GSAP: A GPU-Accelerated Stochastic Graph Partitioner

Chih-Chun Chang
University of Wisconsin-Madison
chih-chun.chang@wisc.edu

Boyang Zhang
University of Wisconsin-Madison
bzhang523@wisc.edu

Tsung-Wei Huang
University of Wisconsin-Madison
tsung-wei.huang@wisc.edu

ABSTRACT

Graph partitioning is essential for understanding the structure of a dataset, such as social networks and web pages. Among various graph partitioners, stochastic block partitioning (SBP) has shown promise in handling complex graphs with varying community sizes or strong intra-community connections. However, the sequential nature of the Monte Carlo Markov Chain iterations and the stochastic proposal generation process limit the efficiency and scalability of SBP. To overcome this limitation, this paper introduces GSAP, a GPU-accelerated stochastic graph partitioner, to enhance the runtime performance of SBP. We propose a parallel algorithm to speed up the generation process of stochastic proposals on GPU. Additionally, we accelerate the calculation of the minimal description length by dividing the formulation into several independent computations. To achieve better performance, we introduce an efficient blockmodel update algorithm to dynamically manage the blockmodel matrix on GPU. Our experimental results on the 2022 HPEC GraphChallenge dataset demonstrate that GSAP can achieve up to 12.3× and 60.9× runtime speedup on a single A4000 GPU compared to two CPU-parallel state-of-the-art SBP algorithms.

ACM Reference Format:
Chih-Chun Chang, Boyang Zhang, and Tsung-Wei Huang. 2024. GSAP: A GPU-Accelerated Stochastic Graph Partitioner. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3673038.3673117

1 INTRODUCTION

Many real-world datasets [50], such as social networks and web pages, use graphs to represent complex structures and relationships. Graph partitioning [49, 75], which divides a graph into several parts, is a crucial technique for understanding these complex datasets. For example, the spectral algorithm [64] utilizes the eigenspectrum of the modularity matrix to partition graphs based on connectivity between vertices. The Girvan–Newman algorithm [65] partitions a graph by iteratively removing edges with the highest betweenness centrality.

SBP [66–68] is one of the graph partitioning algorithms based on iterative inference over stochastic blockmodels [45] to identify the optimal partition of a graph. Stochastic blockmodels represent a graph's structure by a two-dimensional sparse matrix. The elements in the matrix record edge count between blocks. The SBP algorithm consists of three phases: (1) the block-merge phase, (2) the vertex-move phase, and (3) the golden-section search. In the block-merge phase, selected blocks are merged. During the subsequent vertex-move phase, Monte Carlo Markov Chain (MCMC) iterations [18] use the Metropolis-Hastings algorithm to move vertices between blocks, refining the current partition. Next, the golden-section search phase identifies the optimal partition that achieves the minimum description length (MDL) of the blockmodel based on the current partition. Compared to conventional graph partitioning approaches, SBP performs well on complex graphs which have varied community sizes or strong intra-community connections. However, the serial nature of MCMC iterations and the stochastic proposal generation process limit the runtime efficiency and scalability to large graphs.

As a result, the recent IEEE High Performance Extreme Computing Conference (HPEC) introduced the SBP Challenge (SBPC) [44] to seek novel acceleration techniques from the the high-performance computing (HPC) community. F-SBP [70], the 2019 award winner, leverages existing sampling algorithms to enhance SBP runtime. Faster-SPB [69], the 2021 champion, implements an aggressive initial merging strategy to significantly reduce the number of iterations required. H-SBP [73] proposes a hybrid approach by combining MCMC iterations with the asynchronous Gibbs algorithm for parallelization. uSAP [2], the 2023 award winner, introduces a novel initial block-merge strategy based on strongly connected components and employs task graph parallelism to enhance the runtime of SBP. However, these methods are still limited to handling only small to medium-sized graphs. I-SBP [72], 2023 GraphChallenge winner, presents a distributed SBP algorithm to partition larger graphs, while their single-node version show limited performance.

Despite improved runtime performance by existing works, their scalability is largely constrained by CPU parallelism. Compared to CPU, modern GPU offers order-of-magnitude more parallelism and memory bandwidth, which is particularly suitable for handling large graphs. This advantage has inspired us to leverage the power of GPU computing to accelerate SBP. To this end, we propose a GPU-accelerated stochastic graph partitioner called GSAP. To the best of our knowledge, this is an early attempt that successfully accesses SBP using GPU computing. We summarize our contributions as follows:

- We propose a parallel algorithm to accelerate the generation of stochastic proposals on GPU, significantly speeding up the process compared to the CPU-parallel approach.
- We decompose the formulation of MDL into several independent calculations, enabling the parallel computation for each proposal during the block-merge and vertex-move phases. This approach significantly speeds up the heavy workload of MDL computation.
- We introduce an efficient blockmodel update algorithm for dynamically updating the blockmodel matrix during the block-merge and vertex-move phases on GPU. This significantly enhances the runtime performance compared to the CPU-parallel approach.

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673117

We have evaluated GSAP on a single A4000 GPU using the datasets provided by the HPEC SBP Challenge [44] and compared the runtime performance over two state-of-the-art SBP algorithms, uSAP [2] and I-SBP, [72] on a 14-core CPU. The experimental results shows that GSAP achieves 12.3× speedup over uSAP and 60.9× speedup over I-SBP when partitioning graphs of 50K–200K vertices. For the largest graphs of 1M vertices, GSAP can finish SBP in just 15 minutes, whereas uSAP and I-SBP both fail to complete in two hours.

2 BACKGROUND

Graph partitioning aims to identify unique community structures within a graph by determining the partition of each vertex. This partition provides insights into the interactions among vertices and helps identify vertices that belong to specific communities of interest. Some well-known methods adopt the spectral approach [64] to identify partitions in a graph. Some algorithms depend on graph modularity, which measures the density of connections within partitions. High modularity in graphs indicates more intra-partition connections and fewer inter-partition links. Nevertheless, this approach faces challenges, including resolution limits that affect the detectable size of partitions and difficulties in determining the optimal number of communities. To tackle these issues, recent studies have chosen generative statistical models for graph partitioning. Specifically, [66–68] utilize degree-corrected stochastic blockmodel [45] for partition estimation, where each partition is defined as a block.

2.1 Stochastic Block Partitioning

Stochastic Block Partitioning (SBP) utilizes the stochastic blockmodel to identify the optimal partition of a graph, as depicted in Figure 1. This approach models the block structure with a sparse matrix $M_{B\times B}$, where B is the number of identified blocks within the graph. Each matrix element, $M_{a,b}$, records the edge count starting from block a to block b. The log posterior probability function, denoted by $P(G|B)$ in Equation 1, is used to describe the current partition of graph G, given the blockmodel B. $M_{i,j}$ is edges count between blocks i and j. D_i^{in} and D_j^{out} represent the total in-degree and out-degree for blocks i and j, respectively.

$$P(G|B) = \sum_{i,j} M_{i,j} \cdot \log\left(\frac{M_{i,j}}{D_i^{in} D_j^{out}}\right) \tag{1}$$

SBP is divided into three distinct phases: block-merge, vertex-move, and golden-section search. SBP iterates among these phases to discover the ideal partition with the optimal block count, B^*, that minimizes the total description length (MDL), as defined in Equation 2. To overcome the issue of local minima, the algorithm starts with each node assigned to its own block. Then, blocks are progressively merged, followed by Monte Carlo Markov Chain (MCMC) updates on individual vertices to determine the optimal partition for the current number of blocks.

$$MDL = Eh\left(\frac{B^2}{E}\right) + V\log B - P(G|B) \quad \text{where } h(x) = (1+x)\log(1+x) - x\log x \tag{2}$$

In the block-merge phase, a merge is proposed for every block, followed by calculating the resulting change in MDL. The merges resulting in smaller MDL changes are performed according to the reduction rate parameter. In the vertex move phase, a move is proposed for each vertex, followed by the calculation of Metropolis-Hastings acceptance ratio based on MDL to determine whether the move should be accepted or rejected. Subsequently, the blockmodel matrix M is updated. The vertex-move phase is the major bottleneck in SBP due to the sequential MCMC updates that iterate over every vertex until convergence is achieved. The profiling results presented in [73] reveal that the vertex-move phase can account for up to 98% of the total runtime. This high percentage is due to the time-consuming nature of the proposal generation process and the MDL computation for each vertex. Furthermore, the blockmodel matrix must be updated after proposals are accepted during the vertex-move phase. Iterating through these three steps makes the vertex-move phase the bottleneck of the SBP algorithm.

