CS406 HW2

Adnan Kaan Ekiz

March 2019

1 Introduction

For the Homework 2 of the Parallel Computing Course, main idea was to efficiently implement a matrix coloring algorithm by using the greedy approach. During the process of implementing the code, it is important to recognize and briefly mention some of the key parts which seems crucial. For that reason, homework 2 report will present some idea about the algorithm and how it is implemented for sequential and parallel run, some of the concept used throughout the homework, some of the tricks and steps used in the implementation phase, some problems encountered during the implementation phase and time result of improvements with plotted graphs in addition to overall analysis.

2 Concepts

In this section, some concepts will be described first to help the understanding of the topic. These topics are:

- CRS Representation: Format represents a matrix M by three (one-dimensional) arrays, that respectively contain nonzero values, the extents of rows, and column indices.

3 Algorithm

To achieve the ultimate goal of low complexity, for this homework I have decided to use the greedy algorithm given by the homework documentation. Basically algorithm uses greedy approach to calculate the color for each vertex, then uses the adjacent vertices to determine the forbidden color for that matrix resulting with selecting the minimum available color value. For the sequential implementation, pseudocode of the algorithm as follows:

Algorithm 1 Sequential Greed Coloring

```
1: procedure GREEDY [G(V,E)]
2:     for each $v \in V$ do
3:         for each $w \in adj(V)$ do
4:             $forbiddenColors[color[w]] \leftarrow v$
5:         $color[v] \leftarrow min\{c>0: forbiddenColors[c] \neq v\}$
```

Hence, algorithm has a complexity of $O(n^2)$, where n is the maximum number of vertices in the graph which only occurs when the graph is a strongly connected graph. Main idea in the algorithm is to create a correctly colored graph for the calculated CRS representation.

3.1 Sequential Implementation

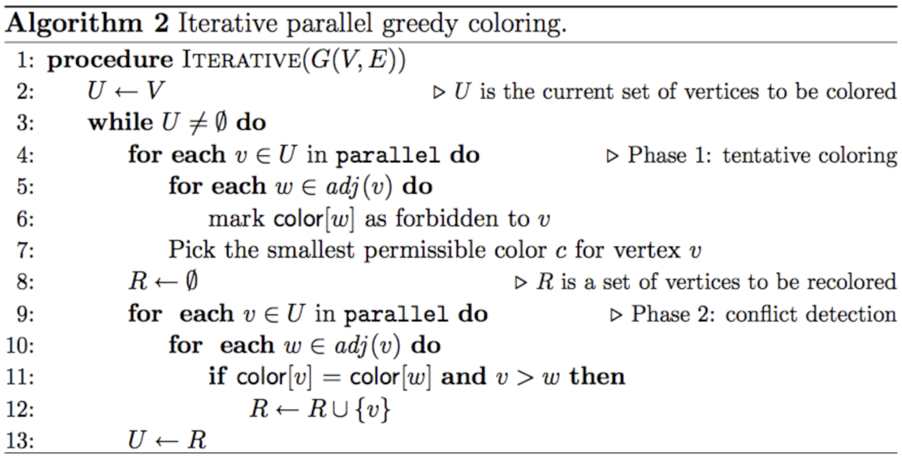
Given the pseudocode of the sequential algorithm above, algorithm has 3 different loop parts for the implemented version:

1. In the outer loop, algorithm was set to iterate through all of the vertices represented in the CRS form.
2. In the first inner loop, at each iteration, algorithm determines whether one of the adjacent vertices has a color assigned to it. If color is assigned then algorithm marks that color as forbidden to the vertex since we don't want adjacent vertices to have the same color.
3. Then, algorithm iterates through forbidden color array in order to find the color with minimum value that is marked as usable.

Overall, algorithm works quite efficient in terms of sequential time with given CRS representation of the graph when compared to given benchmarks.

