

DISTRIBUTED-MEMORY PARALLEL ALGORITHMS FOR DISTANCE-2 COLORING AND THEIR APPLICATION TO DERIVATIVE COMPUTATION*

DORUK BOZDAĞ[†], ÜMİT V. ÇATALYÜREK[‡], ASSEFAW H. GEBREMEDHIN[§],
FREDRIK MANNE[¶], ERIK G. BOMAN^{||}, AND FÜSUN ÖZGÜNER^{**}

Abstract. The distance-2 graph coloring problem aims at partitioning the vertex set of a graph into the fewest sets consisting of vertices pairwise at distance greater than two from each other. Its applications include derivative computation in numerical optimization and channel assignment in radio networks. We present efficient, distributed-memory, parallel heuristic algorithms for this NP-hard problem as well as for two related problems used in the computation of Jacobians and Hessians. Parallel speedup is achieved through graph partitioning, speculative (iterative) coloring, and a BSP-like organization of parallel computation. Results from experiments conducted on a PC cluster employing up to 96 processors and using large-size real-world as well as synthetically generated test graphs show that the algorithms are scalable. In terms of quality of solution, the algorithms perform remarkably well—the number of colors used by the parallel algorithms was observed to be very close to the number used by the sequential counterparts, which in turn are quite often near optimal. Moreover, the experimental results show that the parallel distance-2 coloring algorithm compares favorably with the alternative approach of solving the distance-2 coloring problem on a graph \mathcal{G} by first constructing the square graph \mathcal{G}^2 and then applying a parallel distance-1 coloring algorithm on \mathcal{G}^2 . Implementations of the algorithms are made available via the Zoltan load-balancing library.

Key words. Distance-2 graph coloring; distributed-memory parallel algorithms; Jacobian computation; Hessian computation; sparsity exploitation; automatic differentiation; combinatorial scientific computing

AMS subject classifications. 05C15, 05C85, 68R10, 68W10

1. Introduction. Many algorithms in scientific computing, including algorithms for nonlinear optimization, differential equations, inverse problems, and sensitivity analysis, need to compute the Jacobian or Hessian matrix. In large-scale problems the derivative matrices are typically *sparse*, a property that needs to be exploited to make computation efficient, and in some cases, even feasible. An archetypal model in the efficient computation of sparse Jacobian and Hessian matrices—whether derivatives are calculated using automatic differentiation or estimated using finite differencing—is a *distance-2 coloring* of an appropriate graph [11]. Distance-2 coloring also finds applications in other areas, such as in channel assignment problems in radio networks [18, 19]. In a distance- k coloring, any two vertices connected by a path consisting of at most k edges are required to receive different colors, and the goal is to use as few colors as possible. Distance-1 coloring is used, among others, for discovering concurrency in parallel scientific computing [15, 16, 23].

*This work was supported by the U.S. Department of Energy’s Office of Science through the CSCAPES SciDAC Institute; by the U.S. National Science Foundation under Grants #CNS-0643969 and #CNS-0403342; by Ohio Supercomputing Center #PAS0052; and by the Norwegian Research Council through the Evita program.

[†]Ohio State University, Columbus, Ohio (bozdagd@ece.osu.edu).

[‡]Ohio State University, Columbus, Ohio (umit@bmi.osu.edu).

[§]Purdue University, West Lafayette, Indiana (agebre@purdue.edu).

[¶]University of Bergen, Bergen, Norway (Fredrik.Manne@ii.uib.no).

^{||}Sandia National Laboratories, Albuquerque, New Mexico (egboman@sandia.gov). 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.

^{**}Ohio State University, Columbus, Ohio (ozguner@ece.osu.edu).

In parallel applications where a distance- k coloring is needed, the graph is either already partitioned and mapped or needs to be partitioned and mapped onto the processors of a distributed-memory parallel machine. Under such circumstances, gathering the graph on one processor to perform the coloring sequentially is prohibitively time consuming or infeasible due to memory constraints. Hence the graph needs to be colored in parallel. Finding a distance- k coloring using the fewest colors is an NP-hard problem [20], but greedy heuristics are effective in practice, as they run fast and provide solutions of acceptable quality [7, 11, 14]. They are, however, inherently sequential and thus challenging to parallelize.

We have developed a variety of efficient greedy parallel algorithms for distance-2 coloring on distributed memory environments, and we report in this paper on the design, analysis, implementation, and experimental evaluation of the algorithms. Appropriate variants of the algorithms tailored for Jacobian and Hessian computation are also presented. The algorithms presented here are obtained by extending the parallelization framework we developed in a recent work in the context of distance-1 coloring [5]. The framework is an iterative, data-parallel, algorithmic scheme that proceeds in two-phased rounds. In the first phase of each round, processors concurrently color the vertices assigned to them in a speculative manner, communicating at a course granularity. In the second phase, processors concurrently check the validity of the colors assigned to their respective vertices and identify a set of vertices that needs to be recolored in the next round to resolve any detected inconsistencies. The scheme terminates when every vertex has been colored correctly.

One of the challenges involved in extending the framework outlined above to the distance-2 coloring case is devising an efficient means of information exchange between processors hosting a pair of vertices that are two edges away from each other in the graph. For such pairs of vertices, relying on a direct communication between the corresponding processors would incur unduly high communication cost and locally storing duplicates of distance-2 neighborhoods would require unduly large memory space. Instead, we employ a strategy in which information is relayed via a third processor (the processor owning a mutual distance-1 neighbor of vertices two edges away from each other) as needed. We show that the parallel algorithms designed using this strategy yield good speedup with increasing number of processors while using nearly the same number of colors as a serial greedy algorithm. We also show that the algorithms outperform the alternative approach based on distance-1 coloring of a square graph.

A preliminary version of a small portion of the work presented here has appeared in a conference paper [4]. Compared to [4], the current paper has several new contributions. In terms of the basic distance-2 coloring algorithm for general graphs, the algorithm has been described much more rigorously, its complexity has been analyzed, and possible variations of the algorithm have been outlined; moreover, the experimental performance evaluation of the algorithm is conducted much more thoroughly, and is carried out on a larger set of test problems and on a larger number of processors. In addition, new algorithms for distance-2 coloring of bipartite graphs (for Jacobian computation) and restricted star coloring of general graphs (for Hessian computation) have been presented and experimentally evaluated.

To the best of our knowledge, this paper is the first to present demonstrably efficient and scalable parallel algorithms for distance-2 coloring on distributed-memory architectures. Gebremedhin, Manne, and Pothen [13] have developed shared-memory parallel algorithms for distance-2 coloring. They have also provided a comprehensive

review of the role of graph coloring in derivative computation in [11], and designed efficient serial algorithms for acyclic and star coloring (which are used in Hessian computation) in [14]. The literature on algorithmic graph theory features some work related to the distance-2 coloring problem [1, 2, 18, 19]. Readers are referred to Section 11.4 of the paper [11] and Section 2.3 of the paper [14] for more pointers to theoretical work on distance- k and related coloring problems.

The remainder of this paper is organized as follows. §2 provides background: it includes a self-contained review of the coloring models for sparse derivative matrix computation that are relevant for this paper, and a brief discussion of serial greedy coloring algorithms, since they form the foundation for the parallel algorithms presented here. §3 sets the stage for a detailed presentation of the parallel distance-2 coloring algorithm for general graphs in §4 by discussing several algorithm design issues. §5 shows how the algorithm described in §4 can be adapted for restricted star coloring of general graphs (for Hessian computation) and distance-2 coloring of bipartite graphs (for Jacobian computation). §6 contains a detailed computational evaluation of the performance of the parallel algorithms, and §7 concludes the paper.

2. Background.

2.1. Preliminary concepts and notations. Two distinct vertices in a graph are *distance- k* neighbors if a shortest path connecting them consists of at most k edges. We denote the set of distance- k neighbors of a vertex v by $N_k(v)$. The *degree- k* of a vertex v , denoted by $d_k(v)$, is the number of distinct paths of length at most k edges starting at v . Two paths are distinct if they differ in at least one edge. Note that $d_1(v) = |N_1(v)|$, and $d_2(v) = \sum_{w \in N_1(v)} d_1(w)$. In general, $d_k(v) \geq |N_k(v)|$. We denote the average degree- k in a graph by \bar{d}_k .

A *distance- k coloring* of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a mapping $\phi : \mathcal{V} \rightarrow \{1, 2, \dots, q\}$ such that $\phi(v) \neq \phi(w)$ whenever vertices v and w are distance- k neighbors. A distance- k coloring of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is equivalent to a distance-1 coloring of the k th *power* graph $\mathcal{G}^k = (\mathcal{V}, \mathcal{F})$, where $(v, w) \in \mathcal{F}$ whenever vertices v and w are distance- k neighbors in \mathcal{G} . A distance- k coloring of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ can equivalently be viewed as a partition of the vertex set \mathcal{V} into q *distance- k independent sets*—sets of vertices at a distance greater than k edges from each other. Variants of distance- k coloring are used in modeling partitioning problems in sparse Jacobian and Hessian computation. We review these in the next subsection; for a more comprehensive discussion, see [11, 14].

2.2. Coloring models in derivative computation. The computation of a sparse $m \times n$ derivative matrix A using automatic differentiation (or finite differencing) can be made efficient by first *partitioning* the n columns into q disjoint groups, with q as small as possible, and then evaluating the columns in each group jointly (as a sum) rather than separately. More specifically, the values of the entries of the matrix A are obtained by first evaluating a *compressed* matrix $B \equiv AS$, where S is an $n \times q$ *seed* matrix whose (j, k) entry s_{jk} is such that s_{jk} equals one if and only if column a_j belongs to group k and zero otherwise, and then recovering the entries of A from B .

The specific criteria used to define a seed matrix S for a derivative matrix A depends on whether the matrix A is Jacobian (nonsymmetric) or Hessian (symmetric). It also depends on whether the entries of A are to be recovered from the compressed representation B *directly* (without any further arithmetic), via *substitution* (by implicitly solving a set of simple triangular systems of equations), or via *elimination*

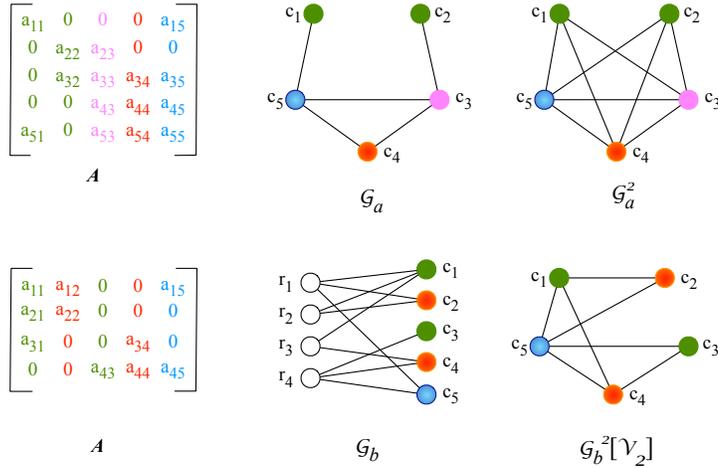


FIG. 2.1. Equivalence among structurally orthogonal column partition of A , distance-2 coloring of $\mathcal{G}(A)$ and distance-1 coloring of $\mathcal{G}^2(A)$. Top: symmetric case. Bottom: nonsymmetric case.