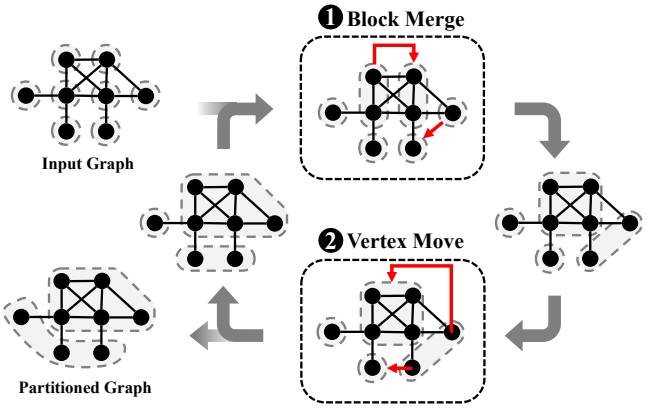

Figure 1: Overview of the SBP algorithm. Initially, each vertex in the graph is assigned to a block. The graph then undergoes iterative block-merge and vertex-move phases until the partitioning process achieves the minimum description length (MDL) through a golden-section search algorithm.

2.2 HPEC SBP Challenge

Despite SBP's ability to handle complex graphs, its major limitation is the long runtime, caused by the sequential nature of the Monte Carlo Markov Chain iterations and the stochastic proposal generation process. To overcome the limitation, the recent IEEE HPEC launched the SBP Challenge (SBPC) to seek novel acceleration approaches for the SBP algorithm. To evaluate the performance of a graph partitioner, SBPC provides a comprehensive evaluation dataset containing synthetic graphs based on the stochastic model [45]. These graphs are generated by sampling the connection between vertices from a Poisson distribution with a correction term to simulate real-world graph characteristics. Table 1 details the attributes of the dataset, which includes four graph categories ranging from 1K to 1M vertices. Each category represents varying degrees of partitioning challenges and graph complexity with the difficulty level of the categories increasing from easiest to hardest.

(1) Low-Low: Low block overlap and low block size variation

(2) Low-High: Low block overlap and high block size variation
(3) High-Low: High block overlap and low block size variation
(4) High-High: High block overlap and high block size variation

Categories	$\|V\|$	$\|E\|$	B	Description
Low-Low	1K	8,067	11	Low level of overlap and low level of size variation between partitions (easiest)
	5K	50,850	19	
	20K	473,914	32	
	50K	1,189,382	44	
	200K	4,750,333	71	
	1M	23,716,108	125	
Low-High	1K	7,892	11	Low level of overlap and high level of size variation between partitions
	5K	50,544	19	
	20K	476,386	32	
	50K	1,193,994	44	
	200K	4,747,902	71	
	1M	23,737,108	125	
High-Low	1K	7,903	11	High level of overlap and low level of size variation between partitions
	5K	51,091	19	
	20K	475,421	32	
	50K	1,183,975	44	
	200K	4,743,468	71	
	1M	23,781,433	125	
High-High	1K	8,032	11	High level of overlap and high level of size variation between blocks (hardest)
	5K	51,157	19	
	20K	473,329	32	
	50K	1,187,682	44	
	200K	4,754,406	71	
	1M	23,772,977	125	

Table 1: The four categories of the SBPC dataset. The Low-Low category represents low complexity, making it the easiest benchmark, while the High-High category represents high complexity, making it the hardest benchmark. Each category contains graphs with vertex counts ranging from 1K to 1M. $|V|$ represents the number of vertices in the graph, $|E|$ denotes the number of edges, and B is the number of partitions.

2.3 Related Works and Their Limitations

Many novel approaches have been proposed to accelerate the SBP algorithm. For example, F-SBP [70] reduces the SBP runtime by utilizing an existing sampling algorithm for subgraph selection and performing partitioning across multi-core clusters. This approach preserves the graph's partition structure and accelerates the algorithm without significantly losing the accuracy. Nevertheless, this method does not achieve a significant reduction in runtime to effectively address the challenges posed by large graphs.

Faster-SPB [69] introduces an aggressive initial merging strategy to considerably decrease the initial block count at the first iteration, thereby reducing the overall number of partitioning iterations needed. However, the aggressive initial merging strategy may merge blocks that cause negative effects on the partition quality. Its parallelism control strategy carefully manages the amount of parallelism during different phases of the algorithm to improve the performance while it does not perform well due to the synchronization overhead.

H-SBP [73] implements the asynchronous Gibbs algorithm to parallelize the MCMC iterations. This innovative approach enables the parallelization of computations that are typically sequential within each MCMC in SBP. By serially processing a select portion of the most influential vertices and parallelizing the remainder, H-SBP maintains accuracy while significantly enhancing efficiency. The method demonstrates strong scaling capabilities, particularly in making the vertex move phase parallelizable. Despite these advances, H-SBP may encounter convergence issues with larger graphs.

uSAP [2] introduces a new strategy based on strongly connected components for initial block merging, decreasing the number of iterations needed for partitioning. To speed up runtime, uSAP implements a dynamic, batch-oriented task graph parallel algorithm for vertex moves, leveraging Taskflow [6, 27, 32, 33, 35] to accelerate SBP.

EDiSt [71] introduces a distributed version of SBP, allowing the algorithm to efficiently scale across a larger number of compute nodes without encountering convergence problems. They also develope a version of SBP optimized for shared memory parallelism on larger clusters. However, the all-to-all communication pattern in EDiSt becomes a significant bottleneck as the number of nodes increases.

I-SBP [72] combines three distinct heuristics [70, 71, 73] to accelerate SBP. However, it still encounters limitations similar to those observed in other approaches.

Despite various approaches developed to accelerate the SBP algorithm, several significant challenges remain unresolved. We highlight three major challenges below:

2.3.1 Limitation to CPU Parallelism. Nearly all existing works leverage multi-core CPU parallelism to accelerate the SBP algorithm. However, the effectiveness of CPU-based parallelism diminishes as graph sizes increase. Therefore, the limitations of CPU parallelism for large graphs remain a critical challenge.

2.3.2 Lack of GPU Methods for SBP. To the best of our knowledge, no existing works focus on developing a GPU-accelerated SBP algorithm. Utilizing GPUs to handle large graphs presents a promising solution due to their ability to massively parallelize computations in SBP, such as generating stochastic proposals and calculating *MDL*.

2.3.3 Challenges with GPU Implementation. Although GPU offers substantial speedup and efficient handling of large volumes of data, designing a GPU-accelerated SBP presents several challenges. For instance, an efficient GPU data structure and kernel algorithm are necessary to dynamically represent partitioned graphs during the SBP process. The irregular data patterns pose challenges in developing GPU kernel algorithms for generating stochastic proposals and computing the *MDL* on GPU. Furthermore, it is critical to update the blockmodel on GPU to minimize the runtime of SBP. Effectively addressing these challenges is key to optimizing the algorithm's performance.

