# uSAP: An Ultra-Fast Stochastic Graph Partitioner

Chih-Chun Chang
*Dept. of Electrical and Computer Engineering*
*University of Wisconsin-Madison*
chih-chun.chang@wisc.edu

Tsung-Wei Huang
*Dept. of Electrical and Computer Engineering*
*University of Wisconsin-Madison*
tsung-wei.huang@wisc.edu

***Abstract*—Stochastic graph partitioning (SGP) plays a crucial role in many real-world applications, such as social network analysis and recommendation systems. Unlike the typical combinatorial graph partitioning problem, SGP presents unique computational difficulties due to time-consuming sampling processes. To address this challenge, the recent HPEC launched the Stochastic Graph Partitioning Challenge (SGPC) to seek novel solutions from the high-performance computing community. Despite many SGP algorithms over the last few years, their speed-ups are not remarkable because of various algorithm limitations. Consequently, we propose uSAP, an ultra-fast stochastic graph partitioner to largely enhance SGP performance. uSAP introduces a novel strongly connected component-based initial block merging strategy to reduce the number of partitioning iterations significantly. To further improve the runtime and memory performance, uSAP adopts a dynamic batch parallel nodal block assignment algorithm and a dynamic matrix representation. We have evaluated uSAP on the 2022 official HPEC SGPC benchmarks. The results demonstrate the promising performance of uSAP on graphs of different sizes and complexities. For example, uSAP achieves $129.4\times$ speed-up over the latest champion on a graph of 50K nodes.**

## I. Introduction

Graph data has become increasingly important in various real-world applications, such as social network analysis and recommendation systems. Graph partitioning can be utilized to analyze such data, dividing the graph into meaningful communities. This enables us to gain valuable insights into the interaction and relationship among the nodes. Among various graph partitioning techniques, *Stochastic Graph Partitioning (SGP)* [1] characterizes the real-world graphs by utilizing Bayesian inferential methods based on the degree-corrected stochastic blockmodel [2]. Unlike the typical combinatorial graph partitioning problem, SGP presents unique computational challenges caused by time-consuming sampling processes. As a result, the recent HPEC launched the Stochastic Graph Partitioning Challenge (SGPC) to seek innovative acceleration methods from the high-performance computing (HPC) community [1].

SGPC provides a baseline sequential partitioner, namely *PEIXOTO* which is developed by [3]–[5]. The baseline algorithm utilizes a Fibonacci search approach [6] to explore different numbers of blocks. It determines the optimal partition by minimizing the overall entropy of a partitioned graph. Each node is initially assigned a unique block and merged with others at each search step, called *block merging*. Subsequently, iterative sequential Monte Carlo Markov Chain (MCMC) updates are applied to assign each node a block, namely *nodal block assignment*. To evaluate the performance of a partitioner over PEIXOTO, SGPC provides a rigorous evaluation metric on synthetic graphs based on the stochastic model in [2]. The model samples the connection between nodes from a Poisson distribution with a correction term to emulate real-world graph characteristics. Table I lists the characteristics of four categories of these graphs from 1K to 200K nodes. The four categories present different levels of partitioning difficulties or graph complexities:

(1) Low block overlap, low block size variation (Low-Low)
(2) Low block overlap, high block size variation (Low-High)
(3) High block overlap, low block size variation (High-Low)
(4) High block overlap, high block size variation (High-High)

| | $\lvert V\rvert$ | $\lvert E\rvert$ | Density | #Blocks | Size |
|---|---|---|---|---|---|
| Low-Low | 1K | 8,067 | $8.0\times10^{-3}$ | 11 | 78 KB |
| | 5K | 50,850 | $2.0\times10^{-3}$ | 19 | 574 KB |
| | 20K | 473,914 | $1.2\times10^{-3}$ | 32 | 6 MB |
| | 50K | 1,189,382 | $4.8\times10^{-4}$ | 44 | 16 MB |
| | 200K | 4,750,333 | $1.2\times10^{-4}$ | 71 | 68 MB |
| Low-High | 1K | 7,892 | $7.9\times10^{-3}$ | 11 | 76 KB |
| | 5K | 50,544 | $2.0\times10^{-3}$ | 19 | 571 KB |
| | 20K | 476,386 | $1.2\times10^{-3}$ | 32 | 6 MB |
| | 50K | 1,193,994 | $4.8\times10^{-4}$ | 44 | 16 MB |
| | 200K | 4,747,902 | $1.2\times10^{-4}$ | 71 | 68 MB |
| High-Low | 1K | 7,903 | $7.9\times10^{-3}$ | 11 | 77 KB |
| | 5K | 51,091 | $2.0\times10^{-3}$ | 19 | 578 KB |
| | 20K | 475,421 | $1.2\times10^{-3}$ | 32 | 6 MB |
| | 50K | 1,183,975 | $4.8\times10^{-4}$ | 44 | 16 MB |
| | 200K | 4,743,468 | $1.2\times10^{-4}$ | 71 | 68 MB |
| High-High | 1K | 8,032 | $8.0\times10^{-3}$ | 11 | 77 KB |
| | 5K | 51,157 | $2.0\times10^{-3}$ | 19 | 578 KB |
| | 20K | 473,329 | $1.2\times10^{-3}$ | 32 | 6 MB |
| | 50K | 1,187,682 | $4.8\times10^{-4}$ | 44 | 16 MB |
| | 200K | 4,754,406 | $1.2\times10^{-4}$ | 71 | 68 MB |

TABLE I: The four categories present different levels of partitioning difficulties provided by the 2022 SGPC [1].

