Scalable parallel graph coloring algorithms

Assefaw Hadish Gebremedhin[*] Fredrik Manne

Abstract

Finding a good graph coloring quickly is often a crucial phase in the development of efficient, parallel algorithms for many scientific and engineering applications. In this paper we consider the problem of solving the graph coloring problem itself in parallel. We present a simple and fast parallel graph coloring heuristic that is well suited for shared memory programming and yields an almost linear speedup on the PRAM model. We also present a second heuristic that improves on the number of colors used. The heuristics have been implemented using OpenMP. Experiments conducted on an SGI Cray Origin 2000 super computer using very large graphs from finite element methods and eigenvalue computations validate the theoretical run-time analysis.

Key words: graph coloring; parallel algorithms; shared memory programming; OpenMP

[*]Department of Informatics, University of Bergen, N-5020 Bergen, Norway. {assefaw, Fredrik.Manne}@ii.uib.no

1 Introduction

The graph coloring problem (GCP) deals with assigning labels (called colors) to the vertices of a graph such that adjacent vertices do not get the same color. The primary objective is to minimize the number of colors used. The GCP arises in a number of scientific computing and engineering applications. Examples include, among others, time tabling and scheduling [14], frequency assignment [6], register allocation [3], printed circuit testing [8], parallel numerical computation [1], and optimization [4]. Coloring a general graph with the minimum number of colors is known to be an NP-hard problem [7], thus one often relies on heuristics to obtain a usable solution.

In a parallel application a graph coloring is usually performed in order to partition the work associated with the vertices into independent subtasks such that the subtasks can be performed concurrently. Depending on the amount of work associated with each vertex, there are basically two coloring strategies one can pursuit. In the first strategy the emphasis is on *minimizing* the number of colors whereas in the second the focus is on *speed*. Ascertaining which is more appropriate depends on the underlying problem one is trying to solve.

If the task associated with each vertex is computationally expensive then it is crucial to use as few colors as possible. There exist several time-consuming (iterative) local improvement heuristics for addressing this need. Some of these heuristics have been shown to be parallelizable [14].

If, on the other hand, the task associated with each vertex is fairly small and one repeatedly has to find new graph colorings then the overall time to perform the colorings might take up a significant portion of the entire computation. See [16] for an example of this case. In such a setting it is more important to compute a usable coloring fast than spending time on reducing the number of colors. For this purpose there exist several linear time, or close to linear time, sequential greedy coloring heuristics. These heuristics have been found to be effective in coloring graphs that arise from a number of applications [4, 11]. Because of their inherent sequential nature, however, these heuristics are difficult to parallelize.

This paper focuses mainly on the latter strategy where the goal is to develop scalable parallel coloring heuristics based on greedy methods. Previous work on developing such algorithms has been performed on distributed memory computers using explicit message-passing. The speedup obtained from these efforts has been discouraging [1]. The main justification for using these algorithms has been access to more memory and thus the ability to solve problems with very large graphs. It is to be noted that the current availability of shared memory computers where the entire memory can be accessed by any processor makes this argument less significant now.

The development of shared memory computers has been accompanied by the emergence of new shared memory programming paradigms of which OpenMP has become one of the most successful and widely used [17]. OpenMP is a directive-based, fork-join model for shared memory parallelism.

In this paper we present a fast and scalable parallel graph coloring algorithm suitable for the shared memory programming model. In our context, scalability of a parallel algorithm is a measure of its capacity to increase speedup as the number of processors is increased for a given problem size. Our algorithm is based on first performing a parallel pseudo-coloring of the graph. The prefix "pseudo" is used to reflect that the coloring might contain adjacent vertices that are colored with the same color. To remedy this we perform a second parallel step where any inconsistencies in the coloring are detected. These are then resolved in a final sequential step. An analysis on the PRAM model using p processors for a graph with n vertices and m edges shows that the expected number of conflicts from the first stage is low and for $p \leq \frac{n}{\sqrt{2m}}$ the algorithm is expected to provide a nearly linear speedup. We also extend this idea and present a second parallel algorithm that potentially uses fewer colors.

The presented algorithms have been implemented in Fortran90 using OpenMP on a Cray Origin 2000 super computer. Experimental results on a number of very large graphs show that the algorithms yield good speedup and produce colorings of comparable quality to that of their sequential counterparts. The fact that we are using OpenMP makes our implementation significantly simpler and easier to verify than if we had used a distributed memory programming environment such as MPI.

The rest of this paper is organized as follows. In Section 2 we give some background on the graph coloring problem and previous efforts made to solve it in parallel. In Section 3 we describe our new parallel graph coloring algorithms and analyze their performance on the PRAM model. Synchronization overhead and OpenMP issues related to our implementation are discussed in Section 4. In Section 5 we present and discuss results from experiments performed on the Cray Origin 2000. Finally, in Section 6 we give concluding remarks.

2 Background

In this section we give a brief overview of previous work done on the development of fast coloring heuristics, both sequential and parallel. We begin by introducing some graph notation used in this paper.

For a graph $G = (V, E)$, we denote $|V|$ by n, $|E|$ by m, and the degree of a vertex v_i by $deg(v_i)$. Moreover, the maximum, minimum, and average degree in a graph are denoted by Δ, δ, and $\overline{\delta}$ respectively.