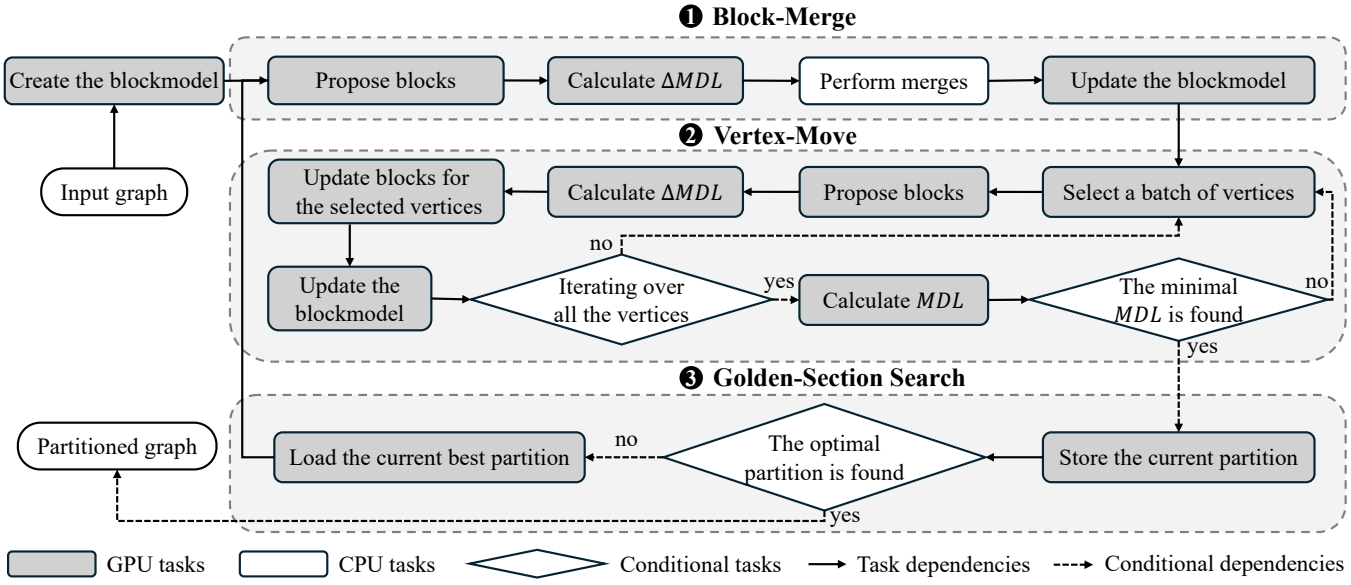

Figure 2: Overview of GSAP. The task graph consists of three steps: (1) The block-merge phase, where each block proposes another block to merge with and calculates the corresponding change in MDL (ΔMDL). Subsequently, these proposals are transferred back to the CPU for merging. (2) The vertex-move phase, which performs parallel MCMC iterations by selecting batches of vertices to propose a move to another block and computing the acceptance probability of each proposal. (3) The golden-section search step, which determines when GSAP should stop by progressively narrowing the search to the number of blocks where the MDL is achieved.

3 GSAP

To overcome the above challenges, we introduce GSAP, a GPU-accelerated stochastic graph partitioner, to speed up the SBP algorithm using GPU. Figure 2 shows the overall task graph of GSAP, where arrows indicate task dependencies. The task graph consists of three steps: the block-merge phase, the vertex-move phase, and the golden-section search. In the block-merge phase, each block stochastically proposes another block to merge with and calculates the corresponding change in *MDL* (ΔMDL). Subsequently, these proposals are transferred back to CPU, where the selected blocks perform merges. In the vertex-move phase, we select a batch of vertices to perform asynchronous Gibbs sampling for the MCMC iterations. Each selected vertex stochastically proposes a block to move to and computes the acceptance probability for this proposal. If the proposal is accepted, the vertex is moved to the proposed block. The proposals in both the block-merge phase and the vertex-move phase are generated in parallel on GPU. The golden-section search step determines when the algorithm will stop. When the *MDL* is achieved, we obtain the optimal partition of the graph. We highlight the GPU-accelerated steps in dark color, while the perform-merge step is executed on the CPU because its does not benefit from GPU acceleration. We aim to address the runtime bottleneck through GPU parallelism.

3.1 Data Representation

The blockmodel matrix is a crucial data structure used to represent the structure of a graph in the SBP algorithm. Elements within the matrix represent the weighted sum of edges between blocks, while

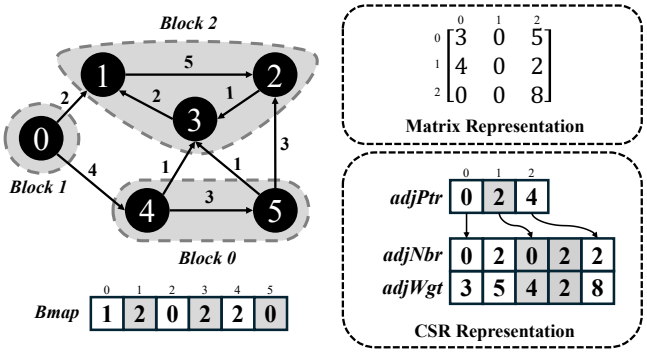

Figure 3: An example of data representation in blockmodels, illustrating both the matrix representation and the Compressed Sparse Row format (CSR) representation.

the diagonal elements denote the weighted sum of edges within each block, indicating self-connectivity. Typically, the blockmodel matrix appears as a sparse matrix, presenting challenges for efficient storage. To reduce the memory footprint and maximize the size of the graph that can be stored, we utilize the Compressed Sparse Row (CSR) format to store the blockmodel in GPU's DRAM. Figure 3 illustrates a blockmodel represented in the CSR format, which consists of three arrays:

- `adjNbr`: The adjacency neighbor list array contains the blocks adjacent to each block.
- `adjPtr`: The adjacency pointer array indexes the start of each block's adjacency list in `adjNbr`.

- `adjWgt`: The adjacency weight array records the weight of each edge connecting each block to its neighboring blocks.

Given that the edges between blocks are directed, it is crucial to maintain information on both in-degree and out-degree neighbors. The CSR structure incorporates six arrays to store this information used by GSAP. For example, in Figure 3, consider the out-degree neighbors of block 0: an edge within the block has a weight of 3, representing self-connectivity. Additionally, block 0 has an out-degree neighbor, block 2, with the weighted sum of edges directed from block 0 to block 2 recorded as 5. Therefore, the adjacency list for block 0 includes entries for both blocks 0 and 2, with corresponding weights of 3 and 5 noted in the adjacency weight array.

To represent the partition of a graph, we use a block mapping array, denoted by `Bmap`, to record the block ID for each vertex. Whenever blocks merge or vertices move, we update this array to reflect the current partition. Additionally, to calculate the ΔMDL for each proposal during both the block-merge and vertex-move phases, we maintain the in-degree and out-degree for each block in two separate arrays, `BdegIn` and `BdegOut`.

3.2 Generating Stochastic Proposals

The iterative proposal-generating process during the block-merge and vertex-move phases of the SBP algorithm is time-consuming. Each block (or vertex) initially selects an adjacent block based on the multinomial probability distribution derived from the weights of connecting edges. If a block (or vertex) has no neighbors, it randomly selects a block. Subsequently, a number x is drawn from a uniform distribution between 0 and 1 and compared to $\frac{B}{\texttt{deg[u]}+B}$, where B represents the number of blocks in the current partition and `deg[u]` is the total degree of block u, calculated as the sum of `BdegIn[u]` and `BdegOut[u]`. If x is greater than the value, the proposal is generated from the adjoining block of u; otherwise, a block is proposed randomly. This stochastic decision is implemented to prevent the proposal-generating process from being trapped in local optima of *MDL*. During the block-merge phase, this process is repeated for each block in the current partition, iterating *num_proposals* times. In the vertex-move phase, each vertex in the graph iteratively generates a proposal until it reaches the MDL.

To leverage GPU parallelism for efficiently solving the iterative proposal-generating process in SBP, we implemented three generators using the NVIDIA cuRAND library. Initially, we launched the random number generator, uniform number Generator, and multinomial distribution generator. These are used to create three lookup tables, which is detailed in Figure 4. This setup significantly reduces the overhead of repeatedly generating random numbers. Specifically, The uniform number generator produces a large batch of x for each proposal. The random number generator is tasked with creating numerous random block IDs. The multinomial distribution generator determines an adjacent block for each block in the graph.

