Improving Graph Coloring on Distributed-Memory Parallel Computers

Ahmet Erdem Sarıyüce*[†], Erik Saule*, and Ümit V. Çatalyürek*[‡]
* *Department of Biomedical Informatics*
[†] *Department of Computer Science and Engineering*
[‡]*Department of Electrical and Computer Engineering*
The Ohio State University
Email: {aerdem,esaule,umit}@bmi.osu.edu

Abstract

Graph coloring is a combinatorial optimization problem that classically appears in distributed computing to identify the sets of tasks that can be safely performed in parallel. Despite many existing efficient sequential algorithms being known for this NP-Complete problem, distributed variants are challenging. Building on an existing distributed-memory graph coloring framework, we investigate two techniques in this paper. First, we investigate the application of two different vertex-visit orderings, namely Largest First and Smallest Last, in a distributed context and show that they can help to significantly decrease the number of colors, on small- to medium-scale parallel architectures. Second, we investigate the use of a distributed post-processing operation, called recoloring, which further drastically improves the number of colors while not increasing the runtime more than twofold on large graphs. We also investigate the use of multicore architectures for distributed graph coloring algorithms.

1. Introduction

Graph coloring is a combinatorial problem which consists of partitioning a graph in a minimum number of independent sets. This problem has numerous applications, the most common one in parallel computing is to represent the tasks of a computation as the vertices of a graph. An edge connects two vertices if these two vertices cannot be computed simultaneously. Finding a coloring of this graph allows to partition the tasks into sets that can be safely computed in parallel. Minimizing the number of colors decreases the number of synchronization points in the computation and increases the efficiency of the parallel platform. Other applications of graph coloring appear in automatic differentiation [1], printed circuit testing [2], frequency assignment [3], register allocation [4], parallel numerical computation [5] and optimization [6] areas.

There exist different variants of the graph coloring problem. The most classical one is sometimes referred to as the distance-1 coloring problem where two adjacent vertices must have different colors. Some applications require that vertices separated by k edges to have different colors. This problem is the distance-k coloring problem. In this paper, we are only interested in distance-1 coloring. However, we believe that all the techniques and results presented in this document can be extended to the other variants of the graph coloring problem.

The distance-1 coloring problem is formally defined as follows. Let $G = (V, E)$ be a graph with $|V|$ vertices and $|E|$ edges. The set of neighbours of a vertex v is $adj(v)$; its cardinality, also called the *degree* of v, is δ_v. The degree of the vertex having the most neighbor is $\Delta = \max_v \delta_v$ A coloring $C : V \to \mathbb{N}$ is a function that maps each vertex of the graph to a color (represented by an integer), such that two adjacent vertices have different colors, i.e. $\forall(u, v) \in E, C(u) \neq C(v)$. Without loss of generality, the number of colors used is $\max_{u \in V} C(u)$. The optimization problem at hand is to find a coloring with as few colors as possible. The problem of finding the minimal number of colors γ a graph can be colored with (also called the chromatic number of the graph) is known to be NP-Complete for arbitrary graphs [7]. Therefore, the problem at hand is NP-Hard. Recently, it has been shown that, for all $\epsilon > 0$, it is NP-Hard to *approximate* the graph coloring problem within $|V|^{1-\epsilon}$ [8].

In this work, we are tackling the distributed memory graph coloring that appears in large scientific parallel applications. In such applications, the computational model (hence the graph) is already distributed onto the nodes of the parallel machine. If the graph is too large to fit in the memory of a single computer, we have no choice but to color it in distributed memory. However, if the graph is sufficiently small, with a naive approach, one can aggregate it on a single node and color it there. A better approach would be taking advantage of the partitioning of the graph, and color the *interior* vertices (vertices for which all their neighbors are local) in parallel and then color the remaining, *boundary vertices* (vertices that have at least one non-local neighbor), sequentially, by aggregating the graph induced by them on a single processor. However, as shown in [9], one can do significantly better by using a real distributed memory coloring algorithm.

Here, we further aim to improve the distributed-memory coloring algorithm presented in [9] in multiple directions. First, we will investigate different vertex-visit ordering strategies, namely Largest First (LF) and Smallest Last (SL) techniques for improving the number of colors. It is intuitive that distributed-memory coloring algorithms increase the number of colors used compared to the sequential greedy algorithm. Therefore, we would like to investigate how increasing the number of processors involved in coloring affects the number of colors. Hence, we will evaluate the algorithms at large scales. Last, but not least, we will investigate the application of recoloring in the distributed-memory setting, to further reduce the number of colors.

The remaining of the document is organized as follows. Section 2 presents the different coloring algorithms existing for sequential and parallel architectures. Section 3 recalls the previous distributed-memory coloring algorithm which is the reference algorithm we use. Section 4 discusses the techniques used to improve coloring in distributed-memory architecture. The proposed techniques are experimentally assessed on real-world graphs and on random graphs in Section 5. Final remarks and ideas to improve further are given in Section 6.

2. Related Work

Graph coloring is one of the well studied problems in the literature [1], [10], [11], [12]. Literature is abundant with many different techniques, such as the one that utilizes greedy coloring [13], [12], cliques [14] and Zykov trees [11].

Despite the pessimistic theoretical results and the existence of more complicated algorithms, for many graphs that arise in practice, solutions that are provably optimal or near optimal can be obtained using a simple *greedy* algorithm [6]. In this algorithm, the vertices of the graph are visited in some *order* and the smallest permissible color at each iteration is assigned to the vertex. Pseudo code of this technique is presented in Algorithm 1. Choosing the smallest permissible color is known as the *First Fit* strategy.

This simple algorithm has two nice properties. First, for any vertex visit ordering, it produces a coloring with at most $1+\Delta$ colors. Second, for some vertex-visit orderings it will produce an optimal coloring [15]. Many heuristics for ordering the vertices have been proposed in the literature [15], [1]. Commonly known vertex orderings are *Largest First* (LF), *Smallest Last* (SL), *Saturation Degree*, and *Incidence Degree* orderings. We refer the reader to [1] for a succinct summary of these ordering techniques. The LF ordering, introduced by Welsh and Powell [16], visits the vertices in non-increasing degree order. SL approaches the ordering from backwards by selecting a vertex with the minimum degree to be ordered last and removing it from the graph for the rest of the ordering phase. Then the next vertex with the minimum degree from the remaining graph is selected to be ordered second-to-last. This procedure is repeated until all vertices are ordered. In other words, the graph is dynamically updated to select a vertex, hence SL produces orderings that are quite different than LF. To the best of our knowledge, the only distributed-memory work that investigated similar ideas to ordering the vertices is [17]. It is a theoretical work where a randomized coloring algorithm is used and ties are broken by choosing the vertex having smaller degree to be recolored. In our work, we will experimentally investigate both the LF and SL orderings in the context of distributed memory parallel coloring.

