EFFICIENT VERTEX-CENTRIC GRAPH COLORING

by

Isaac Willem van Til

HONORS THESIS

Submitted to Texas State University
in partial fulfillment
of the requirements for
graduation in the Honors College
May 2020

Thesis Supervisor:

Dr. Martin Burtscher

TABLE OF CONTENTS

Page

ABSTRACT

Graph coloring is a way of labeling (with labels traditionally referred to as "colors") the vertices of a graph with the constraint that no two adjacent vertices have the same color and that as few colors should be used as possible. In addition to many theoretical problems, graph coloring lends itself to efficiently solving a variety of practical applications (e.g., schedule generation, resource allocation, networking, and solving Sudoku). The problem with graph coloring is that finding a solution with the minimal number of colors is NP-hard, i.e., no known polynomial time algorithm can solve it optimally (so computing the solution cannot be done quickly). We have written a series of algorithms that take advantage of high amounts of parallelization to produce acceptable colorings. An analysis of the colorings and runtimes of each approach compared to other solutions from the literature shows that acceptable colorings are produced in a reasonable time.

I. INTRODUCTION

A graph is a mathematical structure that indicates relationships among pairs of objects. More formally, a graph is a pair G = (V, E) where V is a nonempty set of vertices and E is a set of edges. Coloring a graph generally involves labeling elements of the graph given specific parameters. For the purposes of this paper, we focus solely on the assignment of colors to the vertices of a graph such that (a) no two adjacent vertices share a color, and (b) the number of colors used is minimized. Although this appears useful enough on its own, graph coloring is actually a fundamental integrant in a variety of problems both theoretical and applied.

Take, for example, schedule / timetable generation. If a university wants to create a final exam schedule, they should realize that several students are in more than one class during the current semester. How can the university ensure that no two final exams for classes that share a student are scheduled at the same time? This problem can be represented by a graph with each vertex representing a class and each edge between two vertices indicating a shared student. A correct coloring of this graph produces a schedule that will not have any students miss an exam, and will show the minimum number of timeslots required to administer exams. The wide assortment of other applications where graph coloring is useful includes but is not limited to: schedule generation [12], register allocation [4], networking, data mining, map coloring, and solving Sudoku.

The issue with graph coloring is that it is relatively time consuming to do well. The trivial solution can be calculated immediately- a graph of *n* vertices can obviously be colored with *n* colors, but this is rarely valuable information. To produce a useful coloring with exactly the minimum number of colors requires an algorithm that exists