As mentioned in Section 1 there exist several fast sequential coloring heuristics that are very effective in practice. These algorithms are all based on the same general greedy framework: a vertex is selected according to some predefined criterion and then colored with the smallest valid color. The selection and coloring continues until all the vertices in the graph are colored.

Some of the suggested coloring heuristics under this general framework include Largest-Degree-First-Ordering (LFO) [18], Incidence-Degree-Ordering (IDO) [4], and Saturation-Degree-Ordering (SDO) [2]. These heuristics choose at each step a vertex v from the set of uncolored vertices with the maximum "degree". In LFO, the standard definition of degree of a vertex is used. In IDO, incidence degree is defined as the number of already colored adjacent vertices, whereas in SDO one only considers the number of differently colored adjacent vertices. First Fit (FF) is yet another, simple variant of the general greedy framework. In FF, the next vertex from some arbitrary ordering is chosen and colored. In terms of quality of coloring, these heuristics can in most cases be ranked in an increasing order as FF, LFO, IDO, and SDO. Note that for a graph G the number of colors used by any sequential greedy algorithm is bounded from above by $\Delta + 1$. On the average, however, it has been shown that for *random* graphs FF is expected to use no more than $2\chi(G)$ colors, where $\chi(G)$ is the chromatic number[1] of G [10]. In terms of run time, FF is clearly $O(m)$, LFO and IDO can be implemented to run in $O(m)$, and SDO in $O(n^2)$ [2, 11].

When it comes to parallel graph coloring, a number of the existing fast heuristics are based on the observation that an independent set of vertices can be colored in parallel. A general parallel coloring scheme based on this observation is outlined in Figure 1.

Depending on how the independent set is chosen and colored, Scheme 1 specializes into a number of variants. The Parallel Maximal Independent set (PMIS) coloring is one variant. This is a heuristic based on Luby's maximal independent set finding algorithm [15]. Other variants are the asynchronous parallel heuristic by Jones and Plassmann (JP) [11], and the Largest-Degree-First(LDF) heuristic developed independently by Gjertsen Jr. et al. [12] and Allwright et al. [1].

[1]The chromatic number of a graph is the optimal number of colors required to color it.

Scheme 1

```
$ParallelColoring(G = (V, E))$
begin
    $U \leftarrow V$
    $G' \leftarrow G$
    while ($G'$ is not empty) do in parallel
        Find an independent set $I$ in $G'$
        Color the vertices in $I$
        $U \leftarrow U \setminus I$
        $G' \leftarrow$ graph induced by $U$
    end-while
end
```

Figure 1: A parallel coloring heuristic

Allwright et al. made an experimental, comparative study by implementing the PMIS, JP, and LDF coloring algorithms on both SIMD and MIMD parallel architectures [1]. They report that they did not get speedup for any of these algorithms.

Jones and Plassmann [11] do not report on obtaining speedup for their algorithms either. They state that "the running time of the heuristic is only a slowly increasing function of the number of processors used".

3 Block Partition Based Coloring Heuristics

In this section we present two new parallel graph coloring heuristics and give their performance analysis on the PRAM model. Our heuristics are based on dividing the vertex set of the graph into p successive blocks of equal size. We call this a *block partitioning.* We assume that the vertices are listed in a random order and thus no effort is made to minimize the number of *crossing* edges. A crossing edge is an edge whose end points end up in two different blocks. Obviously, because of the existence of crossing edges, the coloring subproblems defined by each block are not independent.

3.1 The First Algorithm

The strategy we employ consists of three phases. In the first phase, the input vertex set V of the graph $G = (V, E)$ is partitioned into p blocks as $\{V_1, V_2, \ldots, V_p\}$ such that $|V_i| = n/p$, $1 \leq i \leq p$. The vertices in each block are then colored in parallel using p processors. The parallel coloring comprises of n/p parallel steps with synchronization barriers at the end of each step. When coloring a vertex, *all* its previously colored neighbors, both the local ones and those found on other blocks, are taken into account. In doing so, two processors may simultaneously attempt to color vertices that are adjacent to each other. If these vertices are given the same color, the resulting coloring becomes invalid and hence we call the coloring obtained a *pseudo coloring*. In the second phase, each processor p_i checks whether vertices in V_i are assigned valid colors by comparing the color of a vertex against all its neighbors that were colored at the same parallel step in the first phase. This checking step is also done in parallel. If a conflict is discovered, one of the end points of the edge in conflict is stored in a table. Finally, in the third phase, the vertices stored in this table are colored sequentially. Algorithm 1 provides the details of this strategy and is given in Figure 2.

Algorithm 1

```
$BlockPartitionBasedColoring(G, p)$
begin
  1. Partition $V$ into $p$ equal blocks $V_1 \ldots V_p$, where $\lfloor \frac{n}{p} \rfloor \leq |V_i| \leq \lceil \frac{n}{p} \rceil$
     for $i = 1$ to $p$ do in parallel
       for each $v_j \in V_i$ do
         assign the smallest legal color to vertex $v_j$
         barrier synchronize
       end-for
     end-for
  2. for $i = 1$ to $p$ do in parallel
       for each $v_j \in V_i$ do
         for each neighbor $u$ of $v_j$ that has been colored at the same
         parallel step do
           if $color(v_j) = color(u)$ then
             store min $\{u, v_j\}$ in table $A$
           end-if
         end-for
       end-for
     end-for
  3. Color the vertices in $A$ sequentially
end
```

Figure 2: Block partition based coloring

3.1.1 Analysis

Our analysis is based on the PRAM model. Without loss of generality we assume that n/p, the number of vertices per processor, is an integer. Let the vertices on each processor be numbered from 1 to n/p and the parallel time used for coloring be divided into n/p time slots. The processors are synchronized at the end of each time unit t_j. This means, at each time unit t_j, processor p_i colors vertex $v_j \in V_i$, $1 \le j \le n/p$ and $1 \le i \le p$.