With the lookup table in place, we can initiate many GPU threads to generate proposals simultaneously. During the block-merge phase, we scale up the operation by launching threads equal to the product of the number of blocks in the current partition and the *num_proposals* to handle proposals in parallel. For each block, the best proposal is selected based on the *MDL*. In the vertex-move phase, we employ a parallel MCMC approach where a batch of vertices generates proposals simultaneously. This strategy considerably reduces the time required for sequential MCMC iterations. The generated proposals are stored in a proposal array, which is used to calculate the *MDL* for each proposal. The GPU kernel algorithm for generating proposals in the block-merge phase is described in 1. Each block considers its neighbor counts and then uses three pre-generated tables to propose candidates. The algorithm for the vertex-move phase is similar, but each vertex must identify the block ID of its adjacent vertices.

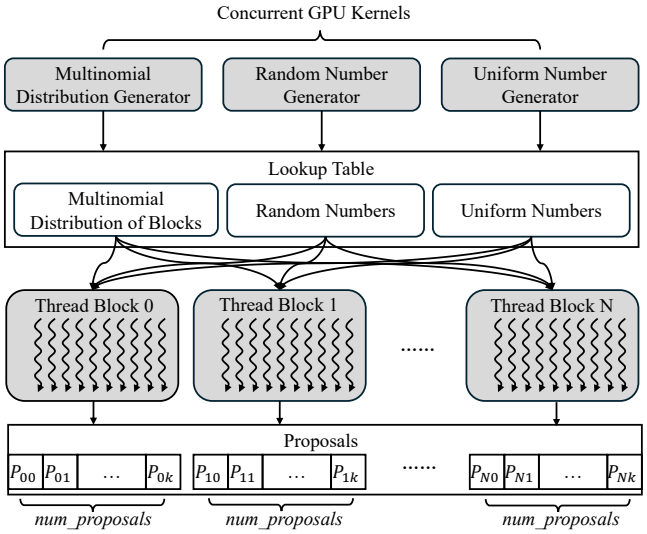

Figure 4: In both the block-merge and vertex-move phases, GSAP initially launches three concurrent kernels to build three lookup tables that store multinomial distributions, uniform numbers, and random numbers. Subsequently, GSAP launches many GPU threads to generate proposals in parallel using these lookup tables.

Algorithm 1: Generating Proposals in Block-Merge

Input: Current block count B, blockmodel in CSR format, lookup tables for multinomial distribution, uniform distribution, and random numbers
Output: A proposed block to merge

```
block_id = blockIdx.x × blockDim.x + threadIdx.x;
if deg[block_id] ≤ 0 then
    return random_table[block_id];
endif
u = multi_table[block_id];
x = uniform_table[block_id];
if x ≤ B/(B+deg[u]) then
    return random_table[block_id];
endif
return multi_table[u];
```

3.3 Computation of MDL

The *MDL* computation is another time-consuming step in SBP, as it is necessary to compute the *MDL* for each proposal. To speed up this heavy workload, we begin by decomposing the formulation into several independent calculations. Instead of calculating the entire *MDL*, we focus on the change in *MDL* caused by each proposal, denoted as ΔMDL and defined in Equation 3. This equation quantifies the difference in the log posterior probability functions for the graph G by comparing the blockmodel B' of the new partition with the blockmodel B of the current partition.

$$\Delta MDL = P(G|B') - P(G|B) \tag{3}$$

During the block-merge phase, the primary change between the original and new partition occurs when a proposed block merges with another block. However, the connections among other blocks remain unchanged. To avoid unnecessary calculations of ΔMDL, it is sufficient to focus only on the incoming and outgoing edges of the merged block b and the proposed block s before and after the merge. To simplify this process, we separate the calculations for incoming and outgoing edges, denoted as C^{in} and C^{out}, respectively. This ensures that each set is evaluated independently, as outlined in Equation 4.

$$\Delta MDL = \left(C^{out}_{b'} + C^{in}_{b'}\right) - \left(C^{out}_{b} + C^{in}_{b}\right) \quad \text{where} \quad b' = b \cup s \tag{4}$$

To calculate $C^{in}b$ and $C^{out}b$ for the original block b, we access the in-degrees D^{in}_i and out-degrees D^{out}_j of b's adjacent blocks from `adjNbr`. We also retrieve the weights of the incoming $M_{i,b}$ and outgoing $M_{b,j}$ edges from `adjWgt`, as outlined in Equation 5. To leverage GPU parallelism, we employ a segmented reduction operation across the current blockmodel, allowing for the parallel computation of C^{in}_b and C^{out}_b. To avoid the duplicated computation of self-connectivity in Equation 5, we must subtract the result of this self-connectivity to maintain the correctness.

$$\begin{aligned} C^{in}_b &= \sum_{i,b} M_{i,b} \cdot \log\left(\frac{M_{i,b}}{D^{in}_i D^{out}_b}\right) \\ C^{out}_b &= \sum_{b,j} M_{b,j} \cdot \log\left(\frac{M_{b,j}}{D^{in}_b D^{out}_j}\right) \end{aligned} \tag{5}$$

To compute $C^{in}_{b'}$ and $C^{out}_{b'}$ for b', a new block formed by merging block s into block b, it is necessary to verify if s and b share the same adjacent blocks. If they do, the connected edges must be integrated, and the total weight of these edges summed up, as detailed in Equation 6. We implement this operation on GPU, where each thread performs a serial merge operation and accumulates the result, as illustrated in Figure 5.

$$\begin{aligned} C^{in}_{b'} &= \sum_{i,b} \left(M_{i,b} + M_{i.s}\right) \cdot \log\left(\frac{\left(M_{i,b} + M_{i.s}\right)}{D^{in}_i \left(D^{out}_b + D^{out}_s\right)}\right) \\ C^{out}_{b'} &= \sum_{b,j} \left(M_{b,j} + M_{s,j}\right) \cdot \log\left(\frac{\left(M_{b,j} + M_{s,j}\right)}{\left(D^{in}_b + D^{in}_s\right) D^{out}_j}\right) \\ &\text{where} \quad b' = b \cup s \end{aligned} \tag{6}$$

The computations of $C^{in}b$, $C^{out}b$, $C^{in}_{b'}$ and $C^{out}_{b'}$ are all independent and can be performed across different blocks without interference. This independence allows us to utilize GPU parallelism to significantly speed up these computations. Once the calculations are completed, the ΔMDL for each block can be computed in parallel. Subsequently, we identify the best proposals for each block based on ΔMDL and then sort them in ascending order. The results are transferred back to the CPU to perform the merge operation, wherein a portion of blocks to be merged is selected based on the *num_blocks_reduction_rate*.

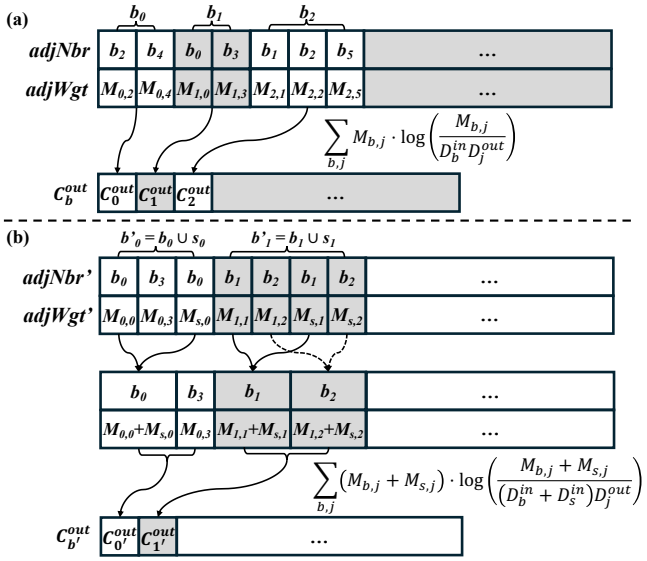

Figure 5: Illustration of how GSAP computes ΔMDL using segmented reduction and merge operations on GPU. (a) Calculating C^{out}_b for each block before merging. (b) Calculating $C^{out}_{b'}$ for each block after merging.

