# HW02 REPORT

**April, 2019**

Yiğit Aras Tunalı
17519
Sabancı University
CS406 - Parallel Computing

# Contents

# 1 Problem Statement

The graph coloring problem is a NP-Hard problem. Since the main approach to solve this problem we will be taking involves a Greedy approach, it is quite hard to implement a parallel algorithm in an efficient manner. There are many challenges to the implementation and they are mainly caused by the inherent dependency of tasks and the sequential nature of the Greedy algorithm.

# 2 Sequential Implementation

The sequential implementation is simply coloring the graph in a greedy manner, using a ForbiddenColours array to track the colours which shouldnt be used for a given vertice, calculated using it's neighbours.

**Algorithm 1** Sequential greedy colouring

1: **for** $v \in V$ **do**
2: **for** $w \in adj(v)$ **do**
3: $forbiddenColours[colour[w]] \leftarrow v$
4: $colour[v] \leftarrow min(c > 0 : forbiddenColours[c] != v)$

The results for the sequential run on the graphs are as follows;

| Result Mat | coPapersDBLP | europe_osm | rmat_b | rmat_er | wiki_topcats |
|---|---|---|---|---|---|
| Time (s) | 0.060267s | 0.834738 | 1.51241 | 2.28713 | 0.13823 |
| Colour Count | 337 | 5 | 85 | 11 | 61 |

# 3 Parallel Implementation and Tricks

One of the first problems to overcome while parallelizing the algorithm was to break the dependency caused by ForbiddenColours array. Each thread randomly accessing and writing to the same array caused a lot of performance issues, through data races and false sharing. So first idea that came to mind was to assign each thread their own ForbiddenColours array and do their neighbour checks independently from each other. The previously introduced problem of conflict check still isn't solved with seperate ForbiddenColours array and requires and additional step in which the conflicts are resolved iteratively. Unfortunately with this approach there is no significant speed-up gained. Main reason is the iterative nature of the approach, coupled with the critical region where the erroneous indices are updated and held. The task distribution during the error correcting parts seems to be unbalanced because of the low number of fixes needed. I have spent quite some time to try and get rid of the critical block, but after taking timings it turns out that critical area takes considerably less time than the colouring part of the code so critical area will remain in the code. The algorithm is as follows;

**Algorithm 2** Parallel Greedy Colouring

```
1: Errors ← V
2: while Errors ≠ 0 do
3:     for v ∈ Errors in parallel do
4:         for w ∈ adj(v) do
5:             forbiddenColours[colour[w]] ← v        ▷ Each thread gets seperate forbiddenColours array
6:         colour[v] ← min(c > 0 : forbiddenColours[c] ≠ v)
7:     Errors ← ∅
8:     for c ∈ V in parallel do
9:         if c has a conflict then
10:            #pragma omp critical
11:            Errors ← c
```

Results after gained after the implementation for -O3 and -O1 are as follows.

| Tables | 2 Threads | 4 Threads | 8 Threads | 16 Threads |
|---|---|---|---|---|
| coPapersDBLP -O3 | 0.170556 | 0.147786 | 0.102480 | 0.085496 |
| coPapersDBLP -O0 | 0.444709 | 0.356112 | 0.200608 | 0.125077 |
| rmat-b -O3 | 4.572170 | 3.808348 | 2.641997 | 1.907850 |
| rmat-b -O0 | 13.626133 | 10.289160 | 8.190430 | 4.656767 |
| rmat-er -O3 | 4.905133 | 2.991285 | 1.749615 | 1.325735 |
| rmat-er -O0 | 30.507933 | 13.498000 | 5.778753 | 2.595760 |
| europe_osm -O3 | 1.115626 | 0.893347 | 0.597050 | 0.587792 |
| europe_osm -O0 | 2.734790 | 1.878003 | 1.168989 | 0.637990 |
| wiki-topcats -03 | 0.354046 | 0.274649 | 0.155468 | 0.119758 |
| wiki-topcats -O0 | 0.856322 | 0.561614 | 0.320650 | 0.213462 |