Our first result gives an upper bound on the *expected* number of conflicts (denoted by K) created at the end of Phase 1 of Algorithm 1 for a graph in which the vertices are listed in a randomly permuted order.

Lemma 3.1 *The expected number of conflicts created at the end of Phase 1 of Algorithm 1 is at most* $\frac{\overline{\delta}(p-1)}{2}(\frac{n}{n-1}) \approx \frac{\overline{\delta}(p-1)}{2}$.

Proof: Consider a vertex $x \in V$ that is colored at time unit t_j, $1 \le j \le n/p$. Since the neighbors of x are randomly distributed, the *expected* number of neighbors of x that are concurrently colored at time unit t_j is given by

$$\frac{p-1}{n-1} deg(x) \tag{1}$$

If we sum (1) over all vertices in G we count each pair of adjacent vertices that are colored simultaneously twice. Moreover each term in the sum represents only a potential conflict since two adjacent vertices could be colored simultaneously and yet be assigned different colors. The sum thus gives an upper bound on the expected number of conflicts. Therefore, we have

$$\begin{aligned} E[K] &\le (1/2)\sum_{x\in V} \frac{p-1}{n-1} deg(x) && (2)\\ &= (1/2)\frac{p-1}{n-1}(2m) && (3)\\ &= (1/2)\overline{\delta}(p-1)(n/n-1) && (4)\end{aligned}$$

In going from (3) to (4), the identity $\overline{\delta} = \frac{\sum_{v\in V} deg(v)}{n} = \frac{2m}{n}$ is used. Note that for large values of n, $(n/n-1) \approx 1$ and the result follows.

□

We now look at the expected run time[2] of Algorithm 1. To do so, we introduce a graph attribute called *relative sparsity* r, defined as $r = \frac{n^2}{m}$. Note that $1/r$, the ratio of the actual number of edges to the total possible number of edges, shows the density of the graph. The following lemma states

[2]We use the prefix E to identify expected time complexity expressions.

that for bounded degree graphs and for $p \leq \sqrt{\frac{r}{2}}$, Algorithm 1 provides an almost linear speedup compared to the sequential First Fit algorithm.

Lemma 3.2 *On a CREW PRAM, Algorithm 1 colors the input graph consistently in $EO(\Delta n/p)$ time when $p \leq \sqrt{\frac{r}{2}}$ and in $EO(\Delta \overline{\delta} p)$ time when $p > \sqrt{\frac{r}{2}}$.*

Proof: Note first that since Phase 3 resolves all the conflicts that are discovered in Phase 2, the coloring at the end of Phase 3 is a valid one. Both Phase 1 and 2 require concurrent read capability and thus the required PRAM is CREW. The overall time required by Algorithm 1 is $T = T_1 + T_2 + T_3$, where T_i is the time required by Phase i. Both Phase 1 and 2 consist of n/p parallel steps. The number of operations in each parallel step is proportional to the degree of the vertex under investigation which is bounded from above by Δ. Thus, $T_1 = T_2 = O(\Delta n/p)$. The time required by the sequential step (Phase 3) is $T_3 = O(\Delta K)$ where K is the number of conflicts discovered in Phase 2. From Lemma 3.1, $E[K] = O(\overline{\delta} p)$. Substituting yields

$$T = T_1 + T_2 + T_3 = EO(\Delta(n/p + \overline{\delta} p)) \tag{5}$$

The overall time T is thus determined by how n/p compares with $\overline{\delta} p$. Using the identity $\overline{\delta} = \frac{2m}{n}$, we see that for $p \leq \sqrt{\frac{n^2}{2m}} = \sqrt{\frac{r}{2}}$, the term n/p dominates giving an overall running time of $EO(\Delta n/p)$. For $p > \sqrt{\frac{r}{2}}$, the term $\overline{\delta} p$ dominates and the overall time becomes $EO(\Delta \overline{\delta} p)$.

□

For most practical applications and currently available parallel computers we expect that both $\overline{\delta} \ll n$ and $p \ll n$ implying that $n > p^2 \overline{\delta}$ and thus giving an overall time complexity of $O(\Delta n/p)$ for Algorithm 1.

The number of colors used by Algorithm 1 is bounded from above by $\Delta + 1$. This follows since the conflicts that arise in Phase 1 are resolved sequentially. However, we note that there exist instances where the coloring produced by Algorithm 1 can be arbitrarily worse than that of the sequential FF algorithm. To see this, consider a complete bipartite graph $G = (V_1, V_2, E)$ with $|V_1| = |V_2| = n/2$, where the vertices in V_1 are ordered before the vertices in V_2 and with $p = 2$. For this setting Algorithm 1 will use $\frac{n}{2} + 1$ colors while sequential FF will color the graph optimally using 2 colors.

