A Fast and Scalable Graph Coloring Algorithm for Multi-core and Many-core Architectures

Georgios Rokos[1], Gerard Gorman[2], and Paul H J Kelly[1]

[1] Software Peroformance Optimisation Group,
Department of Computing
[2] Applied Modelling and Computation Group,
Department of Earth Science and Engineering
Imperial College London,
South Kensington Campus, London SW7 2AZ, United Kingdom,
{georgios.rokos09,g.gorman,p.kelly}@imperial.ac.uk

Abstract. Irregular computations on unstructured data are an important class of problems for parallel programming. Graph coloring is often an important preprocessing step, e.g. as a way to perform dependency analysis for safe parallel execution. The total run time of a coloring algorithm adds to the overall parallel overhead of the application whereas the number of colors used determines the amount of exposed parallelism. A fast and scalable coloring algorithm using as few colors as possible is vital for the overall parallel performance and scalability of many irregular applications that depend upon runtime dependency analysis.
Çatalyürek et al. have proposed a graph coloring algorithm which relies on speculative, local assignment of colors. In this paper we present an improved version which runs even more optimistically with less thread synchronization and reduced number of conflicts compared to Çatalyürek et al.'s algorithm. We show that the new technique scales better on multi-core and many-core systems and performs up to 1.5x faster than its predecessor on graphs with high-degree vertices, while keeping the number of colors at the same near-optimal levels.

Keywords: Graph Coloring; Greedy Coloring; First-Fit Coloring; Irregular Data; Parallel Graph Algorithms; Shared-Memory Parallelism; Optimistic Execution; Many-core Architectures; Intel®Xeon Phi™

1 Introduction

Many modern applications are built around algorithms which operate on irregular data structures, usually in form of graphs. Graph coloring is an important preprocessing step, mainly as a means of guaranteeing safe parallel execution in a shared-memory environment but also in order to enforce neighborhood heuristics, *e.g.* avoid having adjacent graph edges collapse in sequence in graph coarsening [6]. Examples of such applications include iterative methods for sparse linear systems [14], sparse tiling [19, 20], eigenvalue computation [16], preconditioners [18, 12] and mesh adaptivity [7, 10].

Taking advantage of modern multi-core and many-core hardware requires not only algorithmic modifications to deal with data races but also consideration of scalability issues. The exposed parallelism of an irregular algorithm is directly dependent on the number of colors used. The lower this number, the more work-items are available for concurrent processing per color/independent set. Additionally, there is usually some thread synchronization or reduction before proceeding to the next independent set. A poor-quality coloring will only exaggerate the effects of thread synchronization on the parallel scalability of an application. Following this observation, it is obvious that a good coloring algorithm should be fast and scalable itself, so as to minimize its own contribution to the total execution time of the application, and use as few colors as possible.

The simplest graph coloring algorithm is the greedy one, commonly known as *First-Fit* (§2.1). There exist parallel versions for distributed-memory environments, but in this paper we focus on the intra-node, shared-memory case. Probably, the best known parallel algorithm is the one by Jones and Plassmann [13], which in turn is an improved version of the original *Maximal Independent Set* algorithm by Luby [15]. There also exists a modified version of Jones-Plassmann which uses multiple hashes to minimize thread synchronization [3]. A parallel greedy coloring algorithm based on speculative execution was introduced by Gebremedhin and Manne [9]. Çatalyürek *et al.* presented an improved version of the original speculative algorithm in [1] (§2.2). We took the latter one step further, devising a method which runs under an even more speculative scheme with less thread synchronization (§3), without compromising coloring quality.

It must be pointed out that First-Fit variants which use ordering heuristics were not considered here. Despite recent innovations by Hasenplaugh *et al.* [11], those variants take considerably longer to run than the plain greedy algorithm and in many cases do not achieve a sufficiently large improvement in the number of colors to justify their cost. Runtime of coloring for the purpose of dynamic dependency analysis becomes a serious consideration in problems like morph algorithms [17], which mutate graph topology in non-trivial ways and constantly invalidate existing colorings. In those cases, the graph has to be recolored in every iteration of the morph kernel, so coloring becomes a recurring cost rather than a one-off preprocessing step. As shown in [11], heuristic-based algorithms, although achieving some reduction in the number of colors, take 4x-11x longer to run and this would dominate the kernel's runtime. A notable example is the edge-swap kernel from our mesh adaptivity framework PRAgMaTIc[3] [10], in which coloring (using our fast method) already takes up 10% of the total execution time.

