A Scalable Parallel Graph Coloring Algorithm for Distributed Memory Computers

Erik G. Boman[1], Doruk Bozdağ[2], Umit Catalyurek[2,⋆], Assefaw H. Gebremedhin[3,⋆⋆], and Fredrik Manne[4]

[1] Sandia[⋆⋆⋆] National Laboratories, USA
egboman@sandia.gov
[2] Ohio State University, USA
bozdagd@ece.osu.edu, umit@bmi.osu.edu
[3] Old Dominion University, USA
assefaw@cs.odu.edu
[4] University of Bergen, Norway
Fredrik.Manne@ii.uib.no

Abstract. In large-scale parallel applications a graph coloring is often carried out to schedule computational tasks. In this paper, we describe a new distributed-memory algorithm for doing the coloring itself in parallel. The algorithm operates in an iterative fashion; in each round vertices are speculatively colored based on limited information, and then a set of incorrectly colored vertices, to be recolored in the next round, is identified. Parallel speedup is achieved in part by reducing the frequency of communication among processors. Experimental results on a PC cluster using up to 16 processors show that the algorithm is scalable.

1 Introduction

In many parallel scientific computing applications computational dependencies are modeled using a graph, and a coloring of the vertices of the graph is used as a subroutine to identify independent tasks that can be performed concurrently. See [8] and the references therein for examples. In such cases, the computational graph is often distributed among the processors, and hence the coloring itself needs to be performed in parallel. For these applications, fast greedy coloring algorithms that work well in practice are often preferred over slower local improvement heuristics that might use fewer colors.

This paper deals with the parallelization of such fast greedy coloring algorithms and presents an efficient parallel coloring algorithmic scheme designed for distributed memory parallel computers. Several variations of the basic scheme are discussed. Our algorithms are implemented using MPI and experiments conducted on a 16-node PC cluster using several large graphs indicate that our approach is scalable.

⋆ This research was supported in part by Sandia National Laboratories under Doc.No: 283793, Ohio Supercomputing Center #PAS0052.

⋆⋆ Supported by the U.S. National Science Foundation grant ACI 0203722.

⋆⋆⋆ Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 241–251, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The basic idea in the algorithm is to partition the graph among the available processors and let each processor be responsible for the coloring of the vertices assigned to it. Every processor colors its local vertices in steps of s vertices at a time in a sequential fashion. Between each step the processors exchange recent color information. Since a processor colors its local vertices with incomplete color information, conflicts may arise, and these are detected in a separate phase. The algorithm proceeds iteratively by recoloring vertices involved in conflicts. With an appropriate choice of a value for s, the number of ensuing conflicts can be kept low while at the same time preventing the runtime from being dominated by the sending of a large number of small messages.

2 Previous Work

A *coloring* of a graph is an assignment of positive integers (called colors) to its vertices such that no two adjacent vertices receive the same color. Finding a coloring of a general graph that minimizes the number of colors used is an NP-hard problem [6]. Moreover, the problem is difficult to approximate [4]. In practice, however, greedy sequential coloring heuristics have been found to be quite effective [3]. These greedy heuristics are inherently sequential and hence difficult to parallelize.

A number of previously suggested parallel graph coloring algorithms rely on various ways of computing an *independent set* in parallel. A characteristic feature of independent set based parallel coloring algorithms is that a vertex is assigned a color that is never changed at a later point in the algorithm. In such algorithms, while coloring a vertex v, the colors of already colored neighbors of v must be known, and none of the uncolored neighbors of v can be colored at the same time as v. The works of Jones and Plassmann [11], Gjertsen et al. [9], and Allwright et al. [1] are examples of such approaches. All of these algorithms are designed for distributed memory parallel computers and rely on partitioning a graph into the same number of components as there are processors. Each component, including information about its inter- and intra-component edges, is assigned to and colored by one processor.

