Parallel Graph Coloring using OpenMP, CUDA, and MPI

SANJANA KUCHIBHOTLA[1] AND ADITRI GUPTA[2]

[1] Carnegie Mellon University, 15-418, Spring 2025

Compiled June 11, 2025

This document details our work on the 15-418 final project. For our project, we implemented different graph coloring algorithms and parallelized them using OpenMP, MPI and CUDA. In this paper, we discuss our experiments and the results we achieved.

1. SUMMARY

For our project, we implemented multiple different graph coloring algorithms both sequentially and in parallel. Specifically, we first implemented a greedy sequential graph coloring algorithm as our baseline algorithm. We then implemented Jones-Plassmann and parallelized it and then proceeded with parallel versions of the greedy algorithm using speculative coloring and conflict resolution. We parallelized these algorithms on the CPU using OpenMP pragma constructs and MPI. We also implemented a parallel CUDA version of Jones-Plassmann in order to measure how much speedup improved when using a GPU. One of the main aspects of our project was measuring when these different algorithms worked best - whether they worked better on sparse vs. dense graphs, how well they scaled, and how big the graphs needed to be in order to see a significant benefit from parallelism. We also tested with different numbers of threads on the Gates clusters $(1, 2, 4, 8)$ and on the PSC machines. All of this is explored in the following sections.

2. BACKGROUND

The goal of our project is to explore the benefits as well as the difficulties and drawbacks that come with parallelizing graph coloring algorithms. All graph coloring algorithms attempt to color the vertices of a graph in such a way that no neighboring vertices (vertices connected by an edge) are colored using the same color. We explore different algorithms in our code as well as different parallelization schemes.

The data structure that we operate on is a graph. We define a graph as

$$G = (V, E)$$

where V is the set of vertices, and E is the set of edges. We then define a coloring of a graph as a function

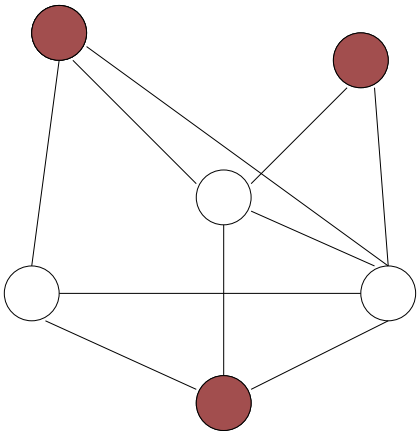
$$c : V \to \mathbb{N} \text{ s.t for every edge } (u, v) \in E, c(u) \neq c(v)$$

We implement the graph as an adjacency list A, where the nodes are integers and $A[u]$ is a list of all the vertices connected to node u by an edge (all the neighbors of u). For each graph, we also maintain a list of colors (as a list of integers) that represents the color that each vertex was assigned at the end of the algorithm.

We also implement a few key operations that can be performed on the graph. Specifically, we have methods for adding an edge between two vertices, getting the neighbors of a given vertex, assigning priorities to vertices (used in algorithms like Jones-Plassmann), and counting and verifying colorings.

The computational bottleneck in graph coloring is assigning colors to vertices while ensuring that the coloring remains valid and there are no conflicts. We want to parallelize by coloring as many vertices as possible at the same time. Parallelism is challenging however, since there are local dependencies between a vertex and all of its neighbors. However, if we find an independent set of vertices, those vertices will not be dependent on each other, and there is an opportunity for parallelism. For example, the following figure shows a set of vertices that can be colored in parallel without dependencies on each other.

Fig. 1. Independent Set of Vertices in a Graph



Essentially in a sequential program, a vertex must go through all of its neighboring vertices in order to see which colors are disallowed. The biggest difficulty when it comes to parallelizing graph coloring is that the problem is inherently sequential - if two neighboring vertices are colored in parallel, there is no guarantee that their colors will be different, resulting in an invalid coloring. Different algorithms have been developed in order to combat this (such as Jones-Plassmann), and Gebremedhin and Manne proposed a class of optimistic graph coloring algorithms. For the different approaches we used, we took inspiration from the work of Gebremedhin and Manne, specifically the ideas of speculative optimistic coloring and subsequent conflict resolution, in order to experiment with different methods.

In addition to this, for testing, we used Erdos-Renyi graphs generated specifically under the $G(N, p)$ model where a generated graph has N vertices and the probability that there is an edge between any two vertices is p. Using this, we want to test multiple different parallel graph coloring algorithms on graphs of different sizes and different densities.

3. APPROACH

A. Sequential Algorithm (Baseline)

The first algorithm we implemented (after implementing our graph class) was the sequential algorithm for graph coloring. For this, we used a greedy approach - we sequentially color all vertices and color each vertex the lowest available color that is not used by any of its neighbors. Pseudocode is shown below:

Algorithm 1. Sequential Greedy Graph Coloring

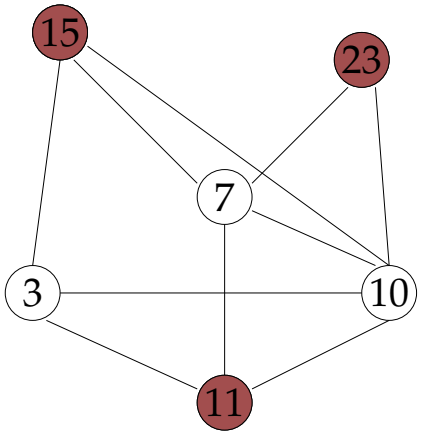
```
1:  function GREEDY_COLOR(G = (V, E))
2:      colors[u] ← −1 for u ∈ V
3:      for u ∈ V do
4:          nbor_colors ← ∅
5:          for v ∈ neighbors(u) do
6:              if colors[v] ≠ −1 then
7:                  Add colors[v] to nbor_colors
8:          color ← 0
9:          while color ∈ nbor_colors do
10:             color ← color + 1
11:         colors[u] ← color
12:     return
```

Developing this algorithm gave us a good idea of how graph coloring worked and what the dependencies in the algorithm were. We also did more research at this point to look into different algorithms and methods that were used to parallelize the greedy approach. This led us to the Jones-Plassmann algorithm. Jones-Plassmann is a randomized algorithm that attempts to increase the amount of opportunities for parallelism in graph coloring.

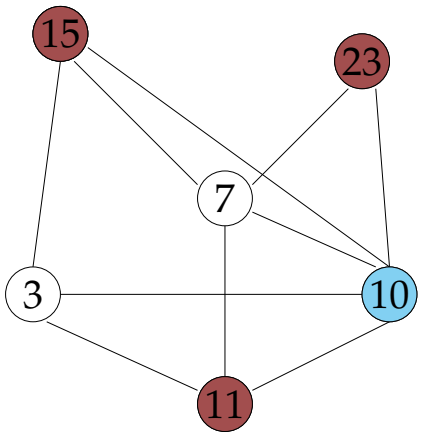
B. Jones-Plassmann

We first worked on developing the sequential version of Jones-Plassmann. We based our initial implementation off of pseudocode from the University of Utah's lecture notes on randomized algorithms [1], from which we were also able to learn about the algorithm. As parallelizing Jones-Plassmann was one of the main things we wanted to focus on, we spent quite a bit of time understanding how the algorithm worked and the looking for opportunities for parallelism in the algorithm. Jones-Plassmann first assigns a unique random priority to all vertices. Then for every vertex, it checks in parallel if that vertex is a local maxima. If it is, it colors it with the lowest available color (similar to greedy sequential coloring). It does this iteratively, considering only vertices that have not been colored yet. This offers a lot of potential opportunities for parallelism, as it attempts to find an independent set on each round. As the maximum parallelism that can be achieved in graph coloring algorithms is through finding the maximal independent set on each round of coloring, Jones-Plassmann gets close to this by picking a large subset of vertices that are a local maxima (priority-wise) among that vertex's uncolored neighbors, which allows for many vertices to be colored in parallel without introducing conflicts. Though this independent set might not be maximal, it will be big enough to at least offer a significant improvement in parallelism.