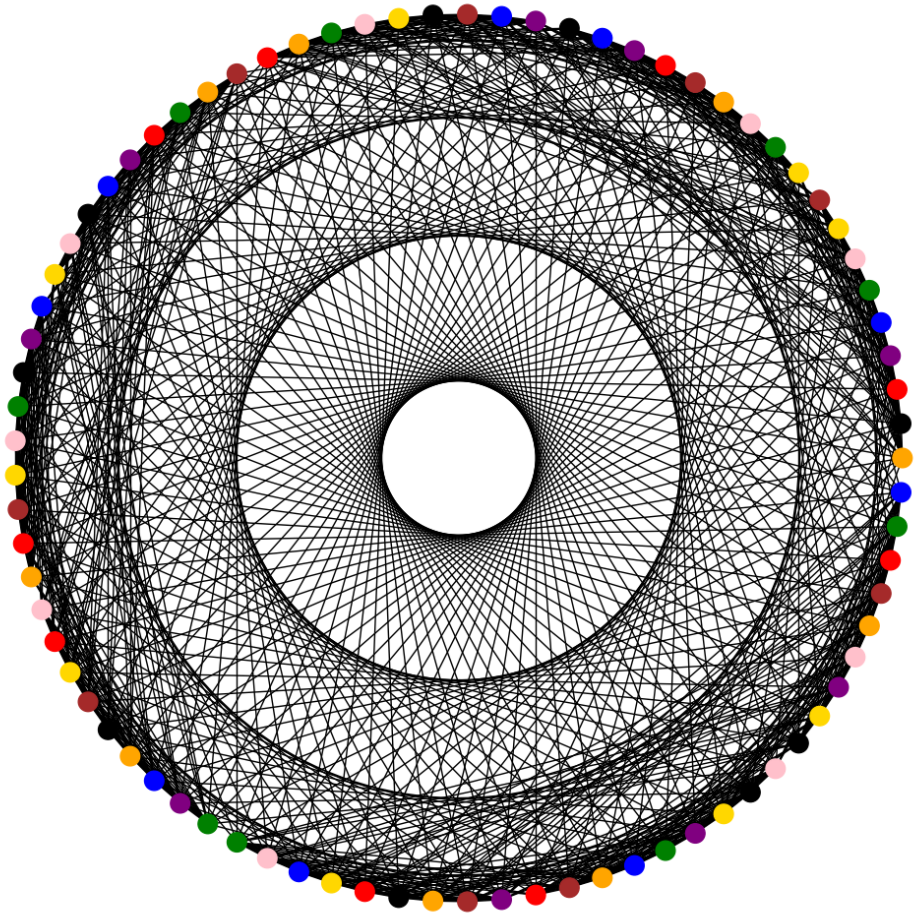

Figure 1: A colored Sudoku graph

outside the realm of polynomial-time execution. One possible way of approaching this problem is to compute the colors for different sections of a graph simultaneously, taking advantage of the parallel capabilities of multi-core and many-core processors. This, of course, spawns a whole slew of other issues- the color of a vertex is dependent on the colors of the vertices adjacent to it, so asynchronous computation requires a more thoughtful plan of attack.

Instead of guaranteeing an optimal solution, we can employ a heuristic approach

which will yield a solution that is good enough for practical purposes. A variety of sequential heuristics exist and have been studied extensively (e.g. the sequential greedy coloring algorithm [2]). In contrast, parallel approaches have not been studied quite as extensively. Some more well-known parallel approaches (Luby [11], Jones & Plassmann [10]) take advantage of independent set creation. Our approach uses large amounts of parallelism to color graphs very quickly with a number of colors reasonably close to the exact minimum. The remainder of this paper is an in depth exploration of the graph coloring problem, parallel computing, and our technique for rectifying the resulting dissonance.

II. BACKGROUND

2.1 Graph Coloring

Formally, a vertex coloring of an undirected graph G = (V, E) is a mapping C from vertices to colors such that $C(i) \neq C(j)$ for every edge $(i, j) \in E$. The smallest number of colors needed to color a graph G is its chromatic number, $\chi(G)$. As calculating $\chi(G)$ is NP-hard, there are no known polynomial-time algorithms that can produce the correct chromatic number. One common heuristic for generating a coloring is the greedy coloring algorithm, which visits each vertex and assigns the best possible color available to that vertex at the time. The quality of coloring depends on the order in which vertices are processed; in the best case scenario, $\chi(G)$ colors will be produced.

For all future examples, the following order of colors will be used:

0: Blue

1: Yellow

2: Red

3: Green

5: Purple

Whenever possible, the first color (blue) must be used- if that is not possible, the next highest color (yellow) must be used, and so on.