In the vertex-move phase, the calculation procedures for C^{in}_b and C^{out}_b remain consistent with those in the block-merge phase. However, the calculation of ΔMDL needs to be modified due to changes in connections between block b and block s resulting from the movement of a vertex v from block b to block s. The blocks of v's adjacent vertices must switch their connections from b to s. For block b, this involves removing edges that connected to the blocks of v's adjacent vertices. For block s, it necessitates adding these edges or integrating them if they already exist in block s. This is accomplished by performing serial merge operations for the block

b and s. The revised ΔMDL for the vertex-move phase is detailed in Equation 7.

$$\Delta MDL = \left(C_{b'}^{out} + C_{b'}^{in} + C_{s'}^{out} + C_{s'}^{in}\right) - \left(C_{b}^{out} + C_{b}^{in} + C_{s}^{out} + C_{s}^{in}\right)$$
$$\text{where} \quad b' = b - \{v\} \quad \text{and} \quad s' = s + \{v\} \tag{7}$$

3.4 Blockmodel Update

The blockmodel matrix is frequently updated during the block-merge and vertex-move phases in the SBP algorithm. For example, as shown in Figure 6, if block 0 is merged into block 1 or vertex 0 moves to block 0, the blockmodel matrix must be updated. To update the blockmodel matrix, which is represented as a conventional two-dimensional matrix, the random access method can be employed. In contrast, this method is not applicable to a blockmodel matrix in CSR format. In addition, the blockmodel matrix requires frequent updates due to the MCMC iterations in the vertex-move phase, which constitute the majority of the runtime in the SBP algorithm. This limitation highlights the critical need for dynamic management and updating of the blockmodel on GPU. Therefore, we propose a blockmodel update algorithm on GPU to enhance computational efficiency and address the challenges posed by the existing data structures.

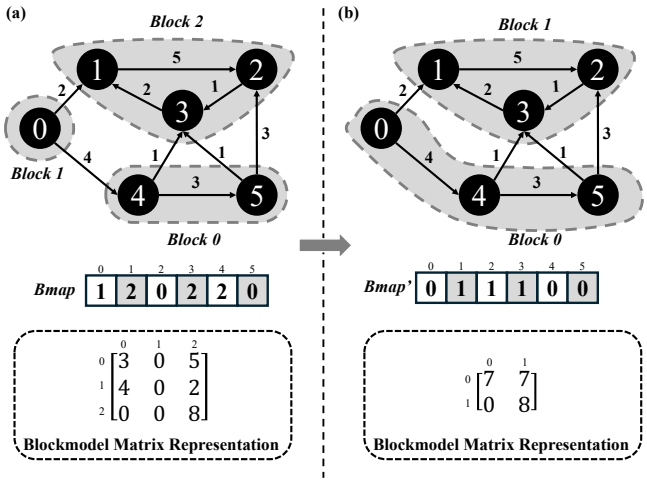

Figure 6: An example of when should update the blockmodel matrix. (a) Block 0 is merged into Block 1. (b) Vertex 0 moves to Block 0. In both cases, the blockmodel matrix must be updated accordingly.

The process of blockmodel update in GSAP begins with the graph's adjacency lists in CSR format, as shown in Step 1 of Figure 7. The second step sorts vertices according to their block IDs, stored in `Bmap`, using the `sort_by_key` operation. Subsequently, we identify each vertex's neighbors and corresponding edges, which are available in `Graph_adjNbr` and `Graph_adjWgt`, and store them in the `nbr` and `nbrWgt` arrays. The next step is to map each neighbor in the `nbr` array to its corresponding block ID using `Bmap`, and then performing the segmented sort operations on the pairs of the `nbr` and `nbrWgt`. During this stage, we detect the starting position of each subsegment for each block section. The results of these positions are stored in `subseg_hd`, which is then used to perform the reduction operation and record the value of `adjPtr` for each block. This is accomplished by utilizing CUDA warp shuffle instructions to compare data across GPU threads. The final stage consists of a segment reduction operation to produce the `adjNbr` and `adjWgt`, which form the CSR-formatted blockmodel. We derive the `adjPtr` by applying an exclusive scan operation to the `subseg_hd` array. The complete method is outlined in Algorithm 2.

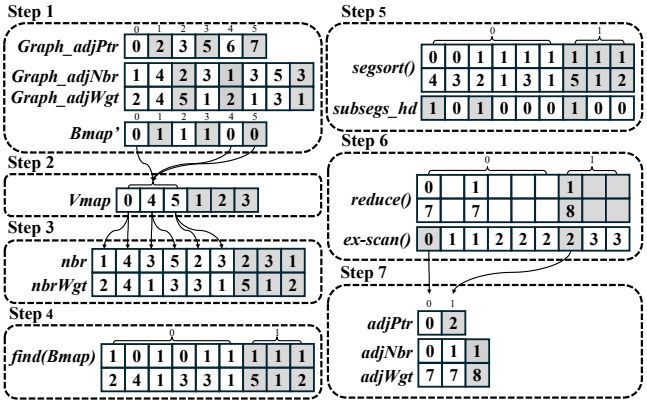

Figure 7: A detailed illustration of the blockmodel update approach based on the example in Figure 6. The final blockmodel in CSR format matches the matrix in Figure 6.

Algorithm 2: Blockmodel Update

Input: CSR arrays of the input graph G and `Bmap`
Output: CSR arrays of the current partition

```
V_map = sort_by_key({0, 1, ..., |V| − 1}, Bmap);
nbr = Graph_adjNbr[Graph_adjPtr[Vmap]];
nbrWgt = Graph_adjWgt[Graph_adjPtr[Vmap]];
find(nbr, Bmap); // Find the block ID for nbr.
segsort(nbr, nbrWgt);
subseg_hds = find_subseg_hds(nbr); // Compare
    adjacent elements to identify the starting
    index of each subsegment.
adjPtr = prefix_scan(subseg_hds);
adjWgt, adjNbr = segreduce(nbr, nbrWgt);
```

4 EXPERIMENTAL RESULTS

We evaluate the performance of GSAP on the SBPC dataset in Table 1. GSAP is implemented using C++17 and NVIDIA CUDA 12.2 and compiled with the nvcc compiler on a host compiler of GNU GCC-12.3.0 with the `-O3` optimization flag enabled. All experiments were carried out on a Ubuntu Linux 6.1.0-1022-oem x86_64 single-node machine equipped with a 14-core (20 CPU threads) 13th Gen Intel(R) Core(TM) i5-13500 operating at 2.5 GHz, 128 GB RAM, and one RTX A4000 16 GB GPU.

4.1 Baseline

We consider uSAP [2] and I-SBP [72] as baselines, which were previous winners in the 2023 SBPC. uSAP is implemented in Taskflow [32] to achieve CPU parallelism, while I-SBP is implemented using OpenMP. uSAP and I-SBP are compiled with GNU GCC-12.3.0 and executed using 20 CPU threads, which is the maximum hardware concurrency supported by our system. Table 2 lists the parameters used for SBP in uSAP, I-SBP, and GSAP.

Parameters	Values
num_blocks_reduction_rate	0.4
num_proposals	10
max_num_nodal_itr	100
delta_entropy_threshold1	0.0005
delta_entropy_threshold2	0.0001
delta_entropy_moving_avg_window	3
num_batches_for_MCMC	4

Table 2: Partitioning parameters used by I-SBP, uSAP, and GSAP.

4.2 Overall Runtime Comparison

Table 3 compares the runtime among uSAP, I-SBP (using 14 CPU cores), and GSAP (using one A4000 GPU) on the SBPC dataset. The results show that GSAP significantly outperforms both uSAP and I-SBP on nearly all graphs, except for the smallest graph containing 1K vertices. For graphs with 1K vertices, GSAP is slower than uSAP due to the overhead associated with memory allocation on GPU and data transfer between CPU and GPU. Additionally, the GPU requires setup time to initiate kernels and configure the execution environment. These overheads dominate the runtime of GSAP for the smallest graphs, leading to slower runtime. However, as graph size increases, the advantages of GSAP become more remarkable. For the largest graphs containing 1M vertices, GSAP achieves optimal partitions in about 15 minutes, whereas I-SBP and uSAP could not complete within 2 hours.