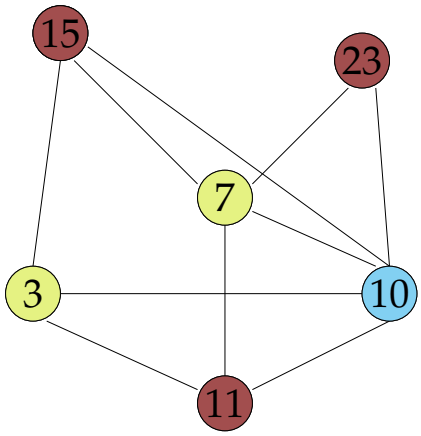
Considering the following graph as an example, these are the vertices that would first be colored:



On the next iteration of Jones-Plassman, the vertex with a priority of 10 would be colored as it is the only uncolored vertex with a local max (with respect to its uncolored neighbors).



Finally, the 7 and 3 can be colored in parallel since they are the only uncolored vertices that are a local max when compared to their uncolored neighboring vertices.



The pseudocode we wrote for the sequential algorithm of Jones-Plassmann is as follows:

For our first parallelization attempt, we noticed that the bulk of the computation lied in the two for loops - one that was computing the edges to color, and the other that was actually coloring the edges. Therefore, as a naive first attempt, we decided to added the OpenMP `#pragma omp parallel for` directives around these loops, using dynamic scheduling in order to balance the workload. However, although this version did work correctly and produce valid colorings, the performance speedup was

Algorithm 2. Sequential Jones-Plassmann

```
1: function SEQUENTIAL_JP(G = (V, E))
2:     priorities[u] ← random unique priority
3:     colors[u] ← −1 for u ∈ V
4:     while there is still a vertex to color do
5:         vertices_to_color = ∅
6:         for u ∈ V do
7:             if u is uncolored local priority max then
8:                 Add u to vertices_to_color
9:         for u ∈ vertices_to_color do
10:            greedily color u
11:    return
```

not as good as we hoped. On a graph with 10,000 vertices and approximately 50,000 edges, the sequential version runs in about 0.098 seconds, and the parallel version ran in 0.087 seconds. Though this was a speedup, it didn't show the speedup we hoped. We then tried to work on optimizing our parallel Jones-Plassmann. First, we realized that one optimization we could make was including only one `#pragma omp parallel for` instead of two. This was because having two separate parallel regions introduced unnecessary overhead since threads were repeatedly spawning and synchronizing. Another optimization we made was to use a different chunksize in dynamic scheduling. Without using any chunksize, threads just grab 1 iteration at a time. Instead, by increasing the chunksize, threads can grab more iterations at a time, especially since the work done in each iteration isn't that much. This reduces the scheduling overhead and the time spent waiting. However, the chunksize cannot be too high, since it might cause poor load balancing as one thread might get more work than another thread. After experimenting with this number, we settled on 64 as a good chunksize. This ended up showing much more improvement compared to our previous naive parallel implementation.

C. Speculative Coloring

After implementing Jones-Plassmann, we wanted to try parallelizing our original sequential greedy approach. We researched ideas provided by Gebremedhin and Manne about speculative coloring as well as conflict resolution and tried to incorporate that into our parallel version of the greedy algorithm. For this approach, the main idea was to keep the greedy approach that colors each vertex the first available color that it can based on the neighboring colors. However, doing this all in parallel in order for each vertex to be colored in parallel can introduce a lot of conflicting vertices due to race conditions. For example, if vertex u and vertex v are neighbors, but both do not see the updates to the others' colors before coloring their own vertex, they might both assign the same color to their own vertex. This would be an invalid graph coloring however. In order to fix conflicting vertices such as this, we also introduce a "conflict resolution" loop that goes through all vertices and their neighbors and if two neighboring vertices are conflicting, it marks one of them as still needing to be colored. Pseudocode for the algorithm is shown below:

Algorithm 3. Parallel Speculative Coloring

```
1: function SPECULATIVE_COLOR(G = (V, E))
2:     colors[u] ← −1 for all u ∈ V
3:     vertices_to_recolor ← V
4:     while there are conflicting vertices do
5:         for all u ∈ vertices_to_recolor do
6:             Color u with the lowest available color not used by any neighbor
7:         vertices_to_recolor ← ∅
8:         for all u ∈ V do
9:             for all v ∈ neighbors(u) do
10:                if u < v and colors[u] = colors[v] then
11:                    colors[v] ← −1
12:                    vertices_to_recolor[v] ← 1
       return
```

In order to parallelize this, we were able to parallelize each of the phases separately. We added one `#pragma omp parallel for` around the for loop that did the speculative coloring, and one around the for loop that did the conflict resolution. Similar to the Jones-Plassmann par-

allelization, a chunksize of 64 worked best with dynamic scheduling. We also tried using static scheduling, as we reasoned that each thread had a similar workload, but dynamic scheduling with a chunksize of 64 showed a slight (though possibly negligible) improvement, so we stuck with the dynamic scheduling.

D. Priority-Based Conflict Resolution

Another idea we wanted to test was a slight modification to Speculative Coloring. The conflict detecting phase during speculative coloring was particularly of interest to us. The first phase was pretty straightforward - it essentially did the same this as greedily coloring the graph. However, when it came to detecting conflicts, there were a variety of heuristics we thought about that could cause the graph coloring algorithm to perform better or worse. The previous speculative coloring algorithm that we implemented always recolored the vertex that was assigned a higher vertex ID. However, this heuristic did not consider anything other than just vertex ID. Though vertices are hypothetically assigned arbitrarily, in reality, this could incorporate some bias into the assignment. For example, lower vertex IDs were likely assigned earlier than higher vertex IDs (at least in our Graph implementation). In large, real-world graphs, this could lead to bias in the coloring scheme and regarding which vertices would be chosen to be recolored. In order to combat this, we thought of a few different schemes: specifically we thought about recoloring with regards to priority and recoloring with regards to vertex degree.

For recoloring with regards to priority, the idea was to essentially "re-randomize" our graph, as priorities were assigned randomly. This would give a more arbitrary bias-free structure to the graph, removing the issue from vertex ID based conflict resolution. This idea is what we ended up implementing. We also wanted to implement recoloring by vertex degree (so vertices with a lower degree would be chosen to be recolored, potentially improving the time required to recolor them). We were not able to implement and test this in time however, but it remains to be a promising method for future experimentation.

E. MPI Jones-Plassmann

We continued our exploration of parallel graph coloring by implementing Jones-Plassmann with the Message Passing Interface. Since the key characteristic of MPI is that the processes are entirely independent and only communicate by explicitly sending and receiving messages to each other, the notable challenge of coordinating priority and color information across independent processes without incurring excessive communication overhead arises. Each process need up-to-date neighbor priorities and boundary colors to decide which of its vertices can be safely colored.

Although MPI and OpenMP differ fundamentally in their memory and synchronization models, our MPI implementation of Jones-Plassmann follows the same high-level steps as the OpenMP psuedocode.