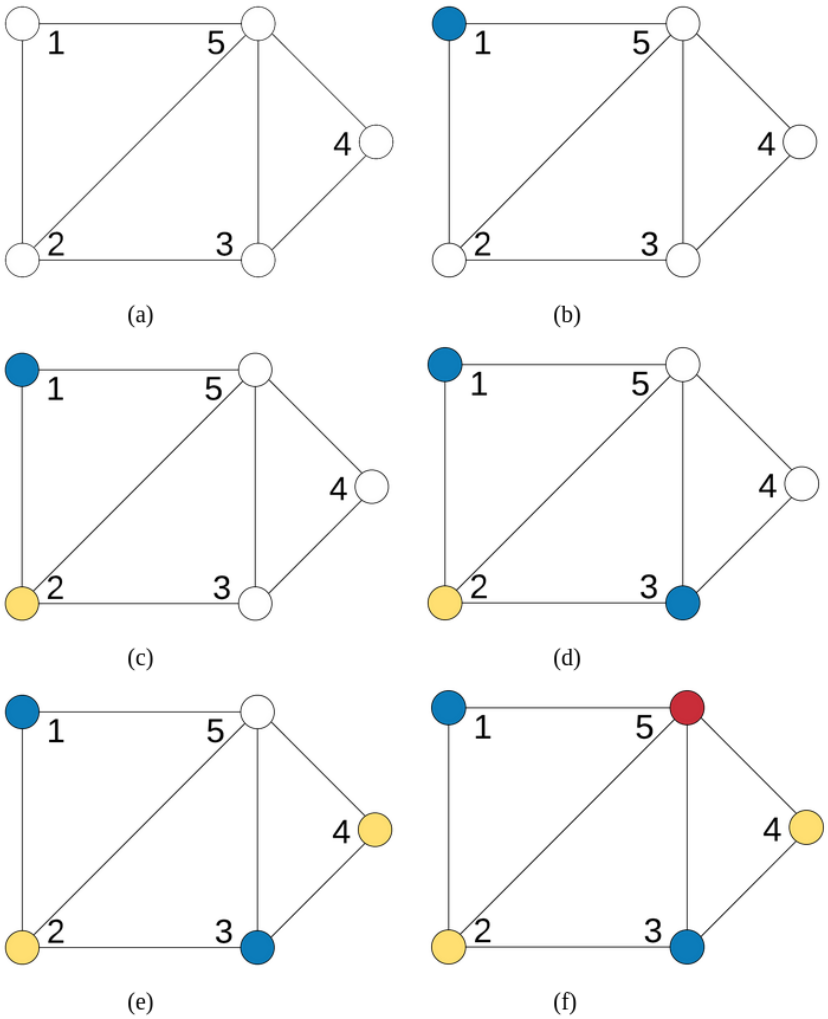

Figure 2: The sequential greedy algorithm

Figure 2 details the steps taken to perform the greedy coloring algorithm. In Figure 2a, no vertices have a color. Starting with vertex number 1, we check its neighbors and give it the best available color (blue) in Figure 2b. To get to Figure 2c we assign vertex 2 the color yellow, as yellow is the best available color not being used by any neighbors of vertex 2. In Figure 2d we perform the same check on the neighbors of vertex 3, giving vertex 3 the color blue as it is the best possible color not used by its neighbors, and so on.

2.2 Parallel Computing

With modern computer hardware (i.e. multi-core CPUs), it is possible to process multiple elements of a problem simultaneously in an attempt to arrive at a solution much faster than traditional sequential computation. Large problems can be divided into smaller chunks, each independently computed by different computational units to produce a correct solution to the overall problem. One relevant restriction to the amount of parallelism that can be achieved (and therefore the amount of potential speedup) is the dependency that an iteration of computation has on previous iterations. If a calculation relies on the result of a previous calculation, it has to wait for that previous calculation to finish before it can produce an output. This has obvious friction with the graph coloring problem, as the color of each vertex depends on the color of the vertices around it.

In the context of this paper, the idea would be to partition a graph so that the color of each vertex could be computed at the same time as the colors of the other vertices. This removes a need for waiting on previous colors to be decided, and allows large sections of a graph to be colored simultaneously and quickly.

III. ALGORITHM DESIGN

Given the problems with the complexity of traditional graph coloring algorithms and their relative lack of portability to parallel systems, we have written a series of algorithms that utilize high amounts of parallelism to calculate acceptable colorings efficiently. The general approach is to (1) color all vertices of a graph with some initial color, then (2) change the color of a vertex based on the colors of its higher priority neighbors. Higher priority is dictated by a higher degree; in the case of a tie, priority is dictated by random hash values and then the index of the vertex.

Our first approach begins with assigning every vertex in a graph with the color 0. Then we visit each vertex and check its higher priority neighbors to see if they share a color with the current vertex- if so, the color of the current vertex is incremented. This is repeated until there is no more work needed, i.e. no vertices need to change color.

The second approach is similar in execution to the first approach, however, the initial coloring is more intelligent. We start each vertex with a color that is a function of its degree, and increment from there. The functions used were log base 10, natural log, square root, and random number between 0 and the square root of the degree.

Finally, the third approach includes the ability to decrement a color. This allows the algorithm to improve the coloring of a graph if the colors of its vertices were initialized or incremented beyond an optimal color with the second approach. This approach begins with some initial coloring, exactly the same as the previous two algorithms. Then, at each vertex, the algorithm determines all of the colors used by the higher priority neighbors and gives the current vertex the lowest unused color. Again, this is repeated until no more vertices need to change a color.

The real power of these approaches is that they have the capacity to assign colors to the vertices of a graph in any order or simultaneously. At the end of each iteration, the updated colors are copied into an array of read-only colors- the algorithm only checks neighboring vertices' colors in the read-only list, ensuring consistent results across parallel executions and serial executions. Furthermore, work done in each iteration is limited only to vertices that could possibly change colors- after the first iteration, only vertices that were lower priority neighbors of vertices that changed colors in the previous iteration are considered for change in the current iteration.