Algorithm 1: Sequential greedy coloring.

```
Data: $G = (V, E)$
for each $v \in V$ do
    for each $w \in adj(v)$ do
        forbiddenColors[color[$w$]] $\leftarrow v$
    color[$v$] $\leftarrow \min\{i > 0 :$ forbiddenColors[$i$] $\neq v\}$
```

Culberson [15] introduces a coloring algorithm, *Iterated Greedy* (IG), where starting with an initial greedy coloring, vertices are iteratively recolored based on the coloring of the previous iteration. Culberson also shows that in recoloring, if the vertices belonging to the same color class (i.e., the vertices of the same color) in previous coloring are colored consecutively, then the number of colors will either decrease or stay the same. Several different heuristics are presented to obtain new permutations of color classes. These heuristics are based on the colors of the vertices, color distribution of the vertices, degree of the vertices and mix of them. Culberson suggests that a hybrid approach, that is, changing the permutation heuristic at each recoloring iteration is more effective than applying the same heuristic at each round. The first work evaluating the effects of recoloring scheme on parallel graph coloring is [18]. In this work, Gebremedhin and Manne use the IG heuristic for graph coloring on shared-memory computers. Apart from them, Goldberg et al. suggest to use recoloring in edge-coloring of multigraphs [19].

Many parallel algorithms for graph coloring [5], [20], [21] are based on finding a maximal (by inclusion) independent set of vertices [22]. A set of vertices are independent if there is no edge between any two vertices, and this set is maximal if no other vertex can be added to this set while keeping it an independent set. An independent set of vertices can be colored in parallel by a small modification of Luby's algorithm [22]. In Luby's algorithm, each vertex is assigned a random number at the beginning. Then for each vertex, the algorithm checks if the random number of this vertex is larger than the random number of its neighbors. If so, this vertex and its neighbors are removed from the graph and the vertex is added to independent set. The same operation is applied recursively until the graph is empty. If the vertex with the largest random number with respect to its neighbors is assigned a smallest permissible color at this instant, instead of being removed, then we get a parallel graph coloring.

3. Distributed-Memory Coloring

Bozdag et al. [9] introduce a distributed-memory parallel graph coloring framework. To the best of our knowledge, their work is the only distributed-memory coloring algorithm that shows a parallel speedup, hence our work is based on their algorithm, which we present here briefly.

The graph is assumed to be originally distributed onto the distributed memory. Each vertex is owned by a single processor. Each processor only knows the edges that connect any vertex it owns. That is to say, if both extremities of an edge are owned by a single processor then only this processor knows about that edge. If an edge connects vertices owned by two different processors, then that edge is only known by these two processors. In the algorithm, each processor is responsible for coloring the vertices it owns. Vertices that are connected to a vertex owned by another processor are said to be *boundary* vertices. If all the neighboring vertices of a vertex are owned by the same processor, this vertex is said to be an *internal* vertex.

The coloring is constructed in multiple rounds. In each round, all the uncolored vertices are tentatively colored using the greedy coloring algorithm. Then each processor independently detects conflicts. When a conflict occurs, one of the vertex will keep its color while the other one is marked for recoloring in the next round. The tie is broken based on a random total ordering generated beforehand. The algorithm iterates until there is no more conflicts.

In order to reduce the number of conflicts at each round, the coloring of the vertices is performed in supersteps. In each superstep, each processor colors a given number of its own vertices. Then, it exchanges the colors of the boundary vertices colored in that superstep, if any, with its *neighbor* processors. If a processor waits for its neighbors to finish their superstep before starting the next one, the process is said to be *synchronous* and it ensures that two vertices can only be in a conflict if they are colored in the same superstep. Otherwise, it is said to be *asynchronous*. The size of the superstep becomes important since a smaller size increases the number of messages exchanged on the network while a larger size is likely to increase the number of conflicts.

There are five main parameters affecting the behavior of the framework. These are superstep size, synchronous or asynchronous execution of supersteps, coloring order of the internal and boundary vertices, color selection strategy, and broadcast or customized communication. Several combinations of those were experimentally investigated to determine the best parameters for reducing the number of colors and the runtime. The study shows that there is no single combination that outperforms the others in both metrics (the number of colors and the runtime). In general, customized communication improves the metrics over broadcast, the best runtime is achieved by coloring internal vertices first using asynchronous communication, whereas the best number of colors is achieved by coloring boundary vertices first.

4. Improving Number of Colors

In this work, we will investigate two methods to decrease the number of colors.

4.1. Vertex-Visit Ordering

As described in Section 2, the performance of the greedy algorithm depends on the order in which vertices are visited. The orderings considered in the parallel graph coloring algorithm presented in [9] only take into account the partitioning of the graph and not the properties of the graph itself. Three orderings were investigated in their work, coloring internal vertices first, boundary vertices first and coloring the vertices in the order they are stored in the memory (which was called *unordered* in [9], here we will call it *Natural* ordering). For boundary or internal first ordering, ordering of the vertices in these subclasses have not been specified, and in the implementation, natural ordering was used. The *Natural* ordering does not introduce any extra computation and is therefore, trivially, computed in $O(|V|)$. Internal first and boundary first orderings require classification of internal and boundary vertices, which requires $O(|E|)$ computation.

In this work we investigates two more successful ordering heuristics, namely *Largest First* (LF) and *Smallest Last* (SL) techniques, as described in Section 2. The LF ordering can be computed using bucket sort or a counting sort since the maximum degree is less than the number of vertices in the graph, leading to a $O(|V|)$ algorithm [16]. The SL ordering requires a data structure that allows to efficiently keep track of how many unordered neighbors a vertex has in order to find efficiently the one with the least number of unordered neighbors. A bucket and a fibonacci heap allow to implement the SL ordering with a complexity of $O(|E|\log|V|)$ [14].