(by solving a rectangular system of equations). In this paper we focus on only direct methods.

2.2.1. Structurally orthogonal partition. Curtis, Powell, and Reid [9] showed that a *structurally orthogonal* partition of a Jacobian matrix A —a partition of the columns of A in which no two columns in a group share a nonzero at the same row index—gives a seed matrix S where the entries of A can be directly recovered from the compressed representation $B \equiv AS$. The structure of a Jacobian matrix A can be represented by the *bipartite* graph $\mathcal{G}_b(A) = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E})$, where \mathcal{V}_1 is the row vertex set, \mathcal{V}_2 is the column vertex set, and $(r_i, c_j) \in \mathcal{E}$ whenever the matrix entry a_{ij} is nonzero. A partitioning of the columns of the matrix A into groups consisting of structurally orthogonal columns is equivalent to a *partial distance-2 coloring* of the bipartite graph $\mathcal{G}_b(A)$ on the vertex set \mathcal{V}_2 [11]. It is called “partial” because the row vertex set \mathcal{V}_1 is left uncolored.

A structurally orthogonal column partition could also be used in computing a Hessian via a direct method, albeit that symmetry is not exploited. Specifically, McCormick [20] showed that a structurally orthogonal partition of a Hessian is equivalent to a distance-2 coloring of its adjacency graph. The *adjacency graph* $\mathcal{G}_a(A)$ of a Hessian A has a vertex for each column, and an edge joins column vertices c_i and c_j whenever the entry a_{ij} , $i \neq j$, is nonzero; the diagonal entries in A are assumed to be nonzero and they are not explicitly represented by edges in the graph $\mathcal{G}_a(A)$. Figure 2.1 illustrates how a structurally orthogonal column partition of a matrix is modeled by a distance-2 coloring in the appropriate graph. The right most subfigures show the equivalent distance-1 coloring formulations in the appropriate square graph.

2.2.2. Symmetry-exploiting partition. Powell and Toint [22] were the first to introduce a symmetry-exploiting technique for computing a Hessian via a direct method. When translated to a coloring ϕ of the adjacency graph, the partition Powell and Toint suggested for a direct Hessian computation requires that (1) ϕ be a distance-1 coloring, and (2) in every path v, w, x on three vertices, the terminal vertices v and x are allowed to have the same color, but only if the color of the

middle vertex w is lower in value. A coloring that satisfies these two requirements has been called a *restricted star coloring* in [14].

Coleman and Moré [8] showed that a *symmetrically orthogonal partition* of a Hessian is sufficient for a direct recovery, and established that such a partition is equivalent to a star coloring of the adjacency graph of the Hessian. A *star coloring* is a distance-1 coloring where, in addition, every path on four vertices uses at least three colors. The name is due to the fact that in a star-colored graph, a subgraph induced by any two color classes is a *collection of stars*. Note that the three coloring models for direct Hessian computation discussed here can be ranked in an increasing order of restriction (decreasing order of accuracy) as *star coloring*, *restricted star coloring*, *distance-2 coloring*.

2.3. Greedy coloring algorithms. An optimization problem associated with distance- k , restricted star, or star coloring asks for an appropriate coloring with the *fewest* colors, and each is known to be NP-hard [8, 14, 20]. In practice, *greedy* algorithms have been found effective in delivering good suboptimal solutions for these problems fast [7, 14]. A greedy algorithm for each of these problems progressively extends a partial coloring of a graph by processing one vertex at a time, in some order; there exist a number of effective ordering techniques that are based on some variation of vertex degree [7, 14]. In the step where a vertex v is colored, first, a set F of *forbidden colors* for the vertex v is obtained by exploring the appropriate neighborhood of v . Then, the *smallest* allowable color (not included in F) is chosen and assigned to v .

In the case of distance-1 coloring, such a greedy algorithm uses at most $\Delta + 1$ colors, where Δ is the maximum degree-1 in the graph. The quantity $\Delta + 1$ is a lower bound on the optimal number of colors needed in a distance-2 coloring. Furthermore, the number of colors used by a greedy distance-2 coloring algorithm is bounded from above by $\min\{\Delta^2 + 1, n\}$, where n is the number of vertices in the input graph. Using this bound, McCormick [20] showed that the greedy distance-2 coloring algorithm is an $O(\sqrt{n})$ -approximation algorithm.

Greedy algorithms for distance-1 and distance-2 coloring can be implemented such that their respective complexities are $O(n\bar{d}_1)$ and $O(n\bar{d}_2)$. Gebremedhin et al. [14] have developed $O(n\bar{d}_2)$ -time greedy algorithms for star and restricted star coloring. Their algorithm for star coloring takes advantage of the structure of the two-colored induced subgraphs—the collection of stars—and uses fairly complex data structures to maintain them. In this paper we develop parallel versions of the greedy algorithms for distance-2 and restricted star coloring. We considered the simpler variant, restricted star coloring, instead of star coloring, since restricted star coloring can be derived via a simple modification of the parallel algorithm for the distance-2 coloring problem, which is the main focus of this paper.

3. Design issues. The parallel distance-2 coloring algorithm proposed in this paper will be presented in detail in §4. Here we discuss the major issues that arise in the design of the algorithm and the assumptions and decisions we made around them.

3.1. Data distribution. The way in which the input graph is partitioned and mapped to processors has implications for both load balance and inter-processor communication. A graph could be partitioned among processors either by partitioning the vertex set or by partitioning the edge set. Traditionally, vertex partitioning has been the most commonly used strategy for mapping graphs (or matrices) to processors [3, 6, 10]. When a vertex partitioning is used, edges are implicitly mapped to

processors, with every crossing edge essentially being duplicated on the two processors to which its endpoints are mapped. For matrices, this corresponds to mapping of entire columns or rows to processors. To make our algorithm and its implementation more readily usable in other parallel codes, we assume that a vertex partitioning is used in distributing the graph among processors. A vertex partitioning classifies the vertices of the graph into two categories: *interior* and *boundary*. An interior vertex is a vertex all of whose distance-1 neighbors are mapped onto the same processor as itself. A boundary vertex has at least one distance-1 neighbor mapped onto a different processor.

3.2. Data duplication and communication mechanism. The next design issue is data duplication and its impact on information exchange.

As stated earlier, when a vertex partitioning is used, every crossing edge is duplicated, and that was the approach used in our earlier work on distance-1 coloring [5]. In such a mapping strategy, it makes sense for each processor to store the colors of the off-processor endpoints of its crossing edges as this would require storing at most one extra color per crossing edge. In terms of communication, such a storage scheme necessitates each processor to send the colors of its boundary vertices to neighboring processors as soon as the colors become available. Each receiving processor could then store the information and use it later while coloring its own vertices. In this way, the color of a boundary vertex is sent at most once for each of its incident crossing edges.

In the distance-2 coloring case, for each vertex, color information about vertices that are two edges away is also needed. One way of acquiring this information would be for each processor to keep a local copy of the subgraph induced by off-processor vertices that are within distance two edges from its own boundary vertices. Then, each processor could store and have access to all the needed color information as soon as the information is received from neighboring processors. Thus, as in the distance-1 coloring case, color information could be sent as soon as it becomes available. This would in turn allow for a flexible coloring order on each processor, since the order in which vertices are colored can be freely determined as the algorithm proceeds. However, this flexibility comes at the expense of extra storage. For relatively dense graphs, large portions of the input graph may need to be duplicated on each processor. In fact, this could happen even if there was just one high degree boundary vertex. For this reason, we chose to duplicate just the boundary vertices and their colors, but not more.

With this design decision in place, each processor will gain local access to distance-1 color information just like in the distance-1 coloring algorithm, and the information can be exchanged among the processors at the earliest possible time. But a mechanism for exchanging color information among vertices that are two edges apart still needs to be devised. Since such colors are not going to be stored permanently on the receiving processor, each color will have to be resent every time it is needed. Here, there are two basic ways in which the communication can be coordinated: one can use either a *request-based protocol* or a *precomputed schedule*. In a request-based protocol, each processor would send a message to its neighboring processors asking for specific color information whenever it needs the information. It then receives the information, uses it, and discards it. With a precomputed schedule, each processor would know the order (at least partially) in which its neighbor processors are going to color their vertices. Thus a processor could itself determine what color information to send to its neighbors and when to do so. A request-based protocol gives rise to more communication than a precomputed schedule, but on the other hand, it is more flexible as it allows for a

Algorithm 1 Overview of the parallel distance-2 coloring algorithm.

```

1: procedure PARALLELCOLORING( $\mathcal{G} = (\mathcal{V}, \mathcal{E}), s$ )
2:   Data distribution:  $\mathcal{G}$  is divided into  $p$  subgraphs  $G_1 = (V_1, E_1), \dots, G_p = (V_p, E_p)$ ,
   where  $V_1, \dots, V_p$  is a partition of the set  $\mathcal{V}$  and  $E_i = \{(v, w) : v \in V_i, (v, w) \in \mathcal{E}\}$ .
   Processor  $P_i$  owns the vertex set  $V_i$ , and stores the edge set  $E_i$  and the ID's of
   the processors owning the other endpoints of  $E_i$ .
3:   on each processor  $P_i, i \in P = \{1, \dots, p\}$ 
4:      $I_i \leftarrow$  interior vertices in  $V_i$ 
5:      $B_i \leftarrow$  boundary vertices in  $V_i$   $\triangleright V_i = I_i \cup B_i$ 
6:     Color the vertices in  $I_i$ 
7:     Assign each vertex  $v \in B_i \cup N_1(B_i)$  a random number  $rand(v)$ , generated using
        $v$ 's ID as seed
8:      $U_i \leftarrow B_i$   $\triangleright U_i$  is the current set of boundary vertices to be colored by  $P_i$ 
9:     while  $\exists j \in P, U_j \neq \emptyset$  do
10:       TENTATIVELYCOLOR( $U_i$ )
11:        $U_i \leftarrow$  DETECTCONFLICTS()

```

completely independent coloring order on each processor. In our algorithm we chose to use a precomputed schedule. Even with a precomputed schedule, there exists an opportunity for using ordering techniques at a local level on each processor, but we fore-go a detailed study of such ordering techniques to limit the scope of this paper.

4. Parallel Distance-2 Coloring of General Graphs. We are now ready to present the new parallel distance-2 coloring algorithm for a general graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. We begin in §4.1 by providing an overview of the algorithm, and then present its details layer-by-layer in §4.2 through §4.4. The complexity of the algorithm is analyzed in §4.5, and a brief discussion of possible variations of the algorithm is given in §4.6. In §5 we will show how the algorithm needs to be modified to solve the restricted star coloring problem on a general graph \mathcal{G} and the partial distance-2 coloring problem on a bipartite graph $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E})$.

4.1. Overview of the algorithm. Initially, the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is assumed to be vertex-partitioned and distributed among the p available processors. The set V_i of vertices in the partition $\{V_1, \dots, V_p\}$ of \mathcal{V} is assigned to and colored by processor P_i . We say P_i *owns* V_i . In addition, processor P_i stores the adjacency list of its vertices and the identities of the processors owning them. This initial data distribution classifies each set V_i into sets of interior and boundary vertices ($V_i = I_i \cup B_i$). We call two processors P_i and P_j *neighbors* if at least one boundary vertex owned by processor P_i has a distance-1 neighbor vertex owned by processor P_j .

Clearly, any two interior vertices owned by two different processors can safely be colored concurrently in a distance-2 coloring. In contrast, a concurrent coloring of a pair of boundary vertices or a pair consisting of one boundary and one interior vertex may not be safe, as the constituents of the pair, while being distance-2 neighbors, may receive the same color and therefore result in a *conflict*. We avoid the latter situation for a potential conflict (due to a pair consisting of one boundary and one interior vertex) by requiring that interior vertices be colored strictly before or strictly after boundary vertices have been colored. Then, a conflict can occur only for pairs of boundary vertices. Thus, the central part of the algorithm being presented is concerned with how the coloring of the boundary vertices is performed in parallel.

The main idea is to perform the coloring of the boundary vertices concurrently in

a speculative manner and then detect and rectify conflicts that may have arisen. The algorithm (iteratively) proceeds in *rounds*, each consisting of a *tentative coloring* and a *conflict detection* phase. Both of these phases are performed in parallel. To reduce the frequency of communication among processors, the tentative coloring phase is organized in a sequence of *supersteps*, a term borrowed from the literature on the Bulk Synchronous Parallel model [3] and used here in a loose sense. Specifically, in each superstep, each processor colors a pre-specified number s of the vertices it owns in a sequential manner, using forbidden color information available at the beginning of the superstep, and only thereafter sends recent color information to *neighboring* processors. In this scenario, if two boundary vertices that are either adjacent or at a distance of exactly two edges from each other are colored during the same superstep, they may receive the same color and thus cause a conflict. The purpose of the subsequent detection phase is to discover such conflicts in the current round and accumulate a list of vertices on each processor that needs to be recolored in the next round to resolve the conflicts.

Given a pair of vertices involved in a conflict, it suffices to re-color only one of them to resolve the conflict. The vertex to be recolored is determined by making use of a *global* random function defined over all boundary vertices. In particular, each processor P_i assigns a random number to each vertex in the set $B_i \cup N_1(B_i)$, where B_i is the set of boundary vertices owned by P_i and $N_1(B_i) = \cup_{w \in B_i} N_1(w)$. Each random number $rand(v)$ is generated using the global ID of the vertex v , to avoid the need for processors to inquire each other of random values. The algorithm terminates when no more vertices to be re-colored are left. A high-level structure of the algorithm is given in Algorithm 1. The routines TENTATIVELYCOLOR and DETECTCONFLICTS called in Algorithm 1 will be discussed in detail in §4.3 and §4.4, respectively. But first we discuss a few fundamental techniques employed in Algorithm 1.

4.2. Conflict detection and relaying distance-2 color information. In addition to the design issues on data distribution, data duplication, and communication protocol discussed in §3, the way in which conflicts are detected is a major issue in the design of Algorithm 1. We employed a strategy in which for every path v, w, x on three vertices, the processor on which the vertex w resides is responsible for detecting not only conflicts that involve the vertex w and an adjacent vertex in $N_1(w)$ but also a conflict involving the vertices v and x . We call the former (involving adjacent vertices) *type 1* conflicts, and the latter (involving vertices two edges apart) *type 2* conflicts. A type 1 conflict is detected by both of the implied processors, whereas a type 2 conflict is detected by the processor owning the middle vertex. Clearly, this way of detecting a type 2 conflict is more efficient than the alternative in which the conflict is detected by both of the processors owning the terminal vertices v and x .

As the termination condition of the while-loop in Algorithm 1 indicates, even if a processor has no more vertices left to be re-colored in a round, *i.e.*, $U_i = \emptyset$, it could still be active in that round, as the processor may need to provide color information to other processors, participate in detecting conflicts on other processors, or both.

Another basic ingredient in the design of Algorithm 1 is the technique used to *build* the list of forbidden colors for a given vertex v in a given superstep. The technique is directly related with the strategies on data duplication and communication protocol employed in the design of the algorithm (these were discussed in §3.2). The next two paragraphs discuss elements of this technique.

Let P_i be the processor that owns the vertex v . The list of forbidden colors for the vertex v consists of (1) colors assigned to adjacent vertices—those in the set

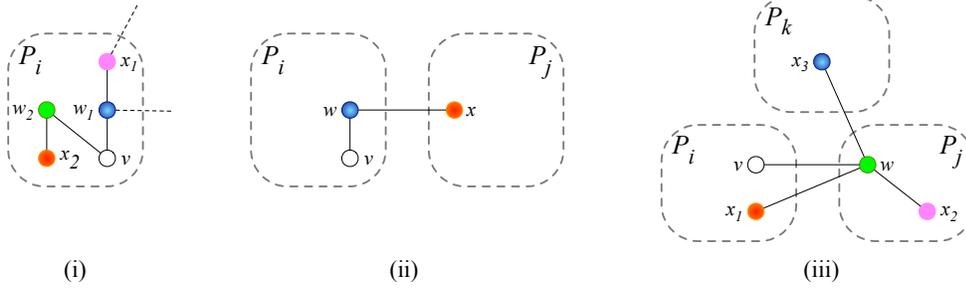


FIG. 4.1. Scenarios depicting the distribution of the distance-2 neighbors of vertex v across processors.

$N_1(v)$ —and (2) colors assigned to vertices exactly two edges away from v . These colors are assigned either in a previous superstep (for boundary vertices) or prior to the iterative coloring (for interior vertices). We classify these colors as *local* or *nonlocal* relative to processor P_i on the onset of the superstep. A color of a vertex u is local to P_i if P_i owns either the vertex u or some distance-1 neighbor of u (in which case P_i would store a copy of u 's color information, which is computed and sent by u 's owner). In contrast, a color is nonlocal to P_i if the information is not locally stored and hence needs to be relayed via an “intermediate” processor.

Figure 4.1 shows the three scenarios in which the vertices on a path v, w, x may be distributed among processors. Case (i) corresponds to the situation where both of the vertices w and x (w_1 and x_1 or w_2 and x_2 in the figure) are owned by processor P_i . Clearly, in this case, the colors of the vertices w and x are both local to P_i . Case (ii) shows the situation where vertex w is owned by processor P_i and vertex x is owned by processor P_j , $j \neq i$. In this case, again, the colors of both vertices w and x are local to P_i . Case (iii) shows the situation where vertex w is owned by processor P_j , and vertices v and x do not have a common distance-1 neighbor owned by processor P_i . As depicted in the figure, vertex x may be owned by any one of the three processors P_i , P_j , or P_k , $i \neq j \neq k$ (shown as vertices x_1 , x_2 , and x_3 , respectively). In this third case, if the vertex x is owned by either of the processors P_j or P_k (shown as x_2 and x_3), then the color of x is nonlocal to processor P_i and needs to be relayed to P_i through processor P_j . On the other hand, if the vertex x is owned by processor P_i (shown as x_1), then the color of x is, strictly speaking, local to P_i and need not be relayed via P_j . However, in the algorithm being described, since processor P_j does not store the adjacency lists of the vertices owned by processor P_i , it would treat the color of x_1 as if it were nonlocal to P_i and send the color information to P_i . In other words, for every edge (v, w) in which vertex v is owned by processor P_i and w is owned by P_j , processor P_j takes the responsibility of building a list of colors used by vertices two edges away from the vertex v (a partial list of forbidden colors to v) and sending the list to processor P_i .

4.3. The tentative coloring phase. Algorithm 2 outlines in detail the routine TENTATIVELYCOLOR run on each processor P_i . The routine starts off by processor P_i determining a coloring-schedule—a breakdown of its current set U_i of vertices to be colored into supersteps (Line 2). Processor P_i then computes and sends schedule information to each of its neighboring processors (Lines 3–5). Similarly, processor P_i receives analogous schedules from each of its neighboring processors (Line 6). This enables each processor to know the distance-2 color information it needs to send in

Algorithm 2 Tentative coloring phase of Algorithm 1 run on processor P_i .

```

1: procedure TENTATIVELYCOLOR( $U_i$ )
2:   Partition  $U_i$  into  $n_i$  subsets  $U_{i,1}, U_{i,2}, \dots, U_{i,n_i}$ , each of size  $s \triangleright n_i = \lceil \frac{|U_i|}{s} \rceil$ .
   Vertices in  $U_{i,\ell}$  will be colored in the  $\ell$ 'th superstep by processor  $P_i$ 
3:   for each processor-superstep pair  $(j, \ell) \in \{\{1, \dots, p\} \times \{1, \dots, n_i\}\}$ ,  $j \neq i$  do
4:      $U_{i,\ell}^j \leftarrow \{v | v \in U_{i,\ell} \text{ and } N_1(v) \cap V_j \neq \emptyset\} \triangleright$  processor  $P_j$  is neighbor to processor  $P_i$ 
5:     Send schedules  $U_{i,\ell}^j$  to each neighbor processor  $P_j$ 
6:     Receive schedules  $U_{j,\ell}^i$  from each neighbor processor  $P_j$ 
7:      $X_{i,\ell} \leftarrow \bigcup_j U_{j,\ell}^i \triangleright$  Vertices in  $X_{i,\ell}$  will be colored in the  $\ell$ 'th step by processors  $P_j, j \neq i$ 
8:     for each  $v \in X_i \cup U_i$ , where  $X_i = \bigcup_\ell X_{i,\ell}$  do
9:        $color(v) \leftarrow 0 \triangleright$  (re)initialize colors
10:     $L \leftarrow \max_{1 \leq j \leq p} \{n_j\} \triangleright L$  is the max number of supersteps over all processors
11:    for  $\ell \leftarrow 1$  to  $L$  do  $\triangleright$  each  $\ell$  corresponds to a superstep
12:      Build lists of forbidden colors for vertices in  $U_{j,\ell}^i$ 
13:      Send the lists to each neighboring processor  $P_j$  where  $\ell \leq n_j$ 
14:      if  $\ell \leq n_i$  then  $\triangleright P_i$  has not finished coloring  $U_i$ 
15:        Receive lists of forbidden colors for vertices in  $U_{i,\ell}^j$  from each neighboring  $P_j$ 
16:        Merge the lists of forbidden colors
17:        Update the lists of forbidden colors with local color information
18:        for each  $v \in U_{i,\ell}$  do
19:           $color(v) \leftarrow c$  such that  $c > 0$  is the smallest “permissible” color for  $v$ 
20:          Send updated colors of vertices in  $U_{i,\ell}^j$  to each neighboring  $P_j$ 
21:          Receive updated colors of vertices in  $U_{j,\ell}^i$  from each neighboring  $P_j$ 

```

each superstep. In particular, using the schedules, for each superstep ℓ , processor P_i constructs a list $X_{i,\ell}$ of vertices that will be colored by some other processor P_j in superstep ℓ and for which it must supply forbidden color information (Line 7). Thus with the knowledge of each $X_{i,\ell}$, processor P_i can be “pro-active” in building up lists of relevant forbidden color information and sending these to neighboring processors in superstep ℓ .

Before the coloring of the vertices in the set U_i by processor P_i commences, impermissible colors assigned in a previous superstep need to be cleared. These consist of colors assigned to vertices in U_i (by processor P_i) and colors assigned to vertices in X_i , which are to be colored by other processors in the current round (Lines 8 and 9).

Since for different processors P_i the number of vertices $|U_i|$ to be colored could differ, the number of supersteps required to color these vertices, $n_i = \lceil \frac{|U_i|}{s} \rceil$, would also vary. Processor P_i needs n_i supersteps to color its vertices, but it may need to supply forbidden color information to a neighboring processor that has not finished coloring its vertices. Therefore, the algorithm overall needs $L = \max_{1 \leq j \leq p} \{n_j\}$ supersteps (see Line 10).

In each superstep, before processor P_i begins to color the set of vertices it owns, it pro-actively builds and sends relevant color information to neighboring processors (Lines 12 and 13). Further, to perform the coloring of its own vertices in a superstep, a processor first gathers color information from other processors to build a partial list of forbidden colors for each of its boundary vertices scheduled to be colored in the current superstep. After the processor has received the partial lists of forbidden colors from all of its neighboring processors, it merges these lists and augments them with local color information to determine a complete list of forbidden colors for its vertices scheduled to be colored in the current superstep. Using this information, the

Algorithm 3 Conflict detection phase of Algorithm 1 run on processor P_i .

```

1: function DETECTCONFLICTS
2:    $W_i \leftarrow \emptyset$  ▷  $W_i$  is the set of vertices  $P_i$  examines to detect conflicts
3:   for  $\ell \leftarrow 1$  to  $L$  do ▷ uses schedules computed in Algorithm 2
4:     for each  $w \in U_{i,\ell}$  where  $w$  has at least one neighbor in  $X_{i,\ell}$  do
5:        $W_i \leftarrow W_i \cup \{w\}$  ▷  $w$  is used for detecting type 1 conflicts
6:       for each  $w \in V_i$  where  $w$  has at least two neighbors in  $X_{i,\ell} \cup U_{i,\ell}$  on different
           processors do
7:          $W_i \leftarrow W_i \cup \{w\}$  ▷  $w$  is used for detecting type 2 conflicts
8:       for each  $j \in P = \{1, \dots, p\}$  do
9:          $R_{i,j} \leftarrow \emptyset$  ▷  $R_{i,j}$  is a set of vertices  $P_i$  notifies  $P_j$  to recolor
10:      for each  $w \in W_i$  do
11:         $encountered[color(w)] \leftarrow w$ 
12:         $lowest[color(w)] \leftarrow w$ 
13:        for each  $x \in N_1(w)$  do
14:          if  $encountered[color(x)] = w$  then
15:             $v \leftarrow lowest[color(x)]$ 
16:            if  $rand(v) \leq rand(x)$  then ▷  $rand(u)$ : random number assigned to  $u$ 
17:              if  $v \neq w$  then
18:                 $R_{i,Id(x)} \leftarrow R_{i,Id(x)} \cup \{x\}$  ▷  $Id(u)$ : ID of processor owning  $u$ 
19:              else
20:                 $R_{i,Id(v)} \leftarrow R_{i,Id(v)} \cup \{v\}$ 
21:                 $lowest[color(x)] \leftarrow x$ 
22:              else
23:                 $encountered[color(x)] \leftarrow w$ 
24:                 $lowest[color(x)] \leftarrow x$ 
25:          for each  $j \in P, j \neq i$  do
26:            Send  $R_{i,j}$  to processor  $P_j$ 
27:          for each  $j \in P, j \neq i$  do
28:            Receive  $R_{j,i}$  from processor  $P_j$ 
29:             $R_{i,i} \leftarrow R_{i,i} \cup R_{j,i}$ 
30:          return  $R_{i,i}$ 

```

processor then speculatively colors the vertices of the current superstep. At the end of the superstep, the new color information is sent to neighboring processors. These actions are performed in the piece of code in Lines 14–20. Regardless of whether a processor has finished coloring all of its vertices or not, it needs to receive updated color information from neighboring processors (see Line 21). This information is needed to enable the processor to compile forbidden color information to be sent to other processors in the next superstep.

4.4. The conflict detection phase. Algorithm 3 outlines the conflict detection routine DETECTCONFLICTS executed on each processor P_i . The routine has two major parts. In the first part, the routine finds a subset $W_i \subset V_i$ of vertices processor P_i needs to *examine* to detect both type 1 and type 2 conflicts. Whenever a conflict is detected, one of the involved vertices is selected to be recolored in the next round to resolve the conflict. The selection makes use of the random values assigned to boundary vertices in Algorithm 1. In the second part, the routine *determines* and returns a set of vertices to be recolored by processor P_i in the next round.

Two vertices would be involved in a conflict only if they are colored in the same superstep. Thus the vertex set W_i need only consist of (1) every vertex $v \in U_i$ that has at least one distance-1 neighbor on a processor $P_j, j \neq i$, colored in the

same superstep as v , and (2) every vertex $v \in V_i$ that has at least two distance-1 neighbors on different processors that are colored in the same superstep, since these might be assigned the same color. To obtain elements of the set W_i that satisfy one or both of these two conditions in an efficient manner, in Algorithm 3, relevant vertices on processor P_i are *traversed a superstep at a time*, using the schedule computed in Algorithm 2. In each superstep ℓ , first each vertex in $U_{i,\ell}$ and its neighboring boundary vertices are marked. Then, for each vertex $v \in X_{i,\ell}$ the vertices in the set $N_1(v)$ owned by processor P_i are marked. If this causes some vertex to be marked twice during the same superstep, then the vertex is added to W_i . The determination of the set W_i is achieved by the piece of code in Lines 3–7 of Algorithm 3; details are omitted for brevity.

Turning to the second part of Algorithm 3, processor P_i accumulates a list $R_{i,j}$ of vertices to be recolored by each processor P_j in the next round. To detect conflicts around a vertex w in the set W_i , we need to look for vertices in the set $N_1(w) \cup \{w\}$ that have the same color. In a valid distance-2 coloring, every vertex in the set $N_1(w) \cup \{w\}$ needs to have a distinct color. If several vertices with the same color are found, we let the vertex with the lowest random value keep its color and re-color the rest. To perform these tasks efficiently, we use two color-indexed, one-dimensional, arrays—*encountered* and *lowest*—that store vertices. The values stored in the two arrays encode information that is updated and used in a for-loop that iterates over each vertex $w \in W_i$. The context in each iteration in turn is a visit through the neighborhood $N_1(w)$ of the vertex w . For each vertex w , $encountered[c] = w$ indicates that at least one vertex in $N_1(w) \cup \{w\}$ having the color c has been encountered, and $lowest[c]$ stores the vertex with the lowest random value among these. Initially, both $encountered[color(w)]$ and $lowest[color(w)]$ are set to be w . This ensures that any conflict involving the vertex w and one of the vertices in the set $N_1(w)$ would be discovered. To detect conflicts involving the neighbors of w , the algorithm checks whether a given color used by a vertex in $N_1(w)$ has been encountered more than once, and if so, the vertex to be recolored is determined using the random values assigned to the vertices and the array *lowest* is updated accordingly. See the for-loop in Lines 10–24 for details.

In Algorithm 3, a type 1 conflict involving adjacent vertices is detected by both of the implied processors. Thus the if-test in Line 17 is included to avoid sending unnecessary notification from one processor to the other. Note also that in Line 13, it would have been sufficient to check for conflicts only using vertices $N_1'(w) \subseteq N_1(w)$ that belong to either U_i or X_i . However, since determining the subset $N_1'(w)$ takes more time than testing for a conflict, we use the larger set $N_1(w)$ in Line 13.

When the lists $R_{i,j}$ processor P_i accumulates are complete, processor P_i sends each list $R_{i,j}$ to processor P_j , $j \neq i$, to notify the latter to do the re-coloring (Lines 25–26). Processor P_i itself is responsible for re-coloring the vertices in $R_{i,i}$ and therefore adds to $R_{i,i}$ notifications $R_{j,i}$ received from each neighboring processor P_j (Lines 27–29).

4.5. Complexity. In Algorithm 2, the overall sequential work carried out by processor P_i and its neighboring processors in order to perform the coloring of the vertices in U_i is $O(\sum_{v \in U_i} d_2(v))$. Summing over all processors, the total work (excluding communication cost) involved in coloring the vertices in the set $U = \cup U_i$ is $O(\sum_{v \in U} d_2(v))$, which is equivalent to the complexity of a sequential algorithm for coloring the vertex set U .

Turning to the communication cost involved in Algorithm 2, note that for each

vertex $v \in U_i$, every neighboring processor sends to processor P_i the union of the colors used by vertices at exactly two edges from the vertex v , while the color of the vertex v is sent to every processor that owns a distance-1 neighbor of v . Thus the total size of data exchanged while coloring the vertex v is bounded by $O(d_2(v))$, which in turn gives the bound $O(\sum_{v \in U} d_2(v))$, where $U = \cup U_i$, on the overall communication cost of Algorithm 2.

In Algorithm 3, in determining the set W_i (in Lines 3–7), at most $|V_i|$ vertices are processed. The per-vertex work involved in this process is proportional to the degree-1 of the vertex. Hence, the time needed to determine W_i is bounded by $O(|V_i|\bar{d}_1)$, where \bar{d}_1 is the average degree-1 in the input graph \mathcal{G} . Further, in each iteration of the for-loop over the set W_i in Lines 10–24, the set of vertices to be recolored is determined in time $O(d_1(w))$, which gives a complexity of $O(|W_i|\bar{d}_1)$ for the entire for-loop. Since $|W_i| \leq |V_i|$ clearly holds, the overall complexity of Algorithm 3 is $O(|V_i|\bar{d}_1)$.

With the complexities of Algorithms 2 and 3 just established and assuming that the number of *rounds* required in Algorithm 1 is sufficiently small, the total work carried out by all of the p processors in coloring the input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is $O(|\mathcal{V}|\bar{d}_2)$, which is the same as the complexity of the sequential algorithm. The experimental results reported in §6 attest that the number of rounds for large-size graphs that arise in practice is indeed fairly small; the observed number was about half a dozen in most cases, and never more than a few dozens, while coloring graphs with millions of edges and employing as many as a hundred processors.

4.6. Variations. Algorithm 1 and its subroutines Algorithms 2 and 3 could be specialized along several axes to result in a variety of derived algorithms. We discuss three of these axes.

First, in Algorithm 1, interior vertices (I_i) are colored *before* boundary vertices (B_i), but the reverse order could also be considered.

Second, while coloring the vertices in a superstep on each processor (see Line 18 of Algorithm 2), the *natural* ordering of the vertices, a *random* ordering, or any other *degree-based ordering* technique could be used [14].

Third, the choice of a color for a vertex in a superstep (see Line 19 of Algorithm 2) could be done in several different ways. For example, a *First Fit* (FF), a *Staggered First Fit* (SFF), or a *randomized* coloring strategy could be used [5]. In the FF strategy, each processor chooses the *smallest* allowable color for a vertex, starting from color 1. In the SFF strategy, each processor P_i chooses the *smallest* permissible color from the set $\{\lceil \frac{iK}{p} \rceil, \dots, K\}$, where the initial estimate K is set to be, for example, equal to the lower-bound $\Delta + 1$ on the distance-2 chromatic number. If no such color exists, then the smallest permissible color in $\{1, \dots, \lfloor \frac{iK}{p} \rfloor\}$ is chosen. If there still is no such color, the smallest permissible color greater than K is chosen. Since the search for a color in SFF starts from different “base colors” for each processor, SFF is likely to result in fewer conflicts than FF.

5. Parallel Restricted Star and Partial Distance-2 Coloring. The algorithms presented in the previous section need to be modified only slightly to solve the two related problems of our concern, restricted star coloring on a general graph (for Hessian computation) and partial distance-2 coloring on a bipartite graph (for Jacobian computation). In this section we point out the specific changes that need to be made in Algorithms 1–3 to address these two problems.

Algorithm 4 Basic difference between distance-2 and partial distance-2 coloring.

<pre> procedure D2COLORING($\mathcal{G} = (\mathcal{V}, \mathcal{E})$) for each $v \in \mathcal{V}$ do for each $w \in N_1(v)$ do Forbid $color(w)$ to v for each $x \in N_1(w)$ do Forbid $color(x)$ to v Assign a color to vertex v </pre>	<pre> procedure PD2COLORING($\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E})$) for each $v \in \mathcal{V}_2$ do for each $w \in N_1(v)$ do $w \in \mathcal{V}_1$ never receives a color for each $x \in N_1(w)$ do Forbid $color(x)$ to v $\triangleright x \in \mathcal{V}_2$ Assign a color to vertex v </pre>
--	---

5.1. Restricted star coloring. As the definition given in §2.2 implies, in a restricted star coloring of a graph, the color assigned to a vertex v needs to satisfy conditions that concern the distance-2 neighbors $N_2(v)$ of the vertex v . The exact same neighborhood is consulted in assigning a color for the vertex v in a distance-2 coloring of the graph. Therefore, the greedy algorithms for the two variants of coloring (as developed in [14]) differ only in the way the set of forbidden colors for the vertex v is determined. We describe this difference below in the sequential setting; the additional differences that arise in the parallel setting are fairly straightforward to implement and their discussion is omitted.

Consider the step of a greedy algorithm (distance-2 or restricted star coloring) in which the vertex v is to be colored, and let v, w, x be a path in the distance-2 neighborhood of the vertex v . In the distance-2 coloring algorithm, both $color(w)$ and $color(x)$ are forbidden to the vertex v , since the path needs to have three distinct colors. In the restricted star coloring algorithm, on the other hand, $color(w)$ would always be forbidden to v , whereas $color(x)$ may or may not be forbidden. The decision in the latter case is made based on further tests:

- If $color(w) = 0$ (i.e., vertex w is not yet colored), then $color(x)$ is forbidden to v . This ensures that any extension v, w, x, y of the path v, w, x would end up using at least three colors, as desired, since in the step in which the vertex w is colored, the distance-1 coloring requirement guarantees that the vertex w gets a color distinct from $color(v)$ and $color(x)$.
- If $color(w) \neq 0$, then $color(x)$ is forbidden to v only if $color(w) > color(x)$.

5.2. Partial distance-2 coloring. Here, the input is a bipartite graph $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E})$, and only the vertex set \mathcal{V}_2 needs to be colored satisfying the condition that a pair of vertices from \mathcal{V}_2 sharing a common distance-1 neighbor in \mathcal{V}_1 receive different colors. If not already distributed, the graph \mathcal{G}_b needs to be partitioned among the processors in a manner that minimizes boundary vertices (and hence the likelihood of conflicts and the size of overall communication). Assuming a vertex partitioning is used, let $V_{1,1}, \dots, V_{1,p}$ denote the partitioning of the set \mathcal{V}_1 , and let $V_{2,1}, \dots, V_{2,p}$ denote the partitioning of the set \mathcal{V}_2 . The subgraph assigned to processor P_i would then be $G_{b,i} = (V_{1,i}, V_{2,i}, E_i)$, where E_i is the set of edges incident on vertices in $V_{1,i} \cup V_{2,i}$. In terms of the underlying matrix, such a partitioning means that each processor owns a subset of the rows as well as a subset of the columns; this is in contrast with the case where either only the columns or only the rows are partitioned. With such a partitioning in place, the only change that needs to be made in Algorithm 1 is to replace the two references to V_i (in Lines 4 and 5) with $V_{2,i}$.

The changes that need to be made in Algorithms 2 and 3 are minimal as well, and they are consequences of the basic difference between distance-2 coloring in a general graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and partial distance-2 coloring in a bipartite graph $\mathcal{G}_b = (\mathcal{V}_1, \mathcal{V}_2, \mathcal{E})$ illustrated by the code fragment in Algorithm 4.

TABLE 6.1

Structural properties of the application (top), and the synthetic (bottom) test graphs used in the experiments. The last column shows the number of edges in $\mathcal{G}^2 = (\mathcal{V}, \mathcal{F})$ compared to $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. (*ST*–Structural Engineering [26], *SH*–Ship Section [28], *CA*–Linear Car Analysis [28], *AU*–Automotive [26], *CE*–Civil Engineering [26]).

app/ class	name	\mathcal{V}	\mathcal{E}	Degree		colors		exec. time (s)		$\frac{ \mathcal{F} }{ \mathcal{E} }$
				max	avg	d1	d2	d1	d2	
ST	nasasrb	54,870	1,311,227	275	48	41	276	0.049	2.237	3.2
	ct20stif	52,329	1,323,067	206	51	49	210	0.063	2.581	3.8
	pwtck	217,918	5,708,253	179	52	48	180	0.229	10.335	2.9
SH	shipsec8	114,919	3,269,240	131	57	54	150	0.128	6.776	3.5
	shipsec1	140,874	3,836,265	101	55	48	126	0.143	7.457	3.1
	shipsec5	179,860	4,966,618	125	55	50	140	0.190	9.852	3.2
CA	bmw7st1	141,347	3,599,160	434	51	54	435	0.167	6.730	3.3
	bmw3_2	227,362	5,530,634	335	49	48	336	0.274	10.077	3.2
	inline1	503,712	18,156,315	842	72	51	843	0.925	55.217	7.0
AU	hood	220,542	5,273,947	76	48	42	103	0.277	9.407	3.2
	msdoor	415,863	9,912,536	76	48	42	105	0.520	17.438	3.2
	ldoor	952,203	22,785,136	76	48	42	112	1.197	40.180	3.2
CE	pkustk10	80,676	2,114,154	89	52	42	126	0.091	3.904	2.9
	pkustk11	87,804	2,565,054	131	58	66	198	0.103	6.041	4.2
	pkustk13	94,893	3,260,967	299	69	57	303	0.155	9.302	6.0
planar	plan-1	4,072,937	12,218,805	40	6	9	41	3.435	18.880	3.4
random	rand-1	400,000	2,002,202	27	10	9	41	0.644	6.676	11
random	rand-2	400,000	4,004,480	45	20	12	101	1.242	21.977	21
s. world	sw-1	400,000	1,998,542	468	10	18	469	0.345	13.909	31
s. world	sw-2	400,000	3,993,994	880	20	27	882	0.632	50.954	59

6. Experiments. In this section we present results on experimental evaluation of the performance of the algorithms presented in the previous two sections. The algorithms are implemented in C using the MPI message-passing library.

6.1. Experimental setup.

Test platform. The experiments are carried out on a 96-node PC cluster equipped with dual 2.4 GHz Intel P4 Xeon CPUs and 4 GB memory. The nodes of the cluster are interconnected via a switched 10Gbps Infiniband network.

Test graphs. Our first test set consists of 15 *real-world* graphs obtained from five different application areas: structural engineering, ship section design, linear car analysis, automotive design, and civil engineering [12, 24, 26, 28]. The top part of Table 6.1 lists the number of vertices, number of edges, maximum degree-1, and average degree-1 in each of these test graphs. The graphs are classified according to the application area they are drawn from. Table 6.1 also lists the number of colors and the execution time in seconds used by greedy sequential distance-1 and distance-2 coloring algorithms when each is run on a single node of our test platform. The last column lists the ratio between the number of edges in the square graph \mathcal{G}^2 and the number of edges in \mathcal{G} . These computed quantities will be used to gauge the performance of the proposed parallel coloring algorithms on the input graphs \mathcal{G} .

To be able to study the scalability of the proposed algorithms on a wider range of graph structures, we have also run experiments on *synthetically* generated graphs, which constitute our second test set. To represent extreme cases, the synthetic graphs considered included instances drawn from three different graph classes: *random*, *planar* and *small-world* graphs. (Loosely speaking, a small-world graph is a graph in which the distance between any pair of non-adjacent vertices is fairly small.) The

random and small-world graphs are generated using the GTgraph Synthetic Graph Generator Suite [27]. The planar graphs are maximally planar—the degree of every vertex is at least five—and are generated using the expansion method described in [21]; the generator is a part of the graph suite for the Second DIMACS Challenge [25]. The structural properties of these graphs as well as the number of colors and runtime used by sequential distance-1 and distance-2 coloring algorithms run on them are listed in the bottom part of Table 6.1.

In the runtimes reported in Table 6.1 as well as in later figures in this section, each individual test is an average of five runs. In the timing of the parallel coloring code, we assume that the graph is initially partitioned and distributed among the nodes of the parallel machine, and thus the times reported concern only coloring. In all of the experiments, the input graph is partitioned using the tool MeTiS [17].