3.2 The Second Algorithm

In this section we show how Algorithm 1 can be modified to use fewer colors. Our method is motivated by the idea behind Culberson's Iterated Greedy coloring heuristic (IG) [5]. IG is based on the following result, stated here without proof.

Lemma 3.3 (Culberson) *Let C be a k-coloring of a graph G, and π a permutation of the vertices such that if $C(v_{\pi(i)}) = C(v_{\pi(l)}) = c$, then $C(v_{\pi(j)}) = c$ for $i < j < l$. Then, applying the First Fit algorithm to G where the vertices have been ordered by π will produce a coloring using k or fewer colors.*

From Lemma 3.3, we see that if FF is re-applied on a graph where the vertex set is ordered such that vertices belonging to the same color class[3] in the previous coloring are listed consecutively, the new coloring is better or at least as good as the previous coloring. There are many ways in which the vertices of a graph can be arranged satisfying the condition of Lemma 3.3. One such ordering is the reverse color class ordering [5]. In this ordering, the color classes are listed in reverse order of their introduction. This has a potential for reducing the number of colors used since one now proceeds by first coloring vertices that could not be colored with low values in the previous coloring.

Our improved coloring heuristic uses Lemma 3.3 and consists of 4 phases, one more phase than Algorithm 1. The first phase is the same as Phase 1 of Algorithm 1. Let the coloring number used by this phase be $ColNum$. During the second phase, the pseudo coloring of the first phase is used to get a reverse color class ordering of the vertices. The second phase consists of $ColNum$ steps. In each step k, the vertices of color class $ColNum - k - 1$ are colored *afresh* in parallel in a similar manner as in Phase 1. The remaining two phases are the same as Phases 2 and 3 of Algorithm 1. The method just described (Algorithm 2) is outlined in Figure 3.

Each color class at the end of Phase 1 is a pseudo independent set. In particular any edge within a color class results from a "conflict" edge from Phase 1. Hence a new block partitioning of the vertices of each color class results in only a few crossing edges. In other words, the number of conflicts expected at the end of Phase 2 (K_2) should be much smaller than the number of conflicts at the end of Phase 1 (K_1). Thus, in addition to improving the quality of the coloring, Phase 2 should also provide a significant reduction in the number of conflicts. Note that conflict checking and removal steps

[3]Vertices of the same color constitute a color class.

Algorithm 2

```
$ImprovedBlockPartitionBasedColoring(G, p)$
begin
 1. As Phase 1 of Algorithm 1
    {At this point we have the pseudo independent
    sets ColorClass(1) ... ColorClass(ColNum) }
 2. for $k = ColNum$ down to 1 do
        Partition $ColorClass(k)$ into $p$ equal blocks $V_1' \dots V_p'$
        for $i = 1$ to $p$ do in parallel
            for each $v_j \in V_i'$ do
                assign the smallest legal color to vertex $v_j$
            end-for
        end-for
    end-for
 3. As Phase 2 of Algorithm 1
 4. As Phase 3 of Algorithm 1
end
```

Figure 3: Modified block partition based coloring

are included in Phases 3 and 4 to ensure that any remaining conflicts are resolved.

The following result gives a bound on the expected number of conflicts at the end of Phase 2 of Algorithm 2.

Lemma 3.4 *The expected number of conflicts created at the end of Phase 2 of Algorithm 2 is at most* $\frac{2p^2\Delta(\Delta+1)}{n} \approx \frac{2p^2\Delta^2}{n}$.

Proof: From Lemma 3.1, the expected number of conflicts at the end of Phase 1 is approximately bounded by $\overline{\delta}p/2$. Noting that there are m edges in the input graph G to Algorithm 2, at the end of Phase 1, the probability that an arbitrary edge in G is in "conflict" is expected to be no more than $\frac{\overline{\delta}p}{2m} = p/n$. Now consider a color class w from the coloring obtained at the end of Phase 1 of Algorithm 2. Let $G' = (V', E')$ be the graph induced by the vertices of this color class and let $n' = |V'|, m' = |E'|$. Further, let x be a vertex in G' and $deg'(x)$ its degree. From the above discussion, we expect that $deg'(x) \leq \frac{p}{n}deg(x)$. Using the same argument as in Lemma 3.1, the expected number of neighbors of x that are concurrently colored at time unit t_j, for $1 \leq j \leq n'/p$, is $\frac{p-1}{n'-1}deg'(x)$. Thus the number of conflicts created due to the vertices of color class w (denoted by K') is bounded as follows.

$$\begin{aligned} E[K'] &\leq \sum_{x \in V'} \frac{p(p-1)}{n(n'-1)} deg(x) & (6) \\ &= \frac{p}{n}(p-1)\frac{\sum_{x \in V'} deg(x)}{n'-1} & (7) \\ &\leq \frac{2p^2\Delta}{n} & (8) \end{aligned}$$

Recall that there are at most $\Delta + 1$ colors at the end of Phase 1. Therefore, K_2, the total number of expected conflicts at the end of Phase 2 , is

$$E[K_2] \leq \frac{2p^2\Delta(\Delta+1)}{n} \qquad (9)$$

$\square$

Noting that $\Delta(\Delta+1) \approx \Delta^2$, (9) can be rewritten as $E[K_2] \leq (\frac{\sqrt{2}p}{\sqrt{n}/\Delta})^2$. This indicates that if $\sqrt{2}p < \sqrt{n}/\Delta$, the expected number of conflicts at the end of Phase 2 is less than 1.

