

III. Method

In this section, we describe our techniques for improving parallel SBP. Our previous work [4] focused on devising a compressed data structure for efficient computations during SBP, dynamically adjusting the amount of parallelism, and reducing the amount of data to be processed by aggressively merging blocks. Since that work, we have made significant enhancements including moving the heavyweight portions of our algorithm from Python to native C code.

However, the techniques presented in this paper are orthogonal to the above mentioned enhancements and focus on changes to the internal data structure and synchronization model to improve shared-memory parallelism during the MCMC nodal movement phase.

A. Source of Nodal Update Contention

The internal state of the SBP algorithm is centered around two entities: a partition array, and an interblock edge count matrix. The partition array simply maps each vertex id to a community id. The interblock edge count matrix is derived from the partition array and keeps track of the counts of edges from every community to every other community. When a vertex u is moved from one block r to another block s, two rows and two columns of the interblock edge count matrix are updated: row r, column r, row s, and column s. The values of the row and column updates correspond to the community block ids of u's neighbors, and the combined edge counts to those neighbor blocks.

These MCMC nodal movements take around 90 percent of the baseline runtime of the algorithm for large-sized graphs, as we reported in our previous work. In our previous implementation of SBP on large-sized graphs, we also observed that there was no overall program speedup when using more than 8 threads for nodal movements. To understand why parallelizing SBP's nodal movements is difficult, consider the naive parallel algorithm shown in Algorithm 1.

In order to propose a nodal movement, the current state of the partition array in the neighborhood of the node index is read. But these neighboring vertices may be moved by other workers. Writing to and reading from this shared state without some form of synchronization can cause corruption that can crash the algorithm. An simple, but ineffective approach is to take a single lock around both reading from and writing to the shared state.

Our prior work solved this issue by buffering nodal movements generated the worker threads, and having a single thread combine all recent updates into global state updates periodically refreshed by each worker, as shown in Algorithm 2. But this approach is only partially effective because while the proposals based on current state can be generated in parallel, the actual updates to global state are still serialized.

B. Split-phase Nodal Updates

To address this data dependency issue, we have devised an approach based on carefully examining the data flow in the SBP algorithm. We observed that updates to shared global state – primarily in the interblock edge count matrix – are commutative addition and subtraction operations. Since these operations are order independent, the instructions needed to update this state need only to have atomicity, not synchronization, and can be executed in parallel by every worker thread. Furthermore, generating proposed updates only requires reading from shared state, not writing. As a result, preventing hazards in this situation only requires that no writers be active while proposals are being generated. Combining these two insights, we devised an approach that splits the parallel nodal move procedure into two phases – a read phase to generate proposed movements and a write phase to carry out the accepted movements. There is a simple barrier in between the phases to provide synchronization. Our improved approach is shown in Algorithm 3, which we refer to as decontentioned parallel nodal movement.

Note that our approach does not entirely eliminate the need for locks. Computing the correct update to shared state while moving a node requires looking at the current partition array at that node's index, as well as the current partition values of each of a node's neighbors. These cannot be modified while reading and still require a lock.

Our split-phase approach has an additional major advantage over the buffered approach in Algorithm 2. The buffered approach requires each worker to have its own private copy of the interblock edge count matrix. This is prohibitively expensive for large sized graphs, even using a compressed representation.

Algorithm 1: Naive Parallel Nodal Movement

```
 1: /* Each worker operates on a range of vertices. */
 2: parallelNodalMovementNaive (start_vert, stop_vert) {
 3:   for ni in range(start_vert, stop_vert) {
 4:     lock.acquire()
 5:     /* propose_node_movement reads different entries
          from the partition array and interblock_edge_cnt
          and returns the new rows and cols it used to
          compute the acceptance probability.
          */
 6:     result = propose_movement(G, ni,
              partition, interblock_edge_cnt)
 7:     r,s,p_accept,new_rows_cols = result
 8:     if (proposal accepted) {
 9:       partition[ni] = s
10:       interblock_edge_cnt[r,:] = new_rows_cols[0]
11:       interblock_edge_cnt[s,:] = new_rows_cols[1]
12:       interblock_edge_cnt[:,r] = new_rows_cols[2]
13:       interblock_edge_cnt[:,s] = new_rows_cols[3]
14:     }
15:     lock.release()
16:   }
17: }
```