Mapping the work to the MPI model was more involved than the OpenMP version. The core data structure, an adjacency-list representation of the graph, the color array and prioritiy array were all stored by each process in its own local memory. The vertices were split into contiguous blocks, where each block's contained about $\frac{N}{P}$ vertices, so esentially each core was working on its own slice of the graph meaning all the data was kept in the core's local cache and memory. We avoided having to broadcast the graph to each process by instead having each processor generate the entire graph locally based on the same random seed. Process 0 assigned priorities to the entire graph and then broadcast te priorities array to all the other processors so they could update their local priorities array. The main iterative loop of the algorithm proceeded as normal, but each process would only scan vertices in its local contiguous block. We merge color updates with `MPI_Allreduce(MPI_MAX)`, so if any process colors vertex i, that assignment is immediately visible to all. For termination, each process sets a `local_done` flag

when it has colored every vertex in its block; an `MPI_Allreduce(MPI_SUM)` then checks if the sum equals P, indicating that all processes are finished. Since each `MPI_Allreduce` also acts as a barrier, there are only two global synchronizations per coloring iteraion with one to merge colors and one to check for completion.

A small optimization made for the MPI implementation was instead of using a `std::set` to track neighbor colors, we used a boolean array `used_colors[MAX_COLORS]` which is statically allocated and allows for faster memory reads and writes. We had also tried sizing our neighbor-color buffer based on the local vertex degree, but this introduced unnecessary complexity, and having an array of size 2000 worked for all of the graphs we tested on.

We approached writing the MPI implementation of Jones-Plassmann with the main guiding idea of "do as much as possible and talk only when you really have to". This principle lead to use prioritizing splitting up the data once and letting each process work on its own chunk in local memory, broadcasting any necessary global info as few times as possible, and keeping our buffers at a fixed size so they could stay in the cache. We wanted to use point-to-point communication and nonblocking receives and sends, yet we kept running into a deadlock or just very poor performance and eventually settled on a more simple and straightforward approach of consolidating inter-process communication through MPI collectives like `MPI_Allreduce` for merging many updates and the integral termination check.

F. CUDA Jones-Plassmann

Following our CPU implementations with OpenMP and MPI, we shifted our focus to GPU acceleration and developed a CUDA-based Jones-Plassmann algorithm on the NVIDIA RTX 2080 (provided on the gates clusters machines).

While coming up with our CUDA implementation, the two main principles we kept in mind were minimizing divergence and managing the memory hierarchy (global vs. shared memory).

In our CUDA Jones-Plassmann implementation, we generated the graph in device global memory with a Compressed Sparse Row (CSR) representation. Two arrays, `adjacency_list[]` and `vertex_offsets` describe every vertex's neighbors. These arrays are a more memory-efficient way of storing relatively sparse graphs (in order $O(n)$ rather than $O(n^2)$), and put each vertex's neighbor list in one contiguous chunk of global memory.

Our thread mapping strategy employs a simple but effective one-to-one correspondence between CUDA threads and graph vertices. Each thread with global ID `i` is responsible for processing exactly one vertex `i` in the graph. At launch time, the host code, based on the user input, sets the block width (128 or 256 threads) and computes the ceiling of `gridDim = (N/block_width)` to ensure sufficient thread coverage across all vertices.

The kernel execution proceeds with each thread independently determining if its corresponding vertex belongs to the current maximal independent set using the priority values. If it does , the thread then proceeds to color its vertex by finding the smallest available color not used by any of its neighbors. Synchronization between iterations occurs on the host side, with kernel launches separated by `cudaDeviceSynchronize()` calls to ensure all threads have completed their work before the next independent set is processed.

This mapping strategy results in pretty good thread utilization when the graph is large and sparse, though some thread divergence is unavoidable due to the irregular nature of graph neighborhood traversals. We chose not to use shared memory for neighborhood data due to its variable size across vertices, which would complicate the implementation without significant performance benefits for our sparse graph representation.

The biggest challenge with trying to implement a graph processing algorithm in CUDA was dealing with the inherent irregular nature of a graph. Essentially, each vertex may have a drastically different number of neighbors, and so when threads process vertices with widely

varying degrees, some threads complete their work quickly while others remain active for much longer, causing thread divergence within warps. Even with our one-to-one thread-to-vertex mapping, the computational work per thread varies significantly based on the number of neighbors each vertex has and the relative priorities between a vertex and its neighbors. Another bottleneck in the efficacy of our implementation is the iterative nature of Jones-Plassmann, which causes it to require global synchronization between independent set coloring rounds. This, in turn, requires entire kernel relaunches rather than more efficient in-kernel synchronization.

4. RESULTS

We ran multiple different experiments on different types of graphs in order to test the amount of speedup for each algorithm achieved on various workloads. This allowed us to compare the effectiveness of the various algorithms, as well as how they performed on denser versus sparser graphs. Specifically we tested the following types of graphs:

- #Vertices: $10,000$ (relatively small), $50,000$ (medium sized), $100,000$ (larger)
- Edge Probability: 0.01 (denser graph), 0.001 (sparser graph)

The graphs we tested on were Erdos-Renyi graphs. Essentially, all graphs were generated with the given number of vertices, and an edge was placed between any two given vertices with the edge probability p given. If given more time and resources, we would want to test on more irregularly generated graphs has well that were more biased to one region over another to simulate real-life social networks and large graphs.

We also tested our implementations for OpenMP, CUDA, and MPI separately in order to analyze the speedup achieved within each framework and to understand the scalability and performance tradeoffs across different levels of parallelism.

As for the number of colors that each algorithm outputted, we did not see much variation in algorithms for colorings on the same graph. Colorings differed by at most 1 across the thread counts no matter how many colors were used.

While we wanted to test our implementations on much larger/denser graphs as well, one limitation we ran into was the initialization time of the graphs - initialization time itself scaled exponentially in the number of nodes, and as a result, even initializing a $500,000$ node graph took approximately 23 minutes. Even after initializing, we ran into memory limitations when trying to run the algorithms, so this was not a viable option given our resources.

A. OpenMP

For OpenMP, we used the sequential greedy solution as our baseline. This allowed us to compare every algorithm to the sequential greedy solution for that algorithm. This showed how some algorithms performed better than the sequential greedy solution with even one thread, and we were able to incorporate that into our graphs. We used $1, 2, 4$, and 8 threads for testing on the Gates clusters.

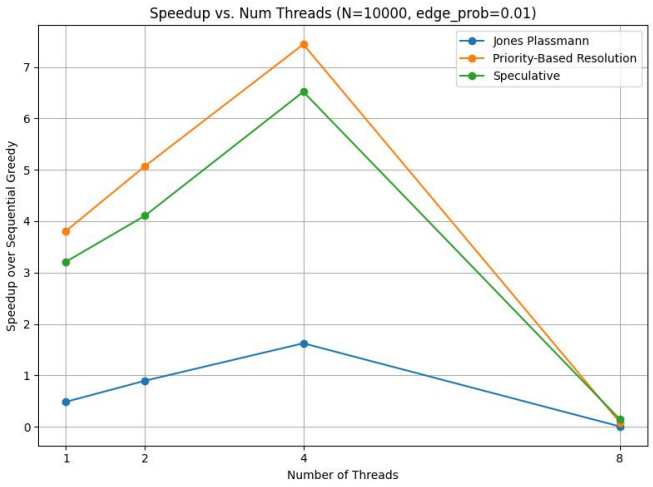


Fig. 2. Small Dense Graph ($N = 10000, p = 0.01$)

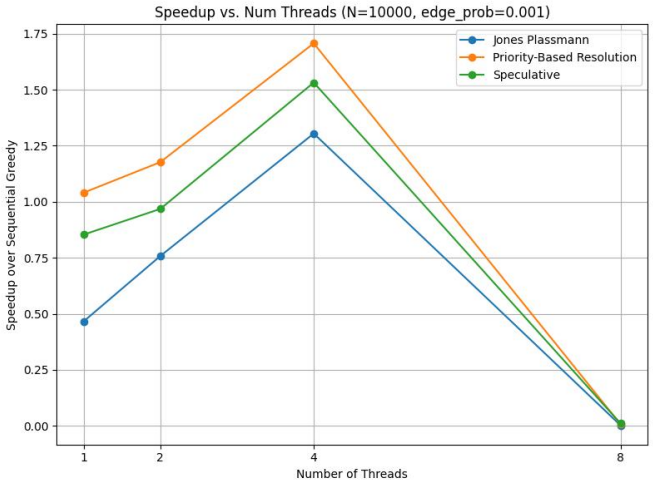


Fig. 3. Small Sparse Graph ($N = 10000, p = 0.001$)

These graphs show the performance of the 3 different OpenMP parallel algorithms we implemented (Jones-Plassmann, Speculative Coloring, and Speculative Coloring with Priority-Based Resolution) on both a sparse and a dense graph. The graphs show interesting results - first, we see that the Priority-Based Resolution method outperformed both other methods no matter what the thread count was. In both sparse and dense graphs, we also see an increase in speedup until the peak at 4 threads, and then a decrease afterwards. This shows that the overhead due to scheduling and thread synchronization, as well as contention at 8 threads outweighs the performance speedup due to parallelism no matter whether the graph is dense or sparse. However, this could change when the size of the graph increases, since the current graph is a relatively small workload.

One interesting result to note is that the speedup on the sparse graph is relatively smaller than the speedup on the dense graph. This is likely due to the fact that there is less work to be done on sparse graphs (e.g. less neighbors to check during conflict resolution). As a result, the amount of parallelism that can be exploited is lower, leading to a smaller speedup.

We also see that Jones-Plassman performs significantly worse than the other two algorithms, especially on the dense graph. This is probably due to the nature of the algorithm requiring local max priority checks which are not parallelized.

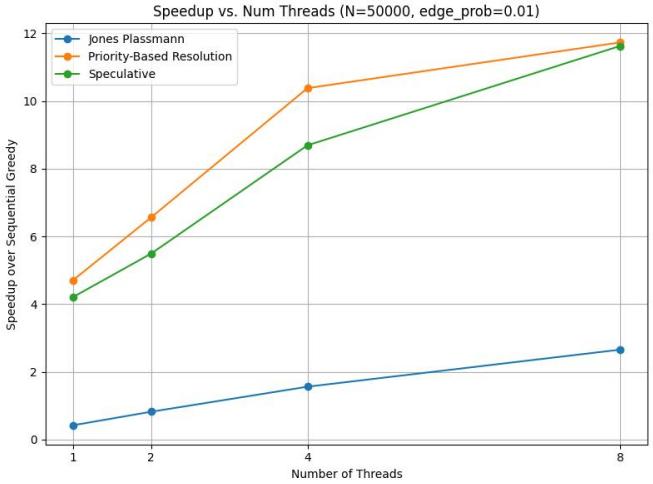


Fig. 4. Medium Dense Graph ($N = 50000, p = 0.01$)

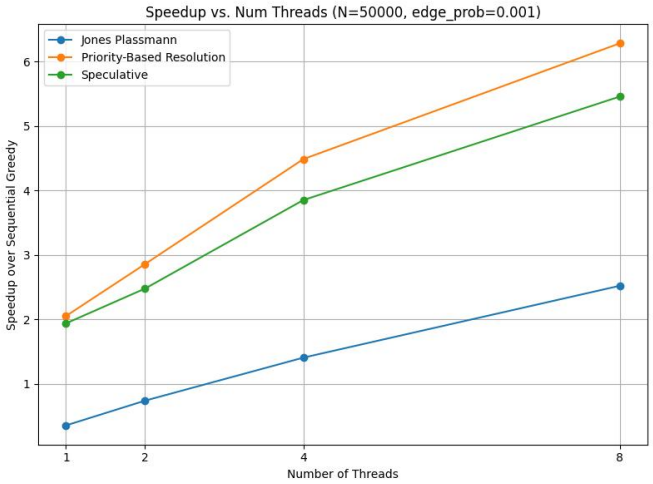


Fig. 5. Medium Sparse Graph ($N = 50000, p = 0.001$)

Looking at the graphs for the medium sized graph with 50,000 vertices, we see a shift from the results for the small graph. One major change is that we do not reach a speedup peak at 4 threads anymore. This shows that as the graph size increases, we are able to exploit parallelism benefits, and the overhead of thread scheduling and synchronization does not dominate the actual benefits of parallelism in these cases. As a result, the graph is more linear-looking than the graphs in the small graph case.

In addition, another interesting point to note is that the general speedup numbers seem to be higher for the medium graph case. While the max speedup for the small graph when it was sparse was 1.75x and when it was dense was $\approx$ 7x when compared to the sequential greedy solution, we see much higher speedups for the medium graph (when compared to the sequential greedy solution). For the medium dense graph, a speedup close to 12x was observed when using 8 threads, and for the medium sparse graph, a speedup close to 6x was observed when using 8 threads.

Other than this, we see a lot of similar results to the small graph case regarding comparisons in the performance of the algorithms themselves.

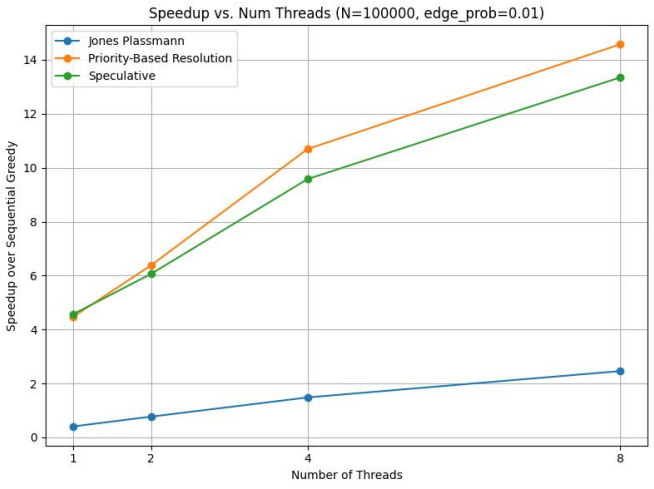


Fig. 6. Large Dense Graph ($N = 100000, p = 0.01$)

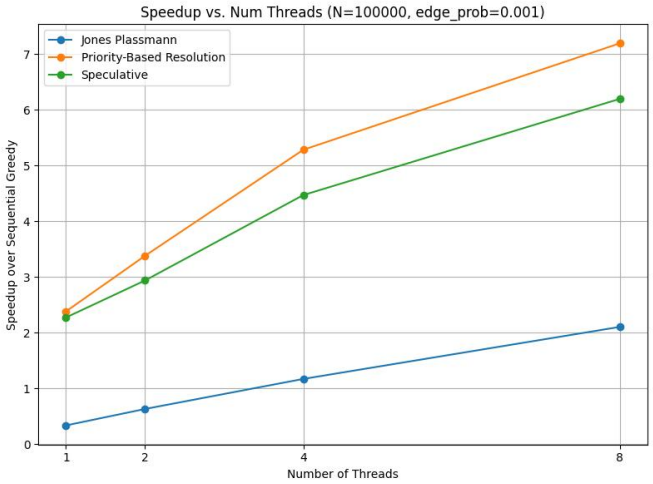


Fig. 7. Large Sparse Graph ($N = 100000, p = 0.001$)

The speedup graphs for a large graph look similar to those for the medium graph. A slight increase in the maximum speedup is observed, but not enough to significantly show a dramatic change. In general, we are able to see from this that these parallel algorithms tend to perform better on larger graphs, and the speedup is especially evident on dense graphs.