In this work, we have designed a customized bucket data structure to implement SL in $O(|E|)$. The key point is that when a vertex is selected for ordering, the degree of all its adjacent vertices needs to be decreased by one. If one can keep this *update* operation proportional to the degree of that vertex, the SL ordering can be done in $O(|E|)$. Updating the neighbors will result in decreasing their degrees by one. By using an auxiliary pointer array to the elements, we can access each element (vertex) in the bucket in constant time. Deletion from bucket i, can be done in $O(1)$ and insertion to bucket $i-1$ can be done in $O(1)$. The *pop-min* operation must be implemented by using a state variable that indicates that the x first buckets are empty. The cost of increasing x is charged on *pop-min* only when it reaches a value it never reached before. Indeed, the other increases of x can be charged on different anterior calls to *update* since each of them decrease at most x by one. Despite a single call to *pop-min* requires $O(\Delta)$ operations in the worst case, all the calls to this function requires only $O(|V|)$ operations. Hence the SL ordering can be implemented in $O(|E|)$.

Notice that the graph is distributed in the memory of the processors. Therefore, building an ordering of the whole graph and enforcing to color the graph in the same order a sequential algorithm would do require numerous communications and sequentializes the coloring process. Therefore, in distributed memory, we opt that each processor computes an ordering of the graph based on the knowledge it possesses, i.e. edges connected to a vertex it owns. Each processor independently obtains an ordering of its own vertices based on local information. Therefore, there is no guarantee that the coloring computed on a single processor and on multiple processors will be the same.

4.2. Recoloring

As mentioned in Section 2, Culberson [15] investigated the use of iterative recoloring for improving the number of colors in a sequential algorithm. Here, we extend this work to distributed-memory graph coloring.

The recoloring idea naturally fits distributed-memory graph coloring. Assume that we have an initial coloring, so a set of vertices with the same color are independent, and hence can be safely colored in parallel without any communication. Our *recoloring algorithm* (RC) proceeds in as many steps as there was color in the initial coloring. In the recoloring process, all the vertices in the same color class (i.e., having the same color in the previous coloring round) are colored at the same step. Then the processors exchange the color information with their neighboring processors at the end of the step. Note that even if a processor does not have any vertices with that color, it waits for other processors to finish coloring at that step. This procedure ensures that no conflict is created by the end of recoloring. Provided an initial coloring, recoloring in sequential and in distributed memory lead to the same solution, making recoloring scalable in terms of number of colors. However, the procedure is fairly synchronous since a processor can not start the i-th step before its neighbors finished their $(i-1)$-th step. Moreover, there is no guarantee that two processors will have a similar number of vertices in each color, thus potentially leading to a load imbalance.

To mitigate some potential load imbalance due to synchronous execution of RC, we also propose a second recoloring approach, named *asynchronous recoloring* (aRC). In this algorithm, after a first parallel coloring step, each processor independently computes their vertex-visit orderings using the initial coloring, and do a second parallel coloring with this new vertex-visit ordering using the algorithm recalled in Section 3. Notice that, conflicts are possible at the end of this process, therefore this second parallel coloring step proceeds in conflict resolution rounds as in the original algorithm. We expect that this approach will not be as good as synchronous recoloring in terms of number of colors, but it might be beneficial for trading the quality for speed.

Name	$\|V\|$	$\|E\|$	Δ	NAT	LF	SL	seq time
auto	448K	3.3M	37	13	12	10	0.1103s
bmw3_2	227K	5.5M	335	48	48	37	0.0836s
hood	220K	4.8M	76	40	39	34	0.0752s
ldoor	952K	20.7M	76	42	42	34	0.3307s
msdoor	415K	9.3M	76	42	42	35	0.1458s
pwtk	217K	5.6M	179	48	42	33	0.0820s

Table 1. Properties of real-world graphs

Name	$\|V\|$	$\|E\|$	Δ	NAT	LF	SL
ER	16,777,216	134,217,624	42	12	10	10
Good	16,777,216	134,181,065	1,278	28	15	14
Bad	16,777,216	133,658,199	38,143	146	89	88

Table 2. Properties of synthetic graphs

In the recoloring process, all the vertices that belong to the same color class must be colored in a consecutive manner. However, one can choose any permutation of the color classes and this choice affects the number of colors significantly. Therefore different permutations of the color classes should be considered for sequential and parallel recoloring. We considered three permutations of color classes: Reverse order (RV) of colors [15], Non-Increasing number of vertices (NI), where the color classes are ordered in the non-increasing order of their vertex counts, and Non-Decreasing number of vertices (ND), which is similarly derived. We investigate these color permutations together with vertex-visit ordering in the context of distributed-memory parallel graph coloring.

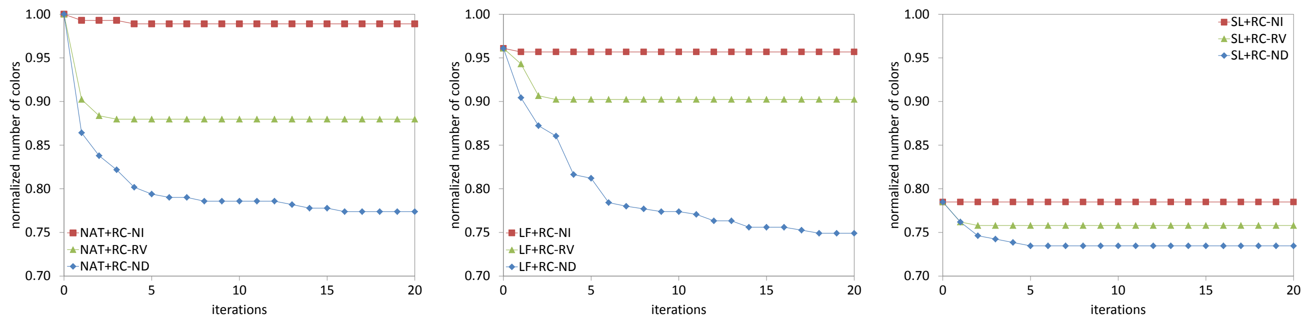
5. Experiments

5.1. Preliminaries

All the algorithms are implemented in Zoltan [23], an MPI-based C library for parallel partitioning, load balancing, coloring and data management services for distributed-memory systems. In the experiments the graphs are partitioned onto the parallel platform either using ParMETIS [24] version 3.1.1 or with a simple block partitioning.

All the algorithms were tested on an in-house cluster which consists of 64 computing nodes. Each node has 2 Intel Xeon E5520 (quad-core clocked at 2.27GHz) processors, 48GB memory, and 500 GB of local hard disk. The nodes are interconnected through 20Gbps DDR oversubscribed InfiniBand. They run CentOS with the Linux kernel 2.6.18. The C code is compiled with GCC 4.1.2 using the -O2 optimization flag. MVAPICH2 in version 1.6 is used to leverage efficiently the InfiniBand interconnect.

We run the experiments on number of processors which are a power of two from 1 to 512 processors. Notice



(a) Natural (NAT) (b) Largest First (LF) (c) Smallest Last (SL)

Figure 1. Sequential study on the real graphs

that each physical machine has 8 cores. When allocating processors on the cluster, we first use processors on different nodes to highlight distributed memory issue. Therefore, when 64 processors are used, each processor (core) is on a different machine. Using 128 processors allocates 2 cores per machine and an allocation of 512 processors uses the 8 cores of the 64 machines.

The experiments are run on six real-world application graphs and three randomly generated graphs. The real-world graphs come from various application areas including linear car analysis, finite element, structural engineering and automotive industry [25], [26]. They have been obtained from the University of Florida Sparse Matrix Collection[1] and the Parasol project. The list of the graphs and their main properties are summarized in Table 1. The number of colors obtained with a sequential run of the three vertex-visit ordering are also listed in the table. Finally, the time to compute the *Natural* coloring in sequential is given. Notice that the largest real world graph takes less than half a second to color sequentially using a *Natural* ordering, making the distributing coloring of all the graphs very challenging. These graphs are partitioned with ParMETIS.

The randomly generated graphs are RMAT graphs, which have been introduced by Chakrabarti et al. [27]. The main idea of RMAT is to recursively subdivide the adjacency matrix in 4 equal parts and to distribute the edges to these parts with a given probabilities, which might differ for each part. This procedure allows the generation of different classes of random graphs. We generated three graphs, namely RMAT-ER, RMAT-Good and RMAT-Bad with the degree distributions in the parts of $(0.25, 0.25, 0.25, 0.25)$, $(0.45, 0.15, 0.15, 0.25)$ and $(0.55, 0.15, 0.15, 0.15)$, respectively. These three graphs are generated to create different challenges to distributed-memory coloring algorithms. The first graph, RMAT-ER, belongs to the class of Erdős-Rényi random graphs which are known to be hard to partition. Hence, almost all vertices are likely to be on the boundary, especially when the number of processors increases. The other two belong to the class of scale-free graphs with power-law distribution and small-world characteristics. These graphs are known to be hard to partition, and they easily create very unbalanced workloads if a vertex partitioning scheme is used. Their properties are summarized in Table 2. These graphs are partitioned using a block partitioning.

For all the experiments we will present both the number of colors obtained and the runtime of the method when the number of processors varies. The real-world graphs all show the same trends and their results are aggregated in the following manner. Each value (number of colors and runtime) is first normalized with respect to the value obtained by the *Natural* ordering of the same graph on one processor. Then the normalized value for different graphs are aggregated using a geometric mean. The three randomly generated graphs are presented independently.

Before studying the impact of our methods in a distributed-memory setting, we present first their impact in a sequential setting on the real-world graphs. Figure 1 presents the impact on the number of colors of vertex-visit ordering and multiple iterations of recoloring. Each chart presents a different vertex-visit ordering and each curve presents a different color permutation in recoloring, e.g., NAT+RC-NI gives *Natural* ordering with recoloring with the Non-Increasing number of vertex order of the color classes. Notice that the charts start with 0 iteration which shows the quality of the vertex-visit ordering only.

First, one can see that the LF vertex ordering leads to lower number of colors than the *Natural* ordering with 0.96 normalized number of colors without recoloring. SL is obviously a much better ordering with a normalized number of colors of 0.78. Second, when recoloring, the three tested permutations of the color classes lead to a decrease of the number of colors. The NI permutation yields the smallest improvement, while the ND permutation obtains the smallest number of colors by reaching a normalized number of colors less than 0.8 after 20 iterations for the three orderings. Finally, the minimum number of colors is achieved by coupling the SL vertex ordering with the ND color permutation, called SL+RC-ND. The reason for the

1. http://www.cise.ufl.edu/research/sparse/matrices/

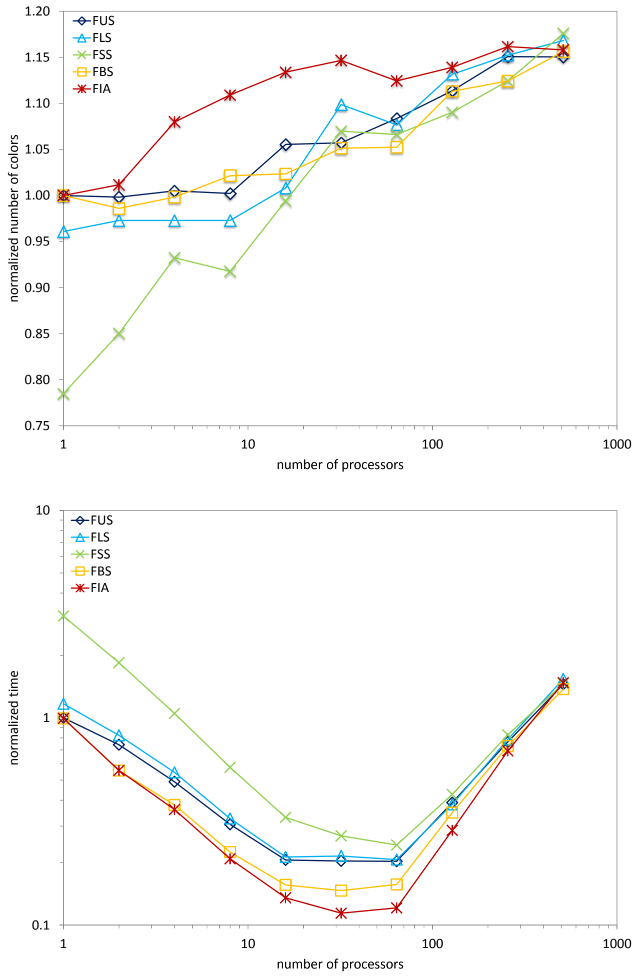


Figure 2. Comparison of ordering on the real graphs

improvement of the ND permutation over the other two relies on the selection of color classes. A permutation is successful if it can removes as many color classes as possible. This color permutation selects first the color classes with fewer vertices, so that the classes with more vertices (which are unlikely to disappear) can merge with them. The NI ordering is likely to lead to a solution very similar to the original First Fit solution, since lower colors are likely to be used more often. The same reason makes the RV better than NI since “reverse” is a reasonable approximation of ND.

