CS406 Parallel Programming - Homework Assignment 2

Baris Sevilmis
Lab Date: 07/04/2019
Due Date: 02/05/2019

1 Problem Definition

Purpose of this homework is to implement Graph Coloring Algorithm as efficient as possible. Graph Coloring is a NP-problem, in which given Graph $G(V, E)$, find minimum color amount to color every vertex $v \in V$ where every adjacent vertex $w \in V$ & $(v, w) \in E$ is required to have a different color. OpenMP library is utilized for parallelization.

2 Data Preprocessing

2.1 Adjacency List

The basic approach of storing a graph is the Adjacency List instead of Adjacency Matrix. This approach is same as we used in our BFS code, only difference is to store every graph as undirected. In adjacency matrix, there is a row and column for each vertex. Considering the fact that the graph is unweighted; if there exists an edge from vertex i to j the corresponding entry in adjacency matrix M, $M[i][j]$ is set to 1. All the other entries are 0. On the other hand, in adjacency list, we append vertex j to the list of vertex i. With this approach, memory requirements are reduced on high levels.

While implementing the adjacency list in C++, the data structure *vector of vector of integers* is used. There is a *vector*, that contains *vector of integers*, for each vertex. If there is an edge from vertex i to j, j is added to the *vector* of i.

2.2 Compressed Row Storage

For sparse matrices, usually the number of edges are $O(n)$, where n denotes the number of vertices. Compressed Row Storage (CRS) of a graph in our case uses two arrays instead of three arrays. First array (C in Figure 1) holds all the contents of the adjacency list added one by one. Second array (R in Figure 1) corresponds to the index of C, where the vertices that has an edge from that vertex start [1]. If $R[j] = k$ and $R[j+1] = k+5$, it means that there is an edge from vertex j to all vertices stored from $C[k]$ to $C[k+4]$. In other words, R is used to distinguish corresponding vertices of C.

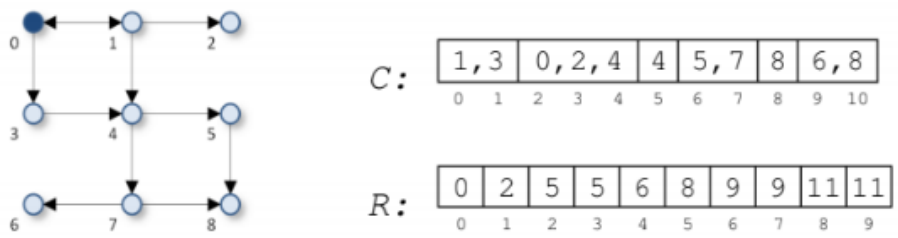

Figure 1: Compressed Row Storage [1]

3 Serial Graph Coloring Algorithm

To solve NP-Problem, Graph Coloring, there are various heuristics and algorithms. One of the best algorithms to solve Graph Coloring is the following Greedy Algorithm in Algorithm 1. In practice, this Greedy Algorithm gives decent solutions, and with the usage of *forbiddenColors* as in Algorithm 1, Serial Greedy Algorithm has the complexity of $O(|E|)$.

Algorithm 1 Serial Greedy Graph Coloring Algorithm

```
1: procedure SERIAL GRAPH COLOR(Graph G(V,E))
2:     for each v ∈ [V] do
3:         for each w ∈ Neighbors[v] do
4:             forbiddenColor[colors[w]] ← v
5:         colors[v] ← min{c >0: forbiddenColor[c] != v}
       return colors
```

4 Iterative Parallel Greedy Graph Coloring Algorithm

To parallelize, Serial Graph Coloring Algorithm, we follow a very basic approach. We will follow the same approach in Algorithm 1 in parallel manner, in which we will color vertices by using distinct *forbiddenColor* arrays for each thread. As each thread colors its own vertex group seperately, there is a high chance that there will be conflicts in some vertices as *forbiddenColor* is not shared. Therefore, in a second part of the code, we will find these conflicts and add them to R array. At the end of the second part, we will equate U to R. If U is empty, then code will terminate, otherwise it will iterate one more time but this algorithm will recolor these conflicting vertices. At the end, all conflicts will be solved, however, in this process minimum color amount that is required has a chance to increase. Following. Algorithm 2 provides iterative parallel greedy graph coloring algorithm.

Algorithm 2 Iterative Parallel Greedy Graph Coloring Algorithm

```
1: procedure ITERATIVE PARALLEL GRAPH COLORING(Graph G(V,E))
2:     U ← V
3:     while U ≠ ∅ do
4:         for each v ∈ [U] in parallel do
5:             for each w ∈ Neighbors[v] do
6:                 forbiddenColor[tid][colors[w]] ← v
7:             colors[v] ← min{c >0: forbiddenColor[tid][c] != v}
8:         R ← ∅
9:         for each v ∈ [U] in parallel do
10:            for each w ∈ Neighbors[v] do
11:                if color[v] == color[w] && v >w then
12:                    R ← R ∪ {v}
13:                    colors[v] ← 0
14:        U ← R
       return colors
```

4.1 NUMA

Using Non Uniform Memory Adress Space in our favor, results in more consistent and efficient results depending on the graphs. If vertices are in a more well connected structure, meaning that memory accesses are relatively closer to each other, than using spreading threads across cores will result in better timing results. Depending on the parallel architecture that is used, NUMA is an effective way to distribute threads across cores or sockets. As mentioned, depending on graph types in terms of connectivity and density, NUMA may have a positive impact.