6.2. Performance of the sequential algorithms. Before we proceed with evaluating the performance of the proposed parallel coloring algorithms, it is worthwhile to observe that the underlying sequential greedy algorithms, in both the distance-1 and the distance-2 coloring cases, performed remarkably well in terms of number of colors used on both the application and synthetic test graphs. Specifically, as Table 6.1 shows, the number of colors used by the greedy sequential distance-1 coloring algorithm is in most cases slightly below the average degree and far below the maximum degree, which is an upper bound on the optimal solution (the distance-1 chromatic number). Even more remarkably, the number of colors used by the greedy sequential distance-2 coloring algorithm in most cases is observed to be very close to the maximum degree, which is a lower bound on the optimal solution (the distance-2 chromatic number). Thus the solution obtained by the greedy distance-2 coloring algorithm is in most cases either optimal or just a few colors more. In both of these greedy algorithms, the vertices were colored in the natural *order* they appear in the input graphs.

6.3. Results on parallel distance-2 coloring. In this section we present detailed experimental results on the parallel distance-2 coloring algorithm for general graphs discussed in §4. In §6.4 we shall present results on the related algorithms, restricted star coloring on general graphs and partial distance-2 coloring on bipartite graphs. In §6.6 we present results on the alternative approach of obtaining a distance-2 coloring by distance-1 coloring the square graph.

6.3.1. Choice of superstep size. Our first set of experiments, the results of which is given in Figure 6.1, is conducted to study the dependence of the performance of Algorithm 1 on the choice of the superstep size s , the number of vertices colored in a superstep before communication takes place. The experiments are conducted using the *application* test graphs and the reported results are *averages* over each class, while using 32 processors. We obtained similar results while experimenting with various other number of processors, but we report only for 32 processors for space considerations.

Figure 6.1(a) shows a plot of the number of conflicts, normalized relative to the total number of vertices in the graph, as a function of superstep size. As one would expect, the normalized number of conflicts and the number of rounds (not shown here) increased as the superstep size s was increased, but the rate of increase remained fairly low once the value of s passed a few hundred. Figure 6.1(b) shows that, for values of s above a few hundred, further increase in s does not significantly influence speedup. Similarly, our experiments (not shown here) showed that the number of colors does

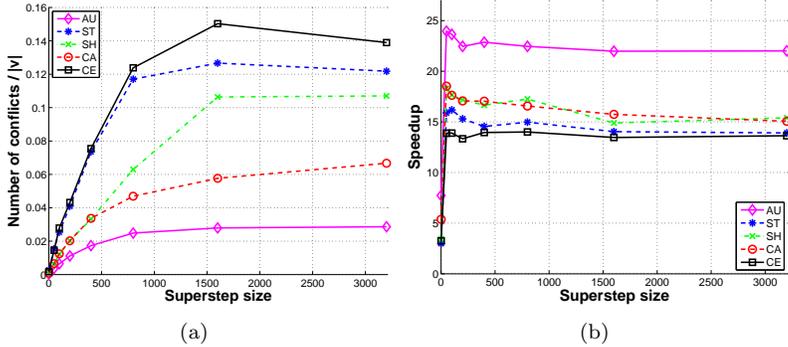


FIG. 6.1. Normalized number of conflicts and speedup, while varying superstep size s for $p = 32$.

not vary significantly with s , and stays within 12% of the number of colors used by the sequential algorithm. In general, an “optimal” value for s is likely to depend on both the properties of the graph being colored (size and density) and the platform on which the algorithm is executed. Based on observations from our parallel experiments, we used a superstep size of 100 for the scalability studies we report on in the rest of this section.

6.3.2. Strong scaling results. Our second set of experiments is concerned with the strong scalability of the parallel distance-2 coloring algorithm as the number of processors used is varied. This set of experiments was conducted on both the application and synthetic test graphs. The results from the experiments on the application graphs are summarized in Figure 6.2, and those from the synthetic graphs are summarized in Figure 6.3. In the results shown in Figure 6.2, the *largest* graph from each application category was used. A more detailed set of results on *all* of the application test graphs for the distance-2 as well as the restricted star coloring algorithm is included in a table in the Appendix.

Results on application test graphs. The results in Figures 6.2(a) and 6.2(b) show that the number of conflicts and the number of rounds increase with increasing number of processors. However, the rate of growth as well as the actual values of both of these quantities was observed to be fairly small for all graphs except `pkustk13` and `inline1`, the densest graphs in our test set. The reason for the relatively large number of rounds required in coloring the graph `inline1` is the existence of vertices with very large degree-1. This phenomenon will be explained further with the help of synthetic graphs shortly.

The metrics number of conflicts and number of rounds are in some sense “intermediate” performance measures for our parallel coloring algorithms. They are included in the reports to help explain results on the two ultimate performance metrics—speedup and number of colors used. The lower row of Figure 6.2 shows results on the latter two metrics. We will use a similar set of four metrics in several other experiments reported in this section.

The speedup results in Figure 6.2(c) demonstrate that our algorithm in general scales well with increasing number of processors. As expected, the obtained speedup is relatively poorer in the cases where the number of conflicts and the number of rounds is relatively higher. The proposed algorithm also performed well in terms of the number of colors used. The results in Figure 6.2(d) show that the number of

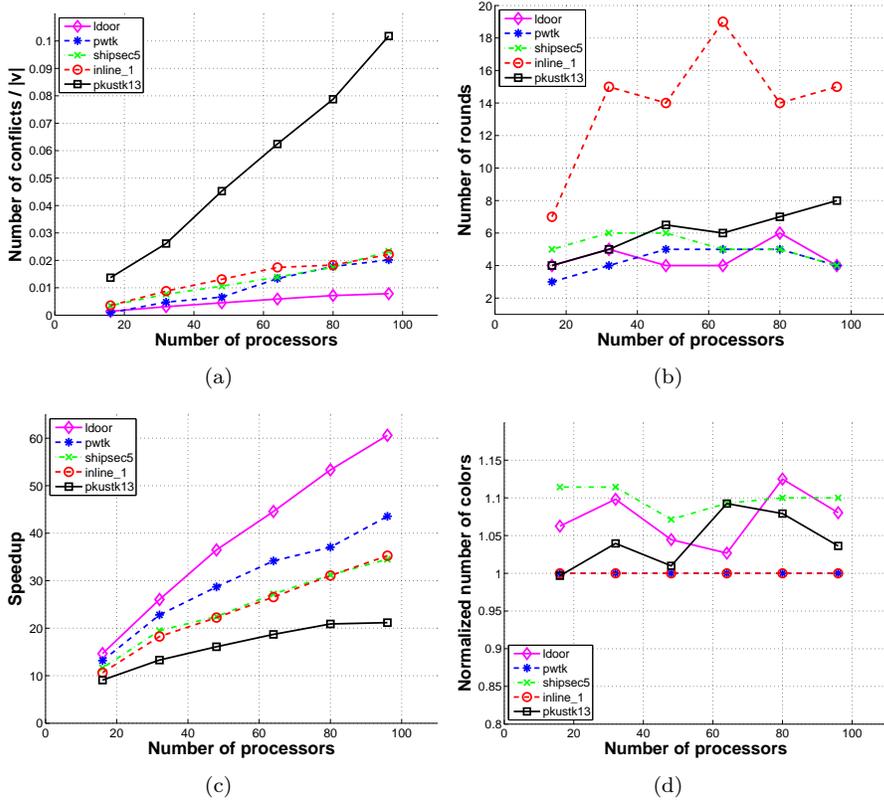


FIG. 6.2. Normalized number of conflicts (a), number of rounds (b), speedup (c), and normalized number of colors (d) while varying the number of processors for the application test graphs. In all cases, superstep size $s = 100$ was used.

colors used by our parallel algorithm, when as many as 96 processors are used, is at most 12% more than that used by the sequential algorithm. Recall that the solution obtained by the sequential algorithm in most cases is nearly optimal.

Results on synthetic test graphs. We now turn attention to results on the synthetic graphs, which are designed to capture “extreme cases”. The planar graph plan-1 represents extremely well partition-able graphs—almost all of the vertices of plan-1 were interior vertices in a partition obtained using the tool MeTiS [17]. For such graphs parallel speedup in the context of Algorithm 1 comes largely from graph partitioning (data distribution) as opposed to the iterative coloring part. As Figure 6.3 shows this results in very small number of conflicts and good speedup.

The random graphs rand-1 and rand-2 represent the opposite extreme—almost all of the vertices in these graphs are boundary regardless of how the graph is partitioned. As expected, Figure 6.3 shows that the number of conflicts is considerably larger for the random graphs. Nonetheless, our algorithm achieved some speedup (a constant around 10) even under such an extreme case. Note that the speedup in this case comes solely from the iterative coloring part.

As mentioned earlier, graphs having vertices with very large degrees comprise a particularly difficult set of instances for the proposed parallel distance-2 coloring algorithm. Resolving conflicts involving the neighbors of such vertices requires a

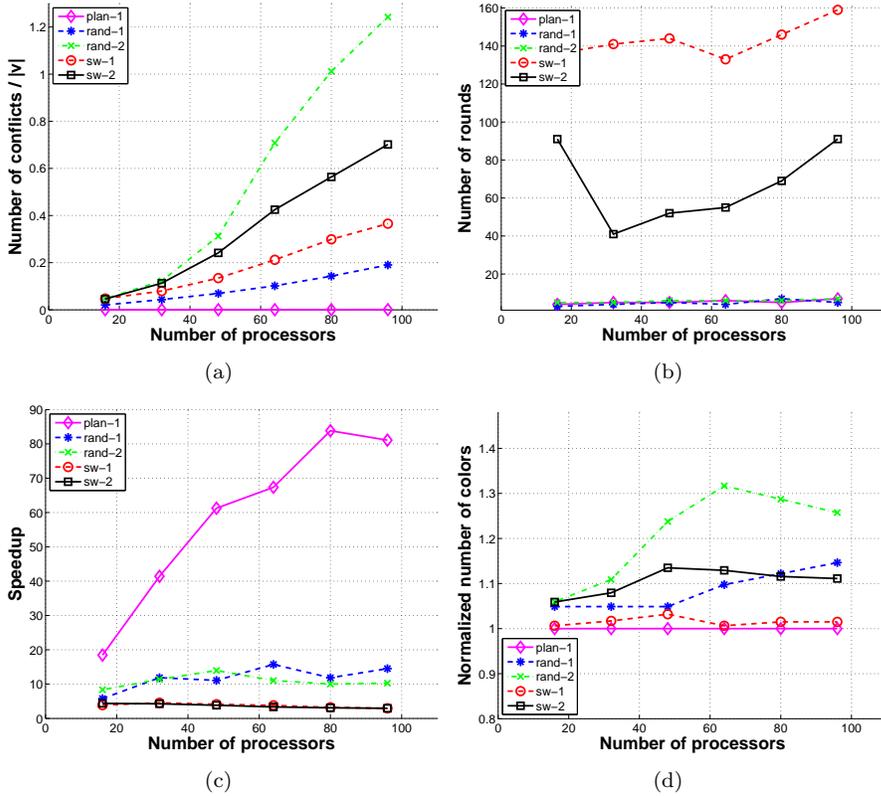


FIG. 6.3. Normalized number of conflicts (a), number of rounds (b), speedup (c), and normalized number of colors (d) while varying the number of processors for the synthetic test graphs. In all cases, superstep size $s = 100$ was used.

large number of rounds which in turn degrades speedup. The third category in our synthetic graphs, the small-world graphs sw-1 and sw-2, are included to represent such a class of pathological instances for our approach. As Figure 6.3 shows, even though the number of conflicts for small-world graphs is not necessarily larger than that for random graphs with similar sizes, coloring small-world graphs required significantly larger number of rounds. Consequently, the speedup achieved was poorer—it was around 3 for all numbers of processors we experimented with.