To overcome the restriction that two adjacent vertices on different processors cannot be colored at the same time, Johansson [10] proposed a distributed algorithm where each processor is assigned exactly one vertex. The vertices are then colored simultaneously by randomly choosing a color from the interval $[1, \Delta + 1]$, where Δ is the maximum vertex degree in the graph. This may lead to an inconsistent coloring, and hence the process needs to be repeated recursively for the vertices that did not receive permissible colors. Finocchi et al. [5] performed extensive sequential simulations of a variant of Johansson's algorithm where the upper-bound on the range of permissible colors is initially set to be smaller than $\Delta + 1$ and then increases only when needed.

Gebremedhin and Manne [8] developed a parallel graph coloring algorithm suitable for shared memory computers. In this algorithm, each processor is assigned equally many vertices to color. A processor colors its vertices in a sequential fashion, at each step assigning a vertex the smallest color not used by any of its neighbors (both on- or off-processor). An inconsistent coloring arises only when a pair of adjacent vertices that reside on different processors is colored simultaneously. Inconsistencies are then detected in a subsequent phase and resolved in a final sequential phase.

3 A New Algorithm

Here we describe a new distributed-memory parallel graph coloring algorithm. In the spirit of the BSP model [2], the algorithm is organized as a sequence of *supersteps*. A superstep has distinct, rather than intermingled, computation and communication phases.

A partitioning of the graph among the processors classifies the vertices into *interior* and *boundary* vertices. An interior vertex is a vertex all of whose neighbors are located on the same processor as itself. A boundary vertex has at least one neighbor located on a different processor. Clearly, the subgraphs induced by interior vertices are independent of each other and hence can be colored concurrently trivially. Coloring the remainder of the graph in parallel requires communication and coordination among the processors and this is the main issue in the algorithm being described.

3.1 The Basic Scheme

At the highest level, our algorithm is iterative—it operates in *rounds*. In each round there are two phases, a *tentative coloring* and a *conflict detection* phase. The former is organized into supersteps while the latter is not, since no communication is required. In every superstep each processor colors s vertices in a sequential manner, where s is an input parameter to the algorithm, using color information available at the beginning of the superstep, and then exchanges recent color information with other processors. In particular, in the communication phase of a superstep, a processor sends the colors of its boundary vertices to other processors and receives relevant color information from other processors. In this scenario, if two adjacent vertices located on two different processors are colored during the same superstep, they may receive the same color and hence cause a conflict. The purpose of the second phase of a round is to detect such conflicts and accumulate a list of vertices on each processor to be recolored in the next round. Since it is not necessary to recolor both endpoints of a conflict edge only one of the involved processors will add a vertex to its list. The processor that will do the recoloring is determined in a random fashion in order to achieve an even distribution of the vertices to be colored in the next round.

The conflict detection phase does not require communication since every processor has acquired a complete knowledge of the colors of the neighbors of its vertices at the end of the tentative coloring phase. The algorithm terminates when there is no more processor with a nonempty list of vertices to be recolored. Algorithm 1 outlines this scheme in more detail.

The rationale for dividing the coloring phase of a round in supersteps, rather than communicating after a single vertex is colored, is to reduce communication frequency and thereby reduce communication time. However the number of supersteps used (equivalently, the number of vertices colored in a superstep) is also closely related to the likelihood of conflicts and consequently the number of rounds. The lower the number of supersteps (the higher the number of vertices colored per superstep) the higher the likelihood of conflicts and hence the higher the number of rounds required. Choosing a value for s that minimizes the overall runtime is therefore a compromise between these two contradicting requirements. An optimal value of s would depend on such factors