In our case our parallel platform consists of 2 sockets with 32 cores on total. Each socket has 16 core on its own. As we distribute our threads at the beginning of the parallel region by using "**proc_bind(spread)**", we achieve higher success. Important thing to remember is to use command "**export OMP_PLACES=cores;**" at the terminal environment, otherwise NUMA won't be utilized. Results of NUMA on top of transpose method can be seen in Results section.

5 Results

C/C++ is used as the programming language for parallel implementation as well as the sequential implementation. Testing of algorithms were done in Nebula with the following specs:

- Model name: Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
- Architecture: x86_64
- CPU op-mode(s): 32-bit, 64-bit
- Byte Order: Little Endian
- CPU(s): 32
- Thread(s) per core: 2
- Core(s) per socket: 8
- Socket(s): 2
- NUMA node(s): 2
- CPU MHz: 1200.117, CPU max MHz: 3000.0000, CPU min MHz: 1200.0000
- L1d cache: 32K, L1i cache: 32K, L2 cache: 256K, L3 cache: 20480K

Compilation was done consecutively for -O3 and -O0 with following line of commands:
•For -O3:
export OMP_PLACES=cores;
g++ -o out3 advparallel.cpp -O3 -fopenmp -std=c++11
•For -O0:
export OMP_PLACES=cores;
g++ -o out0 advparallel.cpp -O0 -fopenmp -std=c++11

Every case was run 10 times with an 5 seperate input graphs on same machine consecutively and average of their results were taken into consideration. Following Table 1 and Figure 2 demonstrate their average run times for -O3. Figure 3 depict speed up results for -O3 cases and Figure 4 depict efficiency results for -O3.

Table 1: Timing results for -O3

Thread Num.	CoPapers	Europe	Wiki-Topcats	RMAT-ER	RMAT-B
1	0.062456s	0.793948s	0.131560s	2.340228s	2.055124s
2	0.080736s	0.705845s	0.169787s	2.359202s	2.645131s
4	0.059798s	0.394966s	0.103169s	1.389503s	1.895098s
8	0.043997s	0.276079s	0.072489s	0.913902s	1.349841s
16	0.035249s	0.203892s	0.058714s	0.730351s	1.214261s

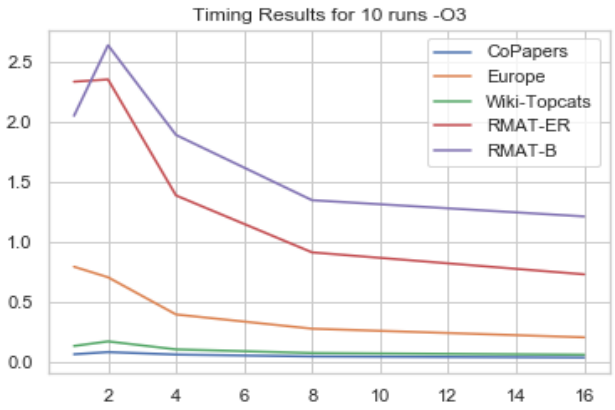

Figure 2: Average Timing Results for -O3 (10 runs)

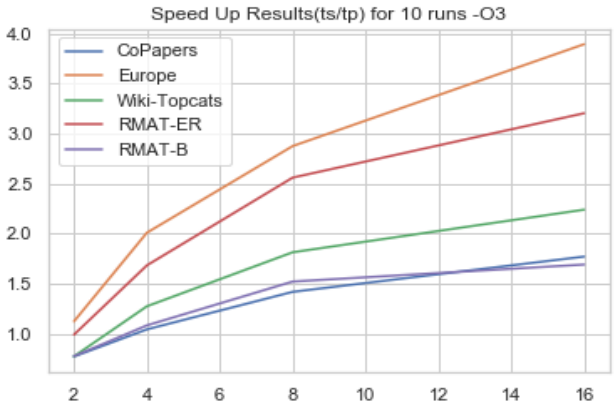

Figure 3: Speed Up Results($\frac{t_s}{t_p}$) for -O3 (10 runs)

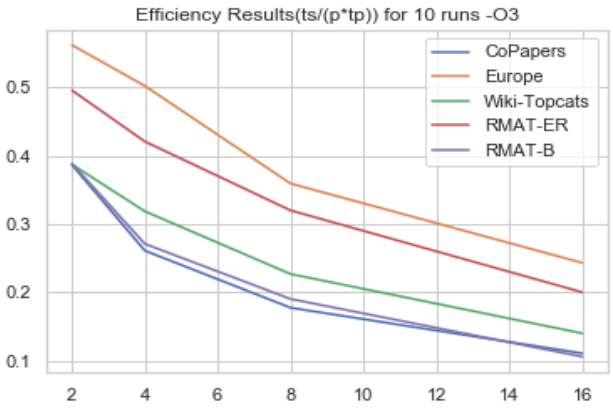

Figure 4: Efficiency Results($\frac{t_s}{p*t_p}$) for -O3 (10 runs)

As well as -O3, every case for -O0 was also run 10 times with same 5 distinct graphs on same machine consecutively and average of their results were taken into consideration. Following Table 2 and Figure 5 demonstrate their average run times for -O0. Figure 6 depict speed up results for -O0 cases and Figure 7 depict efficiency results for -O0.

