Evaluation of Serial and Parallel Shared-Memory Distance-1 Graph Coloring Algorithms

Lukas Gnam[1(✉)], Siegfried Selberherr[2], and Josef Weinbub[1]

[1] Christian Doppler Laboratory for High Performance TCAD, Institute for Microelectronics, TU Wien, Vienna, Austria
{gnam,weinbub}@iue.tuwien.ac.at
[2] Institute for Microelectronics, TU Wien, Vienna, Austria
selberherr@iue.tuwien.ac.at

Abstract. Within the scope of computational science and engineering, the standard graph coloring problem, the distance-1 coloring, is typically used to select independent sets on which subsequent parallel computations can be guaranteed. As graph coloring is an active field of research, various algorithms are available, each offering advantages and disadvantages. We compare several serial as well as parallel shared-memory graph coloring algorithms for the standard graph coloring problem based on reference graphs. Our investigation covers well established as well as recent algorithms and their support for balanced and unbalanced approaches. An overview on speedup, used number of colors, and their respective population for different test graphs is provided. It is shown that the parallel approaches produce similar results as the serial methods, but for specific cases the serial algorithms still remain a good option, when certain properties (e.g., balancing) are of major importance.

Keywords: Graph coloring · Shared-memory · Distance-1 coloring
Parallel algorithm

1 Introduction

The decomposition of computational tasks into independent sets, which pave the way for a subsequent parallelization step, is a widely used approach to exploit parallel computing resources. Examples of such use cases are community detection [9], mesh adaptation [7], and linear algebra [10] algorithms.

In this work, we consider the standard graph coloring problem, the distance-1 coloring. For a general graph $G(V, E)$ a distance-1 coloring is a coloring, where any two adjacent vertices receive different colors. Hence, each color represents an independent set for possible subsequent parallel processing. Usually, the goal of a distance-1 graph coloring problem is to use as few colors as possible: One fact which is often neglected is the population of the resulting sets. Considering the standard formulations of such algorithms, they mostly result in highly

G. Nikolov et al. (Eds.): NMA 2018, LNCS 11189, pp. 106–114, 2019.
https://doi.org/10.1007/978-3-030-10692-8_12

unbalanced populations. A heavily unbalanced coloring can lead to *colors* which contain insufficient workload to achieve acceptable parallel efficiency, thus leading to an undesired bottleneck in a parallel workflow. Therefore, graph coloring algorithms typically aim to use as few colors as possible to enable the subsequent workflow to achieve proper scalability.

Although there have been several efforts to compare different graph coloring algorithms in the past [1,6,9], we pick up on recent developments in this field and provide an overview on some of the newest distance-1 graph coloring algorithms. We show their difference in the number of colors used for different graphs, as well as a comparison of the resulting color populations. Additionally, we investigate the overhead in execution time experienced for different balancing approaches. For the parallel algorithms we also provide speedup and strong scalability data.

In Sect. 2, we briefly discuss the related work and present the coloring algorithms we used in our evaluation, followed by the actual evaluation of the algorithms in Sect. 3.

2 Coloring Algorithms

The most widely used approach to achieve a coloring of a graph is the *Greedy* coloring algorithm [5]. It iterates the graph and assigns the smallest color permissible to the active graph vertex, by checking the color assigned to its neighbors (usually colors are denoted using integers). Thus, for any graph with maximum degree d[1] this algorithm uses at most $d + 1$ colors. One major drawback of this algorithm is, that the highest colors are the ones assigned the least, leading to a skewness in the population of the colors. The *Greedy* algorithm is part of our study.

To alleviate this skewness, the *Greedy* algorithm can be adapted such that it assigns the least used color permissible, leading to a more balanced coloring of the graph [9,11]. This algorithm is also part of our study and henceforth denoted as *Greedy-LU*.

Based on an approach from Gebremedhin and Manne [4], a parallel algorithm for speculative graph coloring was introduced by Çatalyürek et al. [1], which follows a two-step strategy. The first step is to color the graph vertices in parallel without checking for any possible conflicts. In a second step the previously colored graph vertices are checked and, if conflicts occur, the corresponding graph vertices are marked for recoloring in the next iteration. Hence, the number of occurring conflicts defines the number of total iterations, which could lead to performance drawbacks. Catalyürek's algorithm, from now on referred to as *Parallel*, acts as our baseline for the shared-memory parallel coloring approach.