Algorithm 2: Buffered Parallel Nodal Movement

```
/* Range of start_vert to stop_vert is batch size. */
parallelNodalMovementBuffered (start_vert, stop_vert) {
    /* Check for updates from central worker and copy changes. */
    lock.acquire()
    partition_local[:] = partition[:]
    for i in modified_blocks {
        interblock_edge_cnt_local[i, :] =
                interblock_edge_cnt[i, :]
        interblock_edge_cnt_local[:, i] =
                interblock_edge_cnt[:, i]
    }
    lock.release()
    for ni in range(start_vert, stop_vert) {
        proposal = propose_movement(G, ni,
                partition_local, interblock_edge_cnt_local)
        r,s,p_accept,new_rows_cols = proposal
        if (proposal accepted) {
            Buffer ni,r,s into results.
        }
    }
    Send results to central worker.
    /* Prepare to be called again with the next batch. */
}
```

Algorithm 3: Decontentioned Parallel Nodal Movement

```
/* Range of start_vert to stop_vert is batch size. */
parallelNodalMovementDecontentioned (start_vert, stop_vert) {
    for ni in range(start_vert, stop_vert) {
        /* Proposals are generated without taking locks. */
        result = propose_movement(G, ni,
                partition, interblock_edge_cnt)
        if (proposal accepted) { enqueue(Q, proposal) }
    }
    barrier()
    while(Q) {
        ni,r,s,new_rows_cols = dequeue(Q)
        lock.acquire()
        read partition[j] for j in neighbors of ni
        Compute block ids and edge counts from vertex
        neighbors into b_out, count_out, b_in, and count_in.
        partition[ni] = s
        lock.release()
        /* Update shared state in place with no locks */
        for i in range(len(b_out)) {
            interblock_edge_cnt[[r, b_out[i]] -= count_out[i]
            interblock_edge_cnt[[s, b_out[i]] += count_out[i]
        }
        for i in range(len(b_in)) {
            interblock_edge_cnt[[b_in[i], r] -= count_in[i]
            interblock_edge_cnt[[b_in[i], s] += count_in[i]
        }
    }
    /* Prepare to be called again with the next batch. */
}
```

C. Lock-free Compressed Data Structure

Using lock-free instructions to update the interblock edge count matrix is straightforward when this matrix is stored in a dense 2D-array. But this simple representation is not suitable for large sized graphs. At the beginning of SBP, each vertex is assigned to its own community, and thus the initial density of the interblock edge count matrix is very sparse (e.g., 4.1e-4 and 2e-4 for the 50k and 100k node graphs used in our work, respectively). Our previous work focused on devising a compressed representation of the interblock edge count matrix to greatly reduce the amount of storage space needed during processing. Our compressed data structure uses hash tables – one along each axis – to store the interblock edge counts. The tables along each axis are needed because the SBP needs to take slices along both dimensions in order to formulate the entropy changes that would result from block merges and nodal movements.

Implementing a lock-free design is desirable to improve parallelism, but is much harder than with a dense array, especially when hash tables must be re-sized. We have devised a hash table design that uses compare-and-swap (CAS) and double compare-and-swap (DCAS) instructions suitable for updating and resizing in parallel.

Our hash tables implementation uses a linear probing to resolve collisions, that is, first jumping to a slot based on hashing the key, and then scanning until the first unused slot is found. An insertion attempts to CAS a new entry into the first apparently empty slot after hashing the input key. If the CAS fails (i.e. another worker inserted into that open slot first), the next available slot is attempted. Once an insert succeeds, a resizing operation may be needed. Here we make use of a differential reference counting scheme similar to [11] to manage resizing without locks. This scheme uses a pointer to an outer structure that contains a reference counter and a pointer to an internal structure. The internal structure contains the pointer to the hash table itself, and another reference counter. A writer will DCAS the external structure, incrementing the reference counter, and incrementing the internal reference counter when done. Resizes can be done atomically by swapping in both the new external pointer, and the newly-reset external counter together. If a worker sees that a resize is needed, it will allocate a new, bigger, hash table and attempt to swap it in place. Whichever worker wins the race to swap the external structure will then adjust the internal counter, and merge all of the old entries into the new table. We additionally wrote a shared-memory lock-free memory allocator that uses queues of memory pools so hash table resizing can be done entirely without locks.

D. Nodal Update Batch Size

During nodal movements, there is a tradeoff between the overhead of synchronizing global state, and the quality of the proposals generated. The more fresh each worker's view of global state is, the better proposals it generates. Higher-quality proposals mean fewer nodal movements and larger changes in entropy, ultimately leading to faster convergence and better partition accuracy. This batch size parameter is critical and must be measured empirically to tune the algorithm.

In our Decontentioned approach, the group batch size is the size of the range between the start vertex and stop vertex for each parallel worker. In other words, the group batch size is the number of proposals a worker generates before waiting on the barrier and seeing updated global state. The same batch size tradeoff also occurs in our Buffered implementation. In that approach, the group size is the number of proposals evaluated and sent to the central worker before synchronizing from global state. Our previous work [2] found that a batch size of 1 (i.e. the lowest possible granularity) was optimal, but our implementation details and fixed overheads are different now, so we choose to re-evaluate this tradeoff. We found optimal batch sizes for both approaches, and we discuss the details in the next section.

IV. Performance Evaluation

We evaluate our decontentioned approach, as described in Section III with the buffered approach from our prior work [4].

A. Experimental Setup

The datasets we used in our experiments are described in Table I. These include the baseline datasets from the GraphChallenge, supplemented by larger graphs we synthesized using the generator in the GraphChallenge repository. We instrumented our code to measure the overall program runtime, ignoring time to read from disk. Our tests were conducted on a system with two 64-core AMD EPYC 7713 CPUs and 1TB of RAM. For simplicity and because the program runtime is dominated by nodal movements, multi-threaded tests were conducted by setting an equal number of agglomerative merge and nodal movement threads. The serial baseline uses the same native code and compressed data structures, just without the overhead of creating and managing threads. Our code is written in a mixture of Python and C, with the native C code doing the heavy lifting, including computing entropy and implementing the Decontentioned compressed data structure described in Section III. We use Python 3.11.3 and Clang 15.0.7 to build our C code, and Numpy [12] for array processing.

B. Results

1) Group Batch Size: First we set out to characterize the sensitivity of both the Buffered approach and our new Decontentioned approach to the group batch size. We measured the performance of each with a varying number of threads and group sizes. The results for the baseline Buffered approach for the N=20k graph are shown in Figure 1. We see that the optimal number of vertices before synchronizing in the baseline approach can be large, only leveling off at 256 for the smaller number of threads. We found similar results for other input graph sizes, and selected 256 as a reasonable value.

TABLE I
Graph Datasets

$\lvert V\rvert$	$\lvert E\rvert$	Density	Blocks	Serial Runtime(s)
5k	101,973	4.1e−3	19	57.67
20k	408,778	1e−3	32	334.5
50k	1,018,039	4.1e−4	44	1181
100k	2,037,415	2e−4	56	2837
200k	4,064,602	1e−4	71	5596

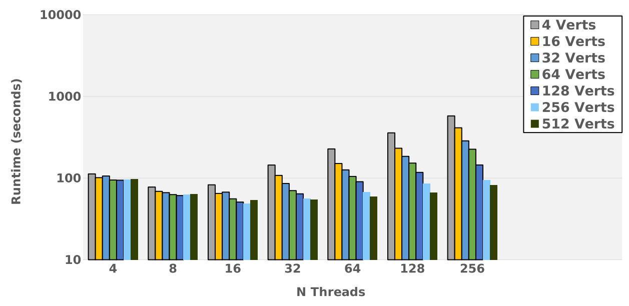

Fig. 1. Performance of parallel Buffered on a N=20k input graph across numbers of threads and varying batch sizes.

The characterization of our Decontentioned approach is shown in Figure 2 and Figure 3 for 20k and 200k graph sizes, respectively. Here we see that a smaller number of vertices is a better batch size compared to Buffered. We found good overall performance at a batch size of 64 vertices, vs. (256 or 512 for Buffered). Our Decontentioned approach performs best under smaller batch sizes, because it has an additional barrier overhead. A worker in Buffered is not blocked waiting for other workers to complete. If a worker has not completed its batch, those results will simply be reported and incorporated later.

2) Decontentioned Performance: Next we look directly at the performance of our Decontentioned approach against the baseline Buffered approach, each configured with an appropriate batch size as determined in Figures 1–3. The speedup of Buffered compared to serial for different numbers of threads

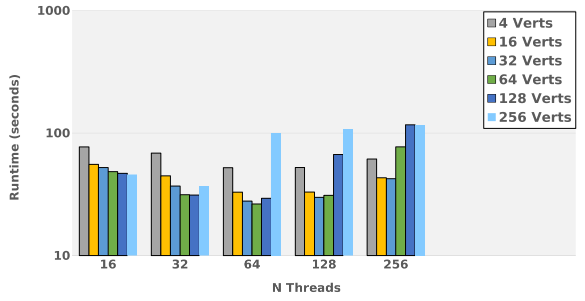

Fig. 2. Performance of parallel Decontentioned on a N=20k input graph across numbers of threads and varying batch sizes.

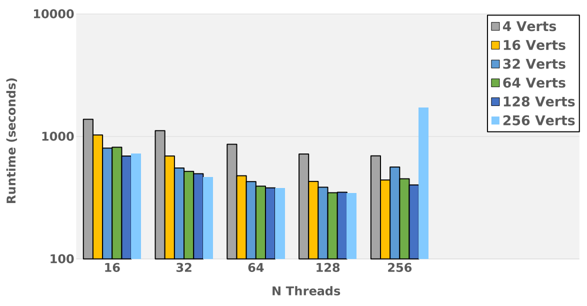

Fig. 3. Performance of parallel Decontentioned on a N=200k input graph across numbers of threads and varying batch sizes.