Table 2: Timing results for -O0

Thread Num.	CoPapers	Europe	Wiki-Topcats	RMAT-ER	RMAT-B
1	0.178173s	1.914981s	0.512050s	6.392378s	4.090160s
2	0.190626s	1.391335s	0.434961s	5.406266s	4.597763s
4	0.141923s	0.902422s	0.260428s	3.140813s	3.661838s
8	0.110750s	0.666428s	0.186906s	2.104826s	4.090221s
16	0.091652s	0.517428s	0.141610s	1.593788s	2.342210s

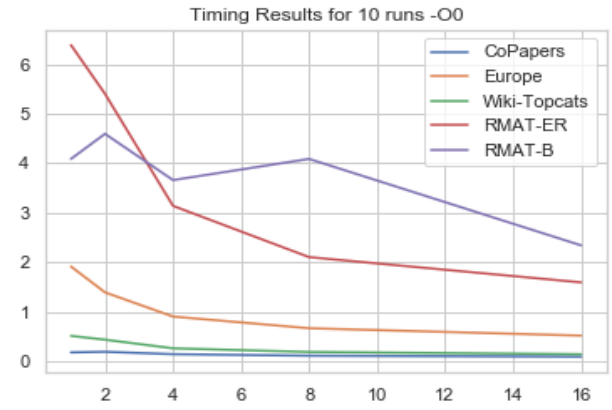

Figure 5: Average Timing Results for -O0 (10 runs)

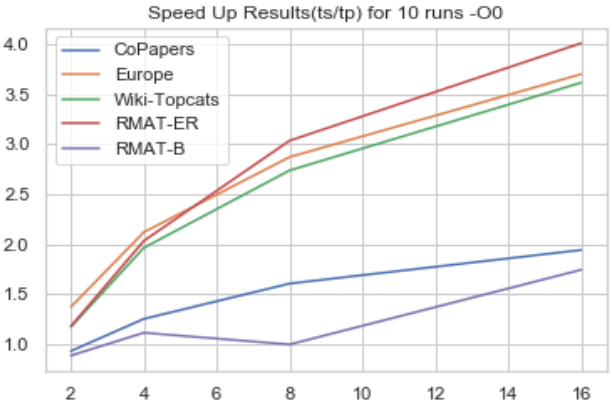

Figure 6: Speed Up Results($\frac{t_s}{t_p}$) for -O0 (10 runs)

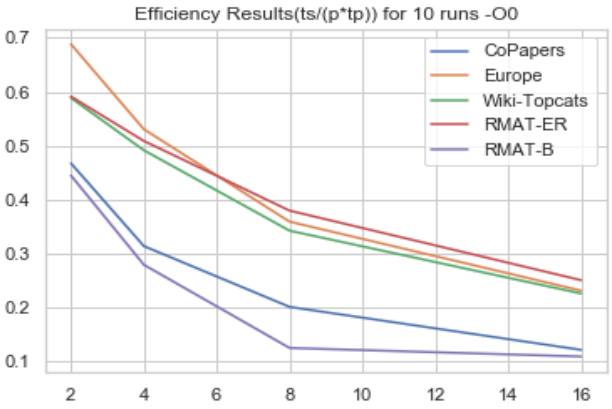

Figure 7: Efficiency Results($\frac{t_s}{p*t_p}$) for -O0 (10 runs)

6 Possible Improvements

6.1 Bitwise *U* & *forbiddenColors*

Representing *forbiddenColors* and *U(uncolored set of vertices)* as bit vectors may result in speed-up, given dense graphs with large amount of vertices. Following approach, will not result in a speed-up in given graphs, that are used for testing. However as mentioned, in graphs where conflict and recoloring size is large, bitwise representations may prove useful. Given N is vertex size, we will create an array of size *N/32*(given 32 bit unsigned integers) or *N/64*(given 64 bit unsigned integers). Every bit within this array will represent a vertex, in which *1* will mean that vertex v needs recoloring and *0* will mean vertex v has been colored correctly. At every iteration of *While* in Algorithm 2, we will go over bits which are *1*. With following approach, memory bound of Algorithm 2 will reduce greatly.
This approach in provided 5 distinct graphs, will not be very efficient. As there are very few conflicts at each iteration and therefore, *N/32* or *N/64* will require more memory than actual required memory amount. As mentioned, in dense graphs with more conflicts bitwise representation approach can reduce the memory boundness of the parallel graph coloring algorithm.

References

[1] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM.