CS 406 Parallel Computing
Project Final Report

Oran Can Oren, Muvaffak Onus, Ege Yapicigolu, Oguz Yuksek

December 2017

1 Problem Statement

Graphs have become a widely used tool for modeling computational problems. One of the important graph problems is the k-coloring problem. Some notable problems that can be reduced to k-coloring are task scheduling, register allocation and social network analysis problems.

Our task is to implement algorithms solving the distance 2 coloring problem, i.e. a coloring of the vertex set such that any pair of distance-1 and distance-2 neighbors receive distinct colors. Graph coloring is a hard problem to compute solutions for, hence our aim is to implement greedy heuristics that utilize parallelism. Parallelism on CPU will be achieved by using OpenMP and by CUDA on GPU.

2 Overview

We have implemented the ideas introduced by [1] to obtain parallelized, efficient and accurate algorithms solving the distance-2 graph coloring problem. In this project, we introduce five different implementations. The first three implementations do not employ use of forbidden array but others do. First one is parallelized on CPU, second operates on GPU and the other is a heterogeneous algorithm dividing the computation workload to both CPU and GPU. The last two implementations that employ forbidden array are only CPU and only GPU. Following subsections describe these algorithms in detail.

2.1 The main algorithm

The algorithm proposed by Deveci, et al.[1] has performed very fast for the distance-1 coloring problem; therefore we have decided to extend the algorithm to solve the distance-2 graph coloring problem. The algorithm is iterative, it applies three main phases at each *round.*

First step is the *assign colors* phase. In this step, for all uncolored vertices v, the algorithm assigns the smallest color v can be assigned to with respect to its forbidden colors. The second step is the *detect conflicts* phase. The algorithm traverses over all vertices v_i that was colored on round i and checks whether a conflict has occurred. If this is the case, then the vertex having the smaller label is dissociated from its color. As for the last step, *forbid colors* phase, algorithm traverses the vertices v_i that was colored on round i once again to forbid the colors of v_i's distance 2 neighbors. These colors are forbidden so that on round $i + 1$ these vertices are not assigned these colors that will lead to same conflicts once again. Algorithm ensures that at least one of the colored vertices at each round will be permanently associated with its color, hence the number of uncolored vertices at each round is strictly decreases.

Algorithm 1: Outline for the main algorithm

```
Result: color $c_i$ for $\forall v_i \in V$
1 while $\exists v_i \in V$ not colored do
2     assignColors();
3     detectConflicts();
4     forbidColors();
5 end
```

2.2 Handling forbidden colors

In our previous report, we have stated our observations with the performance drop down caused by forbidden arrays. Initially for the final step of the project, we have decided to continue with the same approach. In the end, we had three implementations of the algorithm discussed in subsection 2.1. However, execution times of our implementations took very long in comparison to those proposed in the literature.

Section 3 discusses these algorithms that use forbidden array, and showcases their execution times. In order to speed up our implementations, by the guidance of our instructor, we have proceeded by building storage mechanisms for forbidden colors. These algorithms were implemented after the project presentation, where we have comprehended the necessity of forbidden arrays in our implementations.

Our modified CPU implementation now uses thread-private arrays to keep forbidden colors during conflict detection phase; moreover, our CUDA implementation uses an array of 64-bit integers on shared memory, which is on-chip, to hold forbidden colors in form of bit vectors. Details of our modified implementations are located in section 4.

3 Implementations without forbid colors phase

Following subsections showcase the execution times and discusses implementation details of our OpenMP, CUDA and heterogeneous implementations. These algorithms do not perform the *forbid colors* phase.

3.1 OpenMP implementation

The implementation performs *assign colors* and *detect conflicts* over the vertex set until the latter phase cannot dissociate the color of any vertex. Each thread is assigned a portion of the vertex set to perform the aforementioned steps.

At the *assign colors* phase, each thread assigns the minimum available color to the vertices it owns. Minimum color is found by a straightforward traversal strategy. For a vertex v, the thread owning v checks distance-1 and distance-2 neighbors of it until an available color is found. To do so, the operation starts by checking color 1, as it is the smallest color possible. As neighbor checks proceed, when a color match is found, operation restarts with color 2 and goes on until an available color is found.