The rest of this paper is organized as follows: In Section 2 we present the serial greedy coloring algorithm and its parellel implementation by Çatalyürek *et al.*. We explain how the latter can be improved further, leading to our implementation which is described in Section 3 and evaluated against its predecessor in Section 4. Finally, we briefly explain why the class of optimistic coloring algorithms is unsuitable for SIMT-style parallel processing systems in Section 5 and conclude the paper in Section 6.

[3] https://github.com/meshadaptation/pragmatic

2 Background

In this section we describe the greedy coloring algorithm and its parallel version proposed by Çatalyürek *et al.*.

2.1 First-Fit Coloring

Coloring a graph with the minimal number of colors has been shown to be an NP-hard problem [8]. However, there exist heuristic algorithms which color a graph in polynomial time using relatively few colors, albeit not guaranteeing an optimal coloring. One of the most common polynomial coloring algorithms is *First-Fit*, also known as *greedy coloring*. In its sequential form, First-Fit visits every vertex and assigns the smallest color available, *i.e.* not already assigned to one of the vertex's neighbors. The procedure is summarized in Algorithm 1.

Algorithm 1 Sequential greedy coloring algorithm.

Input: $\mathcal{G}(V, E)$
for all vertices $V_i \in V$ **do**
  $\mathcal{C} \leftarrow \{\text{colors of all colored vertices } V_j \in adj(V_i)\}$
  $c(V_i) \leftarrow \{\text{smallest color } \notin \mathcal{C}\}$

It is easy to give an upper bound on the number of colors used by the greedy algorithm. Let us assume that the highest-degree vertex V_h in a graph has degree d, *i.e.* this vertex has d neighbors. In the worst case, each neighbor has been assigned a unique color; then one of the colors $\{1, 2, \ldots, d+1\}$ will be available to V_h (*i.e.* not already assigned to a neighbor). Therefore, the greedy algorithm can color a graph with at most $d + 1$ colors. In fact, experiments have shown that First-Fit can produce near-optimal colorings for many classes of graphs [4].

2.2 Optimistic Coloring

Gebremedhin and Manne introduced an optimistic approach to parallelizing the greedy graph coloring algorithm [9]. They described a fast and scalable version for shared-memory systems based on the principles of speculative (or optimistic) execution. The idea is that we can color all vertices in parallel using First-Fit without caring about race conditions at first (stage 1); this can lead to defective coloring, *i.e.* two adjacent vertices might get the same color. Defects can then be spotted in parallel (stage 2) and fixed by a single thread (stage 3).

Picking up where Gebremedhin and Manne left off, Çatalyürek *et al.* improved the original algorithm by removing the sequential conflict-resolution stage and applying the first two parallel stages iteratively. This work was presented in [1]. Each of the two phases, called *tentative coloring* phase and *conflict detection* phase respectively, is executed in parallel over a relevant set of vertices. Like the original algorithm by Gebremedhin and Manne, the tentative coloring

phase produces a pseudo-coloring of the graph, whereas in the conflict detection phase threads identify defectively colored vertices and append them into a list $\mathcal{L}$. Instead of resolving conflicts in $\mathcal{L}$ serially, $\mathcal{L}$ now forms the new set of vertices over which the next execution of the tentative coloring phase will iterate. This process is repeated until no conflicts are encountered.

Algorithm 2 The parallel graph coloring algorithm by Çatalyürek *et al.*.

```
Input: G(V, E)
U ← V
while U ≠ ∅ do
    #pragma omp parallel for          ▷ Phase 1 - Tentative coloring (in parallel)
    for all vertices V_i ∈ U do                                  ▷ execute First-Fit
        C ← {colors of all colored vertices V_j ∈ adj(V_i)}
        c(V_i) ← {smallest color ∉ C}
    #pragma omp barrier
    L ← ∅                                  ▷ global list of defectively colored vertices
    #pragma omp parallel for          ▷ Phase 2 - Conflict detection (in parallel)
    for all vertices V_i ∈ U do
        if ∃V_j ∈ adj(V_i), V_j > V_i : c(V_j) == c(V_i) then
            L ← L ∪ V_i                                  ▷ mark V_i as defectively colored
    #pragma omp barrier
    U ← L                                  ▷ Vertices to be re-colored in the next round
```

Algorithm 2 summarizes this coloring method. As can be seen, there is no sequential part in the whole process. Additionally, speed does not come at the expense of coloring quality. The authors have demonstrated that this algorithm produces colorings using about the same number of colors as the serial greedy algorithm. However, there is still a source of sequentiality, namely the two thread synchronization points in every iteration of the while-loop. Synchronization can easily become a scalability barrier for high numbers of threads and should be minimized or eliminated if possible.