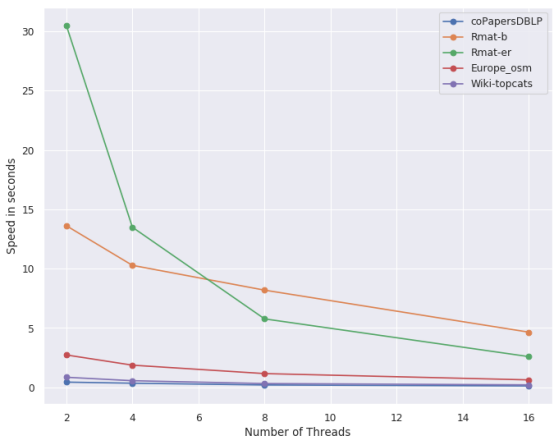



**Figure 1:** First results for the -O0 flag

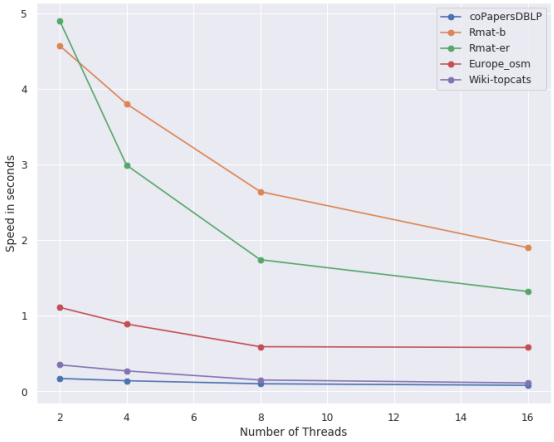



**Figure 2:** First results for the -O3 flag

### 3.1 Usage of Guided Scheduling

One of the tricks that I have used was to utilize the "guided" option of OpenMP scheduling. The children of a node, thus the task size, isn't known before-hand. This makes the option of dividing a for loop "statically" not an optimal approach. Upon comparison $schedule(guided)$ performed better and is utilized in the end code.

### 3.2 Usage of NUMA structure

Since in my approach of the code threads have their respective arrays that they keep track of, splitting them between cores using "OMP_PLACES=cores" and "proc_bind(spread)" improved the overall performance of the algorithm, especially in denser matrices.

### 3.3 Reducing the Task Size in Each Iteration

The main part of the approach that takes a lot of time is the colouring of the graph. At each iteration the size of necessary error correction can be determined at conflict determination phase and thus memory requirement and time requirement in the next iteration can be greatly reduced. This approach introduces a critical zone which only allows 1 thread to access and update the "errors" array, thus makes a time penalty but this is negligible compared to the amount of time the main part of the task takes, thus the critical region has been left untouched.

### 3.4 Memory Efficiency Tricks

After working around with the algorithm for a while, it came to my attention that some of the memory structures could be pre-initialized and used through-out without re-allocating them. To do this they have been placed earlier and just are updated during the algorithm to not effect the performance and the outcome.

Instead of re-allocation, "memset" used and the are that will be used is dynamically realized during run time which also increased the overall performance.

## 4 Final Tweaks and Final Results

After working around the code for a final time, I was able to cut down the working time of the conflict finder loop by limiting the size of it using the size of the set, array in this case, that holds the erroneous vertices. This further improved the performance of the algorithm, yielding better results. This coupled with the small, more pre-memory allocation and smarter looping that limits the range of the loop added to first loop improved the performance slightly further (also removing a slight mistake which was resulting in poor thread performance in the conflict resolution loop) resulted in following performance benchmarks;