From these graphs, we can see that Jones-Plassmann limited speedup due to having more sequential portions of code than the Speculative algorithms. As the speculative algorithms were highly optimistic and color all the vertices at the same time, they are maximally parallelized in this portion (without considering thread scheduling overhead). Though the conflict resolution portion did require some sequential execution, it was parallelized over all vertices. Jones-Plassmann instead had separate portions for figuring out which vertices to color and then the actual coloring of those vertices. As a result, its parallelism was limited, especially due to the structure of the graphs we tested on.

A.1. PSC Machines

We also used the PSC machines in order to test how the data scaled up to 128 threads. Specifically, we tested it on $1, 2, 4, 8, 16, 32, 64$, and 128 threads on the PSC machines. Using this data, we were able to identify a lot of potential benefits with Jones-Plassmann. Some of the data did

not corroborate the results we received on the Gates machines, but we attributed some of that to the poor performance of the PSC machines compared to the gates machines.

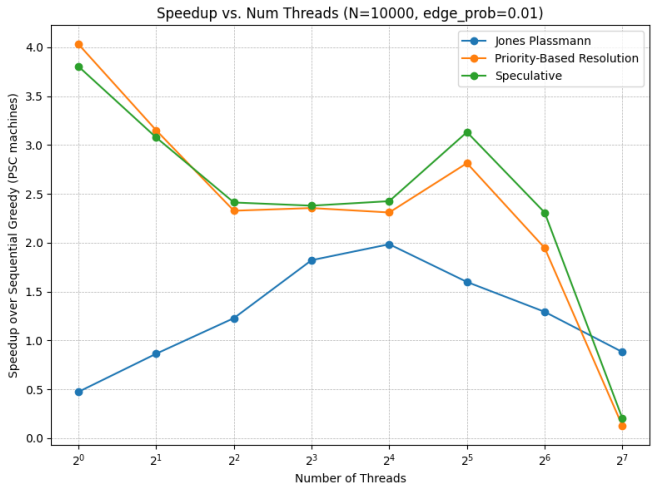


Fig. 8. Small Dense Graph ($N = 10000, p = 0.01$, log scale)

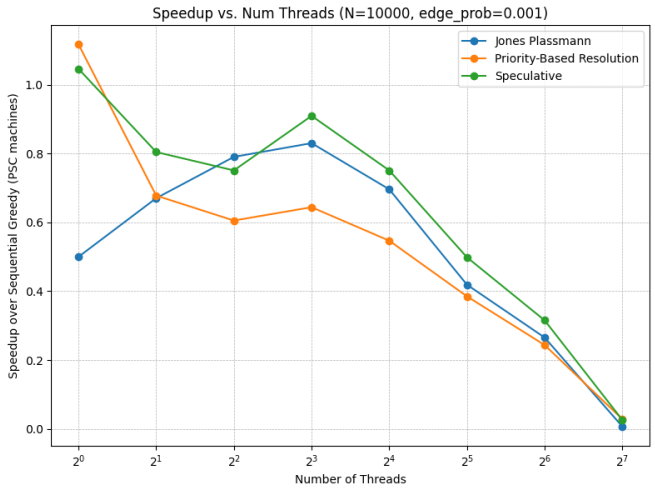


Fig. 9. Small Sparse Graph ($N = 10000, p = 0.001$, log scale)

The PSC machines showed different results than the Gates cluster machines. However, we did notice that the PSC machines ran much slower than the Gates machines and this might have been a reason for the speedup decreasing with more threads for the Speculative versions. We did also notice an increase in speedup with the Jones-Plassmann algorithm. The peak was reached at 16 threads, after which the speedup decreased again. However, the actual speedup numbers themselves are relatively small, possibly explainable by the graph being small, which corroborated results we received on the Gates machines.

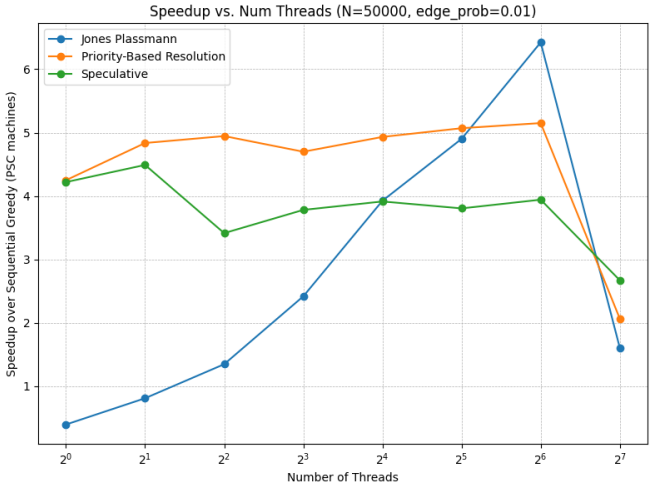


Fig. 10. Medium Dense Graph ($N = 50000, p = 0.01$, log scale)

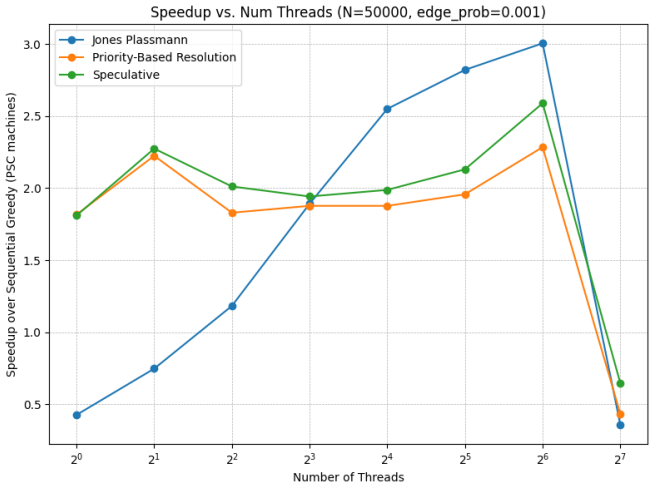


Fig. 11. Medium Sparse Graph ($N = 50000, p = 0.001$, log scale)

The primary result we noticed with the PSC machines by measuring up to 128 threads is that while Jones-Plassmann does start off slow and does not gain much speedup at all (and even performs worse than the sequential greedy algorithm), it does seem to scale pretty well with

the number of threads. Compared to the Speculative algorithms, Jones-Plassmann reaches a high peak at 64 threads before the speedup decreases again. This is due to the fact that Jones-Plassmann benefits mainly from parallelism at high thread counts over methods like speculative coloring. Speculative coloring fails in this aspect - though the optimistic coloring phase is extremely parallelizable, when thread counts grow, there is more contention and synchronization stall time in the conflict resolution phase. However, Jones-Plassmann does not have this issue as it colors vertices correctly the first time and thus scales more at higher thread counts. However, even with the medium graph, Jones-Plassmann reaches a peak at 64 threads and the overhead at 128 threads dominates the speedup achieved due to parallelism.