3 Implementation

Moving toward the direction of removing as much thread synchronization as possible, we improved the algorithm by Çatalyürek *et al.* by eliminating one of the two barriers inside the while-loop. This was achieved by merging the two parallel for-loops into a single parallel for-loop. We observed that when a vertex is found to be defective it can be re-colored immediately instead of deferring its re-coloring for the next round. Therefore, the tentative-coloring and conflict-detection phases can be combined into a single *detect-and-recolor* phase in which we inspect all vertices which were re-colored in the previous iteration of the while-loop. Doing so leaves only one thread synchronization point per round, as can be seen in Algorithm 3. This barrier guarantees that any changes committed by a thread are made visible system-wide before proceeding to the next round.

Algorithm 3 The improved parallel graph coloring technique.

```
Input: G(V, E)
#pragma omp parallel for                    ▷ perform tentative coloring on G; round 0
for all vertices V_i ∈ V do
    C ← {colors of all colored vertices V_j ∈ adj(V_i)}
    c(V_i) ← {smallest color ∉ C}
#pragma omp barrier
U^0 ← V                                     ▷ mark all vertices for inspection
i ← 1                                       ▷ round counter
while U^{i−1} ≠ ∅ do                        ▷ ∃ vertices (re-)colored in the last round
    L ← ∅                                   ▷ global list of defectively colored vertices
    #pragma omp parallel for
    for all vertices V_i ∈ U^{i−1} do
        if ∃V_j ∈ adj(V_i), V_j > V_i : c(V_j) == c(V_i) then   ▷ if they are (still) defective
            C ← {colors of all colored V_j ∈ adj(V_i)}          ▷ re-color them
            c(V_i) ← {smallest color ∉ C}
            L ← L ∪ V_i                     ▷ V_i was re-colored in this round
    #pragma omp barrier
    U_i ← L                                 ▷ Vertices to be inspected in the next round
    i ← i + 1                               ▷ proceed to the next round
```

4 Experimental Results

In order to evaluate our improved coloring method, henceforth referred to as *Reduced Synchronization Optimistic Coloring* (RSOC), and compare it to the previous state-of-the-art technique by Çatalyürek *et al.*, we ran a series of benchmarks using 2D and 3D meshes of triangular and tetrahedral elements respectively (commonly used in finite element and finite volume methods), alongside randomly generated graphs using the *R-MAT* graph generation algorithm [2]. Simplicial 2D/3D meshes are used in order to measure performance and scalability for our target application area ([10]), whereas RMAT graphs were used for consistency with the experimental methodology used in Çatalyürek *et al.*'s publication; the authors state that those RMAT graphs "are designed to represent instances posing varying levels of difficulty for the performance of multithreaded coloring algorithms" [1].

For the 2D case we have used a 2D anisotropic mesh (adapted to the requirements of some CFD problem) named `mesh2d`, which consists of $\approx 250k$ vertices. We also evaluate performance using two 3D meshes, taken from the University of Florida Sparse Matrix Collection [5]. `bmw3_2` is a mesh modelling a BMW Series 3 car consisting of $\approx 227k$ vertices, whereas `pwtk` represents a pressurized wind tunnel and consists of $\approx 218k$ vertices. Finally, we generated three $16M$-vertex, $128M$-edge RMAT graphs, namely `RMAT-ER` (Erdős-Rényi), `RMAT-G` (Good) and `RMAT-B` (Bad), randomly shuffling vertex indices so as to reduce the benefits of data locality and large caches. For more information on those graphs the reader is referred to the original publication by Çatalyürek *et al.* [1].

The experiments were run on two systems: a dual-socket Intel®Xeon® E5-2650 system (Sandy Bridge, 2.00GHz, 8 physical cores per socket, 2-way hyper-threading) running Red Hat®Enterprise Linux® Server release 6.4 (Santiago) and an Intel®Xeon Phi™ 5110P board (1.053GHz, 60 physical cores, 4-way hyper-threading). Both versions of the code (intel64 and mic) were compiled with Intel®Composer XE 2013 SP1 and with the compiler flags `-O3 -xAVX`. The benchmarks were run using Intel®'s thread-core affinity support.