4 Implementation Issues

In this section we address the problem of synchronization overhead and illustrate how OpenMP is used in our Fortran90 implementations.

4.1 Synchronization Overhead

The barrier synchronization in Phase 1 of Algorithm 1 is introduced to identify the parallel step t_j $(1 \leq j \leq n/p)$ during which a vertex is colored. This information is used for two purposes: (i) in Phase 1 to identify already colored neighbors of a vertex, and (ii) in Phase 2 to identify the neighbors of a vertex that are colored at the same parallel step as itself. Although the barrier enables us to realize these purposes, its implementation typically incurs an undesirable large overhead. To overcome this we have developed an asynchronous version of Algorithm 1 (and consequently of Algorithm 2). In the asynchronous version we consider all the neighbors of a vertex under investigation, irrespective of the parallel step during which they are colored. This is done first when determining the color of a vertex and then when checking for consistency of coloring. We have implemented and tested both the asynchronous and synchronous versions of Algorithm 1. In the synchronous version, an OpenMP library routine was utilized to realize barrier synchronization. The obtained results show that the asynchronous version runs faster by a factor of 3 to 5. The relative slow-down

```
!$omp parallel do schedule(static, Bsize)
private(i) shared(vertex)
do i = 1, number_of_vertices
  call assign_color_synch(vertex(i))
  call mp_barrier
enddo
```

```
!$omp parallel do schedule(static, Bsize)
private(i) shared(vertex)
do i = 1, number_of_vertices
  call assign_color_asynch(vertex(i))
enddo
```

Figure 4: OpenMP-sketch of Phase 1 of Algorithm 1, synchronous (left) and asynchronous (right)

factor of the synchronous version depends on how often the OpenMP barrier routine is called. Particularly for a given graph, the relative time spent on synchronization increases with the number of processors.

4.2 OpenMP

Figure 4 provides a sketch of Phase 1 of both the synchronous and asynchronous versions of Algorithm 1. Here the vertices are stored in the integer array *vertex* and the number of vertices per processor is stored in the variable *Bsize*. The routines *assign_color_synch(i)* and *assign_color_asynch(i)*, synchronous and asynchronous versions respectively, assign the smallest valid color to vertex *i*. It should be noted that even though the synchronous algorithm visits fewer vertices both when determining the color of a vertex and when checking for consistency, it incurs an extra initial overhead since one must determine for each of value of p which vertices to check. The routine *mp_barrier* is an OpenMP library routine that enables barrier synchronization.

In addition to the standard OpenMP directives we have used data distribution directives provided by SGI to ensure that most cache misses are satisfied from local memory.

We disallow access to a memory location while it is being written by using the ATOMIC directive in OpenMP. This makes accessing the color of any vertex without reading garbage values possible in Phase 1.

5 Experimental Results

In this section, we experimentally demonstrate the performance of the asynchronous versions of Algorithms 1 and 2. In Section 5.1 we introduce the test graphs used in the experiments and in Section 5.2 we present and discuss the experimental results. The experiments have been performed on a Cray Origin 2000, a CC-NUMA machine consisting of 128 MIPS R10000 processors. The algorithms have been implemented in Fortran90 and parallelized

Set	Problem	n	m	Δ	δ	$\overline{\delta}$	$\sqrt{\frac{r}{2}}$	χ_{FF}	χ_{IDO}
Set I	mrng2	1,017,253	2,015,714	4	2	4	506	5	5
Set I	mrng3	4,039,160	8,016,848	4	2	4	1008	5	5
Set II	598a	110,971	741,934	26	5	13	91	11	9
Set II	m14b	214,765	1,679,018	40	4	16	117	13	10
Set III	dense1	19,703	3,048,477	504	116	309	8	122	122
Set III	dense2	218,849	121,118,458	1,640	332	1,107	14	377	376

Table 1: Test Graphs.

using OpenMP[17]. We have also implemented the sequential versions of FF and IDO to use as benchmarks.

In these experiments, the block partitioning is based on the ordering of the vertices as provided in the input graph. In other words, no random permutation is done on the ordering of the vertices prior to partitioning.

5.1 Test Graphs

The test graphs used in our experiments are divided into three categories as Problem Set I, II, and III (see Table 1). Problem Sets I and II consist of graphs (matrices) that arise from finite element methods [13]. Problem Set III consists of matrices that arise in eigenvalue computations [16]. In addition to providing some statistics about the structure of the test graphs, Table 1 also lists the number of colors required when coloring the graphs using our sequential FF and IDO implementations (shown under columns χ_{FF} and χ_{IDO}, respectively).