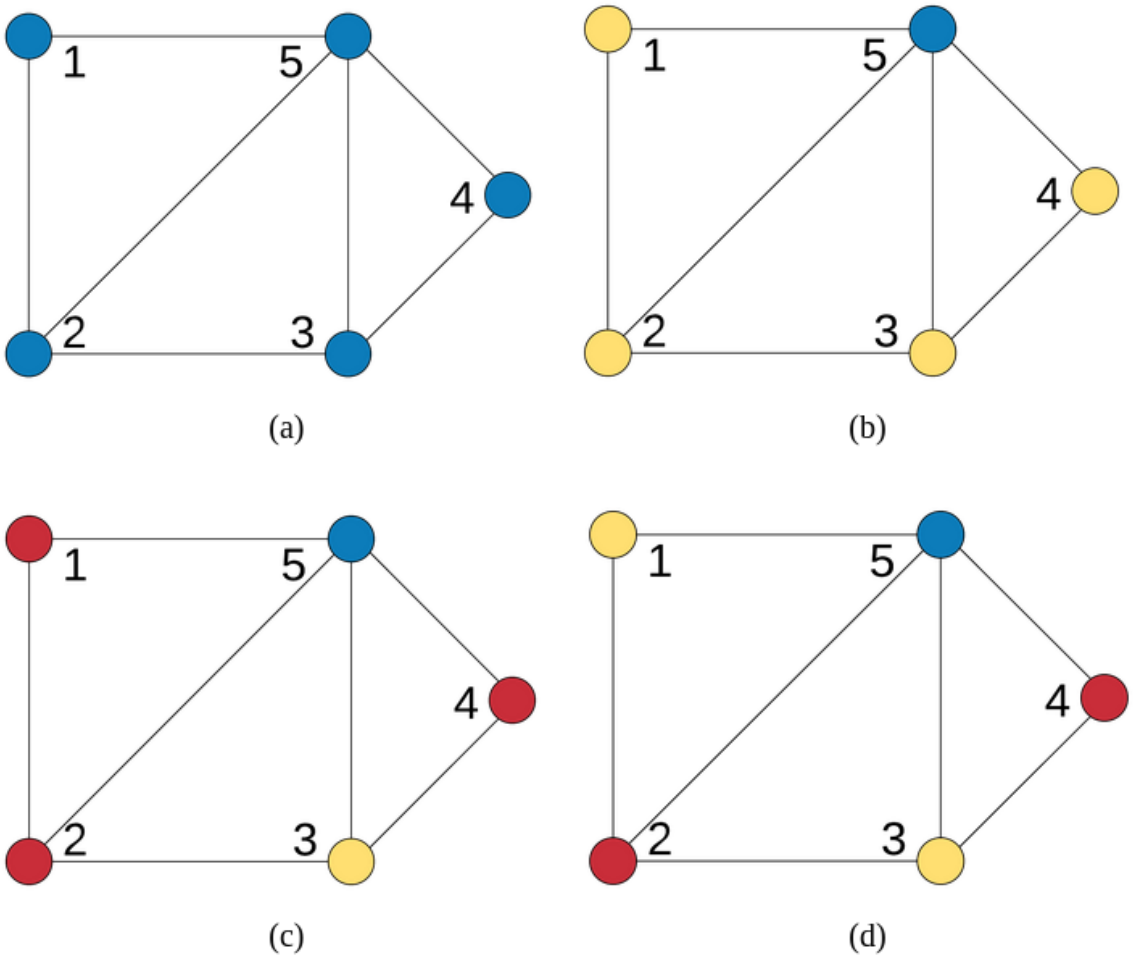

Figure 3: Steps in the third approach

Figure 3a illustrates the initialization step, here all of the vertices are initialized using color 0 (blue). At this point, the algorithm determines all of the colors used by higher priority neighbors of each vertex (in any order). Vertex 1 (in Figure 3a), for example, has higher priority neighbors 2 and 5, both colored blue. According to the coloring scheme outlined previously, vertex 1 should be updated to the color yellow, as that is the lowest color not used by its higher priority neighbors. The colors are all updated, yielding the coloring in Figure 3b. The vertices of the graph in Figure 3b each determine the colors used by their higher priority neighbors, and so on.

IV. RELATED WORK

Previous parallel graph coloring algorithms take advantage of the fact that an independent set of vertices in a graph can be colored in parallel. An independent set is a set of vertices in a graph, no two of which are adjacent. As none of the vertices in an independent set are neighbors, it is possible to give the entire independent set a single color. A variety of algorithms utilize this idea, differing slightly in how the independent sets are determined and how colors are assigned to that set. Luby [11] provides a parallel algorithm for determining the maximal independent set, and Jones and Plassmann [10] take this a step further to color graphs in parallel by finding independent sets. Alabandi et al. provide several shortcuts that increase the amount of parallelism by using shortcutting techniques to color vertices out of order [1]. A variety of other creative heuristics exist, they will generally improve coloring quality at the cost of performance or improve performance at the cost of coloring quality.

V. METHODOLOGY

Table 1 details the input graphs used for gathering results. The graphs come from a variety of sources, including the Center for Discrete Mathematics and Theoretical Computer Science at the University of Rome (Dimacs) [7], the Galois framework (Galois) [8], the Stanford Network Analysis Platform (SNAP) [13], and the SuiteSparse Matrix Collection (SMC) [14]. Table 1 includes the name, type, origin, number of vertices, number of edges, average degree, and maximum degree of each graph.