However, PEIXOTO is extremely time-consuming due to its bottom-up merging strategy and iterative MCMC updates, both of which require a large number of iterations that run *sequentially*. As a result, the algorithm struggles to scale effectively for large graphs. To tackle this problem, SGPC has yielded many solutions [7]–[9], while their speed-ups are not remarkable due to various algorithm limitations (discussed later). For example, the latest SGPC champion [9] only reports up to $3.78\times$ speed-up for a 50K-node graph and cannot complete larger graphs in a reasonable amount of time (e.g., 10hrs).

Consequently, we propose uSAP, an ultra-fast stochastic graph partitioner, to substantially improve the performance of SGP that was previously out of reach. uSAP introduces a novel strongly connected component(SCC)-based initial block merging strategy to largely reduce the partitioning iterations. To further enhance the sampling performance, uSAP adopts a dynamic batch parallel nodal block assignment algorithm. In addition to runtime improvement, uSAP employs a dynamic matrix representation to reduce the memory footprint. We have evaluated uSAP on the 2022 official benchmarks of HPEC SGPC. The results demonstrate the promising performance of uSAP on different graph sizes and categories. For example, uSAP achieves $129.4\times$ speed-up over the latest champion on the graph of 50K nodes. We have made uSAP open-source to facilitate high-performance graph partitioning research [1].

## II. State of the Art

To overcome the scalability challenge, Distributed Sketches [7], the 2020 SGPC champion, proposes the matrix sketches derived from random dimension-reducing projections. They demonstrate the excellent scalability in distributed memory of the linear sketch embedding algorithm. However, the result shows that the pairwise precision and recall (i.e., accuracy) are not promising when partitioning large graphs. Also, dealing with high-degree vertices is still challenging because of the algorithm's communication and computation bottlenecks.

DPGS [8], the honorable mention of 2021 SGPC, introduces a graph summarization technique that preserves the community structure of the graph while reducing its complexity. The result indicates that their algorithm runs faster than PEIXOTO, but the pairwise precision and recall are not significantly better than those achieved by PEIXOTO.

Faster Stochastic Block Partition (FSBP) [9], the 2021 SGPC champion, proposes an aggressive initial merging strategy to considerably decrease the initial block count at the first searching iteration, which in turn reduces the total number of partitioning iterations. Its parallelism control strategy carefully manage the amount of parallelism during different phases of the algorithm to improve the performance. However, the aggressive initial merging strategy may merge blocks that cause substantial changes in entropy, resulting in negative effects on the accuracy. Also, its parallelism control strategy does not perform well due to the synchronization overhead.

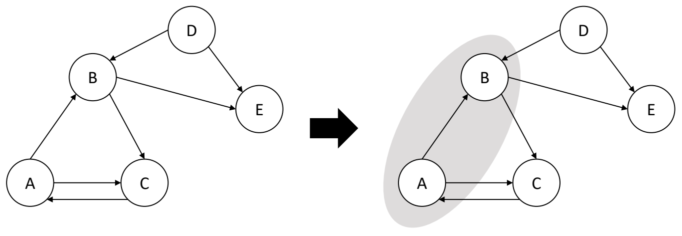
## III. uSAP

To overcome the performance bottleneck of existing partitioners, we present uSAP, an ultra-fast stochastic graph partitioner. We implement uSAP in a task graph programming model to accelerate SGP through an SCC-based initial block merging strategy, a dynamic batch parallel nodal block assignment, and a dynamic matrix representation.

[1]uSAP source code: https://github.com/gary30404/uSAP

### A. SCC-based Initial Block Merging Strategy

In order to reduce the total number of partitioning iterations, we merge the SCC of the graph into blocks in advance. This approach substantially reduces the initial block number and significantly enhances the algorithm's efficiency. In contrast to the greedy strategy of FSBP, which aggressively increases the number of blocks to be merged, our method focuses on merging blocks with stronger connections. The idea is inspired by the degree-corrected stochastic blockmodel [2], which implies that nodes within the same block exhibit stronger connections than nodes across different blocks. The merging scenario, as shown in Figure 1, represents a possible solution of PEIXOTO and FSBP. Merging node B with node A causes an entropy change of $0.35$ according to the entropy definition in [1]. However, merging node C into node A yields a smaller entropy change of $-0.43$, as shown in Figure 2. This indicates that merging node C into node A is a more favorable choice than merging node B into node A because node A and node C are strongly connected. Based on the observation, we adopt the SCC finding algorithm, which has a linear-time complexity, to identify the SCC and guide the initial merging strategy.

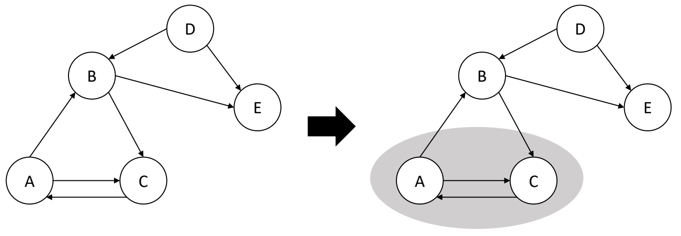


$$M^- = \begin{array}{c} \\ \end{array}\begin{array}{ccccc|c} A & B & C & D & E & \\ 0 & 1 & 1 & 0 & 0 & A \\ 0 & 0 & 1 & 0 & 1 & B \\ 1 & 0 & 0 & 0 & 0 & C \\ 0 & 1 & 0 & 0 & 1 & D \\ 0 & 0 & 0 & 0 & 0 & E \end{array}$$

$$M_1^+ = \begin{array}{ccccc|c} A & B & C & D & E & \\ 1 & 0 & 2 & 0 & 1 & A \\ 0 & 0 & 0 & 0 & 0 & B \\ 1 & 0 & 0 & 0 & 0 & C \\ 1 & 0 & 0 & 0 & 1 & D \\ 0 & 0 & 0 & 0 & 0 & E \end{array}$$

Fig. 1: Illustration of merging node B into node A (a possible solution of PEIXOTO and FSBP) and the resulting block edge count matrix used to calculate the change in entropy ($\Delta E = 0.35$ from [1]).