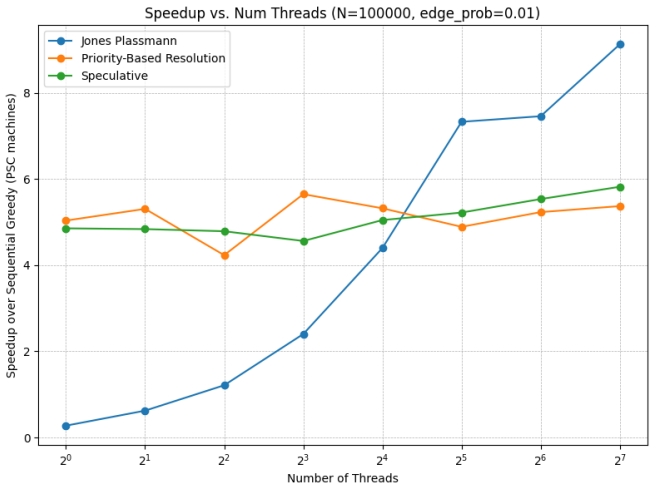


Fig. 12. Large Dense Graph ($N = 100000, p = 0.01$, log scale)

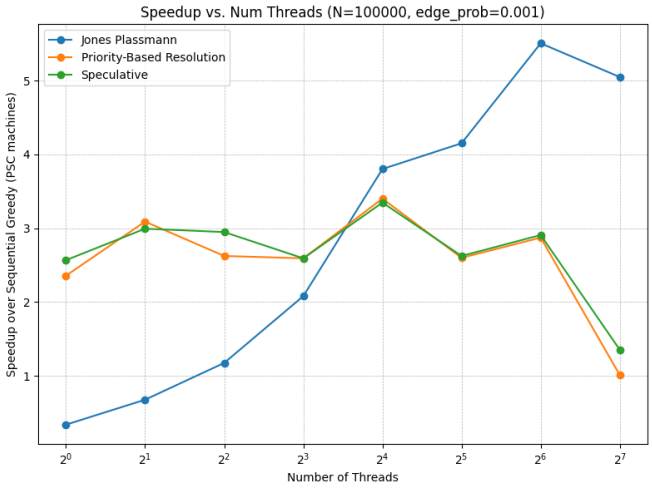


Fig. 13. Large Sparse Graph ($N = 100000, p = 0.001$, log scale)

The large graphs show similar trends as the medium graph. One key difference however, is that the speedup does not plummet at 128 threads as it did with the medium sized graph. With the large graph, the speedup increases even with 128 threads. This shows how the size of the graph definitely has an impact on the speedup at high threads.

Table 1. Total Cache Misses (millions) for `Jones-Plassmann`

Nodes	Edge Prob	1 thread	2 threads	4 threads	8 threads	16 threads	32 threads	64 threads	128 threads
10,000	0.01	2.917	2.637	2.558	2.334	2.614	2.891	3.458	4.282
10,000	0.001	0.192	0.221	0.239	0.281	0.362	0.497	0.651	1.016
50,000	0.01	139.550	115.780	118.493	126.129	124.500	131.591	134.353	157.020
50,000	0.001	10.139	11.067	11.194	12.130	12.713	14.159	15.906	18.198
100,000	0.01	998.063	784.565	724.698	813.832	883.504	796.153	867.697	918.308
100,000	0.001	62.919	65.957	69.811	65.741	76.561	78.211	82.788	89.264

We can see that as the number of vertices grows, the number of cache misses also increases significantly. This is due to the larger working set associated with a larger graph as well as the increased memory usage. Denser graphs with a higher edge probability (meaning more edges) also show many more cache misses than sparse graphs as there are less regular access patterns when graphs are denser. In addition, the number of cache misses also grows a lot with a higher thread count. This suggests that when more threads are used, there is more contention and cache usage is less efficient. The cache numbers show that when we have larger and denser

graphs, memory access patterns and cache access become increasingly important in increasing performance.

B. MPI

For MPI, we used the same sequential greedy algorithm we used to analyze our OpenMP results as a baseline. Below are a few selected graphs showing *speedup vs. thread* count that share both the best speedups observed as well as when we observed quite awful performance "gains" from using the MPI implementation.

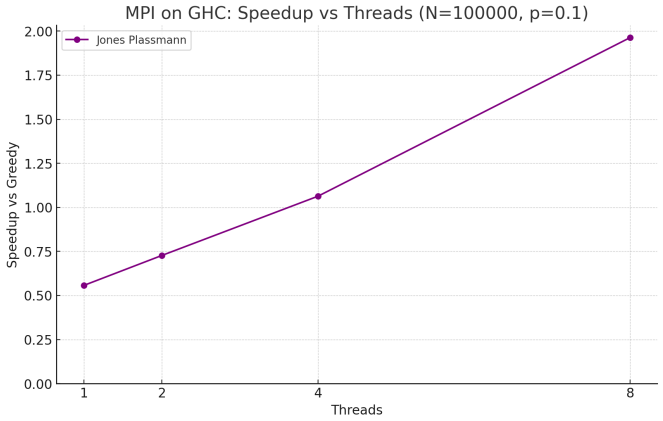


Fig. 14. Large Dense Graph ($N = 100000, p = 0.1$)

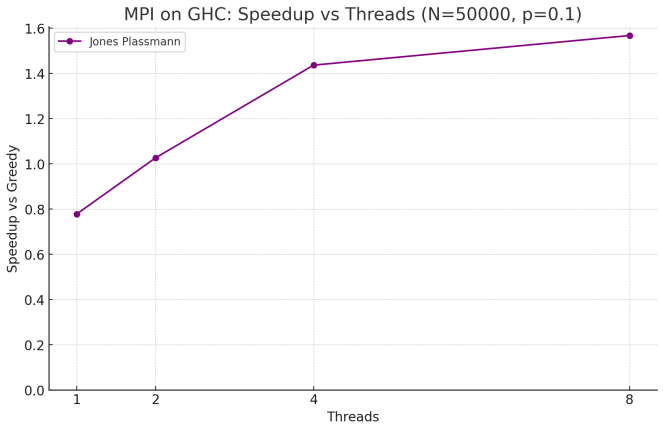


Fig. 15. Medium Dense Graph ($N = 50000, p = 0.1$)

Figures 14 and **15** show the best achieved speedups that we observed on the Gates cluster machines. Our best performance was observed by very large, quite dense graphs- graphs with around 100,000 nodes and a edge-probability of 0.1 which is approximately 500,010,329 edges. Comparable speedup was achieved on the 50,000 node graph with an edge-probability of 0.1 (approx 125,014,822 edges).

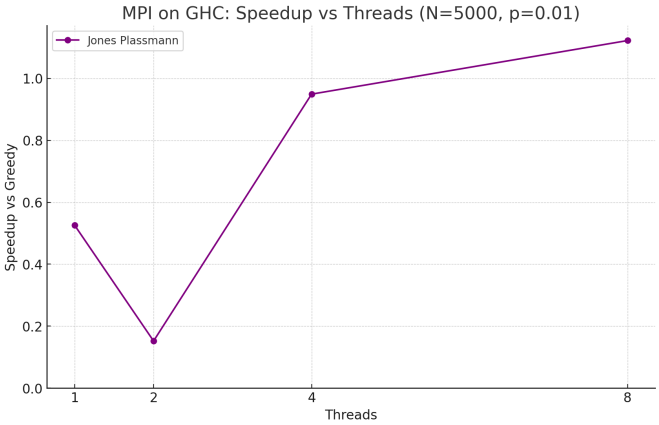


Fig. 16. Very Small Sparse Graph ($N = 5000, p = 0.01$)