Graph name	Type	Origin	Vertices	Edges	davg	dmax
2d-2e20.sym	grid	Galois	1,048,576	4,190,208	4	4
amazon0601	co-purchases	SNAP	403,394	4,886,816	12.1	2,752
as-skitter	Internet topo.	SNAP	1,696,415	22,190,596	13.1	35,455
citationCiteseer	publication citations	SMC	268,495	2,313,294	8.6	1,318
cit-Patents	patent cites	SMC	3,774,768	33,037,894	8.8	793
coPapersDBLP	publication citations	SMC	540,486	30,491,458	56.4	3,299
delaunay_n24	triangulation	SMC	16,777,216	100,663,202	6	26
europe_osm	road map	SMC	50,912,018	108,109,320	2.1	13
in-2004	web links	SMC	1,382,908	27,182,946	19.7	21,869
internet	Internet topo.	SMC	124,651	387,240	3.1	151
kron_g500-logn21	Kronecker	SMC	2,097,152	182,081,864	86.8	213,904
r4-2e23.sym	random	Galois	8,388,608	67,108,846	8	26
rmat16.sym	RMAT	Galois	65,536	967,866	14.8	569
rmat22.sym	RMAT	Galois	4,194,304	65,660,814	15.7	3,687
soc-LiveJournal1	community	SNAP	4,847,571	85,702,474	17.7	20,333
uk-2002	web links	SMC	18,520,486	523,574,516	28.3	194,955
USA-road-d.NY	road map	Dimacs	264,346	730,100	2.8	8
USA-road-d.USA	road map	Dimacs	23,947,347	57,708,624	2.4	9

Table 1: Input graphs

The system used for our experiments has two 3.1GHz Intel Xeon E5-2687W CPUs with 10 cores each. Hyperthreading is enabled, so the 20 cores can run 40 thread simultaneously. The main memory has a 128GB capacity, and the system was running the Fedora 27 operating system with Linux kernel version 4.18.19 and GCC version 7.3.1. The code was compiled with g++ using the "-O3 -march=native" optimizations.

VI. RESULTS

This section is an overview of the results obtained using the final approach detailed in section 3 with an initial coloring of uniform zeros and input graphs from Table 1. This approach produced the exact same final coloring no matter which initial coloring was used, so uniform zero initial coloring was used for simplicity. Parallel results were verified by comparing to serial results, the colorings produced for each graph was the same in parallel as it was in serial.

6.1 Serial Results

Figure 4 compares the number of colors used by our approach, JP-D1 [6], FirstFit [5], and Boost [3]. Figure 5 indicates the throughput of each algorithm in millions of vertices processed per second. We ran each experiment three times for all eighteen graphs listed in Table 1 and recorded the best measured runtime.

The colorings of our approach are, on average, better than other serial approaches from the literature. Our runtimes, on average, are slower than other serial approaches from the literature. The slower timings are to be expected, as the serial version of our approach has no special property that would allow it to execute very fast. In fact, running our approach in serial is almost equivalent to the greedy algorithm from section 2.1.