3.2 Parallel Implementation

In terms of parallel implementation, creation of graph array has some additional work since each thread will cover some part of the graph color array hence resulting in collision. For that reason, algorithm also checks the collision cases in the array and tries to color the graph again until there is no collision left for the graph. Parallel structure and necessary computations are given in below as pseudocode:



Algorithm 2 Iterative parallel greedy coloring.

```
1:  procedure ITERATIVE($G(V, E)$)
2:      $U \leftarrow V$                              ▷ $U$ is the current set of vertices to be colored
3:      while $U \neq \emptyset$ do
4:          for each $v \in U$ in parallel do          ▷ Phase 1: tentative coloring
5:              for each $w \in adj(v)$ do
6:                  mark color[$w$] as forbidden to $v$
7:              Pick the smallest permissible color $c$ for vertex $v$
8:          $R \leftarrow \emptyset$                          ▷ $R$ is a set of vertices to be recolored
9:          for each $v \in U$ in parallel do          ▷ Phase 2: conflict detection
10:             for each $w \in adj(v)$ do
11:                 if color[$v$] = color[$w$] and $v > w$ then
12:                     $R \leftarrow R \cup \{v\}$
13:         $U \leftarrow R$
```

Given pseudocode above, implemented version of the code has following structure:

1. For the outer loop algorithm iterates for thread numbers t=1,2,4,8,16 respectively.

2. At the beginning algorithm runs until there is no need to color any remanining vertex. At the beginning all of the vertices are initialized such that algorithms starts to color all of the vertices in parallel order.

3. For the implemented version, once the parallel region is opened with openmp, algorithm copies the values of available array (is used to show the available colors in order to color the graphColor array) in order to avoid race condition which eventually causes miscalculation for the final result.

4. For the first for loop, implemented algorithm then uses parallel for region to distrubute the workload to threads in static scheduling order to find the available colors for each vertex and assign the minimum available color to that vertex.

5. For the second for loop, algorithm checks the whole graphColor array to see if there is any collisions and tracks the number of collisions in the whole graph for the recoloring process.

6. Once there isn't any vertex to recolor, algorithm finishes the iteration, and runs the pre-processing part of the code in order to use the same parallel algorithm for the next number of threads.

4 Problems & Tricks

4.1 Problems

During the implementation of the code, it was really hard to reduce the running time of the algorithm since there is more than one way for implementing the minimum color value. Hence it was important to use an algorithm that solves the given problem with minimum amount of loops. It was also important to use and efficient algorithm in the parallel implementation to find the vertices that needs to be recalculated in order to avoid the color conflicts in the graphColor array, solution was to implement an algorithm that overwrites the given array to avoid the array construction overhead. Algorithm for finding recolored vertices will be described in the "Tricks" section.

Another important topic is the usage of critical regions which dramatically affects the correctness of the final result. Implemented algorithm uses a critical region inside the second inner loop of the parallel algorithm which took to much time to solve in terms of collision results.

4.2 Tricks

It was important to get rid of the array creation overhead in the second loop of the parallel algorithm since alternative solutions like vectors and another array creation takes too much time and space in practice since creation of row sized arrays for each thread will use considerable amount of space. For that reason, algorithm uses one single array called reColor in order to show the vertices that needs to be re-colored in the next iteration. At the beginning of each iteration 2 variables are being initialized, ct is the number of total vertices to process for the next iteration and a serves as a counter to determine the number of total vertices that needs re-coloring. After that algorithm does the following:

1. Overwriting the reColor array to find vertices for re-coloring process and counting the total number of vertices. This part of the code is processed inside of a critical region to avoid race conditions.

2. At the end of each iteration, algorithm uses sets the new ct to a and a to 0 for the next iteration.

3. When ct becomes 0, algorithm terminates.

Another trick was to use proc_bind for the parallel region since it increased the speed of the parallel results 0.5 times by using export NUM_PLACES = cores which spread the workload between cores.