5.2 Discussion

Algorithm 1. Table 2 lists results obtained using the asynchronous version of Algorithm 1. The number of blocks (processors) is given in column p. Columns χ_1 and χ_3 give the number of colors used at the end of Phases 1 and 3, respectively. The number of conflicts that arise in Phase 1 are listed under the column labeled K. The column labeled $\frac{\overline{\delta}(p-1)}{2}$ gives the theoretically expected upper bound on the number of conflicts as predicted by Lemma 3.1. The time in milliseconds required by the different phases are listed under T_1, T_2, T_3, and the last column T_{tot} gives the total time used. The column labeled S_{par} lists the speedup obtained compared to the time used by running Algorithm 1 on one processor ($S_{par}(p) = \frac{T_{tot}(1)}{T_{tot}(p)}$). The last column, S_{seqFF}, gives the speedup obtained by comparing against a straight forward sequential FF algorithm ($S_{seqFF}(p) = \frac{T_1(1)}{T_{tot}(p)}$).

The results in column K of Table 2 show that, in general, the number of conflicts that arise in Phase 1 is small and grows as a function of the

Problem	p	χ_1	χ_3	K	$\lceil\frac{\overline{\delta}(p-1)}{2}\rceil$	T_1	T_2	T_3	T_{tot}	S_{par}	S_{seqFF}
mrng2	1	5	5	0	0	1190	1010	0	2200	1	0.6
mrng2	2	5	5	0	2	1130	970	0	2100	1.1	0.6
mrng2	4	5	5	0	5	430	280	0	710	3.1	1.7
mrng2	8	5	5	8	11	260	200	0	460	4.8	2.6
mrng2	12	5	5	18	17	200	130	0	330	6.7	3.6
mrng3	1	5	5	0	0	4400	3400	0	7800	1	0.6
mrng3	2	5	5	2	2	2250	1600	0	3850	2	1.1
mrng3	4	5	5	4	5	1300	1000	0	2300	3.4	1.9
mrng3	8	5	5	0	11	630	800	0	1430	5.5	3.1
mrng3	12	5	5	12	17	430	480	0	910	8.6	4.8
598a	1	11	11	0	0	100	80	0	180	1	0.6
598a	2	12	12	4	7	55	40	0	95	2	1.1
598a	4	12	12	12	20	40	20	0	60	3	1.7
598a	8	12	12	36	46	28	15	0	43	4.2	2.3
598a	12	12	12	42	72	20	15	0	35	5.2	2.9
m14b	1	13	13	0	0	200	180	0	380	1	0.5
m14b	2	13	13	2	8	130	120	0	250	1.5	0.8
m14b	4	14	14	14	23	80	50	0	130	3	1.5
m14b	8	13	13	16	53	48	26	0	74	5	2.7
m14b	12	13	13	36	83	40	20	0	60	6.4	3.3
dense1	1	122	122	0	0	200	290	0	490	1	0.4
dense1	2	142	142	30	155	110	140	0	250	2	0.8
dense1	4	137	137	94	464	69	72	0	141	3.5	1.4
dense1	8	129	129	94	1082	53	44	1	97	5.6	2.1
dense1	12	121	124	78	1700	55	90	1	145	3.4	1.4
dense2	1	377	377	0	0	9200	13200	0	22400	1	0.4
dense2	2	382	382	68	553	5160	8040	3	13203	1.7	0.7
dense2	4	400	400	98	1659	2600	4080	4	6684	3.4	1.4
dense2	8	407	407	254	3871	1590	2280	11	3881	5.8	2.4
dense2	12	399	399	210	6083	1090	1420	8	2518	9	3.7

Table 2: Experimental results for Algorithm 1.

number of blocks (or processors) p. This agrees well with the result from Lemma 3.1. We see that for the relatively dense graphs the actual number of conflicts is much less than the bound given by Lemma 3.1.

The run times obtained show that Algorithm 1 performs as predicted by Lemma 3.2. Particularly, the time required for re-coloring incorrectly colored vertices is observed to be practically zero (in the order of a few microseconds) for all our test graphs. This is not surprising as the obtained value of K is negligible compared to the number of vertices in a given graph.

As results in columns T_1 and T_2 indicate, the time used to detect conflicts is approximately the same as the time used to do the initial coloring. This makes the running time of the algorithm using one processor approximately double that of the sequential FF. This in turn reduces the speedup obtained compared to the sequential FF by a factor of 2. The speedup obtained compared to running the parallel algorithm on one processor gets its best