Algorithm 1 An iterative parallel graph coloring algorithm

1: **procedure** PARALLELCOLORING($G = (V, E), s$)
2: *Initial data distribution*: V is partitioned into p subsets $V_1, \ldots, V_p$; processor P_i owns V_i, stores edges E_i incident on V_i, and stores the identity of processors hosting the other endpoints of E_i.
3: **on each** processor P_i, $i \in P = \{1, \ldots, p\}$
4: $U_i \leftarrow V_i$ ▷ U_i is the current set of vertices to be colored
5: **while** $\exists j \in P,\ U_j \neq \emptyset$ **do**
6: **if** $U_i \neq \emptyset$ **then**
7: Partition U_i into ℓ_i subsets $U_{i,1}, U_{i,2}, \ldots, U_{i,\ell_i}$, each of size s
8: **for** $k \leftarrow 1$ **to** ℓ_i **do** ▷ each k corresponds to a superstep
9: **for each** $v \in U_{i,k}$ **do**
10: assign v a permissible color
11: Send colors of boundary vertices in $U_{i,k}$ to relevant processors
12: Receive color information from other processors
13: Wait until all incoming messages are successfully received
14: $R_i \leftarrow \emptyset$ ▷ R_i is a set of vertices to be recolored
15: **for each** boundary vertex $v \in U_i$ **do**
16: **if** $\exists (v, w) \in E$ s.t. $color(v) = color(w)$ and $r(v) \leq r(w)$ **then**
17: $R_i \leftarrow R_i \cup \{v\}$ ▷ $r(v)$ is a random number
18: $U_i \leftarrow R_i$
19:

as the size and density of the input graph, the number of processors available, and the machine architecture and network.

Note that the formulation of Algorithm 1 is general enough to encompass the algorithms of Johannsson [10], Finocchi et al. [5], and Gebremedhin and Manne [8]. Setting $p = n$ (and $s = 1$) and choosing the color of a vertex in Line 10 appropriately, gives the algorithms of Johannsson and Finocchi et al. Setting $s = 1$, restricting Algorithm 1 to *one* round, and resolving conflicts sequentially gives the algorithm of Gebremedhin and Manne.

3.2 Variations

For the sake of generality, Algorithm 1 leaves several issues unspecified. In the sequel, we discuss such issues, in each case pointing out available alternatives.

(i) Initial partitioning. In a parallel application, the graph is usually already distributed among the processors in a reasonable way. However, if this is not the case, a “good” data distribution needs to be computed. The number of conflicts in the algorithm depends on several factors including the number of boundary vertices and the number of edges between these. Thus using a graph partitioner such as Metis [12] should help reduce the number of conflicts as well as the amount of communication.

(ii) Distinguishing between interior and boundary vertices. As mentioned earlier, the subgraphs induced by interior vertices are independent of each other and can therefore

be colored concurrently without any communication. Hence, in the context of Algorithm 1, the interior vertices can be colored *before*, *after*, or *interleaved* with boundary vertices. Algorithm 1 is presented assuming the last option. Coloring the interior vertices first may produce fewer conflicts when using a regular *First-Fit* coloring scheme, since the subsequent coloring of boundary vertices is performed with a larger spectrum of available colors. Coloring boundary vertices first may be advantageous with color selection variants such as Staggered First-Fit (see the discussion later in this section).

(iii) Synchronous vs. asynchronous supersteps. In Algorithm 1, the supersteps can be made to run in a *synchronous* fashion by introducing explicit synchronization barriers at the end of each superstep. An advantage of this mode is that in the conflict detection phase, the color of a boundary vertex needs to be checked only against its neighbors colored at the same superstep. The obvious disadvantage is that the barriers, in addition to the associated overhead, cause some processors to be idle while others complete their supersteps. Alternatively, the supersteps can be made to run *asynchronously*, without explicit barriers at the end of each superstep. Each processor would then only process and use the color information that has been completely received when it is checking for incoming messages. Any color information that has not reached a processor at this stage would thus be delayed from being used until a later superstep. Due to this, in the conflict detection phase, the color of a boundary vertex needs to be checked against all of its off-processor neighbors. Also, it is possible that the asynchronous version results in more conflicts than the synchronous one since a superstep on one processor now can overlap with more than one superstep on another processor.