In our SCC-based initial block merging strategy, we introduce an initial block size threshold denoted as $t_{\text{SCC}}$ to limit the block size accordingly instead of finding all the SCC in the graph. The threshold can be adjusted to accommodate the different complexities of graphs. The overall procedure described in Algorithm 1 begins by applying a depth-first search to obtain the traversal sequence stored in *stack*. Subsequently, we transpose the input graph and perform the second depth-first search on the transposed graph according to the traversal sequence. Once a node is popped from the traversal sequence, its neighbors are identified as the same block. We continue to find the descendants of these neighbors until the step counter exceeds $t_{\text{SCC}}$ or no descendant exists. At this point, an initial block is found. The nodes within the block are then marked as finished, and the step counter is reset in preparation for finding the subsequent initial blocks.



$$M^{-} = \begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{cccccc} A & B & C & D & E & \\ 0 & 1 & 1 & 0 & 0 & A \\ 0 & 0 & 1 & 0 & 1 & B \\ 1 & 0 & 0 & 0 & 0 & C \\ 0 & 1 & 0 & 0 & 1 & D \\ 0 & 0 & 0 & 0 & 0 & E \end{array}$$

$$M_2^{+} = \begin{array}{cccccc} A & B & C & D & E & \\ 2 & 1 & 0 & 0 & 0 & A \\ 1 & 0 & 0 & 0 & 1 & B \\ 0 & 0 & 0 & 0 & 0 & C \\ 0 & 1 & 0 & 0 & 1 & D \\ 0 & 0 & 0 & 0 & 0 & E \end{array}$$

Fig. 2: Illustration of merging node C into node A (the solution of uSAP) and the resulting block edge· count matrix used to calculate the change in entropy (a smaller $\Delta E = -0.43$ than Figure 1).

**Algorithm 1:** Initial-Block-Merging

```
Input: $A$: adjacency list of the input graph
Output: $\Gamma$: block assignment for each node
$block\_id \leftarrow 0$
$B \leftarrow 0$
$finished \leftarrow \{$false$\}$
$stack \leftarrow$ DFS$(A)$
$A' \leftarrow$ Transpose$(A)$
while not $stack.empty()$ do
    $v \leftarrow stack.pop()$
    if not $finished[v]$ then
        $step\_counter \leftarrow 0$
        $stack2 = \{\}$
        $stack2.push(v)$
        while not $stack2.empty()$ do
            if $step\_counter \geq t_{SCC}$ then
                break
            end
            $n \leftarrow stack2.pop()$
            if not $finished[n]$ then
                $finished[n] \leftarrow$ true
                $\Gamma[n] \leftarrow block\_id$
                $++step\_counter$
                for $k \in A'[n]$ do
                    if not $finished[k]$ then
                        $stack2.push(k)$
                    end
                end
            end
        end
        $++block\_id$
        $++B$
    end
end
```

## B. Dynamic Batch Parallel Nodal Block Assignment

To prevent the time-consuming MCMC updates in the early partitioning stage, we propose a dynamic approach to decide when to perform parallel nodal block assignment. We define $t_B$ as the threshold value. When the current number of blocks exceeds the value $\frac{N}{t_B}$, uSAP exclusively performs block merging ($N$ represents the total number of nodes in a graph, and $B$ represents the number of blocks at the current stage). Otherwise, uSAP switches to perform both block merging and nodal block assignment. By dynamically adjusting the level of granularity in the graph partitioning process, uSAP strikes a balance between computational efficiency and accuracy, leading to improved overall performance.

We randomly select a batch of nodes to parallelize the nodal block assignment, as shown in Figure 3. Each thread is assigned a specific set of nodes and performs block assignment independently based on the shared state of the current partitioning result. After completing the computation of the batch, the global shared state is updated, and the overall entropy is calculated to determine whether to continue the nodal block assignment. The parallelization significantly accelerates the time-consuming MCMC updates without affecting accuracy. The foundation to support this argument can be referred to [10], [11].

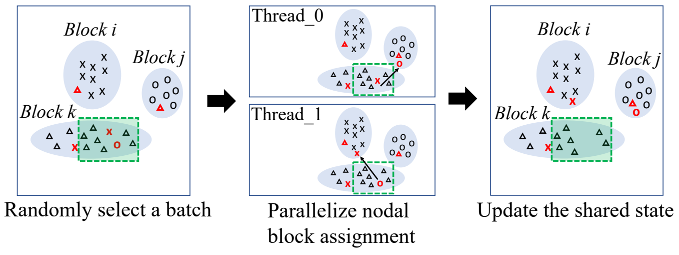


Fig. 3: Illustration of batch parallel nodal block assignment.