5.2. Vertex-Visit Ordering

In the distributed-memory case, [9] concludes that coloring the boundary vertices first then internal ones while using a synchronous communication pattern (denoted FBS) leads to the best number of colors. Also coloring the internal vertices first while using asynchronous communication pattern leads to the best runtime (denoted FIA). *Natural* ordering was also presented and named FUS. Similarly, we name our new variants using the *Largest First*, and *Smallest Last* vertex-visit ordering with synchronous communication pattern as FLS and FSS, respectively. All these charts present results obtained with a superstep size of 500.

Figure 2 presents the obtained normalized number of colors and normalized runtime of the algorithm using different vertex-visit orders on the real graphs. FUS, FBS and FIA almost always obtain a normalized number of colors greater than 1. FIA gets notably bad normalized number of colors by reaching to 1.15 on 32 processors. FLS maintains a normalized number of color below 1 when used on less than 8 processors. FSS, thanks to the *Smallest Last* ordering, obtains much better colors than the other methods on less than 16 processors by achieving a normalized number of colors of less than 1 on 16 processors. When the number of processors increases, the advantages of the SL and LF orderings disappear. Indeed, these two ordering generates a lot of conflicts which reduce the impact of the ordering on the number of colors. The same effect can be seen for the asynchronous communication pattern of FIA which clearly gives worse number of colors than all the synchronous orderings. On 512 processors, the choice of the vertex-visit order does not yield much difference in number of colors.

The runtime of the methods on the real-world graphs, linearly decrease up to 16 processors and keep on decreasing up to 64 processors, but then almost linearly increase up to 512 processors. Note that FSS is the slowest one due to the ordering computation at the beginning but with the increasing number of processor, ordering becomes virtually free and reaches a speedup of 9.37 on 16 processors. As shown in [9] perfect speedup is almost never obtained on fast graph algorithms. After 64 processors, more than a core per machine is used. This reduces the memory bandwidth per processor and increases the access to the MPI subsystem. The runtime of all the vertex-visit ordering variants significantly increase. Here the efficiency of the asynchronous communication pattern of FIA is evident: FIA gets lower runtime than the synchronous methods from 64 processors to 512 processors.

Notice that these results are at a much larger scale than the one presented in [9]. It was claimed that the greedy coloring algorithm scales well, and our results confirm that fact up to 64 processors. On more than 64 processors, our experiments shows that the runtime degrades linearly in the number of processors. One of the reasons is that the machine we are using have multi-core processors and the algorithms use no knowledge of the processor hierarchy. Frequent and small communication stresses both the memory subsystem and the MPI subsystem. The other reason is the increasing number of boundary vertices. As the number of processors increases, the number of boundary vertices also increases because of the partitioning. It leads to more conflicts and the runtime degrades.

The impact of the vertex-visit order is presented on the randomly generated graphs RMAT-ER, RMAT-Good and