(iv) Choice of color. The choice of a permissible color in Line 10 of Algorithm 1 can be made in different ways. The strategy employed affects (1) the number of colors used by the algorithm, and (2) the likelihood of conflicts, and thus the number of rounds required by the algorithm. Both of these quantities are desired to be as small as possible, and a coloring strategy typically reduces one of the quantities at the expense of the other. Here, we present two strategies: *First-Fit* (FF) and *Staggered First-Fit* (SFF). In FF each processor chooses the *smallest* permissible color from the interval $[1, C]$, where C is the current largest color used. If no such color exists, the new color $C + 1$ is chosen. SFF uses an initial estimate K of the number of colors needed for the input graph. Processor P_i chooses the *smallest* permissible color from the interval $[\lceil \frac{iK}{p} \rceil, K]$. If no such color exists, then the smallest permissible color in $[1, \lfloor \frac{iK}{p} \rfloor]$ is chosen. If there is still no such color, the smallest permissible color greater than K is chosen. Unlike FF, the search for a color in SFF starts from different "base colors" for each processor. Hence the latter is likely to result in fewer conflicts than the former. Other color selection strategies that have been suggested include the *randomized* techniques of Gebremedhin et al. [7] and Finocchi et al. [5].

4 Experiments

In this section, we present results from experiments carried out on a 16-node PC cluster equipped with dual 900 MHz Intel Itanium 2 CPUs and 4 GB memory. The nodes of

the cluster are interconnected via switched Myrinet 2000 network. Our test set consists of 19 graphs obtained from molecular dynamics and finite element applications [8, 13]. Table 1 displays the structural properties of the test graphs, including maximum, minimum, and average degree. The table also displays the number of colors and the runtime in seconds used by a sequential FF algorithm when run on a single node of our test platform. All of the results presented in this section are average performance results over all of the graphs presented in Table 1. Each individual test is an average of 5 runs. In the timing of the parallel coloring code, we assume the graph to be initially partitioned and distributed among the nodes of the parallel machine. Hence, the times reported concern only coloring.

Table 1. Properties of the test graphs

name	$\|V\|$	$\|E\|$	Degree			Seq. First-Fit	
			max	min	avg	#colors	time
HIV-2	11,414	15,270	8	1	2.68	5	0.007
HIV-4	11,414	130,332	39	6	22.84	17	0.034
HIV-6	11,414	412,623	116	13	72.30	45	0.099
HIV-10	11,414	1,655,383	454	35	290.06	176	0.387
popc-br-2	24,916	31,449	7	1	2.52	5	0.032
popc-br-4	24,916	255,047	43	2	20.47	21	0.067
popc-br-6	24,916	850,043	125	2	68.23	49	0.206
popc-br-10	24,916	3,587,724	514	2	287.98	173	0.84
er-gre-2	36,573	53,046	8	0	2.90	5	0.022
er-gre-4	36,573	451,355	42	3	24.68	19	0.116
er-gre-6	36,573	1,482,904	116	11	81.09	47	0.357
er-gre-10	36,573	6,511,122	460	79	356.06	174	1.515
apoa1-2	92,224	139,351	8	1	3.02	5	0.057
apoa1-4	92,224	1,131,436	43	2	24.54	20	0.293
apoa1-6	92,224	3,864,429	123	13	83.81	49	0.928
apoa1-10	92,224	17,100,850	503	54	370.85	182	3.993
598a	110,971	741,934	26	5	13.37	12	0.310
144	144,649	1,074,393	26	4	14.86	11	0.219
auto	448,695	3,314,611	37	4	14.77	13	0.984

In our experiments, we considered two ways of partitioning the vertices of a graph. In the first case, the vertex set, with the vertices in their natural order (i.e. the order in which the graphs were supplied), is partitioned into p contiguous blocks of (almost) equal size. Such a *block* partitioning does not attempt to minimize cross-edges, though the structure of the natural order is exploited. In the second case, the vertex set is partitioned into p disjoint subsets of nearly equal size such that the number of cross-edges is small. For this we used the graph partitioning software Metis [12], with an option known as VMetis that also attempts to minimize the communication volume and the number of boundary vertices.

The first set of experiments, shown in Figures 1 and 2, are conducted to assess the effects of the following three issues: block partitioning using the natural order (N)

vs. partitioning using VMetis (V); coloring interior vertices first (I), boundary vertices first (B), or interleaved (U); and using synchronous (S) vs. asynchronous supersteps (A). In all of these experiments we use FF for selecting the color of a vertex. A 3-letter acronym reflecting the options discussed above is used in Figures 1 and 2.