## C. Dynamic Matrix Representation

To increase the memory efficiency of uSAP, we introduce a threshold parameter denoted as $t_M$ to determine the choice between using an adjacency matrix or an adjacency list for representing the block edge count data. The adjacency matrix representation offers the advantage of fast random access to the data, while it is memory inefficient due to the sparse nature of the graph. Additionally, it is time-consuming to iterate a large sparse matrix for computing entropy. On the other hand, the adjacency list provides a more memory-efficient storage option. However, random access to the block edge count matrix is essential for accelerating the entropy calculation. This is because using only the adjacency list representation requires more iterations. As a result, we use $t_M$ to make an informed decision on when to use the adjacency matrix and the adjacency list. This allows uSAP to strike a balance between memory utilization and computational efficiency.

## D. Task Graph Parallelism

To maximize the parallelism of uSAP, we leverage Taskflow [12] to describe our algorithms in a task dependency graph, where dependent tasks and parallel algorithms can be scheduled by the Taskflow runtime across different CPUs with dynamic load balancing. Furthermore, as uSAP incorporates many iterative control flows in the Fibonacci search and MCMC updates, the control taskflow graph (CTFG) programming model of Taskflow allows us to express end-to-end parallelism by integrating control flow into our task graph, largely reducing the threading and synchronization overheads. More details about CTFG and its successful applications can be referred to [12]–[46].

We depict the task graph of uSAP in Figure 4 and present the corresponding overall algorithm in Algorithm 2. uSAP begins with the Initial-Block-Merging (line 1), where the SCC-based initial block merging strategy is employed. This step is followed by initializing block edge count data based on the initial merged blocks. Next, a Fibonacci search (line 4) consists of two steps. The first step is the parallel Block-Merging (line 6), and the second step is batch parallel Nodal-Block-Assignment (line 14). The latter begins with Fetch-Batches (line 10) to obtain batches of nodes and terminates based on the result of Check-Convergence (line 17). Finally, the Prepare-for-Next (line 21) examines the result of the Fibonacci search to determine if the optimal partition has been achieved.

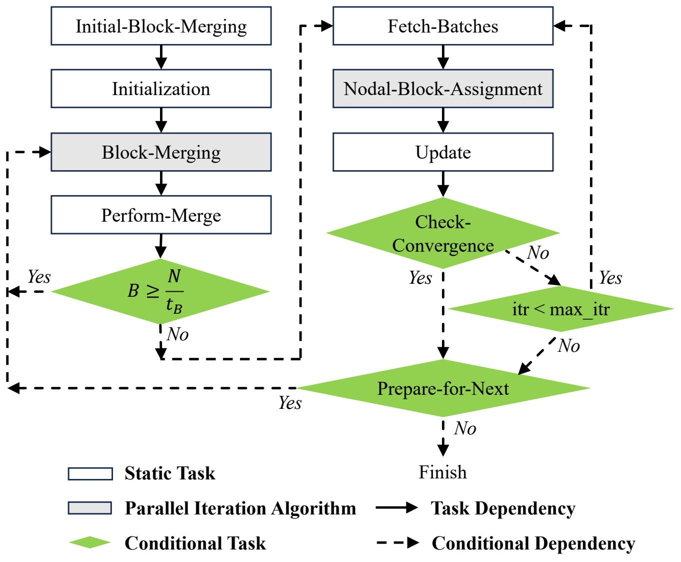


Fig. 4: uSAP leverages Taskflow's control taskflow graph programming model [12] to describe the entire algorithm in an end-to-end taskflow graph. Solid arrows represent a regular task dependency and dashed arrows represent a conditional dependency.

## IV. Experimental Results

We evaluate the performance of uSAP using the official 2022 SGPC dataset. We focus on static graphs under four categories of different graph complexity, as listed in Table I. All experiments were conducted on an Ubuntu Linux 5.15.0-58-generic x86_64 machine. The single-node machine was equipped with a 12-core Intel(R) Core(TM) i7-12700 processor running at 3.4 GHz, with 32GB RAM. We compile all the programs on GNU GCC-11.3.0 with the C++17 standard `-std=c++17` and optimization flags `-O3` enabled. For parameter settings, we use 20 for $t_{\mathrm{SCC}}$, 20 for $t_{\mathrm{B}}$, 1024 for $t_{\mathrm{M}}$, and 0.3 for `num_block_reduction_rate`.

**Algorithm 2:** uSAP

```
   Input: $A$: adjacency list of the input graph
   Output: $\Gamma$: block assignment for each node
1  $\Gamma \leftarrow$ Initial-Block-Merging($A$)
2  $M \leftarrow$ Initialization($\Gamma$)
3  $optimal \leftarrow$ false
4  while not $optimal$ do
5      do in parallel
6          $\Gamma \leftarrow$ Block-Merging($M$)
7      end
8      Perform-Merge($\Gamma$)
9      if $B < \frac{N}{t_B}$ then
           // $N$: the total number of node is a graph
           // $B$: the number of block at the current stage
10         $batches \leftarrow$ Fetch-Batches($A$)
11         for $itr \leftarrow 0; itr < max\_itr; ++itr$ do
12             for $batch \in batches$ do
13                 do in parallel
14                     $\Gamma \leftarrow$ Nodal-Block-Assignment($A$)
15                 end
16                 $\Gamma, M \leftarrow$ Update($\Gamma$)
17                 Check-Convergence($M$)
18             end
19         end
20     end
21     $optimal \leftarrow$ Prepare-for-Next($\Gamma$)
22 end
```

### A. Baseline

We consider two baseline implementations: (1) PEIXOTO sequential partitioner provided by SGPC and (2) FSBP [9] which is the latest champion of SGPC. The aggressive initial merging rate of FSBP is set to 0.75, which is the same as [9]. In addition, we ran FSBP with eight threads, where it achieved the best performance on our machine. Using more threads does not provide any further performance advantage due to its synchronization overhead. In terms of `num_block_reduction_rate`, we use 0.5 for PEIXOTO and FSBP. The other common partition parameters used by PEIXOTO, FSBP, and uSAP are listed in Table II.

| *Update Parameters* | *Values* |
|---|---|
| `num_agg_proposals_per_block` | 10 |
| `max_num_nodal_itr` | 100 |
| `delta_entropy_threshold1` | $5.0 \times 10^{-4}$ |
| `delta_entropy_threshold2` | $1.0 \times 10^{-4}$ |
| `delta_entropy_moving_avg_window` | 3 |

TABLE II: Common partition parameters used by uSAP, FSBP, and PEIXOTO.