5 Results

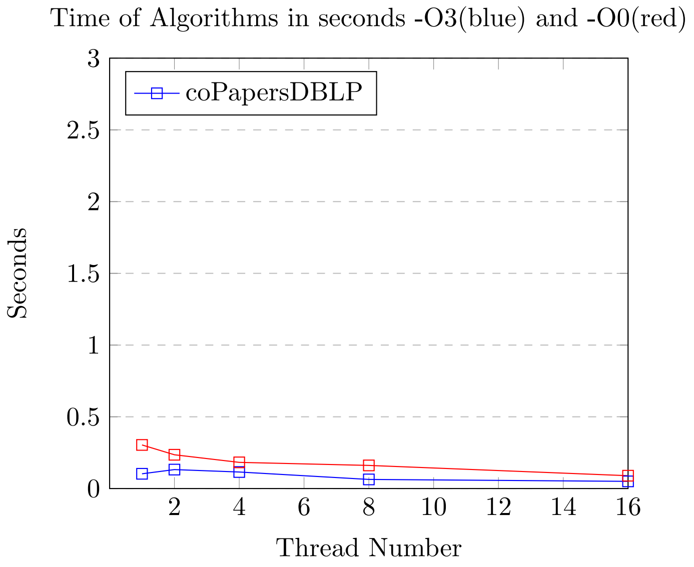
To compare the final results of the parallel algorithm, average of 5 tests for each mtx file will be shown in addition to optimization flags used while compiling the code. Experiment conducted in a way such that average of 5 trials will be used for each mtx file with different compiler options such as -O3 and -O0. Since during the trial there were a lot of process in nebula, results might not indicate the actual times of the matrices.

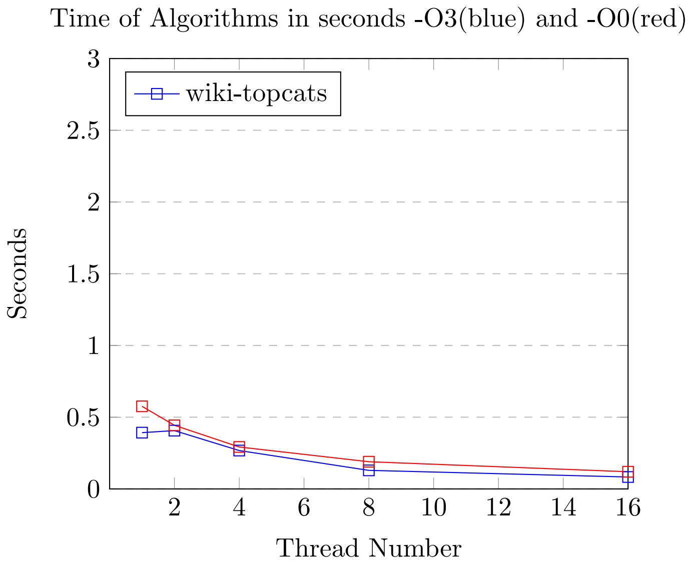
- With 03 compile option: First: export OMP_PLACES = cores
  Second: g++ seq.cpp -std=c++11 -fopenmp -ffast-math -mfpmath=sse -msse2 -O3 -o a

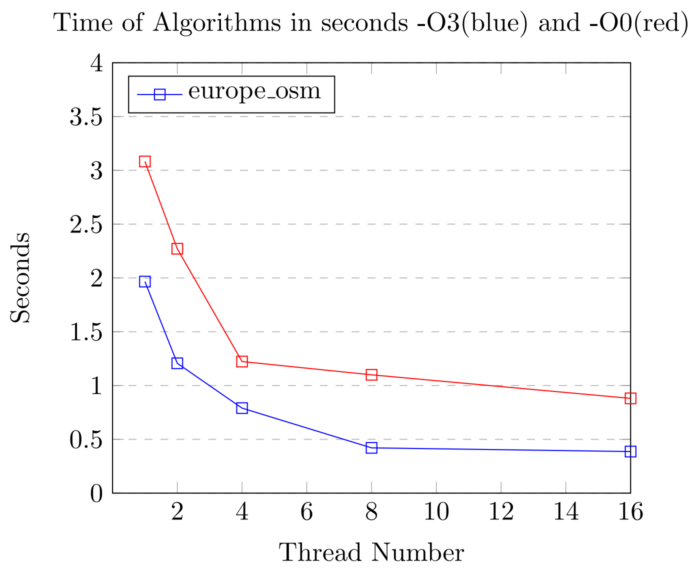
- With 03 compile option: First: export OMP_PLACES = cores
  Second: g++ seq.cpp -std=c++11 -fopenmp -ffast-math -mfpmath=sse -msse2 -O0 -o a