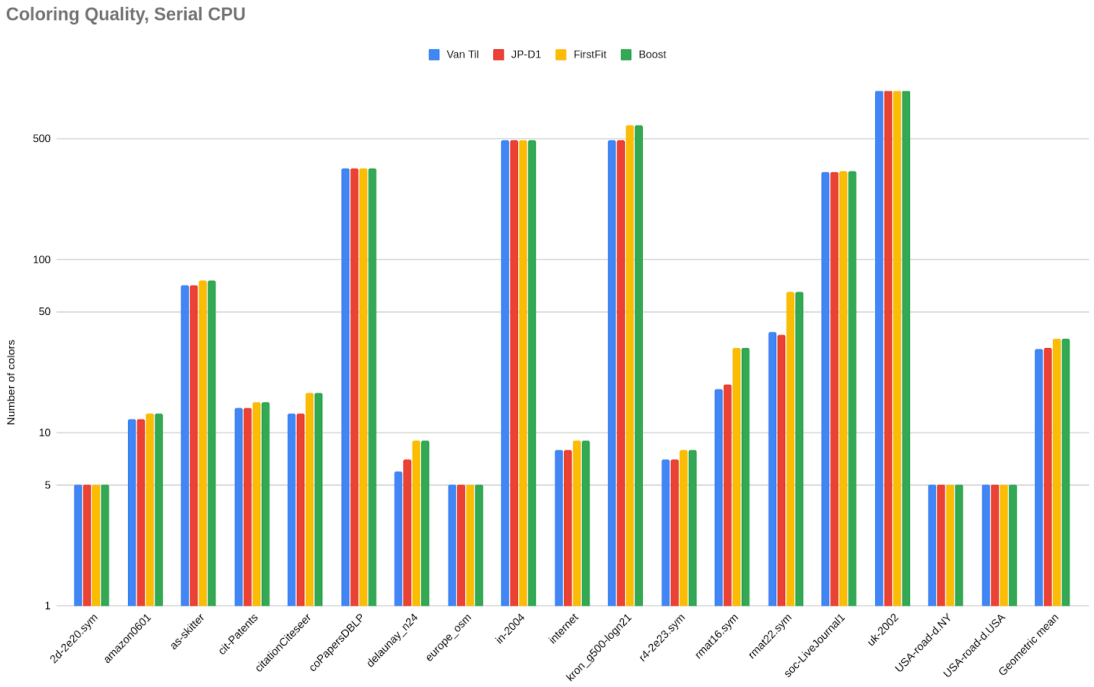

Figure 4: Number of colors used, serial CPU

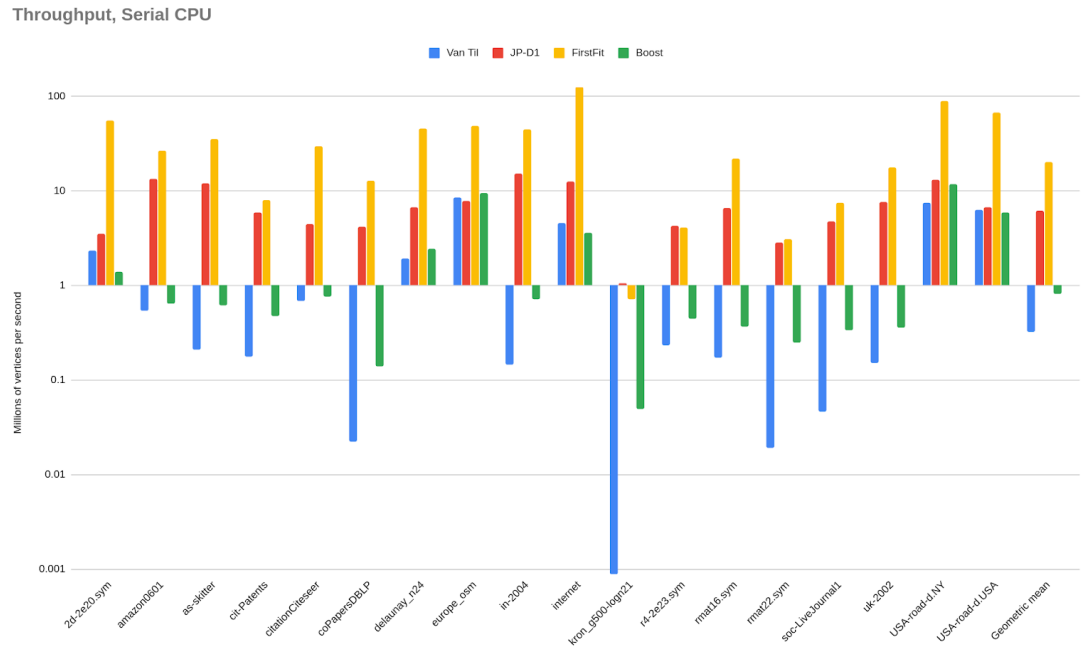

Figure 5: Throughput in millions of vertices processed per second, serial CPU

6.2 Parallel Results

Figure 6 indicates the speedup (i.e. the ratio of serial execution time to parallel execution time) of our approach on 40 threads for each graph. Figure 7 compares the number of colors used by our approach, GMMP-NT [6], FirstFit [5], and Grappolo [9]. Figure 8 indicates the throughput of each algorithm in millions of vertices processed per second. We ran each experiment three times for all eighteen graphs listed in Table 1 and recorded the best measured runtime.

The colorings of our approach are, on average, better than other parallel approaches from the literature. Our runtimes, on average, are slower than other parallel approaches from the literature.