Figure 1 displays the number of conflicts (normalized with respect to the total number of vertices) for the parallel coloring algorithm for different combinations of these options while varying the superstep size and the number of processors. In Figure 1(a), we show results for the case where the number of processors is 8. Similar trends were observed for other number of processors. When varying the number of processors, the superstep size is set to 100. In the interleaved mode the superstep size gives the number of boundary vertices colored in each superstep.

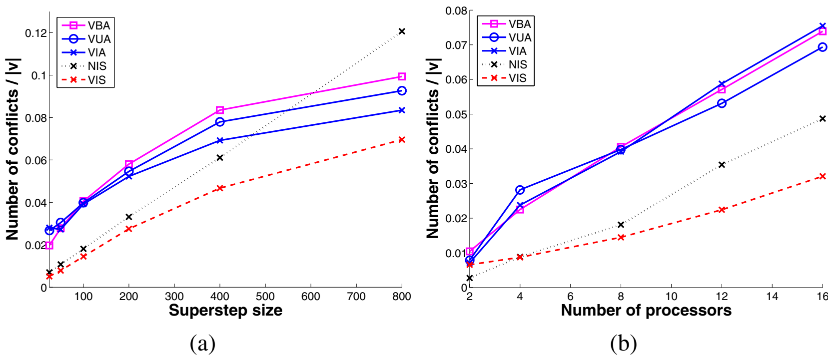


Fig. 1. Number of conflicts while varying (a) superstep size s for $p = 8$, and (b) number of processors for $s = 100$

Figure 1(a) and 1(b) shows that for all configurations the number of conflicts increases as the superstep size and the number of processors, respectively, increases. The two figures also show that asynchronous supersteps result in more conflicts than synchronous supersteps, and that graph partitioning using Metis results in fewer conflicts than block partitioning. In the case where block partitioning is used, only the combination of options (NIS) that gave the fewest conflicts is shown. When using Metis with synchronous supersteps we also only show the configuration (VIS) that gave the least number of conflicts. Using the boundary first and unordered options gave only slightly worse results than the presented ones. In terms of the number of conflicts, the results in Figures 1(a) and 1(b) suggest that the best result is obtained by partitioning the graph using Metis and using a small superstep size while running supersteps synchronously.

As can be observed from the figure in the asynchronous case, the order in which the boundary and interior vertices are colored has no major impact on the number of conflicts.

In all of our experiments, the number of rounds the algorithm has to iterate was observed to be consistently low, varying between two and five, for every configuration we tried. This is a consequence of the fact that the number of initial conflicts is small and

then drops rapidly between successive rounds. As long as Metis is used the total number of conflicts is within 10% of the total number of vertices in all of the configurations considered. Thus more than 90% of the sequential work is performed in the first round. This indicates that the increase in the number of vertices that need to be colored when going from a sequential to a parallel algorithm is fairly low for the test set we use. We also note that the number of colors used stays fairly low in all of our experiments and on the average, it does not increase by more than 4% of that used by the (sequential) FF coloring scheme.

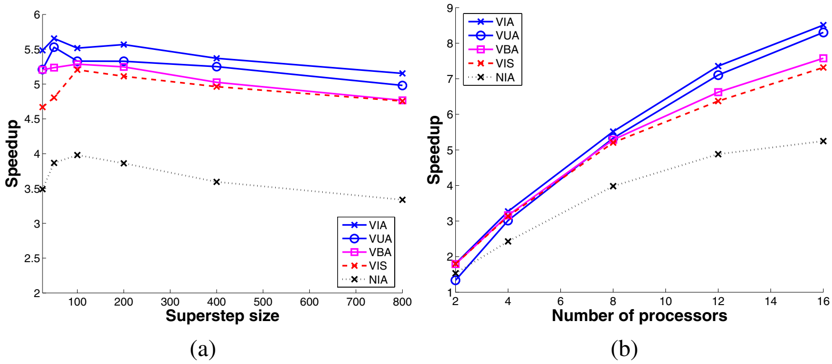


Fig. 2. Speedup while varying (a) superstep size s for $p = 8$, and (b) number of processors for $s = 100$

Figure 2(a) displays speedup values for the several variations of the parallel coloring algorithm while varying the superstep size s for a fixed number of processors $p = 8$. We show the NIA configuration (as opposed to NIS in Figure 1) as it gave the best speedup when not using Metis to partition the graph. As can be seen from the figure, the optimum value for s is close to 100 for all variants. Thus using $s = 100$ seems to be a good compromise between balancing the conflicting issues of increased message startup costs versus the number of conflicts. However, the manner in which the algorithm is configured seems to be more important than the superstep size. It is always better to use asynchronous communication than synchronous. Also, as can be seen from the figure coloring interior vertices first is slightly better than coloring the vertices interleaved which again is better than coloring the boundary vertices first.