| Tables | 2 Threads | 4 Threads | 8 Threads | 16 Threads |
|---|---|---|---|---|
| coPapersDBLP -O3 | 0.122695 | 0.091554 | 0.054536 | 0.066543 |
| coPapersDBLP -O0 | 0.373669 | 0.308577 | 0.217890 | 0.169786 |
| rmat-b -O3 | 2.564628 | 2.065490 | 1.625958 | 1.070088 |
| rmat-b -O0 | 8.633870 | 6.829453 | 5.225500 | 3.538230 |
| rmat-er -O3 | 3.213472 | 1.941690 | 1.013734 | 0.753238 |
| rmat-er -O0 | 11.690400 | 7.992003 | 3.790993 | 2.089340 |
| europe_osm -O3 | 1.032488 | 0.603667 | 0.338980 | 0.257261 |
| europe_osm -O0 | 2.233257 | 1.759103 | 1.156807 | 0.817814 |
| wiki-topcats -03 | 0.197200 | 0.129219 | 0.081993 | 0.073683 |
| wiki-topcats -O0 | 0.497843 | 0.476175 | 0.291855 | 0.182789 |

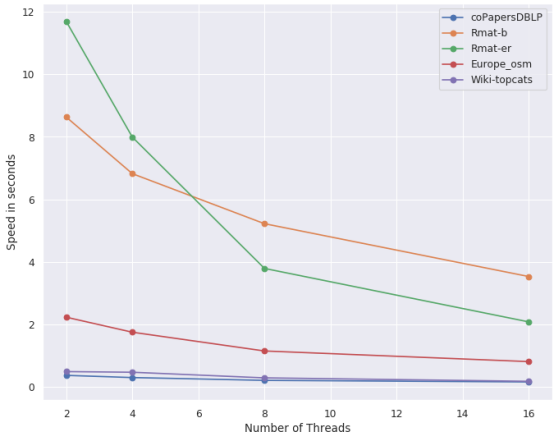



**Figure 3:** Improved results for the -O0 flag

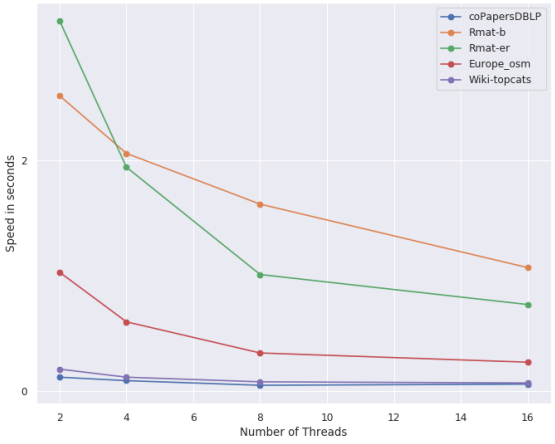
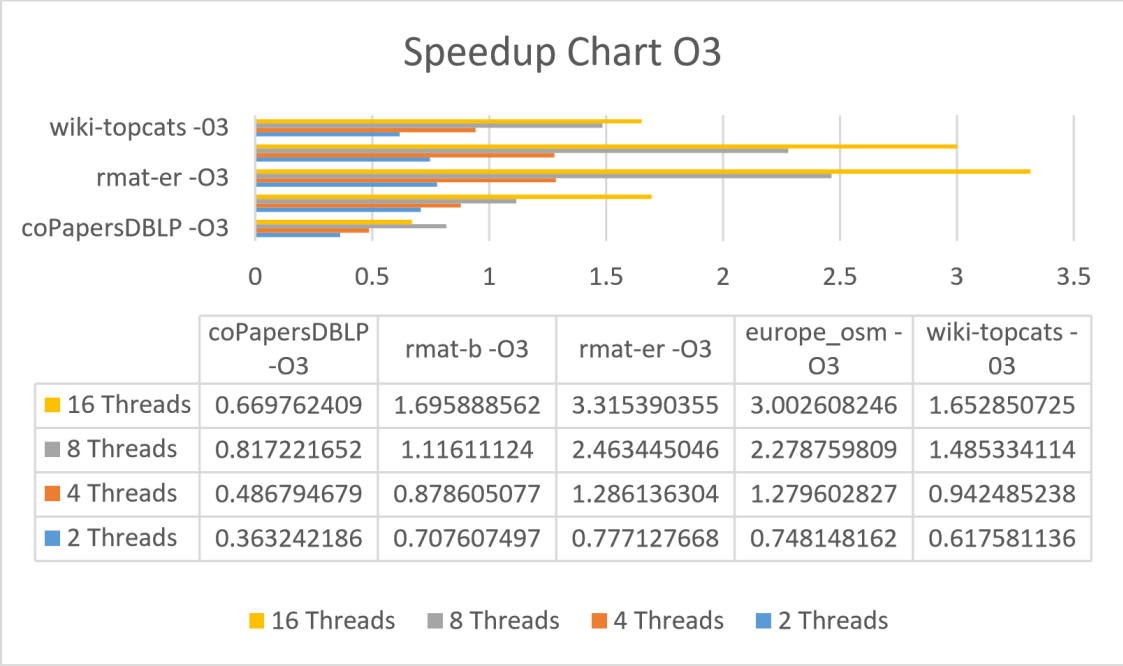
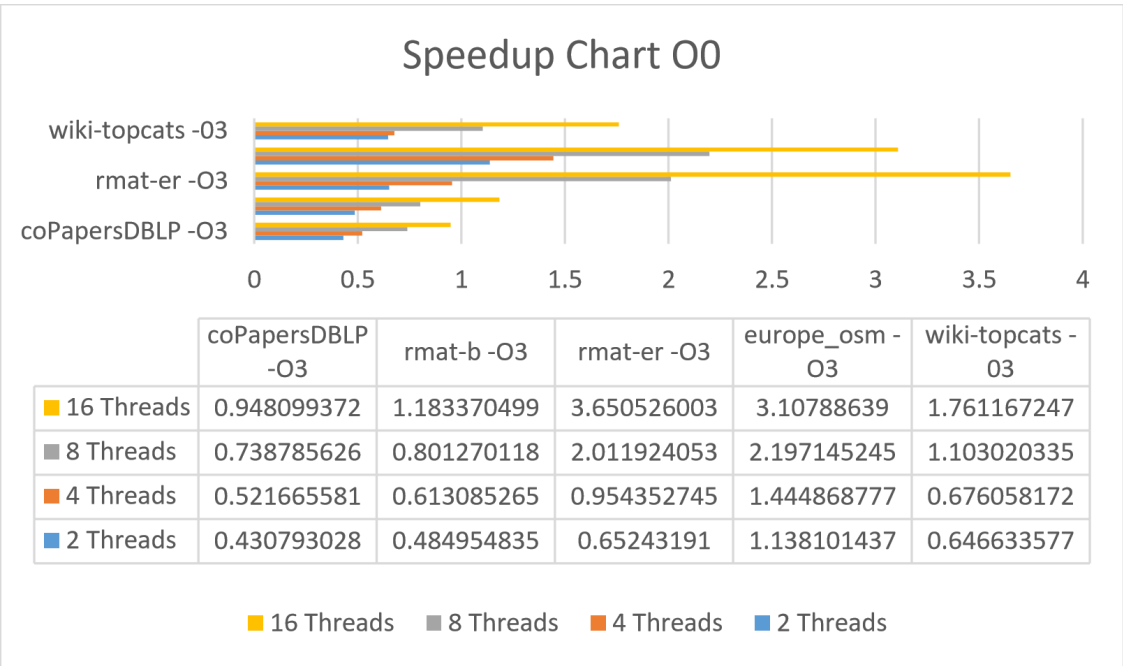