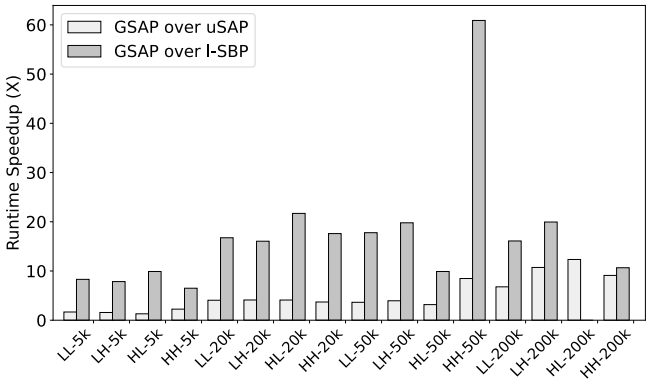

Figure 8: Runtime speedup of GSAP compared to uSAP and I-SBP on the SBPC dataset.

Figure 8 illustrates the runtime speedup of GSAP over uSAP and I-SBP on graphs ranging from 5K to 200K vertices. GSAP achieves a 12.3× speedup over uSAP under the 200K-vertex graph in the high-low category and a 60.9× speedup over I-SBP under the 50K-vertex graph in the high-high category (hardest). For the remaining graphs, GSAP is 4.5× and 14.2× faster than uSAP and I-SBP on average. The runtime advantage of GSAP comes from our stochastic proposal generation approach, parallel computation of MDL, and the blockmodel update algorithm. Figure 9 compares the runtime among GSAP, uSAP, and I-SBP in the low-low category. The runtimes for uSAP and I-SBP when processing graphs with 1M vertices are not reported, as neither algorithm completed the task within two hours.

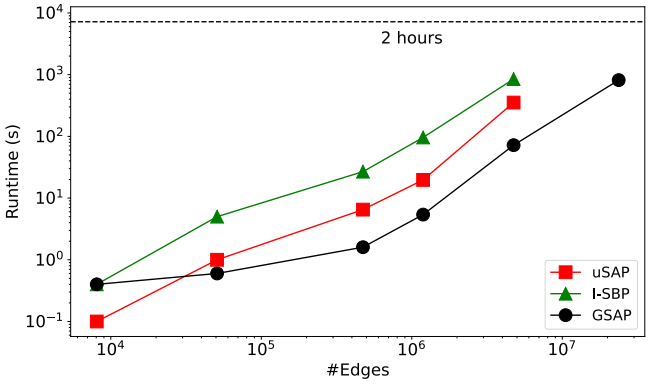

Figure 9: Runtime comparison on the low-low category.

Figure 9 demonstrates GSAP's superior runtime performance, which is attributed to the GPU's order-of-magnitude greater parallelism compared to the CPU-parallel approaches used by uSAP and I-SBP. The advantage of GSAP grows with larger edge counts, highlighting its enhanced scalability. The limitations of the CPU-parallel approaches in uSAP and I-SBP are primarily due to the iterative proposal-making process and the MDL calculation, where each block or vertex must account for every edge connected to its neighbors. This iteration over the entire graph significantly worsens the runtime for uSAP and I-SBP. In contrast, GSAP leverages the GPU's capabilities to process these tasks in parallel, substantially reducing runtime.

4.3 Runtime Breakdown

Figure 10 provides a runtime breakdown of uSAP, I-SBP, and GSAP across 50K- and 200K-vertex graphs. We observe that the runtimes for the block-merge phase in GSAP account for up to 2%, while uSAP and I-SBP reach 4.2% and 7.7%, respectively. For the 200k-vertex graph in the high-low category, the vertex-move phase in GSAP constitutes only 86% of the runtime, compared to at least 95% for uSAP and 92.3% for I-SBP. These results demonstrate that GSAP significantly reduces the time spent in both phases. GSAP does not optimize the golden-section search phase because it achieves better runtime performance on CPU.

Figure 11 shows the average runtimes for each proposal during the block-merge and vertex-move phases. This demonstrates the effectiveness of our parallel algorithm in generating stochastic proposals on GPU. On the Low-High 200K-vertex graph, the average runtime of GSAP is 19.6× and 210.3× faster than uSAP and I-SBP, respectively. This improvement is due to our parallel algorithm

	Graph Categories											
	Low - Low			Low - High			High - Low			High - High		
#Vertices	uSAP	I-SBP	GSAP	uSAP	I-SBP	GSAP	uSAP	I-SBP	GSAP	uSAP	I-SBP	GSAP
1k	**0.1s**	0.4s	0.4s	**0.1s**	1.1s	0.3s	**0.2s**	failed	0.4s	**0.2s**	1.5s	0.4s
5k	1s	5.0s	**0.6s**	1.1s	5.5s	**0.7s**	1.3s	9.9s	**1.0s**	1.8s	5.2s	**0.8s**
20k	6.5s	26.8s	**1.6s**	7.2s	28.9s	**1.8s**	8.2s	43.4s	**2.0s**	7.4s	35.2s	**2.0s**
50k	19.7s	1m36s	**5.4s**	19.4s	1m37s	**4.9s**	41.3s	1m49s	**13.0s**	44.9s	5m23s	**5.3s**
200k	5m53s	14m9s	**52s**	7m9s	13m1s	**40s**	19m45s	failed	**1m36s**	9m52s	11m33s	**1m5s**
1m	>120m	>120m	**13m36s**	>120m	>120m	**12m42s**	>120m	>120m	**15m17s**	>120m	>120m	**13m49s**

Table 3: Runtime comparison among uSAP, I-SBP (using 14 CPU cores), and GSAP (using one NVIDIA A4000 GPU) on the SBPC dataset. 'Failed' indicates the algorithm could not reach the MDL.

	Graph Categories											
	Low - Low			Low - High			High - Low			High - High		
#Vertices	uSAP	I-SBP	GSAP	uSAP	I-SBP	GSAP	uSAP	I-SBP	GSAP	uSAP	I-SBP	GSAP
1k	0.94	0.94	**0.99**	0.87	0.84	**0.92**	0.78	failed	**0.88**	0.76	0.60	**0.80**
5k	0.99	0.99	**1.00**	0.96	0.93	**0.98**	0.92	0.88	**0.95**	0.69	0.69	**0.80**
20k	1.00	0.98	1.00	0.95	**0.97**	0.96	0.98	0.92	**0.99**	0.88	0.85	**0.89**
50k	0.99	0.99	0.99	0.94	0.93	**0.95**	**0.95**	0.72	0.76	**0.82**	0.33	0.78
200k	0.90	0.90	**0.92**	0.89	0.89	0.79	**0.79**	failed	0.76	**0.85**	0.77	0.71
1m	-	-	**0.84**	-	-	**0.91**	-	-	**0.69**	-	-	**0.73**

Table 4: Graph partitioning quality comparison among uSAP, I-SBP (using 14 CPU cores), and GSAP (using one NVIDIA A4000 GPU) in terms of NMI on the SBPC dataset. "Failed" indicates that the algorithm could not reach the MDL.

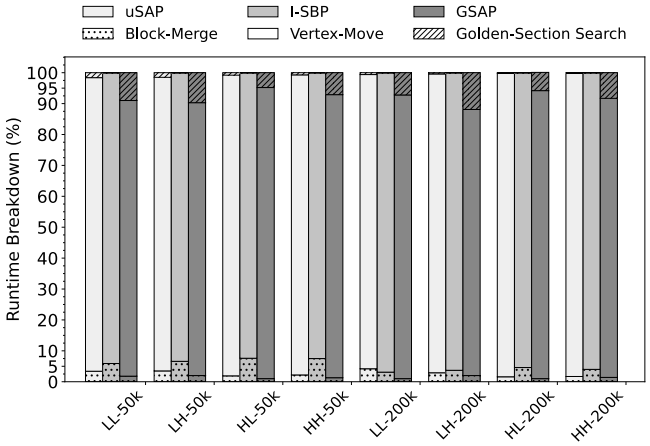

Figure 10: Runtime breakdown of uSAP, I-SBP, and GSAP.

with the pre-generated tables for stochastic proposals and GPU parallelism allows us to generate proposals concurrently. However, each proposal in uSAP and I-SBP requires examining the number of neighbors for each block or vertex and then proposing stochastic candidates, which is very time-consuming.