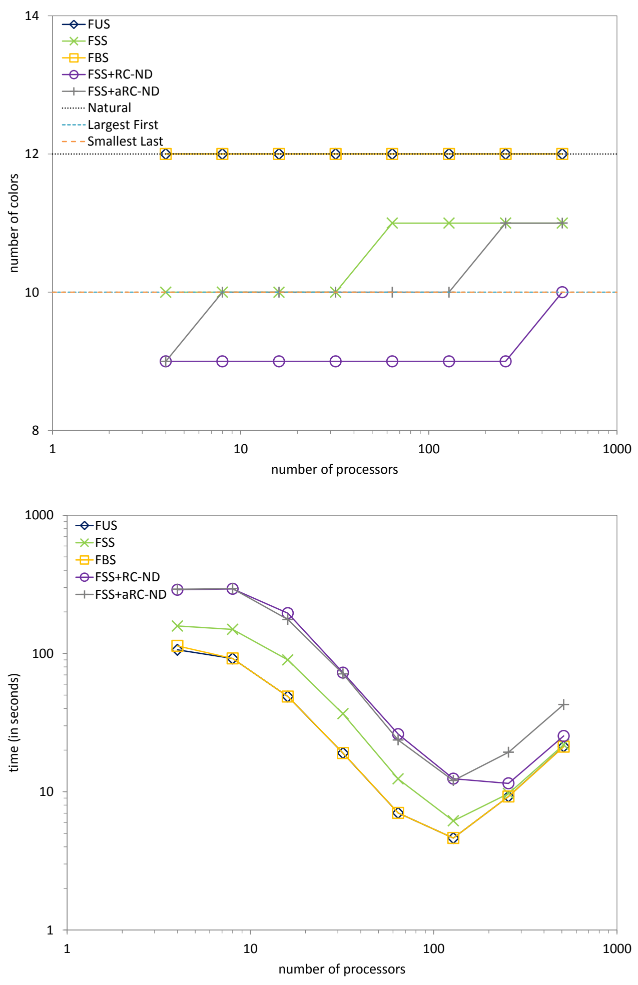


Figure 3. Comparison of ordering and recoloring on RMAT-ER

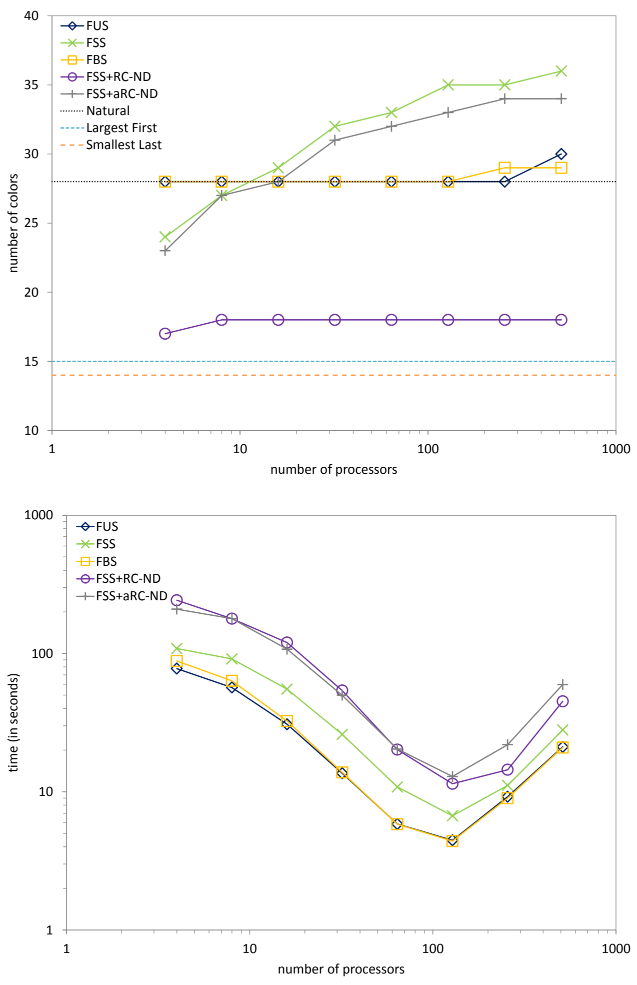


Figure 4. Comparison of ordering and recoloring on RMAT-Good

RMAT-Bad in Figures 3, 4 and 5 respectively. The results of FLS and FIA are omitted from these charts for clarity, but the performance of FLS is very similar to the performance of FSS and the performance of FIA is very similar to the performance of FBS. All the methods scale in the same way on all RMAT graphs up to 64 processors. RMAT-ER and RMAT-Good continue to scale up to 128 processors whereas RMAT-Bad shows an increase after 64 processors. All of the RMAT graphs have increasing runtimes after 128 processors. Just like the real-world graphs, the reason for the increases after 64 processors is the multi-core processor hierarchy and increasing number of boundary vertices.

5.3. Recoloring

Section 5.1 showed that the SL+RC-ND combination outperformed all other vertex-visit ordering and color class permutations presented in the sequential case. We will then focus on comparing the distributed synchronous recoloring (RC) and the asynchronous one (aRC) using the non recolored *Smallest Last* ordering (FSS) as a point of reference.

Figure 6 presents the performance of recoloring on the real-world graphs. As expected, the synchronous recoloring decreases the normalized number of colors significantly allowing to keep the normalized number of colors below the one obtained using the sequential *Largest First* even on 512 processors, bringing a 25% improvement in the number of colors to FSS. However the synchronous recoloring takes more time than the FSS coloring, reaching a normalized runtime of 8.53 while FSS has a normalized runtime of 1.46 on 512 processors. Asynchronous recoloring provides a middle ground by allowing to obtain a better coloring than FSS was able to achieve. Moreover, the asynchronous recoloring is not much different than a regular run of the distributed coloring algorithm and yields the same runtime cost as a second run of the algorithm.

The impact of recoloring is presented on the randomly generated graphs in Figures 3, 4 and 5. First, asynchronous recoloring shows a runtime profile similar to the experiments on the real-world graphs. In terms of number of colors, the

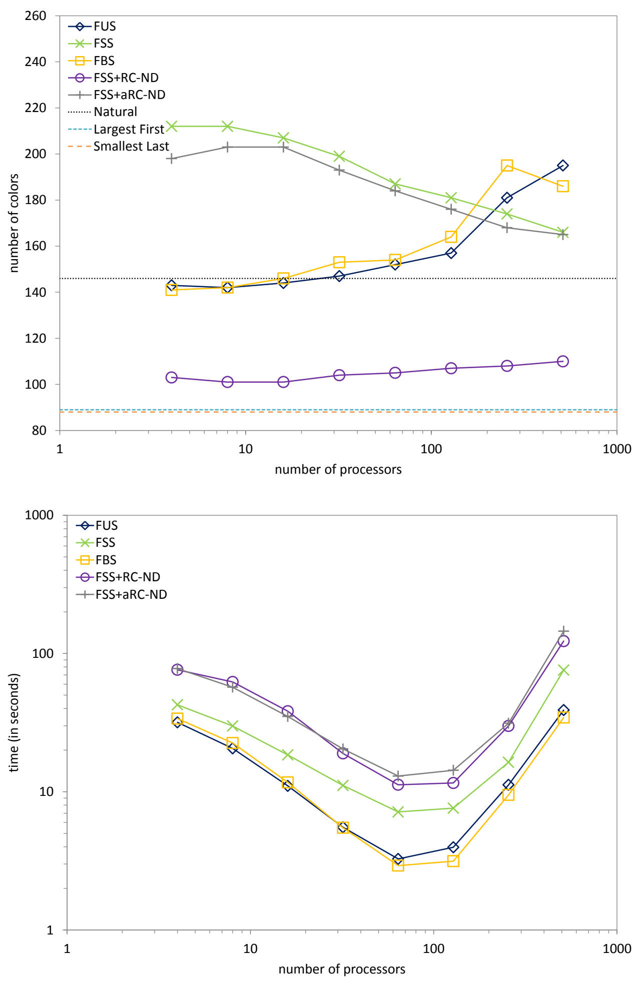


Figure 5. Comparison of ordering and recoloring on RMAT-Bad

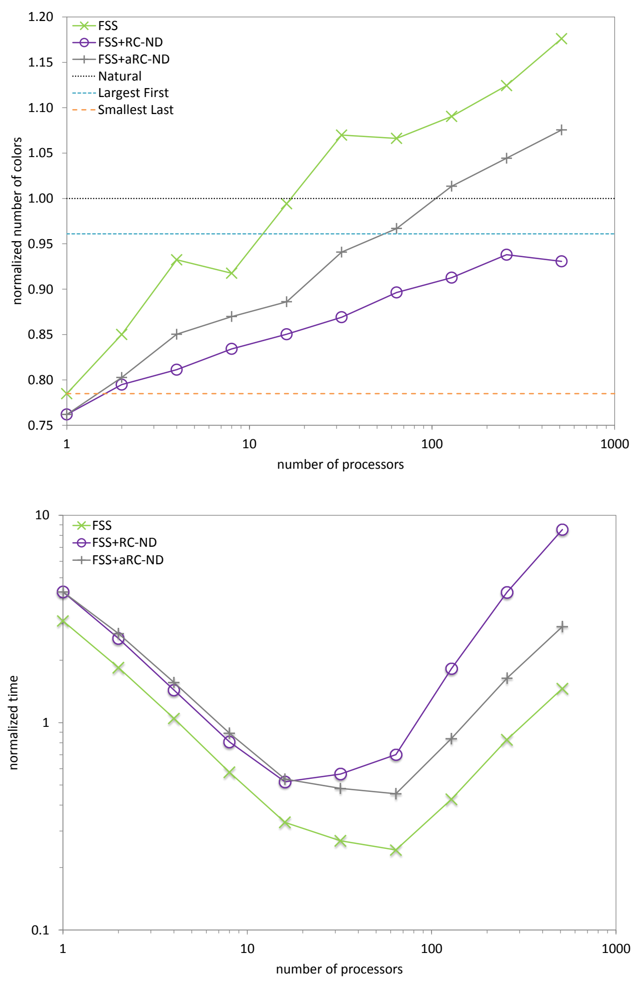


Figure 6. Comparison of recoloring on real-world graphs with Smallest Last ordering

performance of FSS was mainly given by the number of conflicts it generated. The vertex-visit order computed by the asynchronous recoloring does not avoid the majority of these conflicts and the improvement in number of colors compared to FSS is less than 10%.

