Parallel Algorithms for Greedy Graph Coloring

Sarah Pethani (spethani) and Jeff Tan (jefftan)

April 7, 2021

1 Summary

We are going to implement different parallel algorithms for greedy graph coloring in OpenMP and possibly CUDA. We will analyze/compare their performance, speedup, and solution quality (e.g. does one algorithm use less colors than another).

2 Background

Minimal graph coloring is the assignment of colors to the vertices on a graph, such that no two neighboring vertices are assigned the same color. Graph coloring is an NP-hard problem, with applications in job scheduling, register allocation, and many similar problems that are modelable using conflict graphs. For this reason, heuristic algorithms have been developed that make the trade-off between runtime and solution quality. However, many simple greedy approaches to graph coloring are inherently sequential due to dependencies on the colors of existing nodes in the graph.

One such algorithm is the sequential greedy algorithm, which iterates through all vertices and assigns to each vertex the smallest available color that is not shared by any neighbors. While this algorithm is easy to implement and has good sequential performance, it does not produce an optimal solution, and is also very difficult to parallelize due to the explicit time dependency between vertex colors.

Algorithm 1 Sequential Greedy Algorithm [10]

```
1: procedure GREEDY(G(V, E))
2:     for each vertex v ∈ V do
3:         for each vertex w ∈ adj(v) do
4:             colorMask[color[w]] ← v
5:         end for
6:         c ← min {i > 0 : colorMask[i] ≠ v}
7:         color[v] ← c
8:     end for
9: end procedure
```

Two alternative greedy graph coloring algorithms have been proposed to address these issues: the Jones-Plassman (JP) and Gebremedhin-Manne (GM) algorithms.

The JP algorithm attempts to parallelize the graph coloring problem by iteratively finding independent sets of uncolored vertices, and coloring these vertices in parallel. At each step, each remaining

uncolored vertex is assigned a random number, and all vertices whose numbers are a local maximum are colored the next available color. This process repeats until all vertices are colored. Since the computation at each vertex depends only on its neighbors, vertices can be operated on in parallel as long as there is a mechanism for communication between neighbors.

Algorithm 3 Parallel JP Algorithm [7]

```
1:  procedure JP(G(V, E))
2:      W ← V, c ← 1
3:      while W ≠ ∅ do
4:          S ← ∅                          ▷ Initialize the independent set
5:          for each vertex v ∈ W in parallel do
6:              r(v_i) ← random()
7:          end for
8:          for each vertex v ∈ W in parallel do
9:              flag ← true
10:             for each vertex w ∈ adj(v) do
11:                 if r(v) <= r(w) then
12:                     flag ← false
13:                 end if
14:             end for
15:             if flag = true then
16:                 S ← S ∪ {v}
17:             end if
18:         end for
19:         for each vertex v ∈ S in parallel do
20:             color[v] ← c               ▷ Color an independent set
21:         end for
22:         W ← W − S, c ← c + 1
23:     end while
24: end procedure
```

The GM algorithm is a speculative approach that attempts to color as many vertices as possible in parallel, potentially resulting in some conflicts in neighboring vertex colors, which are later resolved. This algorithm has looser restrictions in terms of deciding when to color in a vertex as compared to the JP algorithm, but as a result, has a backtracking aspect to solve problems caused by the looser restrictions.

Algorithm 2 Parallel GM Algorithm [10]

```
1:  procedure GM(G(V, E))
2:      W ← V                              ▷ Initialize the worklist
3:      while W ≠ ∅ do
4:          for each vertex v ∈ W in parallel do
5:              for each vertex w ∈ adj(v) do
6:                  colorMask[color[w]] ← v
7:              end for
8:              c ← min {i > 0 : colorMask[i] ≠ v}
9:              color[v] ← c
10:         end for
11:         R ← ∅                          ▷ Initialize the remaining worklist
12:         for each vertex v ∈ V in parallel do
13:             for each vertex w ∈ adj(v) do
14:                 if color[v] = color[w] and v < w then
15:                     R ← R ∪ {v}
16:                 end if
17:             end for
18:         end for
19:         W ← R                          ▷ Update the worklist
20:     end while
21: end procedure
```

3 Challenge

Though the pseudocode for the Jones-Plassman and Gebremedhin-Manne algorithms seem simple at first glance, there is often need for synchronization in terms of creating new sets of vertices that have yet to be colored by the algorithm. In the JP algorithm, we cannot run another iteration of the main outside loop before all of the threads have completed the previous iteration. There is also a need for updating shared memory which needs to be atomic between different threads. For example, each thread must atomically update the graph at the end of each iteration.

Additionally, the pseudocode for these two algorithms assumes the use of sets for representing groups of vertices. We will have to decide if it is better to use built-in C++ sets or if we should have our own implementation which takes into consideration the characteristics of the coloring problem.

The workload of different threads is not necessarily balanced. In the GM algorithm, if one vertex shares a color with any of its neighbors, we will have to complete additional updates to the vertex set to resolve these conflicts, while other threads will also be contending for access to this vertex set. In addition, since the topology and characteristics of the input graph are unknown, some threads may end up taking much longer than others if we choose a simple static assignment.

4 Resources

We will be using the Gates machines and the Lateday clusters to measure performance and speedup. Additionally, we will be referring to https://chenxuhao.github.io/docs/ipdpsw-2016.pdf for the pseudocode of the different greedy coloring algorithms. We will be writing our code from scratch.

5 Goals and Deliverables

We plan to achieve implementations of Jones-Plassmann and Gebremedhin-Manne in OpenMP along with analysis of their performance and speedup on both the Gates and Latedays machines. We also plan to implement the sequential greedy algorithm in C++ as a baseline for comparison.

We hope to achieve (as a stre implementations of the two parallel algorithms in CUDA and compare the performance and speedup to the OpenMP implementations.

From this, we plan to learn if a parallel backtracking algorithm such as GM outperforms a straightforward parallel implementation such as JP. We also hope to learn about any potential tradeoffs in terms of synchronization requirements, communication requirements, ease of work distribution, and solution quality (how minimal is the coloring).

We plan to report the resulting performance, speedup, and solution differences of both parallel algorithms on varying types of graphs. We will also report any conclusions made about the tradeoffs between the algorithms based on this data.

6 Platform Choice

We choose to complete this project using OpenMP (and maybe CUDA) as they provide a more intuitive approach when using shared memory and parallelizing over iterations of a loop.

7 Schedule

- Week 1 (4/14):
  - Implement sequential baseline and begin implementations of Jones-Plassman and Gebremedhin-Manne.
- Week 2 (4/21):
  - Finish basic implementations of Jones-Plassman and Gebremedhin-Manne in OpenMP.
  - Start writeup for checkpoint.
- Week 3 (4/28):
  - Start analyzing performance and speedup of the two algorithms.
  - Optimize implementations of Jones-Plassman and Gebremedhin-Manne to have better workload balance, less communication, and less memory contention.
- Week 4 (5/5):
  - Generate final data for performance, speedup, and solution quality of the different implementations.
  - Create a final poster and report.