Problem	p	χ_1	χ_2	χ_4	K_1	K_2	T_1	T_2	T_3	T_4	T_{tot}	S_{par}	S_{2seqFF}
mrng2	1	5	5	5	0	0	1050	1700	820	0	3570	1	0.8
mrng2	2	5	5	5	0	0	950	1350	650	0	2650	1.4	1.0
mrng2	4	5	5	5	2	0	470	840	310	0	1620	2.2	1.7
mrng2	8	5	5	5	16	0	300	500	200	0	1000	3.6	2.8
mrng2	12	5	5	5	12	0	250	400	170	0	820	4.4	3.4
mrng3	1	5	5	5	0	0	3700	9500	2600	0	15800	1	0.8
mrng3	2	5	5	5	0	0	1890	4100	1200	0	7190	2.2	1.8
mrng3	4	5	5	5	0	0	1100	2700	750	0	4550	3.5	2.9
mrng3	8	5	5	5	4	0	540	1800	450	0	2790	5.6	4.7
mrng3	12	5	5	5	24	0	450	1900	300	0	2650	6	5.0
598a	1	11	10	10	0	0	100	200	75	0	375	1	0.8
598a	2	12	10	10	14	0	65	105	37	0	207	1.8	1.5
598a	4	11	10	10	22	0	35	90	20	0	145	2.6	2.1
598a	8	12	11	11	40	0	30	99	25	0	154	2.4	2.0
598a	12	12	11	11	50	0	30	110	15	0	155	2.4	2.0
m14b	1	13	11	11	0	0	200	520	190	0	910	1	0.8
m14b	2	13	12	12	2	0	105	240	80	0	425	2.1	1.7
m14b	4	14	12	12	6	0	70	160	40	0	270	3.4	2.7
m14b	8	13	12	12	12	0	45	120	25	0	190	4.8	3.8
m14b	12	13	11	11	22	0	53	150	20	0	223	4	3.2
dense1	1	122	122	122	0	0	180	250	180	0	610	1	0.7
dense1	2	135	122	122	26	0	100	180	140	0	420	1.5	1.0
dense1	4	132	122	122	40	0	80	100	70	0	250	2.5	1.7
dense1	8	126	122	122	104	0	70	80	30	0	180	3.4	2.4
dense1	12	123	121	122	150	2	40	760	30	0	830	0.7	0.5
dense2	1	377	376	376	0	0	9920	13700	7500	0	31120	1	0.8
dense2	2	376	376	376	66	0	5200	6220	4200	0	15620	2	1.5
dense2	4	394	376	376	112	0	2700	3600	2100	0	8400	3.7	2.8
dense2	8	398	376	376	164	0	2000	2000	1800	0	5800	5.4	4.0
dense2	12	399	376	376	232	2	1100	1700	900	0	3700	8.4	6.4

Table 3: Experimental results for Algorithm 2.

values for the two largest graphs mrng3 and dense2.

We see that the number of colors used by Algorithm 1 varies with the number of processors used. Comparing column χ_3 of Table 2 and column χ_{FF} of Table 1, we see that the deviation of χ_3 from χ_{FF} is at most 1 for graphs from Problem Set I and II and at most 16% for the two graphs from Problem Set III.

Algorithm 2. Table 3 lists results of the asynchronous version of Algorithm 2. The number of colors used at the end of Phases 1 and 2 are listed in columns χ_1 and χ_2, respectively. The coloring at the end of Phase 2 is not guaranteed to be conflict-free. Phases 3 and 4 detect and resolve any remaining conflicts. Column χ_4 lists the number of colors used at the end of Phase 4. The number of conflicts at the end of Phases 1 and 2 are listed under K_1 and K_2, respectively. The time elapsed (in milliseconds) in the

various stages are given in columns T_1, T_2, T_3, T_4, and T_{tot}. In order not to obscure speedup results, the time required to build the color classes prior to Phase 2 is not included in T_2. In general the time used for this purpose is in the order of 20% of T_1. Speedup values in column S_{par} are calculated as in the corresponding column of Table 2. The column S_{2seqFF} gives speedups as compared to Culberson's IG restricted only to two iterations ($S_{2seqFF} = \frac{T_1(1)+T_2(1)}{T_{tot}(p)}$).

Results in column χ_2 confirm that Phase 2 of Algorithm 2 reduces the number of colors used by Phase 1. This is especially true for test graphs from Problem Sets II and III, which contain relatively denser graphs than Problem Set I. Comparing the results in column χ_2 with the results in columns χ_{FF} and χ_{IDO} of Table 1, we see that in general the quality of the coloring obtained using Algorithm 2 is never worse than that of sequential FF and in most cases comparable with that of the IDO algorithm. IDO is known to be one of the most effective coloring heuristics [4]. We also observe that, unlike Algorithm 1, the number of colors used by Algorithm 2 remains reasonably stable as the number of processors is increased.

From column K_2 we see that the number of conflicts that remain after Phase 2 of Algorithm 2 is zero for almost all test graphs and values of p. The only occasion where we obtained a value other than zero for K_2 was using $p = 12$ for the graphs dense1 and dense2. These results agree well with the claim in Lemma 3.4.

Figure 5 shows the speedup obtained for the problem dense2 using Algorithm 1 and 2 and how the obtained results compare with the ideal speedups.

6 Conclusion

We have presented two new parallel graph coloring heuristics suitable for shared memory programming and analyzed their performance using the PRAM model.

The heuristics have been implemented using OpenMP and experiments conducted on an SGI Cray Origin 2000 super computer using very large graphs validate the theoretical analysis.

The first heuristic is fast, simple and yields reasonably good speedup for graphs of practical interest run on a realistic number of processors. Generally, the number of colors used by this heuristic never exceeds $\Delta + 1$. For relatively dense graphs, the number of colors used by the heuristic increases slightly as more processors are applied.