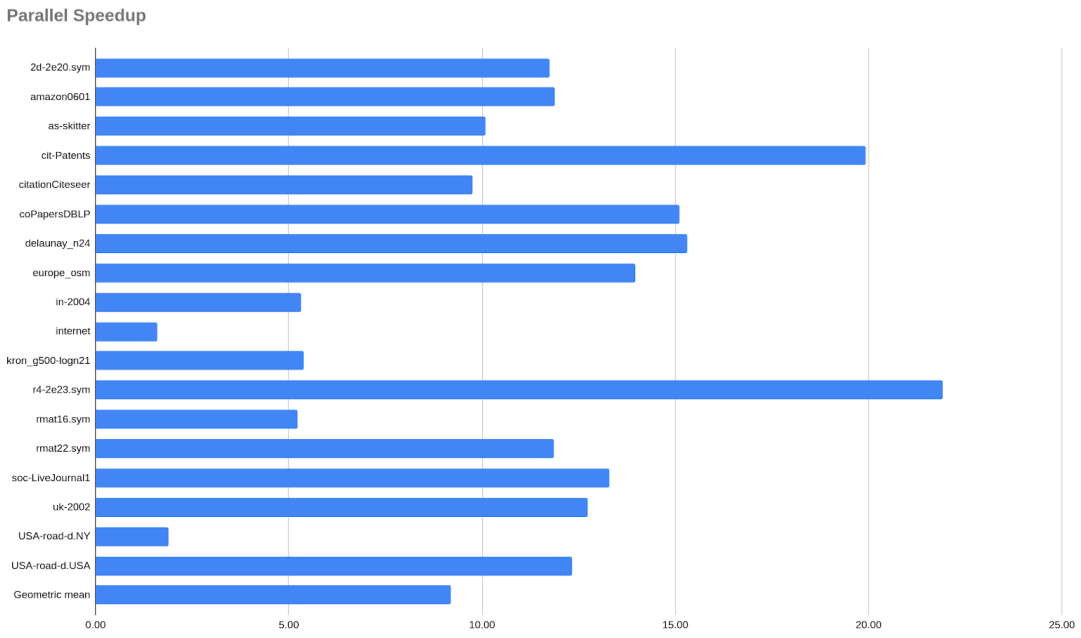

Figure 6: Speedup on 40 threads

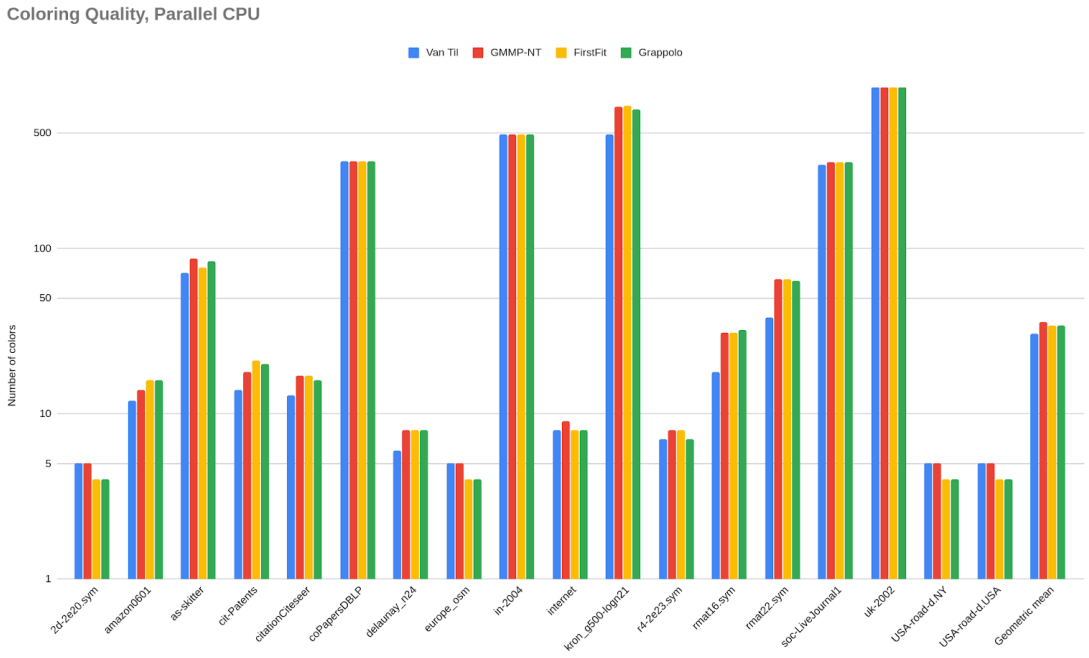

Figure 7: Number of colors used, parallel CPU

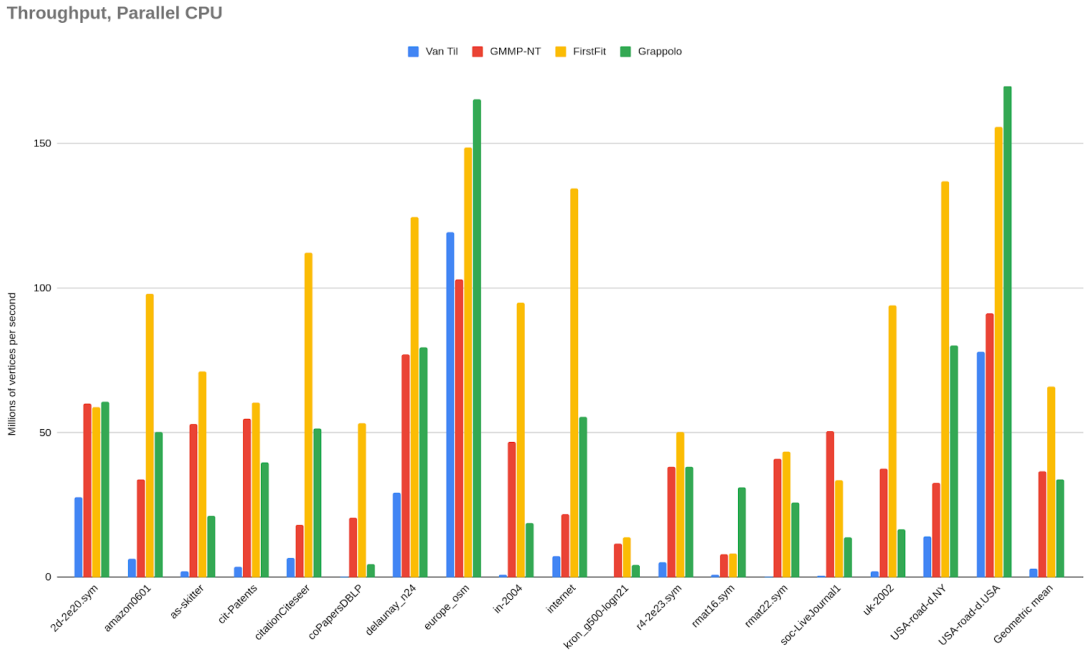

Figure 8: Throughput in millions of vertices processed per second, parallel CPU

VII. CONCLUSION & FUTURE WORK

This paper presents a heuristic for graph coloring that takes advantage of parallelism, producing very good colorings on average. The approach used removes the dependence that is present in many serial graph coloring heuristics, allowing for vertices in a graph to have their color computed simultaneously or in any order. Of the three approaches specified in this paper, the third approach yielded the best coloring quality. In general, our approach is relatively slow compared to other algorithms from the literature. Our approach does produce high quality colorings, on-par with or surpassing other algorithms from the literature.

Future work for this project could include porting the code to a Graphics Processing Unit (GPU), which utilizes very high amounts of parallelism (much higher than 40 concurrent threads). This could potentially speed up the timing in our approach when taking full advantage of the vast parallelism available.

REFERENCES

[1] Alabandi, G., E. Powers, and M. Burtscher. "Increasing the Parallelism of Graph Coloring via Shortcutting." Proceedings of the *2020 ACM Conference on Principles and Practice of Parallel Programming*. February 2020.

[2] Allwright, J. R., R. Bordawekar, P. D. Coddington, K. Dincer, and C. L. Martin. A comparison of parallel graph coloring algorithms. Technical report, SCCS-666, Northeast Parallel Architectures Center at Syracuse University, 1995.