### B. Performance Comparison

Table III compares the runtime performance and memory usage among PEIXOTO, FSBP, and uSAP across different graph sizes and categories. The results clearly demonstrate that uSAP significantly outperforms PEIXOTO and FSBP on all graphs. For example, uSAP is about 1700× faster than FSBP for the graph of 1K nodes under the low-low category. For the same graph category of 5K, 20K, and 50K nodes, uSAP is about 80.2×, 103.3×, and 129.4× faster than FSBP, respectively. When partitioning the largest graph of 200K nodes under the high-high category (the most complicated graph), uSAP can finish in 23 minutes, while PEIXOTO and FSBP fail to complete within 10 hours. Similar data can be observed in the other three categories as well.

The runtime advantage of uSAP is a combination of our SCC-based initial merging strategy, the dynamic batch parallel nodal update approach, and the task graph parallelism. The SCC-based initial merging strategy significantly reduces the number of blocks before the block merging stage, resulting in much fewer partitioning iterations. The results shown in Figure 5 demonstrate that uSAP outperforms PEIXOTO and FSBP by merging up to 80% of nodes into blocks initially, leading to a large reduction in the number of partitioning iterations.

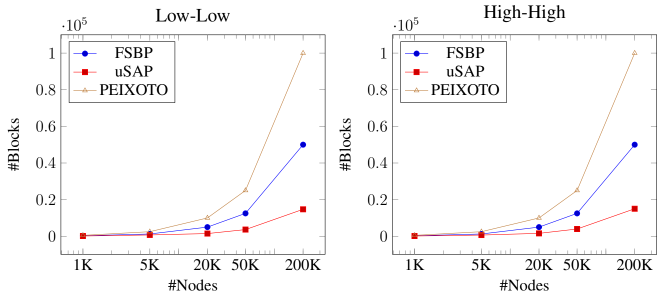


Fig. 5: Number of blocks after the initial merge.

In terms of memory usage, uSAP outperforms PEIXOTO and FSBP on all graphs. For example, uSAP consumes 7.1× less memory than FSBP and 12.55× less than PEIXOTO on the 1K-node low-low graph. For the 20K-node low-low graph, uSAP consumes 2.99× less memory than FSBP and 16.4× than PEIXOTO. The memory efficiency of uSAP can be attributed to its dynamic matrix representation, which efficiently manages memory by employing the adjacency list when the block count $B$ exceeds $t_{\mathrm{M}}$. During this phase, the number of blocks is considerably high and the matrix sparsity is low. The adjacency list representation is more suitable because it only stores the connections between blocks. Conversely, when the block count is below $t_{\mathrm{M}}$, uSAP switches to the adjacency matrix representation for fast random access. This allows uSAP to reach a balance between memory utilization and computational efficiency.

Table IV compares the accuracy in terms of pairwise precision (denoted as *PP*) and pairwise recall (denoted as *PR*) on the four different categories. We observe that uSAP outperforms PEIXOTO and FSBP on nearly all graphs because the SCC-based initial block merging strategy minimizes the change in entropy, and this helps maintain PP and PR. In contrast, the aggressive initial merging approach used in FSBP can harm the accuracy. This approach can significantly change the entropy and degrade PP and PR. Compared to PEIXOTO, our uSAP algorithm incorporates the batch parallel nodal block assignment inspired by [10], [11]. This technique enables the concurrent processing of multiple nodes, leveraging parallelism to accelerate the MCMC updates without compromising the PP and PR. For the high-high 20K-node graph, uSAP exhibits a bit lower PP and PR than PEIXOTO and FSBP. This is because the threshold $t_{\mathrm{B}}$ is fixed across all four categories. Yet, for more complicated graphs like the high-high category, a higher level of granularity in the nodal block assignment is necessary to achieve optimal PP and PR values. Thus, we leave the threshold $t_{\mathrm{B}}$ a tunable parameter where applications can fine-tune it to improve the PP and PR (e.g., reducing $t_{\mathrm{B}}$ for more fine-grained updates).

## C. Scalability

We compare the scalability between uSAP and FSBP over increasing numbers of threads on partitioning the 50K-node graph under the different categories. We do not report the scalability for 200K-node graphs because FSBP cannot complete within a reasonable amount of time. The execution time reported in Figure 6 is presented on a logarithmic scale of base 10. We can observe that uSAP significantly outperforms FSBP, regardless of the number of threads. Regarding scalability, FSBP achieves its best performance when utilizing eight threads. However, when the number of threads exceeds eight, the performance of FSBP begins to degrade. This degradation in performance is due to the lack of an effective scheduling algorithm in FSBP to handle the synchronization overhead that arises with a larger number of threads. To solve this problem, uSAP leverages Taskflow [12] to program our partitioning algorithms in a scalable task dependency graph, where parallel and dependent tasks can be efficiently scheduled by the Taskflow runtime over different CPUs with dynamic load balancing. For example, the runtime of uSAP with eight threads is approximately 8× faster than that of one thread in Figure 6. This significant improvement demonstrates the superior scalability of uSAP, allowing uSAP to effectively utilizes CPUs to achieve performance enhancements for SGP.