Table 1 shows the average execution time over 10 runs of both algorithms on the 2 systems, Intel®Xeon® and Intel®Xeon Phi™, using the 3 finite element/volume meshes and the 3 RMAT graphs. Rows preceded by "C" correspond to the algorithm by Çatalyürek *et al.*, rows preceded by "R" pertain to the improved version. Timings for the meshes are given in milliseconds whereas for the RMAT graphs they are in seconds. As can be seen, RSOC performs faster than Çatalyürek *et al.* for every test graph on both platforms, while scaling better as the number of threads increases, especially on Intel®Xeon Phi™.

Table 1. Execution time of both algorithms on 2 different platforms, Intel®Xeon® and Intel®Xeon Phi™, with varying number of OpenMP threads and using the 3 finite element/volume meshes and the 3 RMAT graphs. Rows preceded by "C" correspond to the algorithm by Çatalyürek *et al.*, rows preceded by "R" pertain to the improved version. Timings for the meshes are given in milliseconds whereas for the graphs they are in seconds.

		Intel®Xeon®						Intel®Xeon Phi™								
		Number of OpenMP threads						Number of OpenMP threads								
		1	2	4	8	16	32	1	2	4	8	15	30	60	120	240
mesh2d	C:	62.7	34.0	19.2	10.2	5.92	4.28	496	252	127	64.9	35.5	19.0	11.7	12.7	73.6
	R:	62.2	31.3	17.7	9.42	5.50	4.05	495	249	125	63.3	34.5	17.9	10.7	10.5	69.4
bmw3_2	C:	58.1	33.5	14.4	7.84	4.73	3.61	468	235	118	60.0	33.1	18.0	11.5	12.7	74.2
	R:	57.8	29.4	12.1	6.48	3.91	3.30	466	234	117	59.2	32.4	17.1	9.88	11.0	54.9
pwtk	C:	40.1	24.0	14.5	8.07	4.96	3.65	465	233	117	59.6	33.2	18.2	11.1	12.9	74.4
	R:	39.8	20.0	11.3	6.08	3.81	3.30	464	232	117	58.9	32.4	17.2	10.6	11.0	59.9
RMAT-ER	C:	6.11	3.21	1.82	1.09	0.79	0.85	196	97.8	48.9	24.6	13.0	6.41	3.16	1.64	0.94
	R:	6.09	3.20	1.81	1.08	0.78	0.85	196	98.0	49.0	24.7	13.1	6.43	3.16	1.64	0.95
RMAT-G	C:	6.10	3.18	1.82	1.08	0.77	0.81	195	97.1	48.6	24.3	12.9	6.34	3.12	1.62	0.93
	R:	6.07	3.17	1.81	1.07	0.77	0.81	195	97.3	48.7	24.4	13.0	6.38	3.13	1.63	0.93
RMAT-B	C:	5.47	2.86	1.62	0.93	0.65	0.64	189	94.1	46.7	23.5	12.3	6.08	3.12	1.90	1.49
	R:	5.46	2.83	1.60	0.92	0.64	0.63	189	94.0	46.9	23.5	12.4	6.02	2.95	1.60	1.00

Figures 1 and 2 show the relative speedup of RSOC over Çatalyürek *et al.* for all test graphs on Intel®Xeon® and Intel®Xeon Phi™, respectively, *i.e.* how much faster our implementation is than its predecessor for a given number of threads. With the exception of `RMAT-ER` and `RMAT-G` on which there is no difference in performance, the gap between the two algorithms widens as the number of threads increases, reaching a maximum value of 50% on Intel®Xeon Phi™ for `RMAT-B`.

Looking at the total number of coloring conflicts encountered throughout the execution of both algorithms as well as the number of iterations each algo-