Figure 12 demonstrates the effectiveness of GSAP's blockmodel matrix update approach by comparing it with CPU-based updates. We observe that the speedups correlate with the increase in edge count. Notably, GSAP achieves a 31.5× speedup on Low-Low 200K-vertex graph. Iterating over all edges in such large graphs can significantly slow down the runtime on CPU. In contrast, GSAP's proposed blockmodel matrix update approach leverages the GPU's capabilities to perform updates on the CSR-formatted blockmodel matrix in parallel, resulting in substantial runtime improvements.

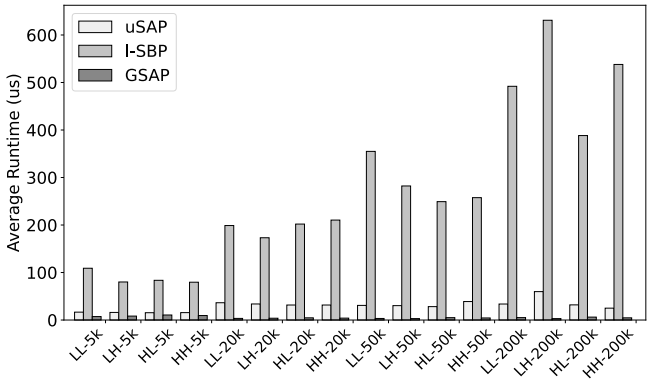

Figure 11: Average runtime of each proposal in the block-merge phase and the vertex-move phase.

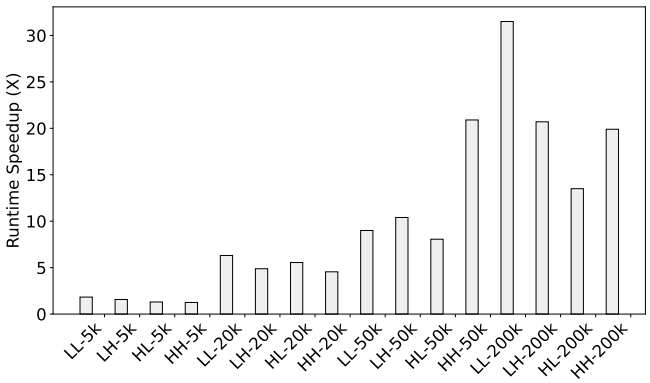

Figure 12: Runtime speedup of the proposed GPU-based blockmodel update over CPU.

4.4 Partition Quality Comparison

The normalized mutual information (NMI) is commonly used to evaluate the quality of partitioning results by comparing them with the ground truth [44]. An NMI value closer to one indicates a perfect correlation between the result and the ground truth, while a value closer to zero indicates no correlation. As shown in Table 4, GSAP generally outperforms uSAP and I-SBP across all graph categories. Specifically, GSAP achieves the highest or nearly the highest NMI scores in most cases for both 1K- and 50K-vertex graphs. For the 50K-vertex graph in the high-low category, GSAP scores 0.78, which is significantly higher than the lowest score of 0.33 achieved by I-SBP in the high-high category for 50K graphs. This implies that despite a drop in performance in this scenario, GSAP still significantly outperforms I-SBP. GSAP's effective performance is attributed to maintaining the original structure of the SBP algorithm, while introducing a parallel version of SBP that adheres to the foundational concepts as proposed in [66–68].

5 CONCLUSION

In this paper, we have introduced GSAP, a GPU-accelerated stochastic graph partitioner, that significantly enhances runtime performance of SBP. Our experimental evaluations on the 2022 GraphChallenge dataset show that GSAP achieves up to 12.3× and 60.9× runtime speedup on a single A4000 GPU compared to the state-of-the-art algorithms on 14 CPU cores. Inspired by the success of GPU computing in graph processing [1–17, 19–26, 28–31, 34–43, 46–49, 51–63, 74–76], our future work plans to incorporate GPU task parallelism using the CUDA Graph to reduce the overhead associated with launching CUDA kernels for larger graphs.

ACKNOWLEDGMENTS

This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

REFERENCES

[1] Che Chang, Tsung-Wei Huang, Dian-Lun Lin, Guannan Guo, and Shiju Lin. 2024. Ink: Efficient Incremental k-Critical Path Generation. In *ACM/IEEE DAC*.

[2] Chih-Chun Chang and Tsung-Wei Huang. 2023. uSAP: An Ultra-Fast Stochastic Graph Partitioner. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[3] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Composing Pipeline Parallelism using Control Taskflow Graph. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[4] Cheng-Hsiang Chiu and Tsung-Wei Huang. 2022. Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms. In *ACM/IEEE Design Automation Conference (DAC)*.

[5] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2021. An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads. In *International Workshop of Asynchronous Many-Task systems for Exascale (AMTE)*.

[6] Cheng-Hsiang Chiu, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Programming Dynamic Task Parallelism for Heterogeneous EDA Algorithms. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[7] Elmir Dzaka, Dian-Lun Lin, and Tsung-Wei Huang. 2023. Parallel And-Inverter Graph Simulation Using a Task-graph Computing System. In *IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSw)*.

[8] Guannan Guo, Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2020. An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints. In *ACM/IEEE Design Automation Conference (DAC)*.

[9] Guannan Guo, Tsung-Wei Huang, Y. Lin, Z. Guo, S. Yellapragada, and Martin Wong. 2023. A GPU-Accelerated Framework for Path-Based Timing Analysis. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)* (2023).

[10] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Critical Path Generation with Path Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[11] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. 2021. GPU-accelerated Path-based Timing Analysis. In *IEEE/ACM Design Automation Conference (DAC)*.

[12] Guannan Guo, Tsung-Wei Huang, and Martin D. F. Wong. 2023. Fast STA Graph Partitioning Framework for Multi-GPU Acceleration. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.

[13] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2020. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[14] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*.

[15] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2021. HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[16] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2023. Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems (TCAD)* (2023).

[17] Zizheng Guo, Tsung-Wei Huang, Jin Zhou, Cheng Zhuo, Yibo Lin, Runsheng Wang, and Ru Huang. 2024. Heterogeneous Static Timing Analysis with Advanced Delay Calculator. In *IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*.

[18] W. K. Hastings. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57, 1 (04 1970), 97–109. https://doi.org/10.1093/biomet/57.1.97

[19] Tsung-Wei Huang. 2020. A General-purpose Parallel and Heterogeneous Task Programming System for VLSI CAD. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[20] Tsung-Wei Huang. 2021. TFProf: Profiling Large Taskflow Programs with Modern D3 and C++. In *IEEE International Workshop on Programming and Performance Visualization Tools (ProTools)*.

[21] Tsung-Wei Huang. 2022. Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[22] Tsung-Wei Huang. 2023. qTask: Task-parallel Quantum Circuit Simulation with Incrementality. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[23] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. 2021. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021).

[24] Tsung-Wei Huang and Leslie Hwang. 2022. Task-parallel Programming with Constrained Parallelism. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[25] Tsung-Wei Huang, Chun-Xun Lin, , and Martin Wong. 2019. Distributed Timing Analysis at Scale. In *ACM/IEEE Design Automation Conference (DAC)*.

[26] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2018. A General-purpose Distributed Programming System using Data-parallel Streams. In *ACM Multimedia Conference (MM)*.

[27] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[28] Tsung-Wei Huang, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2019. Essential Building Blocks for Creating an Open-source EDA Project. In *ACM/IEEE Design Automation Conference (DAC)*.

[29] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2017. DtCraft: A Distributed Execution Engine for Compute-intensive Applications. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[30] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2019. DtCraft: A High-performance Distributed Execution Engine at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2019).