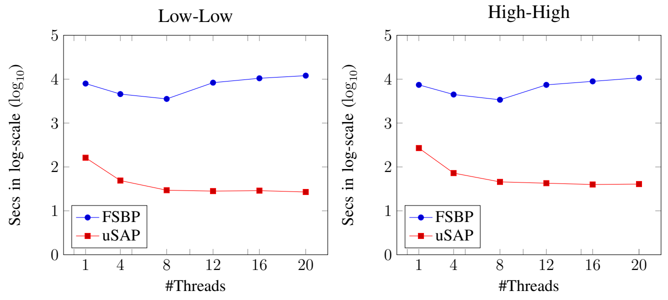


Fig. 6: Scaling results under different numbers of threads of uSAP and FSBP on 50K-node graphs.

## D. Effect of Dynamic Matrix Representation Strategy

We study the effect of our dynamic matrix representation strategy under different $t_{\mathrm{M}}$ values. In Table V, we only consider the effect of the dynamic matrix representation without other optimization mentioned in this paper. Using adjacency lists alone takes about 0.45s, 4.9s, 60s, and 363s to partition the four

| | Static Graph Categories | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Low-Low | | | Low-High | | | High-Low | | | High-High | | |
| Nodes | PEIXOTO | FSBP | uSAP | PEIXOTO | FSBP | uSAP | PEIXOTO | FSBP | uSAP | PEIXOTO | FSBP | uSAP |
| | sec | sec | sec | sec | sec | sec | sec | sec | sec | sec | sec | sec |
| | MB | MB | MB | MB | MB | MB | MB | MB | MB | MB | MB | MB |
| 1K | 47.3 | 8.5 | **0.005** | 46.8 | 8.6 | **0.006** | 59.9 | 9.9 | **0.007** | 58.9 | 10.1 | **0.007** |
| | 86.6 | 49.1 | **6.9** | 86.4 | 48.0 | **7.3** | 86.7 | 49.3 | **7.4** | 86.6 | 48.8 | **7.2** |
| 5K | 470.0 | 56.2 | **0.7** | 522.8 | 70.5 | **0.6** | 596.5 | 75.8 | **1.2** | 550.7 | 71.3 | **1.1** |
| | 244.6 | 78.9 | **13.6** | 244.7 | 78.5 | **15.7** | 249.9 | 78.6 | **14.3** | 239.1 | 79.3 | **16.7** |
| 20K | 5305.5 | 640.6 | **6.2** | 5450.9 | 641.2 | **7.0** | 5681.6 | 689.2 | **10.4** | 5176.6 | 607.7 | **10.1** |
| | 2074.4 | 378.2 | **126.1** | 1951.7 | 386.4 | **136.1** | 2073.3 | 385.3 | **136.1** | 2033.6 | 382.0 | **132.6** |
| 50K | >36000 | 3558.6 | **27.5** | >36000 | 3462.6 | **27.4** | >36000 | 3381.4 | **53.5** | >36000 | 3419.2 | **41.4** |
| | - | 942.6 | **318.9** | - | 981.5 | **310.5** | - | 947.3 | **310.5** | - | 970.8 | **322.5** |
| 200K | >36000 | >36000 | **299.8** | >36000 | >36000 | **338.8** | >36000 | >36000 | **1243.8** | >36000 | >36000 | **1381.2** |
| | - | - | **1222.6** | - | - | **1228.8** | - | - | **1228.8** | - | - | **1334.5** |

TABLE III: Overall runtime performance (seconds) and memory efficiency (MBs) of PEIXOTO (SGPC sequential baseline), FSBP (2021 champion), and uSAP on static graphs of 2022 HPEC SGPC datasets [1].

| | Static Graph Categories | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Low-Low | | | Low-High | | | High-Low | | | High-High | | |
| Nodes | PEIXOTO | FSBP | uSAP | PEIXOTO | FSBP | uSAP | PEIXOTO | FSBP | uSAP | PEIXOTO | FSBP | uSAP |
| | PP | PP | PP | PP | PP | PP | PP | PP | PP | PP | PP | PP |
| | PR | PR | PR | PR | PR | PR | PR | PR | PR | PR | PR | PR |
| 1K | 0.994 | 0.994 | **0.998** | 0.939 | 0.811 | **0.952** | 0.717 | 0.719 | **0.747** | 0.686 | 0.710 | **0.843** |
| | 0.996 | 0.996 | **0.998** | 0.990 | **0.995** | 0.993 | 0.883 | 0.878 | **0.971** | **0.972** | 0.653 | 0.954 |
| 5K | 0.940 | **1.000** | **1.000** | 0.976 | 0.970 | **0.987** | 0.861 | 0.641 | **0.983** | 0.676 | 0.689 | **0.861** |
| | **1.000** | **1.000** | **1.000** | 0.769 | 0.850 | **0.998** | 0.816 | 0.666 | **0.997** | 0.789 | 0.721 | **0.801** |
| 20K | 0.984 | **1.000** | **1.000** | 0.721 | 0.921 | **0.948** | 0.950 | 0.875 | **0.982** | **0.889** | 0.789 | 0.803 |
| | **1.000** | **1.000** | **1.000** | 0.671 | 0.479 | **0.999** | 0.740 | 0.705 | **0.999** | **0.995** | 0.943 | 0.778 |
| 50K | - | 0.988 | **1.000** | - | 0.849 | **0.921** | - | 0.757 | **0.946** | - | **0.766** | 0.456 |
| | - | 0.855 | **1.000** | - | 0.997 | **0.998** | - | 0.862 | **0.994** | - | 0.64 | **0.981** |
| 200K | - | - | **0.983** | - | - | **0.768** | - | - | **0.832** | - | - | **0.386** |
| | - | - | **1.000** | - | - | **0.922** | - | - | **0.314** | - | - | **0.885** |