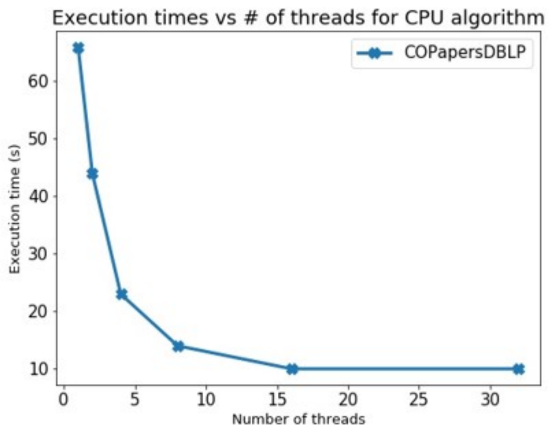
The *detect conflict* phase is essentially similar. For each vertex, distance-1 and distance-2 neighbors are checked to find matching colors. When a match is found among two vertices, the vertex having smaller id is dissociated from its color.

As section 2 describes, both above-mentioned phases are performed until no conflicts are detected. We have tried various task scheduling methods. OpenMP's dynamic schedule has yielded the best results. An execution time table and its plot is located below. The program outputs 3100-3300 colors.

Table 1: Execution times vs threads for coPapers

1 th	**2 th**	**4 th**	**8 th**	**16 th**	**32 th**
66 s	44 s	23 s	14 s	10 s	10 s

Figure 1: Execution times vs threads for coPapers



```
Algorithm 2: Iterative find min color for $v$
    input: $v$: vertex id
           $C$: color array
 1  match_found = true;
 2  current_color = 1;
 3  while match_found do
 4      match_found = false;
 5      for $n_i^1 \in N[v]$ and not match_found do
 6          if $C[n_i^1] = C[v]$ then
 7              match_found = true;
 8              current_color += 1;
 9              break;
10          end
11          for $n_i^2 \in N[n_i^1]$ do
12              if $C[n_i^2] = C[v]$ then
13                  match_found = true;
14                  current_color += 1;
15                  break;
16              end
17          end
18      end
19  end
```

3.2 CUDA implementation

We have obtained GPU-parallelized implementation by assigning each GPU thread a vertex. The strategies are very similar to those of the OpenMP implementation, as described in subsection 3.1. Each thread is assigned the task of assigning colors and detecting conflicts to the vertex they are given.

Initially for the color assignment phase, same strategy for finding the minimum available color was used as in the OpenMP implementation; however, this strategy performed very slow. To overcome this, we have completely removed the color assignment phase and modified conflict detection phase. The modified conflict detection function now increments the color of the vertex having smaller label instead of dissociating its color when a conflict is detected. If this incremention results in another conflict, further invokations of the modified conflict detection function are left to fix it.

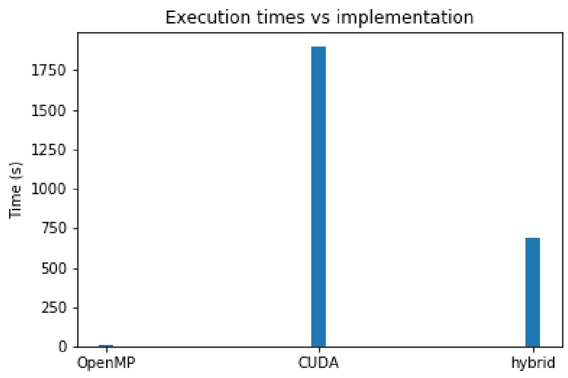
The implementation resulted in taking about 1800-2000 seconds to terminate for the coPapers graph, yielding 5952 colors. This implementation is problematic because the number of conflicts do not strictly decrease by iterations, as the modifed conflict detection function simply increments the color of the vertex having smaller label. We have tried to use the traditional conflict detection algorithm, where in case of a conflict the color of vertex having smaller label is reset to unknown. However, this algorithm did not terminate in an acceptable time period and thus the incremention strategy took place in our implementation.

3.3 Hybrid implementation

Due to execution time issues encountered during the conflict detection phase, we have moved the computation of conflict detection phase to CPU. The *assign colors* and *detect conflicts* functions are same as in those described by subsections 3.1 and 3.2. This implementation simply combines the GPU implementation of color assignment phase and CPU implementation of conflict detection phase; down-side of this implementation is the memory copy operation for the colors array after each color assignment and conflict detection phase.

The implementation takes about 685 seconds to terminate, yielding 3319 colors with 16 CPU threads.