In Figure 2(b) the speedup obtained as the number of processors is varied while using a superstep size of 100 is shown. The trends observed in Figure 2(b) are similar to those in Figure 2(a). The best *average* speedup, over all test cases, was about 8.5 while using 16 processors. However, for particular test cases, we have observed a speedup value as high as 12.5 while using 16 processors. The worst result observed was a speedup of 3.2 on 16 processors although this was a clear outlier. "Medium" dense graphs tend to give better speedup values than very sparse or very dense graphs.

Our next set of experiments concerns the different coloring schemes as discussed in Section 3. The results are shown in Figure 3(a) (conflicts), and Figure 3(b) (speedup).

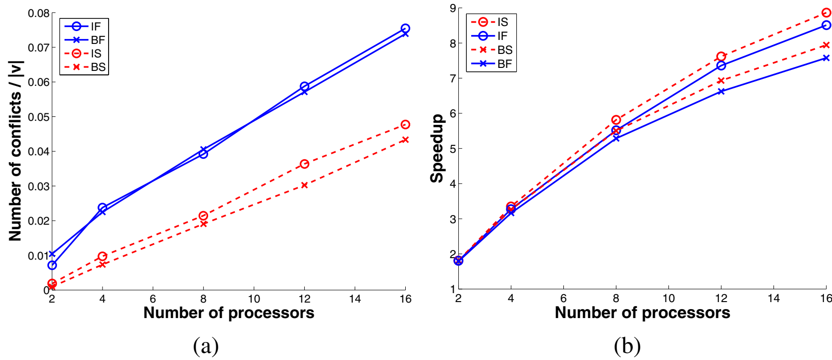
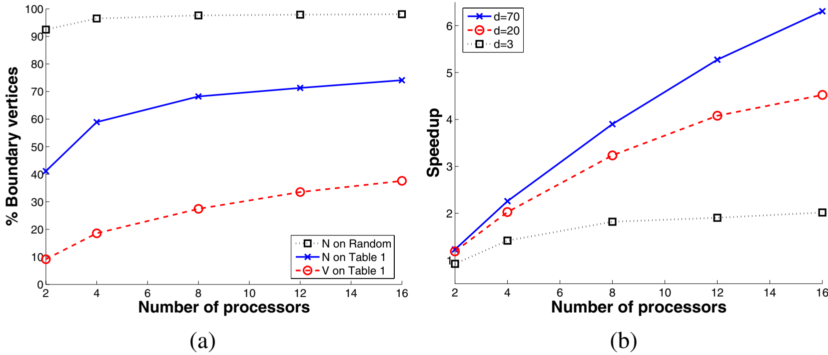


Fig. 3. Effect of the color selection algorithm on (a) the number of conflicts, and (b) speedup while using a superstep length of $s = 100$

In the figures the labels I and B show whether the interior vertices or the boundary vertices are colored first, while the second letter correspond to the FF (F) and the SFF (S) color selection scheme. In all of these experiments Metis is used for partitioning and the communication is done asynchronously. For SFF we use the number of colors found by sequential FF as our initial estimate of the number of colors. Coloring the vertices in an interleaved fashion gave similar results as those in the figures and are not shown here.

As expected, the SFF scheme gives fewer conflicts than the FF scheme. But as can be seen from Figure 3(b) in terms of speedup this is offset by the higher overhead associated with determining the correct color in the SFF scheme. Also, the SFF scheme has the disadvantage of requiring an a priori estimate on the expected number of colors.