TABLE IV: Comparison of the pairwise precision (PP) and pairwise recall (PR) values among PEIXOTO, FSBP, and uSAP of the four categories.

| $\lvert V\rvert$ | Adjacency List | Dynamic Matrix Representation ($t_{\mathrm{M}}$) | | | |
|---|---|---|---|---|---|
| | | 512 | 1024 | 2048 | 4096 |
| 1K | 0.45 | 0.43 | 0.42 | 0.44 | 0.43 |
| 5K | 4.9 | 3.6 | 3.7 | 4.1 | 4.7 |
| 20K | 60.2 | 46.1 | 43.1 | 43.3 | 45.7 |
| 50K | 363.0 | 291.6 | 271.9 | 273.5 | 278.8 |

TABLE V: Comparison of runtime (secs) under different $t_{\mathrm{M}}$ in low-low graphs.

low-low graphs 1K, 5K, 20K, and 50K, respectively. With our proposed dynamic strategy, we observe that the best runtime improvements can achieve $19.6\% - 25\%$ on the 50K-node graph under different $t_{\mathrm{M}}$ (512, 1024, 2048, and 4096). The value of $1024$ strikes a balance between memory access time and the number of iterations to read each row of the matrix for calculating the entropy.

## V. Conclusion and Future Works

In this paper, we have introduced uSAP, an ultra-fast stochastic graph partitioner targeting HPEC SGPC. uSAP has introduced a novel SCC-based initial block merging strategy to significantly reduce the number of partitioning iterations. In addition, uSAP has adopted a dynamic batch parallel nodal block assignment algorithm and a dynamic matrix representation to improve runtime and memory performance. We have evaluated uSAP on the 2022 official HPEC SGPC benchmarks. The results have demonstrated the promising performance of uSAP on graphs of different sizes and complexities. For example, uSAP achieves $129.4\times$ speed-up over the 2021 champion on a graph of 50K nodes.

Our future work will extend uSAP to handle streaming graphs and leverage GPU [36] with data-parallel distributed computing [47]–[50] to gain further acceleration and performance improvements.

## Acknowledgment

We are grateful for the support of National Science Foundation of US (CCF-2126672, CCF-2144523 (CAREER), OAC-2209957, TI-2229304, and DMR-2235276). Special thanks go to reviewers for their help with improving this manuscript.

## References

[1] E. Kao, V. Gadepally, M. Hurley, M. Jones, J. Kepner, S. Mohindra, P. Monticciolo, A. Reuther, S. Samsi, W. Song *et al.*, "Streaming graph challenge: Stochastic block partition," in *2017 IEEE High performance extreme computing conference (HPEC)*. IEEE, 2017, pp. 1–12, http://graphchallenge.mit.edu/challenges.

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

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

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

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

[6] W. H. Press, "Saul. a. teukolsky, william t. vetterling and brian p. flannery,"numerical recipes in c"," 1994.

[7] B. W. Priest, A. Dunton, and G. Sanders, "Scaling graph clustering with distributed sketches," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–7.

[8] L. Durbeck and P. Athanas, "Dpgs graph summarization preserves community structure," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–9.

[9] A. J. Uppal, J. Choi, T. B. Rolinger, and H. H. Huang, "Faster stochastic block partition using aggressive initial merging, compressed representation, and parallelism control," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2021, pp. 1–7, https://github.com/iHeartGraph/GraphChallenge/tree/master/StochasticBlockPartition/code/python.

[10] A. Terenin, D. Simpson, and D. Draper, "Asynchronous gibbs sampling," in *International Conference on Artificial Intelligence and Statistics*. PMLR, 2020, pp. 144–154.

[11] 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.

[12] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, vol. 33, no. 6. IEEE, 2022, pp. 1303–1320.

[13] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," 2019, pp. 974–983.

[14] C.-X. Lin, T.-W. Huang, G. Guo, and M. D. F. Wong, "A Modern C++ Parallel Task Programming Library," in *ACM MM*, 2019, p. 2284–2287.

[15] ——, "An Efficient and Composable Parallel Task Programming Library," in *IEEE HPEC*, 2019, pp. 1–7.

[16] C.-X. Lin, T.-W. Huang, and M. D. F. Wong, "An Efficient Work-Stealing Scheduler for Task Dependency Graph," in *IEEE ICPADS*, 2020, pp. 64–71.

[17] T.-W. Huang, "A General-Purpose Parallel and Heterogeneous Task Programming System for VLSI CAD," in *IEEE/ACM ICCAD*, 2020.

[18] T.-W. Huang, Y. Lin, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale," *IEEE TCAD*, vol. 40, no. 8, pp. 1687–1700, 2021.

[19] T.-W. Huang and L. Hwang, "Task-Parallel Programming with Constrained Parallelism," in *IEEE HPEC*, 2022, pp. 1–7.

[20] T.-W. Huang, "Enhancing the Performance Portability of Heterogeneous Circuit Analysis Programs," in *IEEE HPEC*, 2022, pp. 1–2.