Recently, Lu et al. presented several serial shared-memory parallel algorithms for distance-1 graph coloring [9]. Following their results, we selected the *Scheduled Reverse* algorithm to be included in our study. This particular approach uses the *Greedy* algorithm to obtain an initial coloring and improves the balancing by moving vertices from colors with high population to colors with low

[1] The degree of a vertex of a graph is the number of incident edges [3].

population, without introducing new colors. We chose to limit the algorithm to three iterations, following the results and suggestions presented in [9], which is a reasonable compromise between color balancing and computational overhead. Within the remainder of this work, we will refer to this algorithm as *Parallel Recolor*.

3 Evaluation

3.1 Benchmark Platform

We used a single compute node of the Vienna Scientific Cluster 3 (VSC-3). A node offers two Intel Xeon E5-2650v2 Ivy Bridge EP processors running with 2.6 GHz and a total of 64 GB of main memory. Hence, 16 physical and 32 logical cores are available. The benchmarks were compiled using Intel's C++ compiler, version 17.0.4, with `-O3` optimization. Additionally, we made use of Intel's thread-core affinity capabilities using the *KMP_AFFINITY* environment variable to ensure proper thread pinning to avoid thread migration. We used a `static` OpenMP loop scheduling, as the problems do not pose a load-balancing issue.

3.2 Test Graphs

For our investigations we used four different graphs, where three of them were created with the parallel graph generation software PaRMAT [8] following the approach from Catalyürek [1]. We varied the graph parameters a, b, and c (see Table 1) while keeping a constant total number of vertices of 16 777 216. The first graph, RMAT-ER, is a graph belonging to the so-called Erdös-Rényi class with a normal degree distribution, whereas the other two, RMAT-G and RMAT-B, have multiple local maxima of the degree distribution (for more details see [1]). The highest number of neighbors for a vertex (maximum vertex degree) is observed in the RMAT-B graph with 49 212. The fourth test graph is taken from the University of Florida Sparse Matrix Collection [2] and is a mesh modeling a BMW 3 series car. An overview on the graph properties is shown in Table 1.

Table 1. Properties of the four graphs used in this evaluation study as well as the graph parameters for the RMAT graphs used within PaRMAT.

Graph	Vertices	Avg. degree	Max. degree	a	b	c
RMAT-ER	16 777 216	16	42	0.25	0.25	0.25
RMAT-G	16 777 216	16	41 938	0.45	0.15	0.25
RMAT-B	16 777 216	16	49 212	0.55	0.15	0.15
BMW	227 362	48.65	335	-	-	-

3.3 Coloring Quality

In Fig. 1 we show the maximum number of colors used by each of the individual algorithms for the four input graphs. For the RMAT-ER graph the *Greedy-LU* algorithm uses the most colors, i.e., 23, compared to 12 for all others, including the *Parallel* algorithm with 32 threads. Due to its recoloring approach the *Parallel Recolor* algorithm uses always the same number of colors as the *Greedy* algorithm, because the latter acts as the initial input coloring and no colors are added while balancing. This is independent of the number of threads being used. Additionally, it can be observed that the actual number of colors is always way below the maximum vertex degree, and often also in the range of the nearly optimal *Greedy* coloring algorithm. In the case of the RMAT-G graph, *Greedy-LU* produces an output using 70 colors, whereas the other algorithms use only 26. The resulting colorings require about 1600 times less colors than the maximum vertex degree occurring in this graph, except of the *Greedy-LU* algorithm which uses nearly 600 times less colors. The difference results from the *Greedy* coloring approach applied in the two parallel algorithms, which is unbalanced in contrast to the *Greedy-LU* algorithm. Considering the RMAT-B graph, the *Greedy-LU* algorithm requires the most colors, i.e., 395, which is about four times more than the demand from the *Parallel* algorithm using 32 threads. The results for the three-dimensional BMW mesh show that *Parallel* with 32 threads uses the least number of colors. This is a result of the parallel execution of the *Greedy* coloring approach in its tentative coloring phase, yielding a better outcome in the number of used colors and their population for this specific graph.

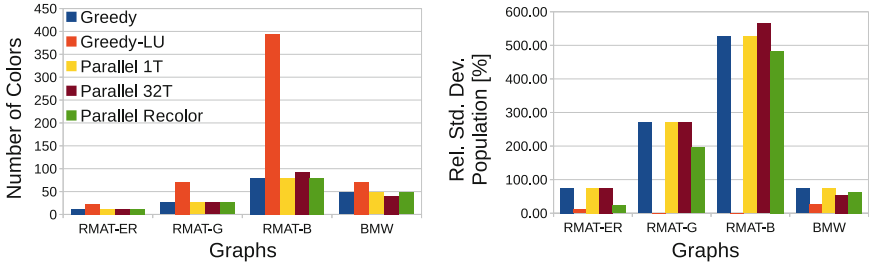

Fig. 1. Maximum number of colors (left) and relative standard deviation of the color population (right) resulting from the investigated algorithms for each of the input graphs. For the *Parallel* algorithm the results obtained with 1 (1T) and 32 threads (32T) are depicted. Note that the single-threaded version produces the same results as the *Greedy* algorithm. Since the maximum number of colors used by the *Parallel Recolor* algorithm does not depend on the number of threads only one result bar is shown in each figure. (Color figure online)