The second heuristic is relatively slower, yields reasonable speedup and improves on the quality of coloring obtained from the first one in that it uses

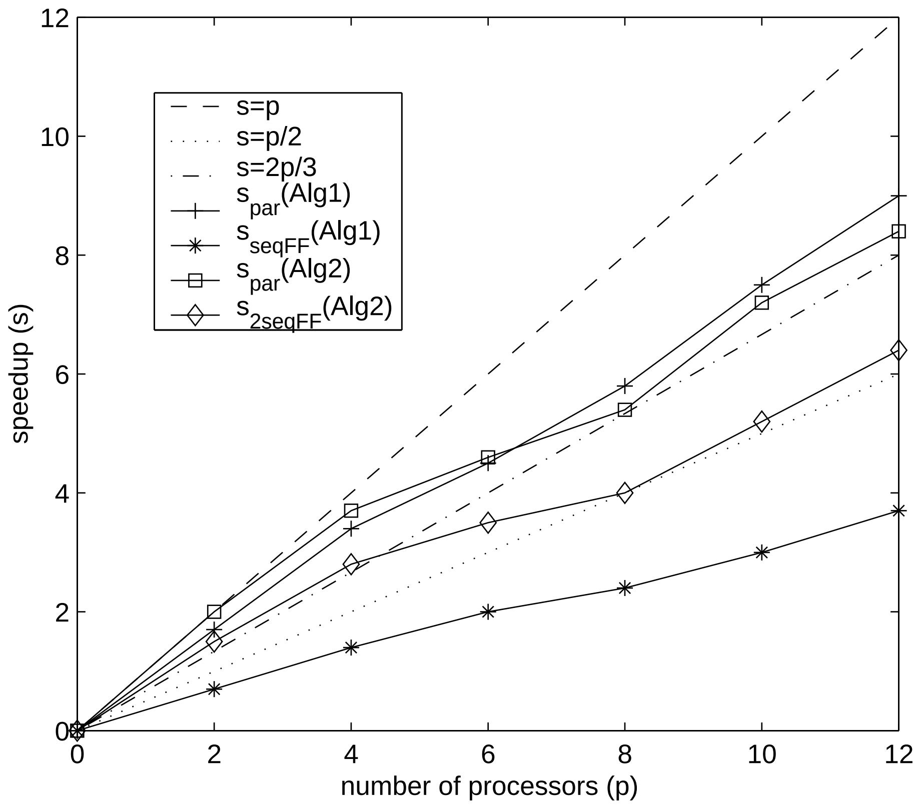

Figure 5: Comparison of speedup curves for dense2.

fewer colors. The number of colors used is also more stable as the number of processors is increased. For the test graphs used in this experiment, the number of colors used by this heuristic is in most cases comparable with that of sequential IDO.

One of the main arguments against using OpenMP has been that it does not give as good speedup as a more dedicated message passing implementation using MPI. The results in this paper show an example where the opposite is true, the OpenMP algorithms have better speedup than existing message passing based algorithms. Moreover, implementing the presented algorithms in a message passing environment would have required a considerable effort and it is not clear if this would have led to efficient algorithms. Implementing these algorithms using OpenMP is a relatively straight forward task as all the communication is hidden from the programmer.

We point out that in a recent development the algorithms presented in this paper have been adapted to the CGM model [9].

We believe that the general idea in these coloring heuristics of allowing inconsistency for the sake of concurrency can be applied to develop parallel algorithms for other graph problems and we are currently investigating this in problems related to sparse matrix computations.

Acknowledgements We thank the referees for their helpful comments and George Karypis for making the test matrices in Problem Sets I and II available.

References

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

[2] D. Brelaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4), 1979.

[3] G.J. Chaitin, M. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[4] T.F. Coleman and J.J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20(1):187–209, 1983.

[5] J.C. Culberson. Iterated greedy graph coloring and the difficulty landscape. Technical Report TR 92-07, Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada, June 1992.

[6] A. Gamst. Some lower bounds for a class of frequency assignment problems. *IEEE Transactions of Vehicular Technology*, 35(1):8–14, 1986.

[7] M.R. Garey and D.S. Johnson. *Computers and Intractability.* W.H. Freeman, New York, 1979.

[8] M.R. Garey, D.S. Johnson, and H.C. So. An application of graph coloring to printed circuit testing. *IEEE Transactions on Circuits and Systems*, 23:591–599, 1976.

[9] A.H. Gebremedhin, I.G. Lassous, J. Gustedt, and J.A. Telle. Graph coloring on a coarse grained multiprocessor. To be presented at WG 2000, 26th International Workshop on Graph-Theoretic Concepts in Computer Science, June 15–17, 2000, Konstantz, Germany.

[10] G.R. Grimmet and C.J.H. McDiarmid. On coloring random graphs. *Mathematical Proceedings of the Cambridge Philosophical Society*, 77:313–324, 1975.

[11] M.T. Jones and P.E. Plassmann. A parallel graph coloring heuristic. *SIAM Journal of Scientific Computing*, 14(3):654–669, May 1993.

[12] R.K. Gjertsen Jr., M.T. Jones, and P. Plassman. Parallel heuristics for improved, balanced graph colorings. *Journal of Parallel and Distributed Computing.*, 37:171–186, 1996.

[13] G. Karypis. Private Communication.

[14] G. Lewandowski. *Practical Implementations and Applications Of Graph Coloring*. PhD thesis, University of Wisconsin-Madison, August 1994.

[15] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[16] F. Manne. A parallel algorithm for computing the extremal eigenvalues of very large sparse matrices (extended abstract). *Lecture Notes in Computer Science, Springer*, 1541:332–336, 1998.

[17] OpenMP. A proposed industry standard API for shared memory programming. *http://www.openmp.org/*.

[18] D.J.A. Welsh and M.B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *Computer Journal*, 10:85–86, 1967.