Figure 2: Execution times vs threads for coPapers



Algorithm 3: Unified conflict detection / color assignment algorithm

```
input: $v$: vertex id
       $C$: color array
1  thread private forbidden_colors;
2  for $n_i^1 \in N[v]$ do
3      forbidden_colors[C[$n_i^1$]]=true;
4      for $n_i^2 \in N[n_i^1]$ do
5          forbidden_colors[$C[n_i^2]$]=true;
6      end
7  end
8  if $forbidden\_colors[C[v]]$ then
9      for $c_i \in forbidden\_colors$ do
10         if $c_i = 0$ then
11             $C[v] = i$;
12             break;
13         end
14     end
15 end
```

4 Implementations with forbid colors phase

The implementations discussed in the following subsections have a storage mechanism for forbidden colors per thread. Therefore, the third phase of the algorithm discussed in section 2, *forbid colors*, is included in these algorithms.

4.1 OpenMP implementation

In this implementation, each thread keeps a thread-private array for keeping the forbidden colors. Having this array, it is sufficient to traverse the distance-1 and distance-2 neighbors once to populate the forbidden colors, so that further availability checks for different colors can be done in constant time.

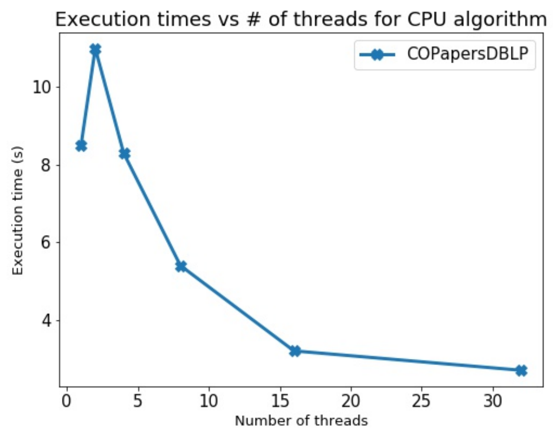
For a given vertex, the algorithm traverses over distance-1 and distance-2 neighbors to mark the forbidden colors by using their id as index. Afterwards, a single traversal over this array yields the minimum available color. This approach has increased the performance significantly. Furthermore, we have removed the color assignment function and left its task to a the modified conflict detection function. In order to achieve this, we populate the result array before the iterations by assigning color numbers to distance-1 neighbors as their indexes without checking for conflicts.

In the end, the procedure for each thread is to fill the forbidden color array and check if there is a conflict. If there is a conflict, then a traversal of the forbidden array commences. The thread checks for the minimum available color in this array and when it's found, this color is assigned to the vertex. A table and its plot showcasing the execution times of the algorithm is provided below.

Table 2: Execution times vs threads for coPapers

1 th	**2 th**	**4 th**	**8 th**	**16 th**	**32 th**
8.5 s	11 s	8.3 s	5.4 s	3.2 s	2.7 s

Figure 3: Execution times vs threads for coPapers



4.2 CUDA implementation

The forbidden array approach in OpenMP implementation is not applicable in CUDA; as each thread is responsible from a vertex, having an array for each thread would yield tremendous memory requirements.. To overcome this challenge, we used 64-bit integers to represent bit vectors. Each block stores an array of size 5120 storing such integers. Afterwards, we use a padding scheme for this array, allowing each thread to use a 4096-bit portion of the array. The reason to use 5120 long integer is the constraint of the hardware, that is shared memory space per block is 48KB and we use only 40KB of that to prevent occupancy problems. Since number of bits that each thread needs is constant, we had to decrease number of threads per block to 80.

This implementation doesn't have an explicit color assignment phase, as in the OpenMP implementation. Instead, each thread is assigned possible colors initially. Afterwards, we invoke a modified version of the conflict detection phase until all vertices are successfully colored. The modified conflict detection function is very similar to that of the OpenMP impelementation's. First, the forbidden array is filled. Afterwards, if the color of the vertex is marked in the forbidden color array, a traversal to find the smallest available color starts. Once the color is found, the vertex is assigned this color.

We used the block-shared cache to store the forbidden arrays because it is the fastest memory access structure available in current Nvidia GPU architecture. In contrast to our expectations, the implementation performed very slow.

References

[1] Deveci, Mehmet Boman, Erik D. Devine, Karen Rajamanickam, Siva. (2016). Parallel Graph Coloring for Manycore Architectures. 892-901. 10.1109/IPDPS.2016.54.