The speedup achieved by our approach stems from two sources: partitioning and the "core" algorithm. Partitioning using Metis makes a trivial parallelization of the coloring of interior vertices possible. The "core" algorithm is a nontrivial way of coloring the boundary vertices in parallel. Figure 4(a) shows the percentage of boundary vertices for the graphs in Table 1 when using block partitioning with the natural vertex ordering, and when using Metis. As one can see the number of boundary vertices increases with the number of processors being used. Thus it is difficult to measure the particular speedup from coloring just the boundary vertices since the amount of work performed changes with the number of processors. In order to give some indication of the performance of the algorithm on the boundary vertices we present Figure 4(b). This shows the speedup when coloring three random graphs each containing 32000 vertices and with average vertex degrees 3, 20, and 70 respectively. For these experiments we used the NIA configuration with the vertices colored according to the SFF scheme. Since the vertices are ordered according to their natural order almost all the vertices become boundary vertices (see the topmost curve in Figure 4(a)). Thus this can be viewed as applying more processors while keeping the number of boundary vertices fixed. Since we are in effect traversing the graph at least twice (for coloring and verification) we cannot expect to get a speedup of more than $p/2$. Based on this the observed maximum speedup of more than 6 when using 16 processors is quite good.



(a) (b)

Fig. 4. (a) Percentage of boundary vertices for graphs in Table 1 (N = natural ordering, V = ordering given by Metis), and random graphs. (b) Speedup for random graphs of various average degrees

5 Conclusion

We have developed an efficient and truly scalable parallel graph coloring algorithm suitable for a distributed memory computer. The algorithm is flexible and can easily be tuned to suit the nature of the graph to be colored and the specifics of the hardware being used. The scalability of the algorithm has been experimentally demonstrated. This should be seen in light of the fact that previous distributed-memory parallel coloring algorithms, such as the algorithm of Jones and Plassmann [11], did not give any speedup when coloring the boundary vertices as more processors are applied.

Even though our main objective has been to achieve parallel speedup, being able to perform coloring in a distributed setting where the graph is already partitioned among the processors is an important functionality in itself.

In the future we plan to experiment with more sophisticated color selection schemes that may further reduce the number of conflicts. We are also considering how to generalize the algorithm to other coloring problems such as distance-2 graph coloring and hypergraph coloring, both of which have important applications in scientific computing.

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 NPAC technical report SCCS-666, Northeast Parallel Architectures Center at Syracuse University, 1994.
2. Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford, 2004.
3. T. F. Coleman and J. J More. Estimation of sparse jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, 1(20):187–209, 1983.
4. Pierluigi Crescenzi and Viggo Kann. A compendium of NP optimization problems. http://www.nada.kth.se/~viggo/wwwcompendium/.

5. Irene Finocchi, Alessandro Panconesi, and Riccardo Silvestri. Experimental analysis of simple, distributed vertex coloring algorithms. In *Proc. 13th ACM-SIAM symposium on Discrete Algorithms (SODA 02)*, 2002.
6. M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, 1979.
7. Assefaw Gebremedhin, Fredrik Manne, and Alex Pothen. Parallel distance-k coloring algorithms for numerical optimization. In *proceedings of Euro-Par 2002*, volume 2400, pages 912–921. Lecture Notes in Computer Science, Springer, 2002.
8. Assefaw Hadish Gebremedhin and Fredrik Manne. Scalable parallel graph coloring algorithms. *Concurrency: Practice and Experience*, 12:1131–1146, 2000.
9. Robert K. Gjertsen Jr., Mark T. Jones, and Paul Plassmann. Parallel heuristics for improved, balanced graph colorings. *J. Par. and Dist. Comput.*, 37:171–186, 1996.
10. Öjvind Johansson. Simple distributed $\delta + 1$-coloring of graphs. *Information Processing Letters*, 70:229–232, 1999.
11. Mark T. Jones and Paul Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, 1993.
12. G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1), 1999.
13. Michelle Mills Strout and Paul D. Hovland. Metrics and models for reordering transformations. In *Proceedings of the The Second ACM SIGPLAN Workshop on Memory System Performance (MSP)*, pages 23–34, June 8 2004.