**Figure 4:** Improved results for the -O3 flag

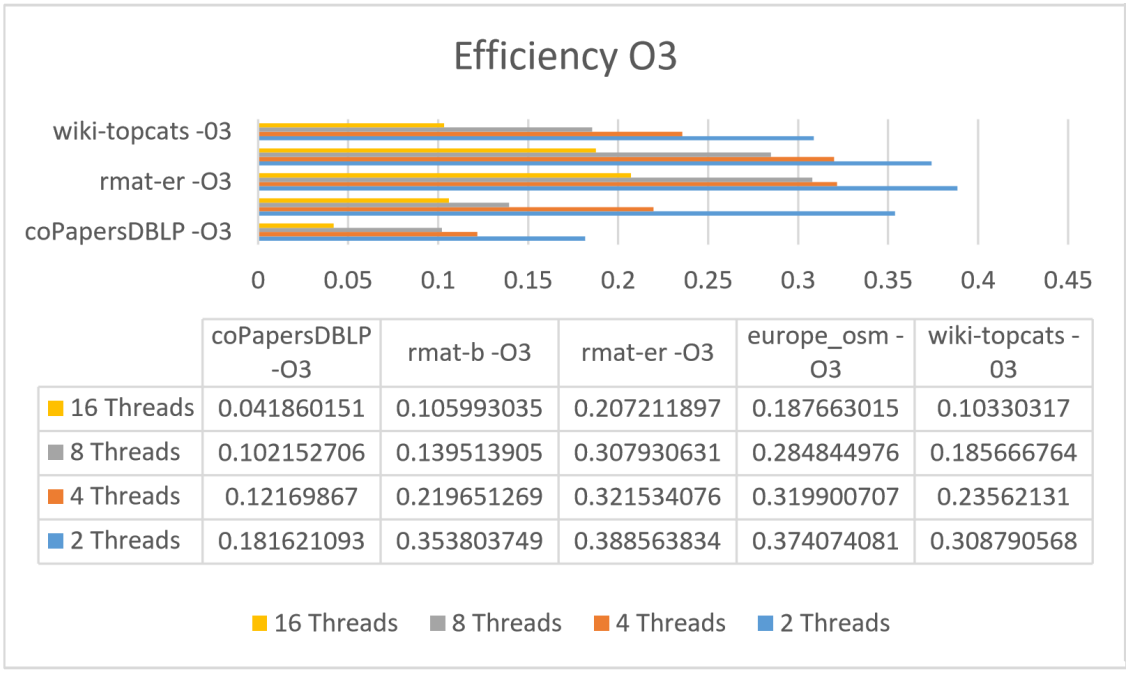
Efficiency and speed-up resulst of the tests are as follows as well (Note that there are results for 5 graphs but only the names of 3 are written they are in order wiki,rmat-b,rmat-er,europe and copaper from top to bottom);