Synchronous recoloring shows a different picture. While FSS obtained bad number of colors because of a high number of conflicts on RMAT-Good and RMAT-Bad, the synchronous recoloring does not yield any conflict. Therefore, it obtains a much better number of colors, close to the sequential *Largest First* and *Smallest Last* orderings and up to 40% improvement in number of colors compared to FUS and FBS and up to 50% improvement compared to FSS. In terms of runtime, the absence of conflict and the size of the graphs make the synchronous recoloring procedure very scalable in these cases inducing a very low overhead compared to the original coloring when the number of processors is high.

Provided the gain shown by the recoloring procedure, one might be interested in reproducing the sequential improvement on the number of colors obtained by running the recoloring procedure multiple times. The study of the impact of multiple iterations of recoloring in a distributed-memory setting is presented in Figure 7 on the real-world graphs. While a single iteration of recoloring can limit drastically the overhead in terms of number of colors induced by the conflict that appears, subsequent iterations still significantly improve the number of colors. On 512 processors, running 10 iterations of the recoloring procedure allows to reach a normalized number of colors close to the one obtained with the sequential *Smallest Last* vertex-visit order.

5.4. Processor allocation policy

The results of Figure 2 show that the runtime of the algorithms dramatically decrease when more than one processor per machine is used. To study this effect we ran FSS using 4 different processor allocation policies. They allocate respectively, 1, 2, 4 and 8 processors per node of the cluster in a round robin fashion. Which means that when

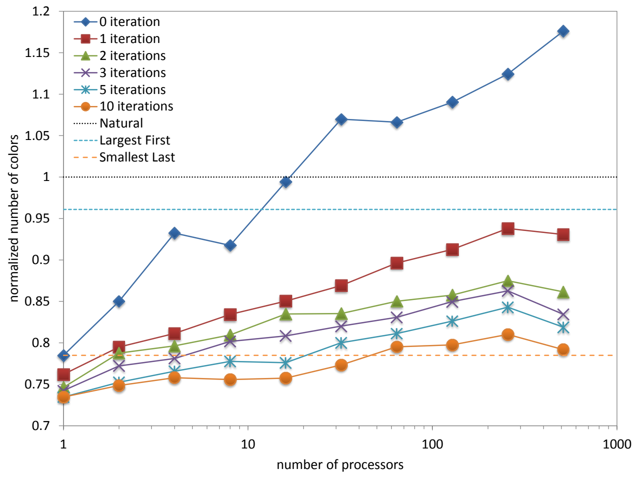


Figure 7. Impact of the number of iterations on real-world graphs in distributed memory

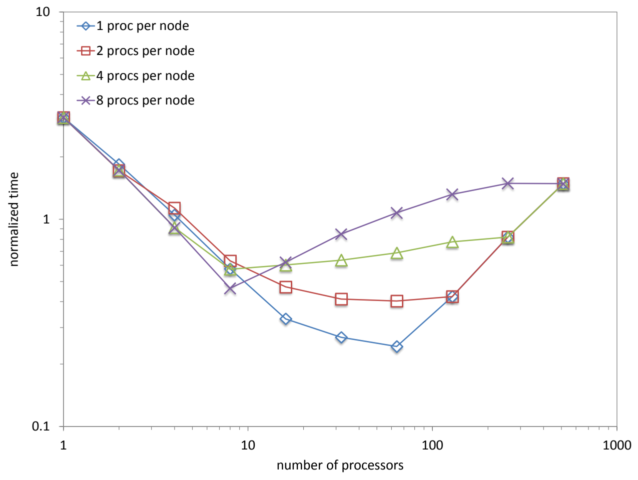


Figure 8. Impact of the processor allocation policy on real-world graphs in distributed memory

allocating 32 processors, the first one uses one processor from 32 nodes. The second one uses 2 processors from 16 nodes. The last one uses 8 processors from 4 nodes. When allocating 256 processors, the first three policies allocate 4 processors from 64 nodes while the last policy allocates 8 processors from 32 nodes.

Figure 8 presents the results of this experiments. It shows that up to 8 processors allocated, the best configuration uses only one node. The performance of allocating the processors by 8 degrades and becomes the worst one when allocating 32 processors or more. Allocating processors one by one obtains the best performance between 16 and 64 processors. On more than 64 processors, the different processor allocation policies start producing the same allocation and obtain the same results.

The two most common explanations for these types of behavior are memory contention and network contention. Memory contention would explain why using fewer nodes on the 32 processors case obtains the worst runtime. But allocating only one node when using 8 processors should not lead to the best results. If it was network contention, one would expect that the first allocation policy which generates the most network traffic to obtain the worst results. There might be a bandwidth problem on one node.

Another explanation would be that when 8 processes access the MPI subsystem on one node, there is contention on the subsystem itself and the performance decreases. When 8 processes access the MPI and network subsystem, some of them can not access the subsystem and got unscheduled by the system. Most likely an hybrid implementation of the algorithm would more intelligently access the MPI and network subsystem. It would lessen the impact of that contention and improve the performance.

6. Conclusion

In this paper we investigated two different, but compatible, ways of improving the number of colors in distributed-memory graph coloring algorithms. We showed that using a vertex-visit ordering that takes into account the properties of the graph instead of the properties of the partition can yield major improvements when the size of the boundary is small, e.g., when the number of processors is small or when the graph is regular. We also investigated recoloring by leveraging the independent sets of vertices exposed by an existing solution to color large number of vertices independently with little synchronization or with little conflicts. We showed that graphs that induce a larger number of conflicts benefit heavily from synchronous recoloring in terms of the number of colors. The runtime overhead being small in such cases, multiple iterations of recoloring can be used to obtain even fewer colors. When the overhead of the synchronous recoloring is too high, asynchronous recoloring can be used to obtain a solution faster but of lesser quality.