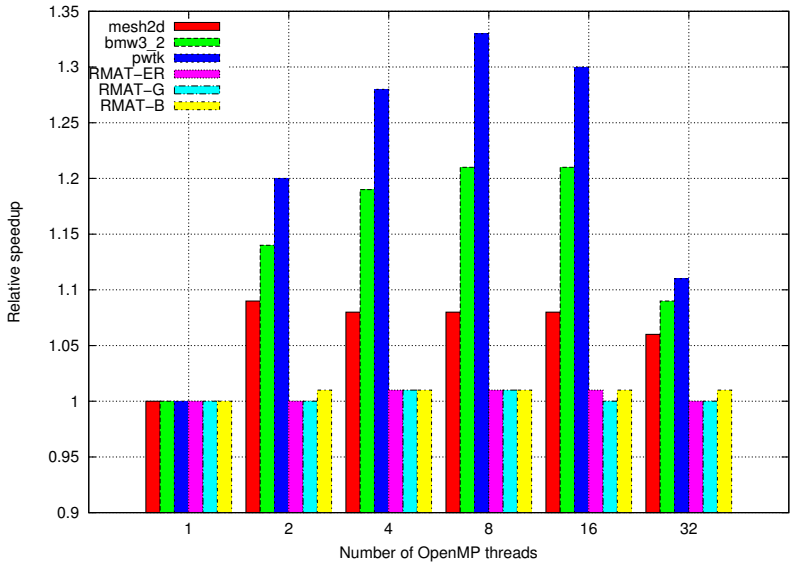

Fig. 1. Speedup of RSOC relative to Çatalyürek *et al.* as the number of threads increases on Intel®Xeon® E5-2650.

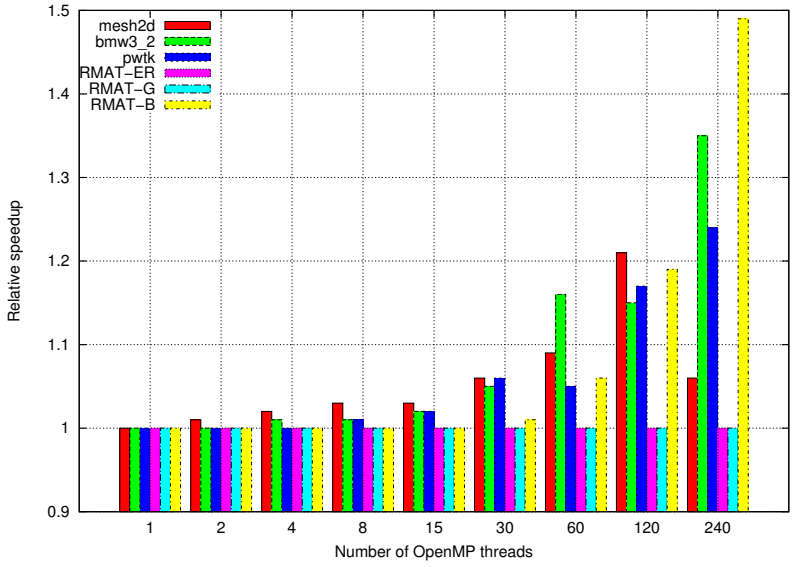

Fig. 2. Speedup of RSOC relative to Çatalyürek *et al.* as the number of threads increases on Intel®Xeon Phi™ 5110P.

rithm needs in order to resolve them, we can identify an additional source of speedup for our algorithm (apart from the absence of one barrier). We will use the Intel®Xeon Phi™ system for this study, as it is the platform on which the most interesting results have been observed. Figures 3 and 4 depict the total number of conflicts for the three meshes and the RMAT graphs, respectively. When using few threads both algorithms produce about the same number of conflicts. However, moving to higher levels of parallelism reveals that RSOC results in much fewer defects in coloring for certain classes of graphs.

This observation can be explained as follows: In Çatalyürek *et al.* all threads synchronize before entering the conflict-resolution phase, which means that they enter that phase and start resolving conflicts at the very same time. Therefore, it is highly possible that two adjacent vertices with conflicting colors will be processed by two threads simultaneously, which leads once again to new defects. In our improved algorithm, on the other hand, a conflict is resolved as soon as it is discovered by a thread. The likelihood that another thread is recoloring a neighboring vertex at the same time is certainly lower than in Çatalyürek *et al.*.