[31] Tsung-Wei Huang, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer v2: A Parallel Incremental Timing Analysis Engine. *IEEE Design and Test (DAT)* (2021).

[32] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2022).

[33] Tsung-Wei Huang, Dian-Lun Lin, Yibo Lin, and Chun-Xun Lin. 2022. Taskflow: A General-purpose Parallel and Heterogeneous Task Programming System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2022).

[34] Tsung-Wei Huang and Yibo Lin. 2022. Concurrent CPU-GPU Task Programming using Modern C++. In *IEEE International Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS)*.

[35] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin Wong. 2021. Taskflow: A General-purpose Parallel Task Programming System at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021).

[36] Tsung-Wei Huang and Martin Wong. 2015. OpenTimer: A High-Performance Timing Analysis Tool. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[37] Tsung-Wei Huang and Martin Wong. 2016. UI-Timer 1.0: An Ultra-Fast Path-Based Timing Analysis Algorithm for CPPR. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2016).

[38] Tsung-Wei Huang, Martin Wong, D. Sinha, K. Kalafala, and N. Venkateswaran. 2016. A Distributed Timing Analysis Framework for Large Designs. In *IEEE/ACM Design Automation Conference (DAC)*.

[39] Tsung-Wei Huang, P.-C. Wu, and Martin Wong. 2014. Fast Path-Based Timing Analysis for CPPR. In *IEEE/ACM ICCAD*.

[40] Tsung-Wei Huang, Pei-Ci Wu, and Martin D. F. Wong. 2014. UI-Timer: An ultra-fast clock network pessimism removal algorithm. In *IEEE/ACM ICCAD*.

[41] Tsung-Wei Huang, Boyang Zhang, Dian-Lun Lin, and Cheng-Hsiang Chiu. 2024. Parallel and Heterogeneous Timing Analysis: Partition, Algorithm, and System. In *ACM International Symposium on Physical Design (ISPD)*.

[42] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. GLARE: Accelerating Sparse DNN Inference Kernels with Global Memory Access Reduction. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[43] Shiu Jiang, Tsung-Wei Huang, and Tsung-Yi Ho. 2023. SNICIT: Accelerating Sparse Neural Network Inference via Compression at Inference Time on GPU. In *ACM International Conference on Parallel Processing (ICPP)*.

[44] Edward Kao, Vijay Gadepally, Michael Hurley, Michael Jones, Jeremy Kepner, Sanjeev Mohindra, Paul Monticciolo, Albert Reuther, Siddharth Samsi, William Song, et al. 2017. Streaming graph challenge: Stochastic block partition. In *2017 IEEE High performance extreme computing conference (HPEC)*. IEEE, Waltham, MA, USA, 1–12.

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

[46] Kuan-Ming Lai, Tsung-Wei Huang, and Tsung-Yi Ho. 2019. A General Cache Framework for Efficient Generation of Timing Critical Paths. In *ACM/IEEE Design Automation Conference (DAC)*.

[47] Kuan-Ming Lai, Tsung-Wei Huang, Pei-Yu Lee, and Tsung-Yi Ho. 2021. ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[48] T.-Y. Lai, Tsung-Wei Huang, , and Martin Wong. 2017. Libabs: An Effective and Accurate Macro-modeling Algorithm for Large Hierarchical Designs. In *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*.

[49] Wan Luan Lee, Dian-Lun Lin, Tsung-Wei Huang, Shui Jiang, Tsung-Yi Ho, Yibo Lin, and Bei Yu. 2024. G-kway: Multilevel GPU-Accelerated k-way Graph Partitioner. In *ACM/IEEE Design Automation Conference (DAC)*.

[50] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[51] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. A Modern C++ Parallel Task Programming Library. In *ACM Multimedia Conference (MM)*.

[52] Chun-Xun Lin, Tsung-Wei Huang, Guannan Guo, and Martin Wong. 2019. An Efficient and Composable Parallel Task Programming Library. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[53] Chun-Xun Lin, Tsung-Wei Huang, and Martin Wong. 2020. An Efficient Work-Stealing Scheduler for Task Dependency Graph. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*.

[54] Chun-Xun Lin, Tsung-Wei Huang, Ting Yu, and Martin Wong. 2018. A Distributed Power Grid Analysis Framework from Sequential Stream Graph. In *ACM Great Lakes Symposium on VLSI (GLSVLSI)*.

[55] Dian-Lun Lin and Tsung-Wei Huang. 2020. A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism. In *IEEE High-performance and Extreme Computing Conference (HPEC)*.

[56] Dian-Lun Lin and Tsung-Wei Huang. 2021. Efficient GPU Computation using Task Graph Parallelism. In *European Conference on Parallel and Distributed Computing (Euro-Par)*.

[57] Dian-Lun Lin and Tsung-Wei Huang. 2022. Accelerating Large Sparse Neural Network Inference using GPU Task Graph Parallelism. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* (2022).

[58] Dian-Lun Lin, Tsung-Wei Huang, Joshua San Miguel, and Umit Ogras. 2024. TaroRTL: Accelerating RTL Simulation using Coroutine-based Heterogeneous Task Graph Scheduling. In *International European Conference on Parallel and Distributed Computing (Euro-Par)*.

[59] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *ACM International Conference on Parallel Processing (ICPP)*.

[60] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Shih-Hsin Wang, Brucek Khailany, and Tsung-Wei Huang. 2023. GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs. In *ACM/IEEE Design Automation Conference (DAC)*.

[61] Shiju Lin, Guannan Guo, Tsung-Wei Huang, Weihua Sheng, Evangeline Young, and Martin Wong. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.

[62] Chedi Morchdi, Cheng-Hsiang Chiu, Yi Zhou, and Tsung-Wei Huang. 2024. A Resource-efficient Task Scheduling System using Reinforcement Learning. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.

[63] McKay Mower, Luke Majors, and Tsung-Wei Huang. 2021. Taskflow-San: Sanitizing Erroneous Control Flow in Taskflow Programs. In *IEEE Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*.

[64] Mark EJ Newman. 2006. Finding community structure in networks using the eigenvectors of matrices. *Physical review E* 74, 3 (2006), 036104.

[65] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Physical review E* 69, 2 (2004), 026113.

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

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

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

[69] Ahsen J. Uppal, Jaeseok Choi, Thomas B. Rolinger, and H. Howie Huang. 2021. Faster Stochastic Block Partition Using Aggressive Initial Merging, Compressed Representation, and Parallelism Control. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–7. https://doi.org/10.1109/HPEC49654.2021.9622836

[70] Frank Wanye, Vitaliy Gleyzer, and Wu-chun Feng. 2019. Fast Stochastic Block Partitioning via Sampling. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–7. https://doi.org/10.1109/HPEC.2019.8916542

[71] F. Wanye, V. Gleyzer, E. Kao, and W. Feng. 2023. Exact Distributed Stochastic Block Partitioning. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. 25–36.

[72] Frank Wanye, Vitaliy Gleyzer, Edward Kao, and Wu-chun Feng. 2023. An Integrated Approach for Accelerating Stochastic Block Partitioning. In *2023 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[73] Frank Wanye, Vitaliy Gleyzer, Edward Kao, and Wu-chun Feng. 2023. On the Parallelization of MCMC for Community Detection. In *Proceedings of the 51st International Conference on Parallel Processing*. Article 87, 13 pages.

[74] Yasin Zamani and Tsung-Wei Huang. 2021. A High-Performance Heterogeneous Critical Path Analysis Framework. In *IEEE High-Performance Extreme Computing Conference (HPEC)*.

[75] Boyang Zhang, Dian-Lun Lin, Che Chang, Cheng-Hsiang Chiu, Bojue Wang, Wan Luan Lee, Chih-Chun Chang, Donghao Fang, and Tsung-Wei Huang. 2024. G-PASTA: GPU Accelerated Partitioning Algorithm for Static Timing Analysis. In *ACM/IEEE DAC*.

[76] Kexing Zhou, Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. 2022. Efficient Critical Paths Search Algorithm using Mergeable Heap. In *IEEE/ACM Asia and South Pacific Design Automation Conference (ASP-DAC)*.