Multiple other improvements should be investigated. The tested vertex-visit orderings currently take into account either the properties of the partition or the properties of the graph. Investigating vertex-visit orderings that take both information into account could lead to better number of colors and runtime in a large number of use cases. Recoloring proved to be a scalable concept for the number of colors when number of processors increases, thanks to the equivalence to a sequential procedure. However it currently suffers from a lack of communication and computation overlapping that impacts its scalability in terms of runtime. Then, one can investigate combination of a naive but faster first coloring algorithm together with a scalable recoloring procedure to obtain both a good runtime and number of colors, even when the number of processors increases. Finally, MPI is currently used to perform communication, leading to multiple MPI processes per physical machine.

An hybrid implementation where MPI is used for distributed memory inter-node communication and OpenMP is used for shared memory intra-node communication, should improve the load balance and communication by limiting the number of parts the partitioner generates.

Acknowledgments

This work was partially supported by the U.S. Department of Energy SciDAC Grant DE-FC02-06ER2775 and NSF grants CNS-0643969, OCI-0904809 and OCI-0904802.

References

[1] A. H. Gebremedhin, F. Manne, and A. Pothen, "What color is your jacobian? Graph coloring for computing derivatives," *SIAM Review*, vol. 47, no. 4, pp. 629–705, 2005.

[2] M. Garey, D. Johnson, and H. So, "An application of graph coloring to printed circuit testing," *Circuits and Systems, IEEE Transactions on*, vol. 23, no. 10, pp. 591–599, Oct. 1976.

[3] A. Gamst, "Some lower bounds for a class of frequency assignment problems," *Vehicular Technology, IEEE Transactions on*, vol. 35, no. 1, pp. 8–14, Feb. 1986.

[4] G. J. Chaitin, "Register allocation & spilling via graph coloring," *SIGPLAN Not.*, vol. 17, pp. 98–101, Jun. 1982.

[5] J. Allwright, R. Bordawekar, P. D. Coddington, K. Dincer, and C. Martin, "A comparison of parallel graph coloring algorithms," Northeast Parallel Architectures Center at Syracuse University (NPAC), Tech. Rep. SCCS-666, 1994.

[6] T. F. Coleman and J. J. More, "Estimation of sparse Jacobian matrices and graph coloring problems," *SIAM Journal on Numerical Analysis*, vol. 1, no. 20, pp. 187–209, 1983.

[7] D. W. Matula, "A min-max theorem for graphs with application to graph coloring," *SIAM Review*, vol. 10, pp. 481–482, 1968.

[8] D. Zuckerman, "Linear degree extractors and the inapproximability of max clique and chromatic number," *Theory of Computing*, vol. 3, pp. 103–128, 2007.

[9] D. Bozdag, A. Gebremedhin, F. Manne, E. Boman, and U. Catalyurek, "A framework for scalable greedy coloring on distributed memory parallel computers," *Journal of Parallel and Distributed Computing*, vol. 68, no. 4, pp. 515–535, 2008.

[10] J. A. Ellis and P. M. Lepolesa, "A las vegas graph colouring algorithm," *The Computer Journal*, vol. 32, pp. 474–476, Oct. 1989.

[11] R. D. Dutton and R. C. Brigham, "A new graph colouring algorithm," *The Computer Journal*, vol. 24, no. 1, pp. 85–86, 1981.

[12] A. V. Kosowski and K. Manuszewski, "Classical coloring of graphs," *Graph Colorings*, pp. 1 – 19, 2004.

[13] D. W. Matula, G. Marble, and J. Isaacson, "Graph coloring algorithms," *Graph Theory and Computing*, pp. 109–122, 1972.

[14] J. S. Turner, "Almost all k-colorable graphs are easy to color," *J. Algorithms*, vol. 9, pp. 63–82, Mar. 1988.

[15] J. C. Culberson, "Iterated greedy graph coloring and the difficulty landscape," University of Alberta, Tech. Rep. TR 92-07, Jun. 1992.

[16] D. J. A. Welsh and M. B. Powell, "An upper bound for the chromatic number of a graph and its application to timetabling problems," *The Computer Journal*, vol. 10, no. 1, pp. 85–86, 1967.

[17] J. Hansen, M. Kubale, L. Kuszner, and A. Nadolski, "Distributed largest-first algorithm for graph coloring," in *Euro-Par 2004 Parallel Processing*, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds., 2004, vol. 3149, pp. 804–811.

[18] A. H. Gebremedhin and F. Manne, "Parallel graph coloring algorithms using OpenMP (extended abstract)," in *In First European Workshop on OpenMP*, 1999, pp. 10–18.

[19] M. K. Goldberg, "Edge-coloring of multigraphs: Recoloring technique," *Journal of Graph Theory*, vol. 8, no. 1, pp. 123–137, 1984.

[20] M. Jones and P. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[21] R. K. Gjertsen Jr., M. T. Jones, and P. Plassmann, "Parallel heuristics for improved, balanced graph colorings," *Journal of Parallel and Distributed Computing*, vol. 37, pp. 171–186, 1996.

[22] M. Luby, "A simple parallel algorithm for the maximal independent set problem," *SIAM journal on computing*, vol. 15, no. 4, pp. 1036–1053, 1986.

[23] E. Boman, K. Devine, R. Heaphy, B. Hendrickson, V. Leung, L. A. Riesen, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco, *Zoltan 3.0: Parallel Partitioning, Load Balancing, and Data-Management Services; User's Guide*, Sandia National Laboratories, Albuquerque, NM, 2007, tech. Report SAND2007-4748W.

[24] G. Karypis, K. Schloegel, and V. Kumar, "ParMETIS: Parallel graph partitioning and sparse matrix ordering library, version 3.1," Dept. Computer Science, University of Minnesota, Tech. Rep., 2003.

[25] A. H. Gebremedhin and F. Manne, "Scalable parallel graph coloring algorithms," *Concurrency: Practice and Experience*, vol. 12, pp. 1131–1146, 2000.

[26] M. M. Strout and P. D. Hovland, "Metrics and models for reordering transformations," in *Proceedings of the The Second ACM SIGPLAN Workshop on Memory System Performance (MSP)*, June 8 2004, pp. 23–34.

[27] Y. Z. D. Chakrabarti and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of SIAM Data Mining*, 2004.