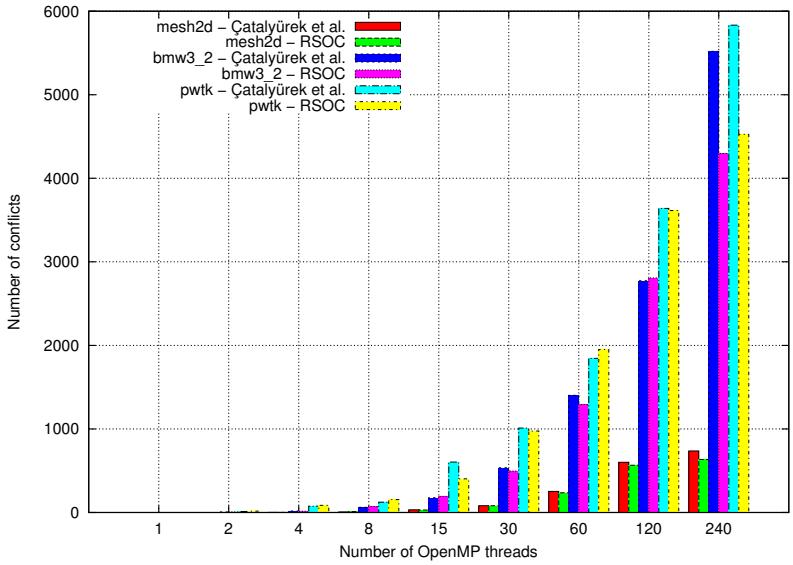

Fig. 3. Number of conflicts on Intel®Xeon Phi™ 5110P using `mesh2d`, `bmw3_2` and `pwtk`.

The reduced number of conflicts also results in fewer iterations of the algorithm, as can be seen in Figures 5 and 6. Combined with the absence of one barrier from the while-loop, it is only expected that our new algorithm ultimately outperforms its predecessor. A nice property is that both algorithms produce colorings using the same number of colors, *i.e.* quality of coloring is not compromised by the higher execution speed.

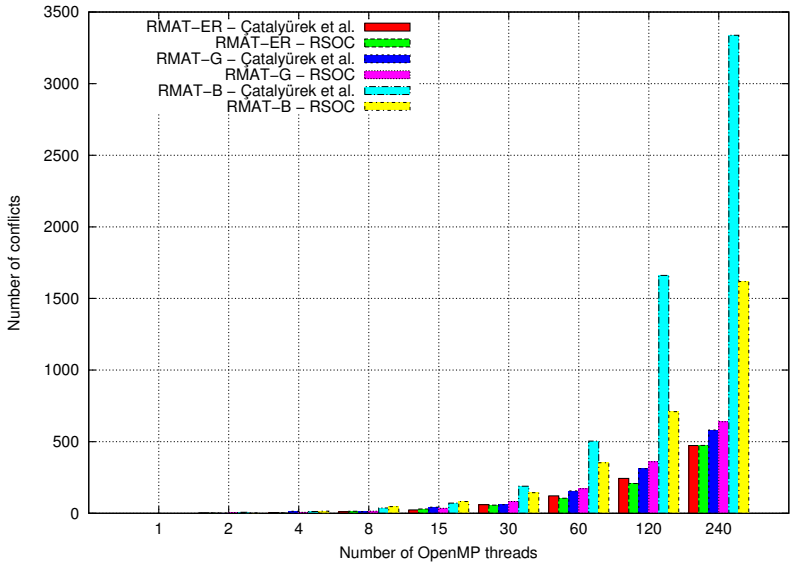

Fig. 4. Number of conflicts on Intel®Xeon Phi™ 5110P using `RMAT-ER`, `RMAT-G` and `RMAT-B`.

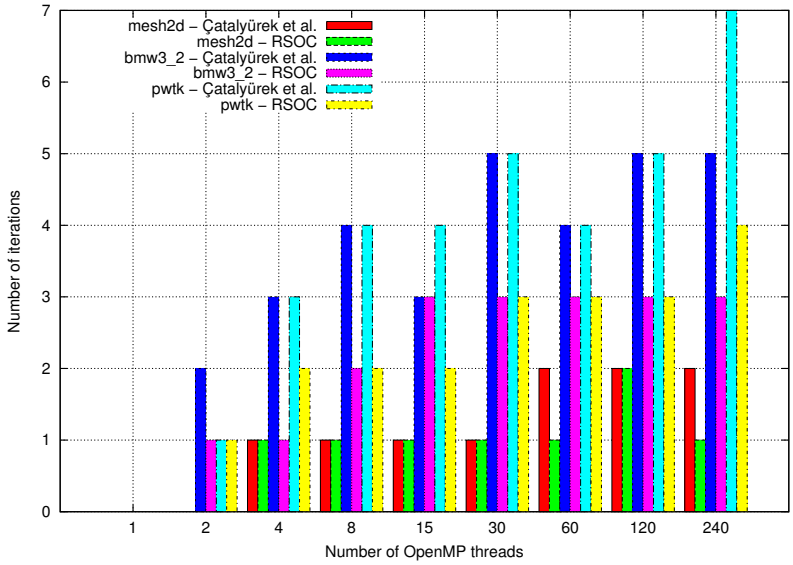

Fig. 5. Number of iterations on Intel®Xeon Phi™ 5110P using `mesh2d`, `bmw3_2` and `pwtk`.