[21] T.-W. Huang, D.-L. Lin, Y. Lin, and C.-X. Lin, "Taskflow: A General-Purpose Parallel and Heterogeneous Task Programming System," *IEEE TCAD*, vol. 41, no. 5, pp. 1448–1452, 2022.

[22] G. Guo, T.-W. Huang, C.-X. Lin, and M. Wong, "An Efficient Critical Path Generation Algorithm Considering Extensive Path Constraints," in *ACM/IEEE DAC*, 2020, pp. 1–6.

[23] Z. Guo, T.-W. Huang, and Y. Lin, "GPU-Accelerated Static Timing Analysis," in *IEEE/ACM ICCAD*, 2020.

[24] G. Guo, T.-W. Huang, and M. Wong, "Fast STA Graph Partitioning Framework for Multi-GPU Acceleration," in *IEEE/ACM DATE*, 2023, pp. 1–6.

[25] C.-H. Chiu and T.-W. Huang, "Composing Pipeline Parallelism Using Control Taskflow Graph," in *ACM HPDC*, 2022, p. 283–284.

[26] ——, "Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms: Late Breaking Results," in *ACM/IEEE DAC*, 2022, p. 1388–1389.

[27] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Critical Path Generation with Path Constraints," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[28] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating Common Path Pessimism Removal with Heterogeneous CPU-GPU Parallelism," in *IEEE/ACM ICCAD*, 2021, pp. 1–9.

[29] ——, "A Provably Good and Practically Efficient Algorithm for Common Path Pessimism Removal in Large Designs," in *ACM/IEEE DAC*, 2021, pp. 715–720.

[30] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "GPU-accelerated Path-based Timing Analysis," in *IEEE/ACM DAC*, 2021, pp. 721–726.

[31] Z. Guo, T.-W. Huang, and Y. Lin, "Accelerating Static Timing Analysis using CPU-GPU Heterogeneous Parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023.

[32] D.-L. Lin and T.-W. Huang, "A Novel Inference Algorithm for Large Sparse Neural Network using Task Graph Parallelism," in *IEEE HPEC*, 2020, pp. 1–7.

[33] ——, "Accelerating Large Sparse Neural Network Inference Using GPU Task Graph Parallelism," *IEEE TPDS*, vol. 33, no. 11, pp. 3041–3052, 2022.

[34] C.-H. Chiu, D.-L. Lin, and T.-W. Huang, "An Experimental Study of SYCL Task Graph Parallelism for Large-Scale Machine Learning Workloads," in *Euro-Par Workshop*, 2022.

[35] C.-H. Chiu and T.-W. Huang, "Composing Pipeline Parallelism Using Control Taskflow Graph," in *ACM HPDC*, 2022, p. 283–284.

[36] D.-L. Lin and T.-W. Huang, "Efficient GPU Computation Using Task Graph Parallelism," in *Euro-Par*, 2021.

[37] T.-W. Huang and M. D. F. Wong, "OpenTimer: A High-Performance Timing Analysis Tool," in *IEEE/ACM ICCAD*, 2015, p. 895–902.

[38] T.-W. Huang, G. Guo, C.-X. Lin, and M. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," in *IEEE TCAD*, 2021, pp. 776–789.

[39] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "OpenTimer v2: A Parallel Incremental Timing Analysis Engine," *IEEE Design and Test*, vol. 38, no. 2, pp. 62–68, 2021.

[40] E. Dzaka, D.-L. Lin, and T.-W. Huang, "Parallel And-Inverter Graph Simulation Using a Task-graph Computing System," in *IEEE IPDPSw*, 2023, pp. 923–929.

[41] T.-W. Huang, "qTask: Task-parallel Quantum Circuit Simulation with Incrementality," in *IEEE IPDPS*, 2023, pp. 746–756.

[42] D.-L. Lin, H. Ren, Y. Zhang, B. Khailany, and T.-W. Huang, "From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–12.

[43] D.-L. Lin, Y. Zhang, H. Ren, S.-H. Wang, B. Khailany, and T.-W. Huang, "GenFuzz: GPU-accelerated Hardware Fuzzing using Genetic Algorithm with Multiple Inputs," in *ACM/IEEE DAC*, 2023.

[44] S. Jiang, T.-W. Huang, B. Yu, and T.-Y. Ho, "Snicit: Accelerating sparse neural network inference via compression at inference time on gpu," in *ACM ICPP*, 2023, p. 51–61.

[45] K. Zhou, Z. Guo, T.-W. Huang, and Y. Lin, "Efficient Critical Paths Search Algorithm using Mergeable Heap," in *IEEE/ACM ASPDAC*, 2022, pp. 190–195.

[46] K.-M. Lai, T.-W. Huang, P.-Y. Lee, and T.-Y. Ho, "ATM: A High Accuracy Extracted Timing Model for Hierarchical Timing Analysis," in *IEEE/ACM ASPDAC*, 2021, p. 278–283.

[47] T.-W. Huang, C.-X. Lin, and M. D. F. Wong, "Dtcraft: A distributed execution engine for compute-intensive applications," in *IEEE/ACM ICCAD*, 2017, pp. 757–765.

[48] ——, "DtCraft: A High-Performance Distributed Execution Engine at Scale," *IEEE TCAD*, vol. 38, no. 6, pp. 1070–1083, 2019.

[49] T.-W. Huang and M. D. Wong, "Accelerated Path-Based Timing Analysis with MapReduce," in *ACM ISPD*, 2015, p. 103–110.

[50] T.-w. Huang and M. D. F. Wong, "On fast timing closure: speeding up incremental path-based timing analysis with mapreduce," in *ACM/IEEE SLIP*, 2015, pp. 1–6.