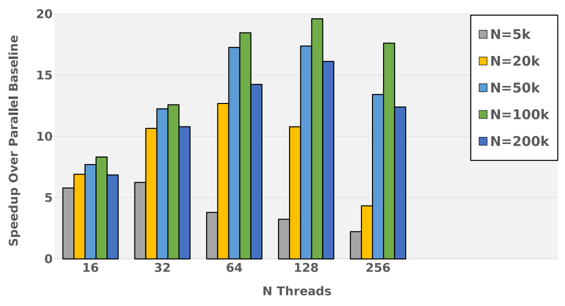

Fig. 5. Parallel Decontentioned speedup over serial across numbers of threads.

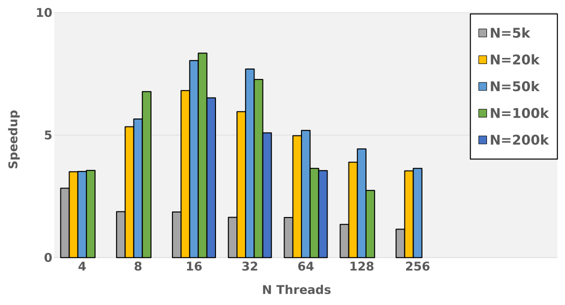

Fig. 4. Parallel Buffered speedup over serial across numbers of threads.

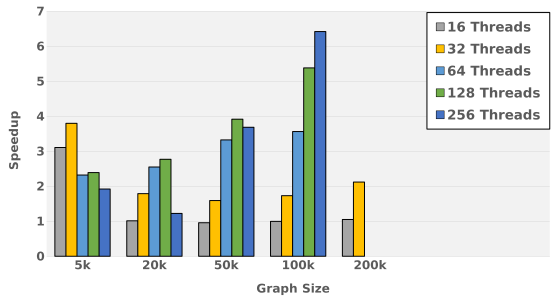

Fig. 6. Decontentioned speedup over Buffered across different input graphs and numbers of threads.

is shown in Figure 4. Here we see that the maximum speedup is achieved at a low number of threads (16) compared to the number of CPU cores available (128 physical cores), for every input graph. This corresponds to our previous work showing that nodal movement parallelism is limited [4]. Note that Buffered failed to run on the larger 100k and 200k node graphs with a large number of threads because it exhausted available system memory (which is 1TB) due to duplicated state in the worker threads.

The speedup of our new Decontentioned over serial is shown in Figure 5. Unlike Buffered, the maximum speedup is achieved at 128 threads, corresponding to the number of CPU cores available on the system. Furthermore, the speedups are much higher. Next we directly compare the performance of Buffered and Decontentioned. The speedup of Decontentioned over Buffered is shown in Figure 6. Here we see the maximum speedup over Buffered at 128 threads is 5.38x. At 256 threads, Decontentioned is no longer at its fastest, but has a speedup of 6.42x compared to Buffered. This indicates a more graceful performance degradation on an over-scheduled system. Furthermore, Decontentioned can actually process the larger 100k and 200k graphs with a large number of threads, because of its more efficient use of memory.

V. Conclusions and Future Work

We have described a new approach to that improves the parallelism of Stochastic Block Partition by leveraging a novel lock-free compressed data structure, splitting the critical nodal movement operations into separate read and write phases, and optimizing the granularity of nodal movement updates across workers. Our current work forms the basis for further algorithmic and implementation improvements in the future. In particular, we would like to combine our new approach to shared-memory parallelism on one node with message-passing based parallelism across nodes. We also plan to investigate the use of fine-grained locking to further enhance the parallelism of the algorithm. Finally, we plan to apply adaptive techniques based on vertex connectivity and asynchronous Gibbs sampling [13] to dynamically adjust the nodal update batch size.

VI. Acknowledgements

This work was supported in part by the National Science Foundation, 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, 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

[2] 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.

[3] 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.

[4] A. J. Uppal, J. Choi, T. B. Rolinger, and H. Howie Huang, "Faster stochastic block partition using aggressive initial merging, compressed representation, and parallelism control," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, pp. 1–7.

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

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

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

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

[9] 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.

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

[11] A. Williams, *C++ Concurrency in Action: Practical Multithreading*. Manning, 2012.

[12] 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.

[13] F. Wanye, V. Gleyzer, E. Kao, and W.-c. Feng, “On the parallelization of mcmc for community detection,” in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–13.