To compare the population of the colors produced by the implemented algorithms we use the relative standard deviation of the coloring results (see Fig. 1). The different distributions of vertex degrees in each graph (see Table 1) strongly

influence the resulting color populations: Vertices with a high degree compared to the graph's average degree lead to a higher number of colors with very low population. This is especially observable for the RMAT graphs, where for RMAT-ER the deviations range between 11–75%, for RMAT-G between 0–270%, and for RMAT-B between 0–565%. For the BMW graph the deviations are between 26–74%. As can be seen, the best results (i.e., the least deviation) are obtained using the *Greedy-LU* algorithm, since it initially tries to balance the colors. In case of the three RMAT graphs the *Parallel Recolor* algorithm returns the second best deviation results, followed by the *Greedy* and the *Parallel* algorithm. The *Parallel* algorithm using 32 threads produces similar results as *Greedy*, except for the RMAT-B graph, where its deviation is larger than the deviation resulting from the *Greedy* algorithm. For the BMW graph the *Parallel* algorithm using 32 threads produces the best deviation results after the *Greedy-LU* algorithm, followed by the *Parallel Recolor* algorithm. Because it assigns the smallest color permissible, the *Greedy* algorithm produces the highest skewness. Regarding the color population for the BMW graph, *Parallel* also performs better than *Parallel Recolor*. Nevertheless, the relative standard deviation must not be viewed as a single quality metric for the population deviation of the used colors, because there can still be very large differences between specific colors. Figure 2 shows these differences in the color populations occurring for the different graphs. Since the *Greedy* algorithm uses the smallest color permissible for coloring a vertex, the color population decreases for increasing color indices when applying the *Greedy* as well as the *Parallel* algorithm. Therefore, we observe a high skewness in the results produced by the *Greedy* and the *Parallel* algorithm. Since the *Parallel Recolor* algorithm does not add new colors, strong *jumps* of color populations can be observed, especially for high color indices, but it alleviates the high skewness produced initially with the *Greedy* algorithm. This effect is shown in Fig. 2, where *Parallel Recolor* produces well balanced colorings for the RMAT-ER and RMAT-G graphs, whereas for the other two cases it results in significant population differences for higher colors (e.g., 264 times higher for color 54 than for color 53 in the RMAT-B graph). Regarding the results obtained with the *Parallel* algorithm, our investigations show that it produces similar results as the *Greedy* algorithm, since the *Parallel* algorithm uses a *Greedy* approach in its parallel coloring step. As expected, the *Greedy-LU* algorithm produces the best balanced colorings for all test graphs but at the price of using 1.5 to almost 5 times more colors than the unbalanced *Greedy* algorithm.

3.4 Strong Scaling Analysis

Since not only the resulting number of colors and their respective population can have a strong impact on the overall application performance, but also the execution time of the coloring algorithm itself, we additionally investigated the strong scaling capabilities of the *Parallel* and the *Parallel Recolor* algorithm based on the reference graphs. For the latter there is, in addition to the initial use of the *Greedy* coloring algorithm, also some serial part in the algorithm which prepares the data for parallel execution. Therefore, it can be expected