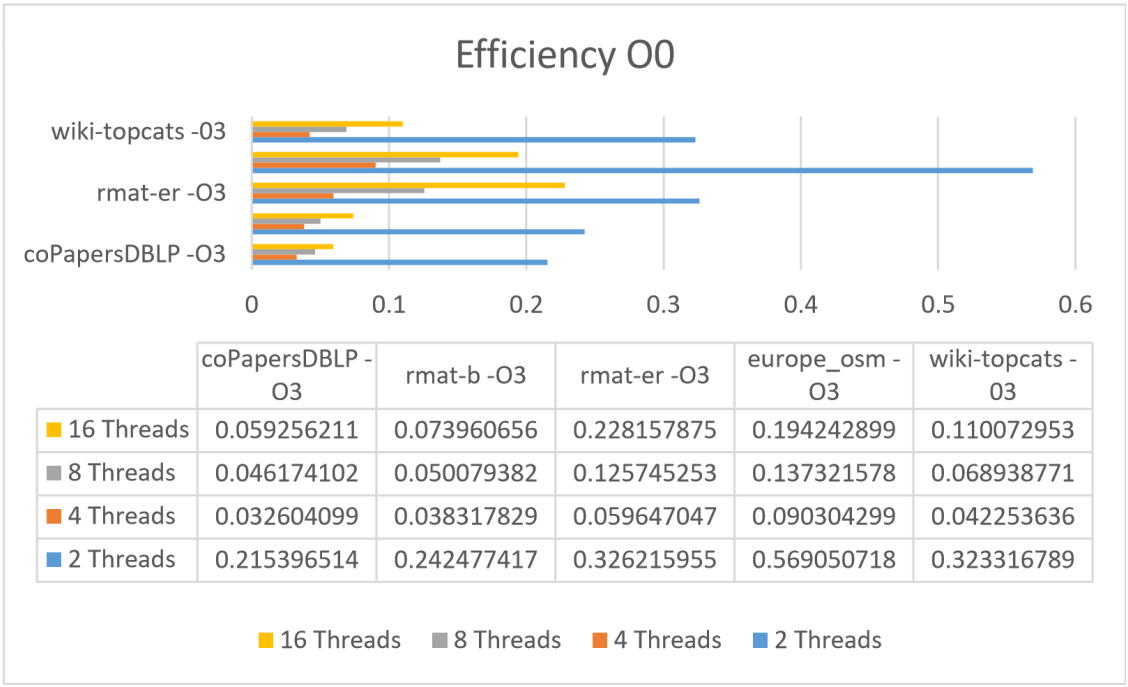

| | coPapersDBLP -O3 | rmat-b -O3 | rmat-er -O3 | europe_osm -O3 | wiki-topcats -03 |
|---|---|---|---|---|---|
| 16 Threads | 0.669762409 | 1.695888562 | 3.315390355 | 3.002608246 | 1.652850725 |
| 8 Threads | 0.817221652 | 1.11611124 | 2.463445046 | 2.278759809 | 1.485334114 |
| 4 Threads | 0.486794679 | 0.878605077 | 1.286136304 | 1.279602827 | 0.942485238 |
| 2 Threads | 0.363242186 | 0.707607497 | 0.777127668 | 0.748148162 | 0.617581136 |




| | coPapersDBLP -O3 | rmat-b -O3 | rmat-er -O3 | europe_osm -O3 | wiki-topcats -03 |
|---|---|---|---|---|---|
| 16 Threads | 0.948099372 | 1.183370499 | 3.650526003 | 3.10788639 | 1.761167247 |
| 8 Threads | 0.738785626 | 0.801270118 | 2.011924053 | 2.197145245 | 1.103020335 |
| 4 Threads | 0.521665581 | 0.613085265 | 0.954352745 | 1.444868777 | 0.676058172 |
| 2 Threads | 0.430793028 | 0.484954835 | 0.65243191 | 1.138101437 | 0.646633577 |




| | coPapersDBLP -O3 | rmat-b -O3 | rmat-er -O3 | europe_osm -O3 | wiki-topcats -03 |
|---|---|---|---|---|---|
| 16 Threads | 0.041860151 | 0.105993035 | 0.207211897 | 0.187663015 | 0.10330317 |
| 8 Threads | 0.102152706 | 0.139513905 | 0.307930631 | 0.284844976 | 0.185666764 |
| 4 Threads | 0.12169867 | 0.219651269 | 0.321534076 | 0.319900707 | 0.23562131 |
| 2 Threads | 0.181621093 | 0.353803749 | 0.388563834 | 0.374074081 | 0.308790568 |




| | coPapersDBLP -O3 | rmat-b -O3 | rmat-er -O3 | europe_osm -O3 | wiki-topcats -03 |
|---|---|---|---|---|---|
| 16 Threads | 0.059256211 | 0.073960656 | 0.228157875 | 0.194242899 | 0.110072953 |
| 8 Threads | 0.046174102 | 0.050079382 | 0.125745253 | 0.137321578 | 0.068938771 |
| 4 Threads | 0.032604099 | 0.038317829 | 0.059647047 | 0.090304299 | 0.042253636 |
| 2 Threads | 0.215396514 | 0.242477417 | 0.326215955 | 0.569050718 | 0.323316789 |

## 5 FINAL REMARKS AND HOW TO RUN THE CODE

To run the code please use "export OMP_PLACES=cores" directive first and compile with "g++ -fopenmp -std=c++11" flags. The overall performance of the algorithm, didn't go as high as I thought it would. I have looked into the results and the code for a long time but couldn't find the reason why it wouldn't be. Since the main code consists of two parts, 1st where the remaining vertices are coloured and 2nd where the errors are found and added as remaining vertices for the next iteration, I have timed their timings and tried to improve the most taxing step, which was the initial colouring phase. The timings were always as follows; X seconds for initial colouring, X/4 seconds for the first conflict finding and more than halved at each consecutive step. With the current algorithm I am working with, I wasn't able to find further improvements on how to improve the performance of the initial colouring phase, if the performance of the phase were to be improved the overall timing would drastically improve as well.