As can be seen from Figure 6.3(d), the normalized number of colors for all the synthetic graphs except rand-2 is within 12% of the sequential algorithm. For rand-2, there are relatively many conflicts but few rounds. This entails that many processors recolor a fairly large number of vertices in each round to resolve the conflicts. Over multiple rounds, this results in a significant increase in the number of colors used. This is in contrast to small world graphs, where many fewer vertices are recolored in many more rounds. This resembles sequential coloring, hence the number of colors increases only slightly.

6.3.3. Weak scaling. Our next set of experimental results is on weak scalability, where both problem size and number of processors are increased in proportion so as to result in nearly constant runtime. The experiments were conducted on five instances from each of random and planar graph classes. The structural properties of these

name	\mathcal{V}	\mathcal{E}	Degree	
			max	avg
plan-2	1,625,972	4,877,910	37	6
plan-3	3,250,939	9,752,811	43	6
plan-4	4,888,479	14,665,431	43	6
plan-5	6,513,589	19,540,761	38	6
plan-6	8,150,267	24,450,795	39	6
rand-3	160,000	1,600,528	45	20
rand-4	320,000	3,201,327	46	20
rand-5	480,000	4,803,946	43	20
rand-6	640,000	6,403,242	44	20
rand-7	800,000	8,005,505	45	20

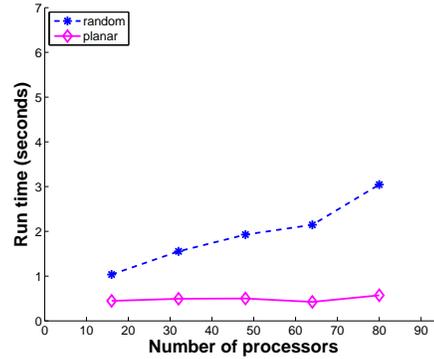


FIG. 6.4. Properties of the synthetic test graphs used in weak scaling tests (left) and results for distance-2 coloring (right).

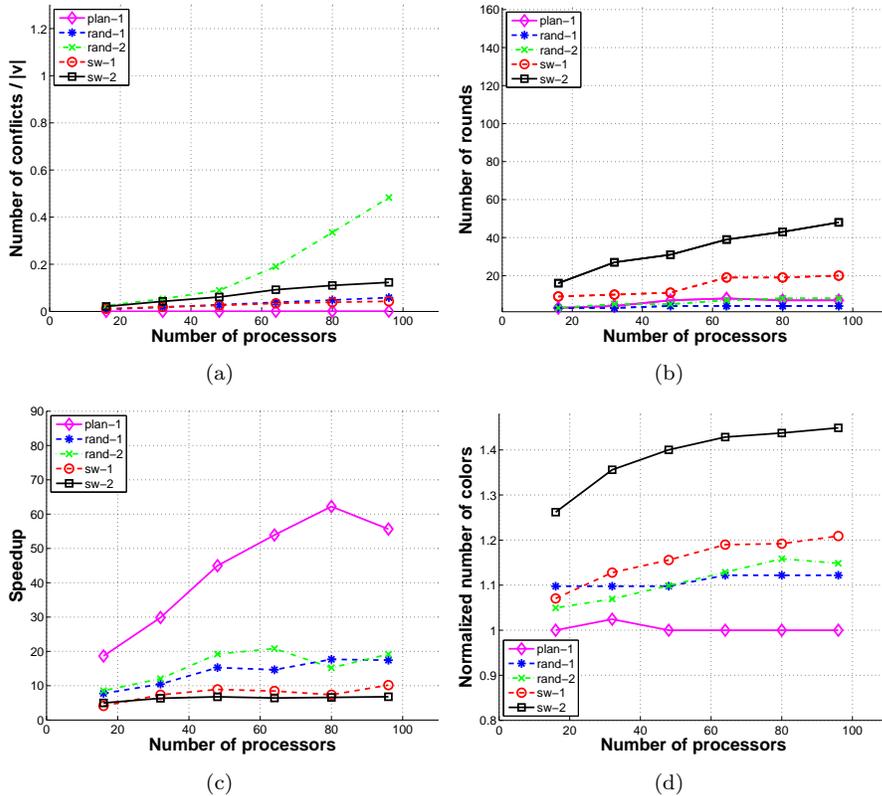


FIG. 6.5. Normalized number of conflicts (a), number of rounds (b), speedup (c), and normalized number of colors (d) while using SFF color selection strategy and varying the number of processors for the synthetic graphs in Table 6.1. Superstep size $s = 100$.

graphs and the experimental results are summarized in Figure 6.4. It can be seen that the algorithm behaved almost ideally for planar graphs—the runtime remained nearly constant as the problem size and the number of processors were increased in proportion. For the random graphs, the runtime grew, but it did so fairly slowly.

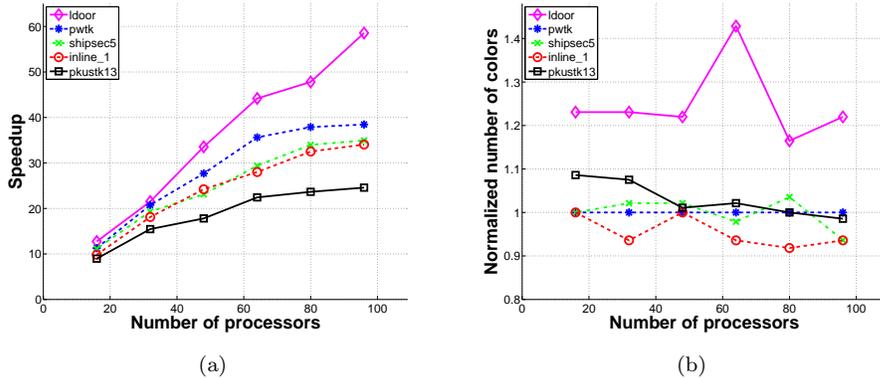


FIG. 6.6. Speedup (a) and normalized number of colors (b) for the restricted star coloring algorithm.

6.3.4. Variations of the algorithm. In the results presented thus far in this section, a variant of the proposed distance-2 coloring algorithm with the following combination of parameters was used: interior vertices were colored *before* boundary vertices; the *natural* ordering of the vertices was used while coloring; and a *FF* color choice strategy was employed. In addition to this “default” variant, we experimented with *seven* other variations in which interior vertices are colored *after* boundary vertices, a *random* vertex ordering is used while coloring, and a *SFF* color choice strategy (as discussed in §4.6) is used. The experiments showed that the only option that resulted in better performance in the majority of the test cases compared to the default variant was the use of the SFF coloring strategy. Figure 6.5 shows the performance of the SFF-coloring based variant on the synthetic test graphs. Results on the application graphs are omitted as no significant improvement over the FF-based variant was observed.

As Figure 6.5 shows, the performance improvement with SFF was especially significant for the random and small-world graphs. The number of conflicts and the number of rounds were much smaller for these graphs when SFF is used instead of FF; hence better speedup was achieved. For small-world graphs, the SFF strategy required significantly more colors than FF, whereas for random graphs SFF required about the same number of colors for the random graph rand-1, and fewer colors for the denser random graph rand-2.

6.4. Results on parallel restricted star coloring. As discussed in §5, a parallel restricted star coloring of a graph can be obtained via a slight modification of the parallel distance-2 coloring algorithm. We have performed the necessary modifications and carried out experiments on the scalability of the resulting restricted star coloring algorithm. Figure 6.6(a) shows speedup results obtained when the restricted star coloring algorithm was run on the largest graph from each of the application graph classes listed in Table 6.1. As expected, these speedup results are very similar to those for distance-2 coloring (Figure 6.2(c)). The normalized number of colors used by the restricted star coloring algorithms are given in Figure 6.6(b). As results in the table in the appendix show, for many of the test graphs, restricted star coloring gives fewer colors than distance-2 coloring, demonstrating the advantage of exploiting symmetry.

TABLE 6.2

Structural properties of the additional test graphs [26] used for partial distance-2 coloring and experimental results.

name	$ \mathcal{V}_1 $	$ \mathcal{V}_2 $	$ \mathcal{E} $	sequential		speedup/num. of colors		
				cols.	time	p=16	p=48	p=96
lhr71c	70,304	70,304	1,528,092	65	1.43	11.3/66	13.8/67	6.8/66
stormG2	528,185	1,377,306	3,459,881	48	0.85	7.5/48	7.1/48	3.0/48
cont11_l	1,468,599	1,961,394	5,382,999	8	0.52	5.2/12	6.9/13	5.3/13
cage13	445,315	445,315	7,479,343	118	2.27	4.6/114	4.2/119	2.7/113
cage14	1,505,785	1,505,785	27,130,349	136	8.86	3.4/129	4.9/131	3.5/130

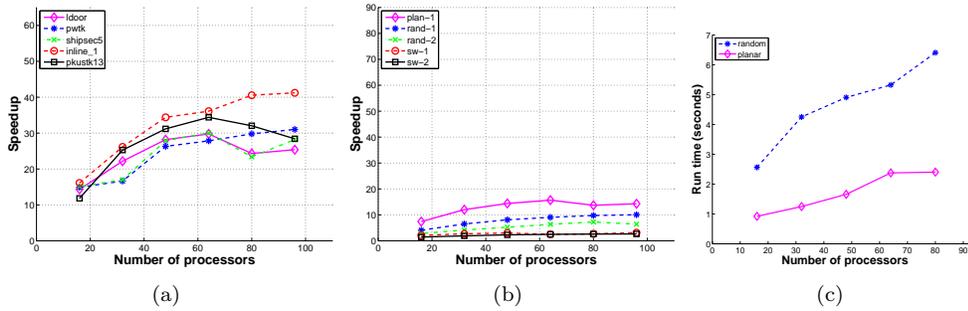


FIG. 6.7. Speedup for distance-1 coloring of \mathcal{G}^2 on the application test graphs (a) and the synthetic test graphs (b) listed in Table 6.1. Weak scaling for distance-1 coloring of \mathcal{G}^2 on the synthetic graphs given in Figure 6.4 (c).