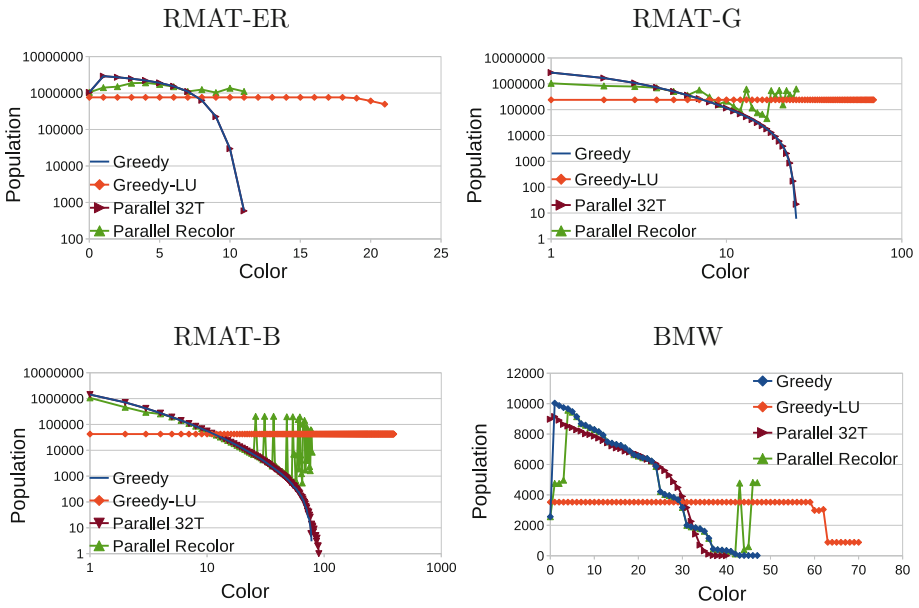

Fig. 2. Resulting color populations for the four test graphs evaluated in our study. For the *Parallel* algorithm we show the results obtained using 32 threads (32T), since the single threaded version produces the same output as the *Greedy* algorithm. The coloring results for the *Parallel Recolor* algorithm are independent from the number of threads. (Color figure online)

that the recoloring approach of the *Parallel Recolor* algorithm is most likely to experience parallel performance limitations. Figure 3 and Table 2 show this expected behavior. All timings are averaged based on three iterations.

The execution times for the single-threaded versions show major differences for the three RMAT test graphs. The *Parallel Recolor* algorithm is nearly two times faster than the *Parallel* algorithm for the RMAT-ER graph, and 1.3 times faster for the other two RMAT graphs, because the *Parallel* algorithm iterates at least twice over the graph (coloring and checking), whereas *Parallel Recolor* retains the nearly optimal initial *Greedy* coloring. As shown in Table 2, the single-threaded execution times for the BMW graph are similar, due to the smaller number of vertices: 0.132 s for the *Parallel* algorithm and 0.137 s for the *Parallel Recolor* algorithm. As expected, when increasing the number of threads *Parallel* outperforms *Parallel Recolor* for more than 2 threads for the RMAT-ER and the RMAT-G graph and for more than 4 threads for the RMAT-B graph. However, for the BMW graph the *Parallel Recolor* algorithm scales best for up to 16 threads, albeit breaking down for 32 threads, because of the increased number of recoloring conflicts occurring with a higher number of threads for this graph.

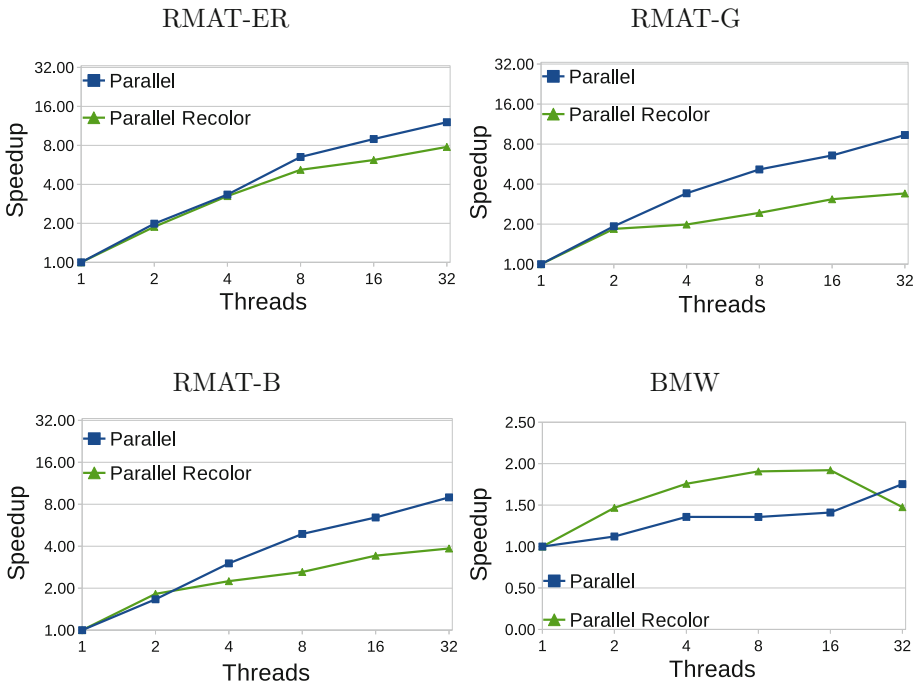

Fig. 3. Speedups of the parallel algorithms for the different test graphs.

Table 2. Execution times in seconds of the algorithms for the test graphs, with the fastest time for each graph in bold. For the *Parallel* and *Parallel Recolor* (*Par.Rec.*) algorithms the results obtained with 1 (1T) and 32 threads (32T) are shown.