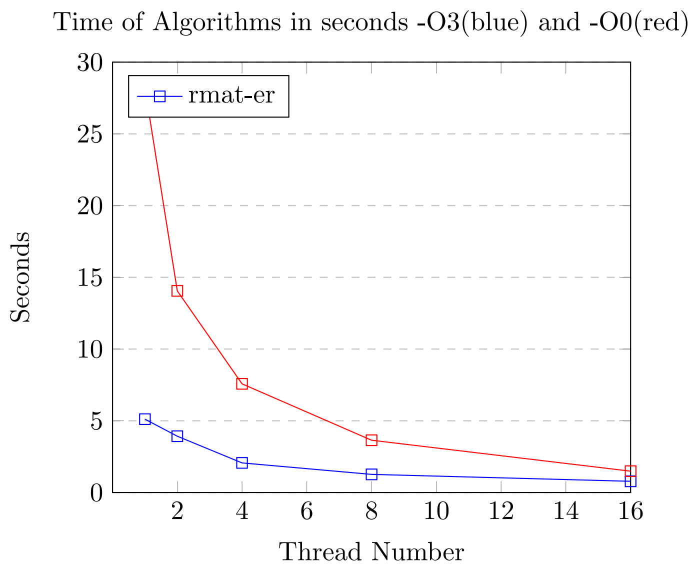
Graph	europe_osm	coPapers	wiki-topcats	rmat-er
Sequential(-O3)	0.701647	0.0586255	0.130402	2.47897
Sequential(-O0)	1.36494	0.156453	0.299206	5.7029
Using -O0	Average Time(s)			
Thread Num(1)	3.08239	0.303858	0.575947	28.2199
Thread Num(2)	2.27121	0.235336	0.443631	14.0548
Thread Num(4)	1.22284	0.182377	0.292457	7.57526
Thread Num(8)	1.09964	0.1611	0.189738	3.64595
Thread Num(16)	0.88068	0.0894226	0.119898	1.48861
Using -O3	Average Time(s)			
Thread Num(1)	1.96613	0.102731	0.392386	5.11428
Thread Num(2)	1.20668	0.132073	0.406488	3.92557
Thread Num(4)	0.790124	0.115142	0.267594	2.06431
Thread Num(8)	0.420751	0.063453	0.129428	1.26488
Thread Num(16)	0.386512	0.0500074	0.0831505	0.783446

Finally, rmat-b.mtx didn't really worked well with the code since algorithm freezes up when reading the mtx file as a result of unexpected error occurred during the reading process.









6 Conclusion

In conclusion, if matrix has a high amount of connected property, speed-up decreases and total color value will increase, hence resulting with more overhead. In addition to that, total speed gain with -O3 compiler was at 6 and 3 times for max and min speed-up respectively.

7 Side Notes

To run the code, uncomment the last part in the code for parallel function. To run the sequential one, please comment out the commented lines that consists functions and cout statement in the last part of the int main function.

8 References

1 http://www.wikizero.biz/index.php?q=aHR0cHM6Ly9lbi53a
WtpcGVkaWEub3JnL3dpa2kvU3BhdGlhbF9yZWZlcmVuY2Vfc3lzdGVt