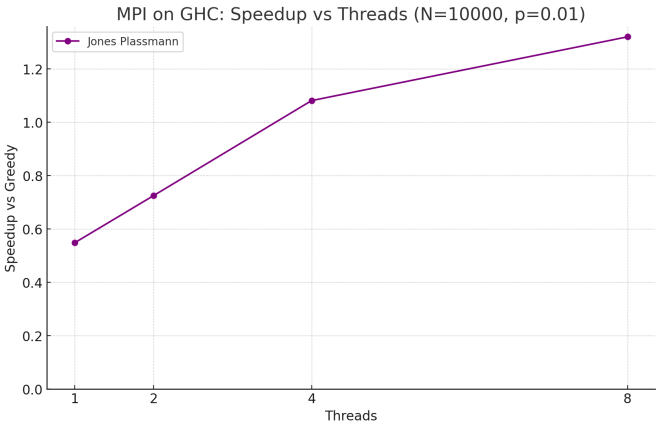


Fig. 17. Small Sparse Graph ($N = 5000, p = 0.01$)

Figures 16 and **17** show the our worst performing tests. MPI Jones-Plassmann showed significantly worse performance in terms of execution time (not including initialization time) than the greedy sequential version, even when 4 processors were used. Only when 8 processors were being utilized was there any speedup, and even then, it was very minimal (~1.2x speedup) and far below linear speedup. These graphs were much smaller than the ones from Figures

14 and 15. The very small (relatively speaking) sparse graph has about 5000 nodes with an edge-probability of 0.01, meaning it has approximately 125,102 edges. The other small, sparse graph has 10,000 nodes with the same edge probability, yielding it about 501,057 edges.

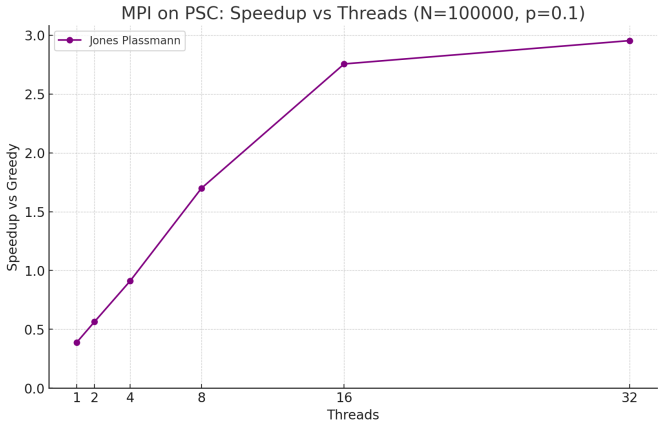
We see the best speedups on large, dense graphs because each MPI process has a lot of work to do between the two global synchronizations. On a 100 000-node graph with p=0.1, each process scans and colors tens of millions of edges every iteration, so the cost of our two `MPI_Allreduce` calls is small compared to that local computation. When the graph is smaller or sparser, each process does much less work per wave and the fixed synchronization latency dominates, killing any speedup.

Dense graphs also give us more parallel work in each coloring round. Even though higher edge probability can limit how many vertices qualify as "locally maximal," there are still enough of them that all processors stay busy. On small or very sparse inputs, the number of vertices we can color in parallel drops quickly, so we end up doing the same synchronizations over and over on smaller and smaller chunks of work.

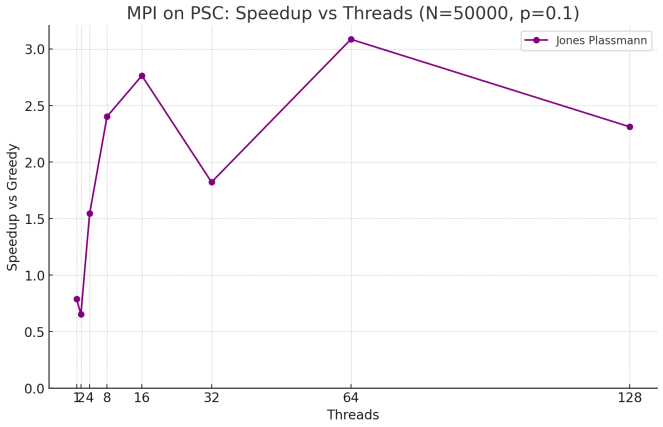
B.1. PSC Machines

We also used the PSC machines in order to test how the implementation scaled up to 128 threads. Specifically, we tested it on $1, 2, 4, 8, 16, 32, 64$, and 128 processors on the PSC machines.

For our well-performing test graphs, a similar trend as observed in the gates machines was visible for the PSC machines' data. However, for the poorer performing graphs, there was a very noticeable peak when the number of processors was 8, and then the implementation began doing very poorly for higher processor counts (np=16,32,64,128) when compared to the greedy sequential algorithm.



(a) Small Sparse Graph ($N = 10000, p = 0.01$)



(b) Large Dense Graph ($N = 100000, p = 0.1$)

Fig. 18. Comparison of scaling on PSC machines for different graph sizes and densities.

Figure 18 shows that the same trend observed on the GHC machines holds true for very large and dense graphs on the PSC machines even at higher processor counts.

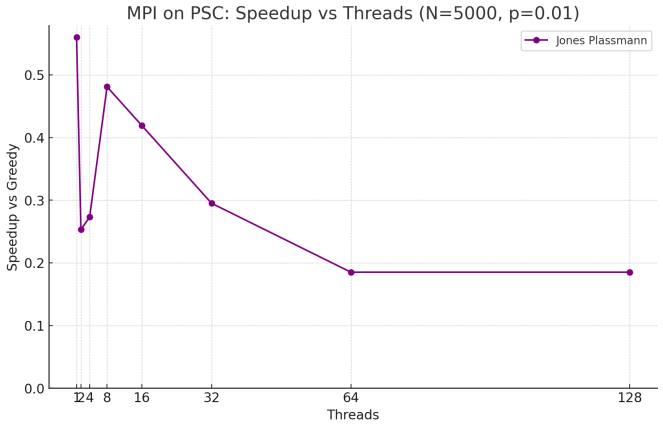


Fig. 19. Very Small Sparse Graph ($N = 50000, p = 0.1$)

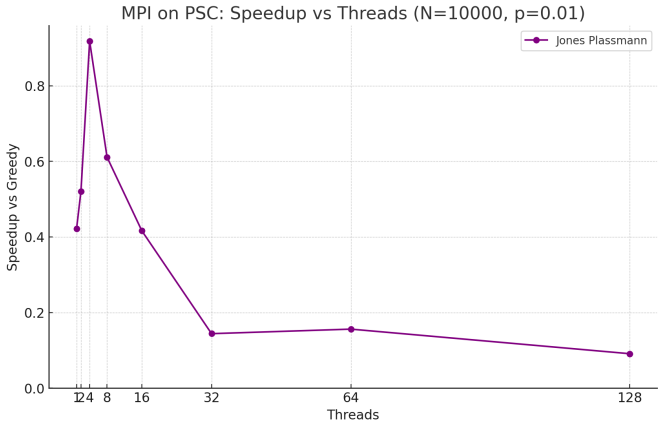


Fig. 20. Small Sparse Graph ($N = 100000, p = 0.1$)

Figure 20 and **21** show that for the smaller, sparse graphs, the performance worsens as the number of processors increases on the PSC machines. Our understanding for why this is happening is that once we go past eight processors on those small, sparse graphs, each process only has a few hundred vertices to color per iteration, so the actual work it does shrinks dramatically—yet we still perform the same two global `MPI_Allreduce` synchronizations each round. As the useful work per process decreases, the fixed cost of those collectives no longer gets hidden behind computation and quickly dominates the loop, causing the overall runtime to increase instead of decrease.

One promising optimization would be to only exchange the colors of "boundary" vertices—those that have neighbors on other processs—instead of performing a full-array `MPI_Allreduce` each iteration. By first collecting the indices and colors of these boundary vertices and then using custom point-to-point or neighborhood-collective calls to synchronize just that subset, we could drastically cut the volume of data sent and reduce synchronization overhead. Investigating this boundary-only communication strategy is a good starting point for improving our MPI implementation.