[3] Boost, https://www.boost.org/doc/libs/1_63_0/libs/graph_parallel/doc/html/index.html, last accessed on 12/28/2019.

[4] Chaitin, G. J. (1982), "Register allocation & spilling via graph colouring", *Proc. 1982 SIGPLAN Symposium on Compiler Construction* (1982): 98–105.

[5] Chen, Xuhao, Pingfan Li, Jianbin Fang, Tao Tang, Zhiying Wang, and CanqunYang. “Efficient and high-quality sparse graph coloring on GPUs.” *Concurrency and Computation: Practice and Experience* 29, no. 10 (2017): e4064.

[6] ColPack, Combinatorial Scientific Computing and Petascale Simulations, https://github.com/CSCsw/ColPack, last accessed on 12/28/2019.

[7] DIMACS, Center for Discrete Mathematics and Theoretical Computer Science, http://www.dis.uniroma1.it/challenge9/download.shtml, last accessed on 12/28/2019.

[8] Galois, ISS -The University of Texas at Austin, https://iss.oden.utexas.edu/?p=projects/galois, last accessed on 12/28/2019.

[9] Grappolo, the Grappolo graph toolkit, https://github.com/luhowardmark/GrappoloTK, last accessed on 12/28/2019.

[10] Jones, Mark T., and Paul E. Plassmann. “A parallel graph coloring heuristic.” *SIAM Journal on Scientific Computing* 14, no. 3 (1993): 654-669.

[11] Luby, Michael. “A simple parallel algorithm for the maximal independent set problem.” *SIAM journal on computing* 15, no. 4 (1986): 1036-1053.

[12] Marx, Dániel, "Graph colouring problems and their applications in scheduling", *Periodica Polytechnica, Electrical Engineering* 48 (2004): 11–16.

[13] SNAP, Stanford Large Network Dataset Collection, https://snap.stanford.edu/data/, last accessed on 12/28/2019.

[14] SuiteSparse Matrix Collection, https://sparse.tamu.edu/, last accessed on 12/28/2019.