Graph	*Greedy*	*Greedy-LU*	*Parallel 1T*	*Parallel 32T*	*Par.Rec.1T*	*Par.Rec.32T*
RMAT-ER	7.213	12.32	26.71	**2.21**	13.97	10.03
RMAT-G	5.765	140.21	19.55	**2.10**	15.11	9.53
RMAT-B	6.372	29.02	19.50	**2.17**	15.43	10.21
BMW	**0.036**	0.378	0.132	0.075	0.137	0.242

4 Conclusions

As shown in this work, the resulting colorings depend heavily on the type and properties of the respective input graph (e.g., see Figs. 1 and 2). In order to make an adequate choice it is therefore necessary to determine the requirements of the specific task for which the coloring should be used. If a balanced color distribution is the primary metric of interest, then the *Greedy-LU* algorithm is the best choice. However, if the application in mind has to repeatedly execute the coloring algorithm, it is likely that the execution time spent in coloring becomes more and more dominant: As the timings in Table 2 indicate, there

are cases where the *Greedy-LU* algorithm takes about 24 times longer than the unbalanced serial *Greedy* algorithm. This results from the fact, that the *Greedy-LU* algorithm has to maintain a list of how often each color has already been assigned, such that the least used color permissible is picked.

This introduces the execution performance - and by extension parallel scalability - as a potential core metric. As shown in Figs. 2 and 3, as well as in Table 2, in most cases the use of a parallel coloring strategy, like the *Parallel* algorithm using 32 threads, can save up to 70% of time spent in coloring compared to the *Greedy* algorithm while producing similar results. In our test cases the speed of the *Parallel Recolor* algorithm suffers from its serial preparation step, yielding inferior execution performance than the *Parallel* algorithm. Nevertheless, if balancing of the colors *and* execution time are of importance, the recoloring approach remains still a reasonable choice, because in some cases it is more than 14 times faster than the *Greedy-LU* algorithm and guarantees that the number of colors used is the same as with the *Greedy* algorithm.

In all cases, the *Greedy-LU* algorithm proves to give the best balanced colorings, while in most cases the *Parallel* algorithm guarantees a proper tradeoff between execution time and the number of colors. For the BMW graph, the serial *Greedy* algorithm provides a well-balanced tradeoff between all three performance parameters.

Acknowledgements. The financial support by the *Austrian Federal Ministry for Digital and Economic Affairs* and the *National Foundation for Research, Technology and Development* is gratefully acknowledged. The computational results presented have been achieved using the Vienna Scientific Cluster (VSC).

References

1. Çatalyürek, Ü.V., Feo, J., Gebremedhin, A.H., Halappanavar, M., Pothen, A.: Graph coloring algorithms for multi-core and massively multithreaded architectures. Parallel Comput. **38**(10), 576–594 (2012)
2. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011)
3. Diestel, R.: Graph Theory, 5th edn. Springer, Heidelberg (2017)
4. Gebremedhin, A.H., Manne, F.: Scalable parallel graph coloring algorithms. Concurr. Pract. Exp. **12**(12), 1131–1146 (2000)
5. Gyárfás, A., Lehel, J.: On-line and first fit colorings of graphs. J. Graph Theory **12**(2), 217–227 (1988)
6. Hawick, K., Leist, A., Playne, D.: Parallel graph component labelling with GPUs and CUDA. Parallel Comput. **36**(12), 655–678 (2010)
7. Ibanez, D., Shephard, M.: Mesh adaptation for moving objects on shared memory hardware. In: Proceedings of the International Meshing Roundtable (2016)
8. Khorasani, F., Gupta, R., Bhuyan, L.N.: Scalable SIMD-efficient graph processing on GPUs. In: Proceedings of the International Conference on Parallel Computing Technologies, pp. 39–50 (2015)
9. Lu, H., Halappanavar, M., Chavarría-Miranda, D., Gebremedhin, A.H., Panyala, A., Kalyanaraman, A.: Algorithms for balanced graph colorings with applications in parallel computing. IEEE Trans. Parallel Distrib. Syst. **28**(5), 1240–1256 (2017)

10. Manne, F.: A parallel algorithm for computing the extremal eigenvalues of very large sparse matrices. In: Kågström, B., Dongarra, J., Elmroth, E., Waśniewski, J. (eds.) PARA 1998. LNCS, vol. 1541, pp. 332–336. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0095354
11. Manne, F., Boman, E.: Balanced Greedy colorings of sparse random graphs. In: Proceedings of the Norwegian Informatics Conference, pp. 113–124 (2005)