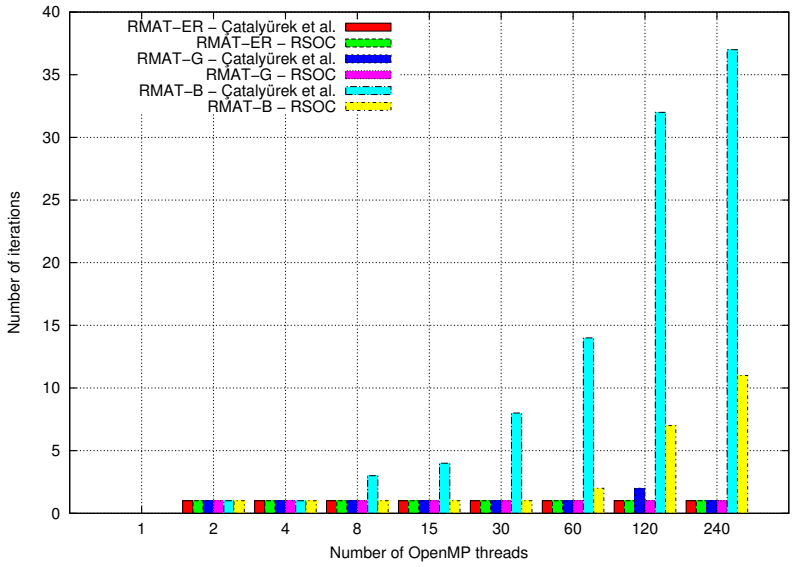

Fig. 6. Number of iterations on Intel®Xeon Phi™ 5110P using `RMAT-ER`, `RMAT-G` and `RMAT-B`.

5 SIMT restrictions

Trying to run the optimistic coloring algorithms using CUDA on an Nvidia GPU revealed a potential weakness. Neither algorithm terminated; instead, threads spun forever in an infinite loop. This is due to the nature of SIMT-style multithreading, in which the lockstep warp execution results in ties never being broken. An example of why these algorithms result in infinite loops in SIMT-style parallelism can be seen in Figure 7, where we have a simple two-vertex graph and two threads, each processing one vertex (this scenario is likely to actually occur at a later iteration of the while-loop, where the global list of defects $\mathcal{L}$ is left with a few pairs of adjacent vertices). At the beginning (a), both vertices are uncolored. Each thread decides that the smallest color available for its own vertex is red. Both threads commit their decision at the same clock cycle, which results in the defective coloring shown in (b). In the next round the threads try to resolve the conflict and decide that the new smallest color available is green. The decision is committed at the same clock cycle, resulting once again in defects (c) and the process goes on forever.

Theoretically, this scenario is possible for CPUs as well, although the probability is extremely low. We believe that there will always be some randomness (*i.e.* lack of thread coordination) on CPUs which guarantees convergence of the optimistic algorithms. This randomness can also be "emulated" on GPUs by having a dynamic assignment of vertices to threads and making sure that two adjacent vertices are always processed by threads of different warps.

(a) Graph (b) Round 1 (c) Round 2 (d) Round 3

Fig. 7. Example of an infinite loop in SIMT-style parallelism when using one of the optimistic coloring algorithms.

6 Conclusions

In this article we presented an older parallel graph coloring algorithm and showed how we devised an improved version which outperforms its predecessor, being up to 50% faster for certain classes of graphs and scaling better on manycore architectures. The difference becomes more pronounced as we move to graphs with higher-degree vertices (3D meshes, RMAT-B graph).

This observation also implies that our method (with the appropriate extensions) could be a far better option for d-distance colorings of a graph $\mathcal{G}$, where $\mathcal{G}^d$ is considerably more densely connected than $\mathcal{G}$ (graph $\mathcal{G}^d$, the d^{th} power graph of $\mathcal{G}$, has the same vertex set as $\mathcal{G}$ and two vertices in $\mathcal{G}^d$ are connected by an edge if and only if the same vertices are within distance d in $\mathcal{G}$).

Speed and scalability stem from two sources, (a) reduced number of conflicts which also results in fewer iterations and (b) reduced thread synchronization per iteration. Coloring quality remains at the same levels as in older parallel algorithms, which in turn are very close to the serial greedy algorithm, meaning that they produce near-optimal colorings for most classes of graphs.

Acknowledgments

The authors gratefully acknowledge funding from EPSRC grants EP/I00677X/1 and EP/L000407/1 supporting this work.