C. Cache Misses

Quickly scanning the tables below makes it clear that for all of the graphs - small or large, dense or sparse, the general trend is that the total number of cache misses increases as the number of processors increases, but the cache misses per processor stays roughly the same.

Table 2. MPI: Total Cache Misses (millions)

Nodes	Density	1 Proc	2 Proc	4 Proc	8 Proc	16 Proc	32 Proc	64 Proc	128 Proc
5000	Dense	5.16	9.39	15.50	29.03	55.44	125.36	270.71	684.86
5000	Sparse	1.78	4.02	7.86	16.57	32.83	79.27	173.61	484.66
10000	Dense	24.26	42.11	68.75	106.96	176.32	356.98	725.75	1576.59
10000	Sparse	3.60	7.77	15.26	28.09	50.38	113.34	238.35	612.04
50000	Dense	880.65	1518.70	2529.93	4671.07	8232.70	15778.51	31520.27	67150.21
50000	Sparse	175.80	229.92	377.57	530.55	946.84	1816.20	3955.10	8576.37
100000	Dense	4009.71	6601.38	10660.58	17941.62	32535.55	63126.23	–	–
100000	Sparse	1103.47	1435.62	1989.21	2790.50	4352.08	7797.48	15656.80	34657.00

Table 3. MPI: Cache Misses per Processor (millions)

Nodes	Density	1 Proc	2 Proc	4 Proc	8 Proc	16 Proc	32 Proc	64 Proc	128 Proc
5000	Dense	5.16	4.70	3.88	3.63	3.47	3.92	4.23	5.35
5000	Sparse	1.78	2.01	1.97	2.07	2.05	2.48	2.71	3.79
10000	Dense	24.26	21.06	17.19	13.37	11.02	11.16	11.34	12.32
10000	Sparse	3.60	3.89	3.82	3.51	3.15	3.54	3.72	4.78
50000	Dense	880.65	759.35	632.48	583.88	514.54	493.08	492.50	524.61
50000	Sparse	175.80	114.96	94.39	66.32	59.18	56.76	61.80	67.00
100000	Dense	4009.71	3300.69	2665.15	2242.70	2033.47	1972.69	1079.46	544.95
100000	Sparse	1103.47	717.81	497.30	348.81	272.01	243.67	244.64	270.76

Since each MPI process keeps its own full copy of the graph data structures — adjacency lists, color array, priority array, when you double the number of processes, you effectively double the total memory footprint and the amount of cache activity happening on the machine. On every coloring iteration, we run a full-array `MPI_Allreduce`, which causes every process to write out every color entry and then reload it on the next pass. As the Allreduce grows from 8 processes to 16, 32, 64, and beyond, that collective touches more endpoints and issues more memory writes, so each process ends up fetching more data out of main memory even though it's coloring fewer vertices, causing a nearly linear rise in total cache misses.

At the per-process level, cache misses stay roughly constant instead of dropping. Even though each process's local vertex block shrinks, the Allreduce forces a full invalidation of its cached color array every iteration, so each core

must reload that data, no matter how small its private workload is.

D. CUDA

Tables 4 and 5 show that our CUDA implementation delivers its best speedups on sparse graphs. When the edge probability is p=0.001, there are much fewer edges and so each thread's neighbor list is very short, so the `color_independent_set` kernel function colors most vertices in one or two launches. This lets the GPU hide memory latency behind thousands of concurrent threads and amortize the cost of kernel launches, yielding up to ∼4x speedup over the greedy implementation and over 30× over sequential Jones–Plassmann for $N = 50{,}000$. Even for medium sizes (N=10000), we see 3–3.5x speedups with only 128 or 256 threads. The number of threads has no substantial impact on the speedup.

In contrast, dense graphs (p=0.01 and above) force each thread to scan long CSR slices, causing significant warp divergence. Also since more iterations are required before Jones-Plassmann terminates there is much more synchronization overhead associated with repeatedly launching the kernel function. As a result, speedup versus the greedy baseline often drops below 1x for large, dense inputs (for example. N=100,000), even though the GPU still outperforms sequential Jones–Plassmann by 3–13×.

This behavior differs with our MPI results. In MPI, sparse graphs suffered from fixed-cost collectives dominating small workloads, while dense graphs benefited from high compute-to-communication ratios. On the GPU, the launch cost and divergence overhead make dense graph coloring less effective. But sparse graphs map perfectly to CUDA's SIMT model, with minimal per-thread work and few synchronizations, delivering the largest performance gains.

Table 4. CUDA-JP Speedup (128 threads)

Vertices	Edges	Edge Prob	**Speedup vs Greedy**	Speedup vs Seq JP
5,000	12,606	0.001	**1.81244**	6.49218
5,000	124,834	0.01	**1.26159**	3.06706
10,000	50,038	0.001	**3.52695**	11.01224
10,000	500,764	0.01	**1.35579**	3.69871
50,000	1,250,966	0.001	**3.96502**	29.28163
50,000	12,507,465	0.01	**0.63645**	12.95158
100,000	5,002,375	0.001	**3.87039**	32.69369
100,000	12,502,609	0.0025	**2.04431**	29.22989
100,000	25,006,004	0.005	**1.03794**	22.79843
100,000	49,982,163	0.01	**0.44771**	12.56272

Table 5. CUDA-JP Speedup (256 threads)

Vertices	Edges	Edge Prob	**Speedup vs Greedy**	Speedup vs Seq JP
5,000	12,580	0.001	**1.61522**	6.07356
5,000	124,622	0.01	**1.22057**	3.06706
10,000	50,139	0.001	**3.31622**	10.62849
10,000	500,265	0.01	**1.34622**	3.44118
50,000	1,251,369	0.001	**4.01577**	29.16895
50,000	12,496,680	0.01	**0.64923**	12.97750
100,000	4,999,845	0.001	**3.64796**	30.87780
100,000	50,005,336	0.01	**0.43711**	12.48814

REFERENCES

1. H. Sundaram, *Randomized Algorithms*, University of Utah, Lecture Notes, 2015. [Online]. Available: https://users.cs.utah.edu/~hari/teaching/bigdata/06_Randomized_Algorithms.pdf
2. J. Gebremedhin and F. Manne, "Scalable Parallel Graph Coloring Algorithms," *Concurrency and Computation: Practice and Experience*, vol. 12, no. 12, pp. 1131-1146, 2000.
3. J. Cohen and S. Toledo, "Graph Coloring: More Parallelism for Incomplete LU Factorization," NVIDIA Developer Blog, 2016. [Online]. Available: https://developer.nvidia.com/blog/graph-coloring-more-parallelism-for-incomplete-lu-factorization/
4. D. Sunderland, M. Hoemmen, and K. Teranishi, "Performance Portability Strategies for Graph Coloring on KokkosKernels," Sandia National Laboratories, Technical Report, 2016. [Online]. Available: https://www.osti.gov/servlets/purl/1246285

E. Work Performed by Each Student

The work was distributed $50\% - 50\%$ between us.

- Graph Interface Implementation + Testing Suite (Sanjana)
- Sequential Solution (Sanjana)
- Jones-Plassmann Sequential + Initial Parallel using OpenMP (Sanjana, Aditri)
- Jones-Plassmann OpenMP optimized (Sanjana)
- Speculative Coloring using OpenMP (Sanjana)
- Priority-Based Conflict Resolution using OpenMP (Sanjana)
- CUDA Jones-Plassmann (Aditri)
- MPI Jones-Plassmann (Aditri)
- Benchmark numbers, results, report, poster (Sanjana, Aditri)