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 Computing 38(10–11), 576–594 (2012)
2. Chakrabarti, D., Faloutsos, C.: Graph mining: Laws, generators, and algorithms. ACM Computing Surveys 38(1), 2 (2006)
3. Cohen, J., Castonguay, P.: Efficient Graph Matching and Coloring on the GPU. `http://on-demand.gputechconf.com/gtc/2012/presentations/S0332-Efficient-Graph-Matching-and-Coloring-on-GPUs.pdf`
4. Coleman, T., Moré, J.: Estimation of sparse jacobian matrices and graph coloring problems. SIAM Journal on Numerical Analysis 20(1), 187–209 (1983), `http://dx.doi.org/10.1137/0720013`
5. Davis, T.A., Hu, Y.: The University of Florida Sparse Matrix Collection. ACM Transactions on Mathematical Software 38(1), 1:1–1:25 (Dec 2011), `http://doi.acm.org/10.1145/2049662.2049663`

6. De Cougny, H., Shephard, M.S.: Parallel refinement and coarsening of tetrahedral meshes. International Journal for Numerical Methods in Engineering 46(7), 1101–1125 (1999)
7. Freitag, L.F., Jones, M.T., Plassmann, P.E.: The Scalability Of Mesh Improvement Algorithms. In: IMA Volumes In Mathematics And Its Applications. pp. 185–212. Springer-Verlag (1998)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
9. Gebremedhin, A.H., Manne, F.: Scalable parallel graph coloring algorithms. Concurrency: Practice and Experience 12(12), 1131–1146 (2000), http://dx.doi.org/10.1002/1096-9128(200010)12:12<1131::AID-CPE528>3.0.CO;2-2
10. Gorman, G.J., Rokos, G., Southern, J., Kelly, P.H.J.: Thread-parallel anisotropic mesh adaptation. Accepted for Publication in Proceedings of the 4th Tetrahedron Workshop on Grid Generation for Numerical Computations (2015)
11. Hasenplaugh, W., Kaler, T., Schardl, T.B., Leiserson, C.E.: Ordering heuristics for parallel graph coloring. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures. pp. 166–177. SPAA '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2612669.2612697
12. Hysom, D., Pothen, A.: A scalable parallel algorithm for incomplete factor preconditioning. SIAM J. Sci. Comput. 22(6), 2194–2215 (Jun 2000), http://dx.doi.org/10.1137/S1064827500376193
13. Jones, M.T., Plassmann, P.E.: A parallel graph coloring heuristic. SIAM J. SCI. COMPUT 14, 654–669 (1992)
14. Jones, M.T., Plassmann, P.E.: Scalable iterative solution of sparse linear systems. Parallel Comput. 20(5), 753–773 (May 1994), http://dx.doi.org/10.1016/0167-8191(94)90004-3
15. Luby, M.: A simple parallel algorithm for the maximal independent set problem. In: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing. pp. 1–10. STOC '85, ACM, New York, NY, USA (1985), http://doi.acm.org/10.1145/22145.22146
16. Manne, F.: A parallel algorithm for computing the extremal eigenvalues of very large sparse matrices. In: Proceedings of the 4th International Workshop on Applied Parallel Computing, Large Scale Scientific and Industrial Problems. pp. 332–336. PARA '98, Springer-Verlag, London, UK, UK (1998), http://dl.acm.org/citation.cfm?id=645781.666669
17. Nasre, R., Burtscher, M., Pingali, K.: Morph algorithms on GPUs. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 147–156. PPoPP '13, ACM, New York, NY, USA (2013), http://iss.ices.utexas.edu/Publications/Papers/nasre13-ppopp.pdf
18. Saad, Y.: ILUM: A Multi-Elimination ILU Preconditioner for General Sparse Matrices. SIAM Journal on Scientific Computing 17(4), 830–847 (1996), http://dx.doi.org/10.1137/0917054
19. Strout, M.M., Carter, L., Ferrante, J., Freeman, J., Kreaseck, B.: Combining Performance Aspects of Irregular Gauss-Seidel via Sparse Tiling. In: 15th Workshop on Languages and Compilers for Parallel Computing (LCPC). College Park, Maryland (July 25-27, 2002)
20. Strout, M.M., Luporini, F., Krieger, C.D., Bertolli, C., Bercea, G.T., Olschanowsky, C., Ramanujam, J., Kelly, P.H.J.: Generalizing run-time tiling with the loop chain abstraction. In: Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS) (May 2014)