6.5. Results on parallel partial distance-2 coloring of bipartite graphs.

We tested the partial distance-2 coloring algorithm on bipartite graphs of five non-symmetric matrices obtained from [26]. Structural properties of these graphs as well as the sequential and parallel partial distance-2 coloring results are given in Table 6.2. In each test case except for cont11_l, the number of colors used by the parallel algorithm was observed to be within 2% of the sequential algorithm. For cont11_l, the number of colors increased from 8 up to 13 when using the parallel algorithm. The speedup here is poorer than the distance-2 coloring case on general graphs in part because the partitioner is less suitable for bipartite graphs.

6.6. Results on parallel distance-1 coloring of \mathcal{G}^2 . As discussed in §2, the distance-2 coloring problem on a graph \mathcal{G} can be solved by constructing and then distance-1 coloring the square graph \mathcal{G}^2 . This alternative method has the same overall asymptotic time complexity as a greedy distance-2 coloring on the graph \mathcal{G} , but the actual runtimes of the two approaches could differ substantially.

We have done experiments on this alternative approach using the graphs listed in Table 6.1. For these graphs, the ratio of the number of edges in \mathcal{G}^2 to that in \mathcal{G} is listed in the last column of Table 6.1. As one can see from the table, the storage requirement for \mathcal{G}^2 could be up to 7 times larger than what is needed for \mathcal{G} for the application graphs, and up to 59 times larger for the synthetic graphs.

To construct \mathcal{G}^2 in parallel, each processor requires the adjacency lists of distance-1 neighbors of its boundary vertices. This requires communication between processors similar to the forbidden color communication in a single round of the proposed parallel distance-2 coloring algorithm. However, instead of a union of colors, adjacency lists are exchanged. Thus the communication cost is larger.

Figure 6.7 shows strong and weak scaling results of the approach based on distance-1 coloring of \mathcal{G}^2 . The timing for this method includes parallel construction of \mathcal{G}^2 and parallel distance-1 coloring of \mathcal{G}^2 using the algorithm presented in [5]. While computing speedup, the sequential greedy distance-2 coloring algorithm is used as the reference. Therefore, the results in Figures 6.7(a) and 6.7(b) are directly comparable to those in Figures 6.2(c) and 6.3(c), respectively. The results show that the alternative approach discussed in this section is slower and scales worse than the proposed parallel distance-2 coloring algorithm.

7. Conclusion. We have presented efficient distributed-memory parallel algorithms for three closely related coloring problems that arise in the efficient computation of sparse Jacobian and Hessian matrices using automatic differentiation or finite differencing. The problems are distance-2 coloring of general graphs, restricted star coloring of general graphs, and partial distance-2 coloring of bipartite graphs. The scalability of the proposed algorithms has been demonstrated on a variety of application as well as synthetic test graphs. MPI implementations of the algorithms have been made available to the public through the Zoltan library.

As the experimental results showed, the number of rounds required in coloring graphs in which a small fraction of the vertices are of relatively very large degree (e.g. small world graphs) could be fairly high (in the order of a hundred). A separate treatment of such vertices could improve the performance of the proposed algorithm. We plan to explore this in a future work.

Acknowledgments. We thank the anonymous referees for their valuable comments, which helped us improve the presentation of this paper.

REFERENCES

- [1] G. AGNARSSON, R. GREENLAW, AND M.M. HALLDÓRSSON, *On powers of chordal graphs and their colorings*, Congr. Numer., 100 (2000), pp. 41–65.
- [2] G. AGNARSSON AND M.M. HALLDÓRSSON, *Coloring powers of planar graphs*, SIAM J. Discr. Math., 16 (2003), pp. 651–662.
- [3] R.H. BISSELING, *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*, Oxford, 2004.
- [4] D. BOZDAĞ, U. CATALYUREK, A.H. GEBREMEDHIN, F. MANNE, E.G. BOMAN, AND F. ÖZGÜNER, *A parallel distance-2 graph coloring algorithm for distributed memory computers*, in Proceedings of HPCC 2005, vol. 3726 of Lecture Notes in Computer Science, Springer, 2005, pp. 796–806.
- [5] D. BOZDAĞ, A. H. GEBREMEDHIN, F. MANNE, E. G. BOMAN, AND U. V. CATALYUREK, *A framework for scalable greedy coloring on distributed-memory parallel computers*, Journal of Parallel and Distributed Computing, 68 (2008), pp. 515–535.
- [6] U. V. CATALYUREK AND C. AYKANAT, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Transactions on Parallel and Distributed Systems, 10 (1999), pp. 673–693.
- [7] T. F. COLEMAN AND J. J. MORE, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. Numer. Anal., 1 (1983), pp. 187–209.
- [8] T. F. COLEMAN AND J. J. MORÉ, *Estimation of sparse Hessian matrices and graph coloring problems*, Math. Program., 28 (1984), pp. 243–270.
- [9] A. R. CURTIS, M. J. D. POWELL, AND J. K. REID, *On the estimation of sparse Jacobian matrices*, J. Inst. Math. Appl., 13 (1974), pp. 117–119.
- [10] K. DEVINE, E. BOMAN, R. HEAPHY, B. HENDRICKSON, AND C. VAUGHAN, *Zoltan data management services for parallel dynamic applications*, Computing in Science and Engineering, 4 (2002), pp. 90–97.
- [11] A.H. GEBREMEDHIN, F. MANNE, AND A. POTHEN, *What color is your Jacobian? Graph coloring for computing derivatives*, SIAM Rev., 47 (2005), pp. 629–705.

- [12] A. H. GEBREMEDHIN AND F. MANNE, *Scalable parallel graph coloring algorithms*, *Concurrency: Practice and Experience*, 12 (2000), pp. 1131–1146.
- [13] A. H. GEBREMEDHIN, F. MANNE, AND A. POTHEN, *Parallel distance- k coloring algorithms for numerical optimization*, in proceedings of Euro-Par 2002, vol. 2400, Lecture Notes in Computer Science, Springer, 2002, pp. 912–921.
- [14] A. H. GEBREMEDHIN, A. TARAFDAR, F. MANNE, AND A. POTHEN, *New acyclic and star coloring algorithms with application to computing Hessians*, *SIAM J. Sci. Comput.*, 29 (2007), pp. 1042–1072.
- [15] D. HYSOM AND A. POTHEN, *A scalable parallel algorithm for incomplete factor preconditioning*, *SIAM J. Sci. Comput.*, 22 (2001), pp. 2194–2215.
- [16] M.T. JONES AND P.E. PLASSMANN, *Scalable iterative solution of sparse linear systems*, *Parallel Computing*, 20 (1994), pp. 753–773.
- [17] G. KARYPIS AND V. KUMAR, *A fast and high quality multilevel scheme for partitioning irregular graphs*, *SIAM J. Sci. Comput.*, 20 (1999).
- [18] S. O. KRUMKE, M.V. MARATHE, AND S.S. RAVI, *Models and approximation algorithms for channel assignment in radio networks*, *Wireless Networks*, 7 (2001), pp. 575 – 584.
- [19] V. S. A. KUMAR, M. V. MARATHE, S. PARTHASARATHY, AND A. SRINIVASAN, *End-to-end packet-scheduling in wireless ad-hoc networks*, in SODA 2004: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, 2004, pp. 1021–1030.
- [20] S. T. MCCORMICK, *Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem*, *Math. Programming*, 26 (1983), pp. 153 – 171.
- [21] C. A. MORGENSTERN AND H. D. SHAPIRO, *Heuristics for rapidly four-coloring large planar graphs.*, *Algorithmica*, 6 (1991), pp. 869–891.
- [22] M. J. D. POWELL AND P. L. TOINT, *On the estimation of sparse Hessian matrices*, *SIAM J. Numer. Anal.*, 16 (1979), pp. 1060–1074.
- [23] Y. SAAD, *ILUM: A multi-elimination ILU preconditioner for general sparse matrices*, *SIAM J. Sci. Comput.*, 17 (1996), pp. 830–847.
- [24] M. M. STROUT AND P. D. HOVLAND, *Metrics and models for reordering transformations*, in Proceedings of the Second ACM SIGPLAN Workshop on Memory System Performance, 2004, pp. 23–34.
- [25] *The Second DIMACS Challenge*. <http://mat.gsia.cmu.edu/challenge.html>.
- [26] *University of Florida sparse matrix collection*. Available at <http://www.cise.ufl.edu/research/sparse/matrices/>.
- [27] *GTgraph: A suite of synthetic graph generators*. <http://www-static.cc.gatech.edu/~kamesh/GTgraph/>.
- [28] *Test data from the Parasol project*. Available at <http://www.parallab.uib.no/projects/parasol/data/>.

Appendix A.

TABLE A.1

Results for parallel distance-2 coloring and restricted star coloring for the application graphs using $s = 100$. In columns 6 and 10, the number of colors used by the respective sequential greedy coloring algorithm is given in parentheses. Abbreviations: normalized (*norm*), number (*nr*), conflicts (*cflicts*), rounds (*rnds*), speedup (*spdup*), colors (*cols*), number of processors (*p*).

p	graph name	distance-2 coloring				restricted star coloring			
		norm. cflicts	nr. of rnds	spdup	nr. of cols	norm. cflicts	nr. of rnds	spdup	nr. of cols
16	nasasrb	1.21%	5	10.1	276 (276)	1.98%	4	9.8	276 (100)
	ct20stif	1.59%	6	8.8	210 (210)	1.32%	4	9.3	177 (210)
	pwtk	0.09%	3	13.2	180 (180)	0.66%	4	11.3	180 (180)
	shipsec8	0.87%	5	10.2	158 (150)	0.95%	4	10.8	158 (144)
	shipsec1	0.85%	4	11.4	141 (126)	0.74%	4	11.0	126 (126)
	shipsec5	0.35%	5	11.6	156 (140)	0.56%	4	11.2	141 (141)
	bmw7st_1	0.85%	12	11.5	435 (435)	0.66%	3	11.8	432 (408)
	bmw3.2	0.54%	4	11.2	336 (336)	0.64%	4	12.0	336 (204)
	inline_1	0.36%	7	10.6	843 (843)	0.26%	4	9.8	843 (843)
	hood	0.69%	4	13.4	108 (103)	0.67%	4	11.6	105 (79)
	msdoor	0.31%	4	14.0	112 (105)	0.29%	4	12.5	105 (98)
	ldoor	0.14%	4	14.7	119 (112)	0.14%	4	12.7	112 (91)
	pkustk10	1.28%	5	11.0	127 (126)	0.81%	4	10.1	120 (114)
pkustk11	1.49%	5	9.4	219 (198)	0.75%	4	8.5	183 (174)	
pkustk13	1.37%	4	9.1	302 (303)	0.95%	4	9.0	303 (279)	
48	nasasrb	6.00%	24	8.9	276 (276)	4.87%	10	13.8	189 (100)
	ct20stif	6.21%	9	12.9	210 (210)	4.42%	5	15.7	146 (210)
	pwtk	0.66%	5	28.7	180 (180)	0.75%	4	27.7	180 (180)
	shipsec8	2.14%	4	21.4	161 (150)	1.76%	4	20.9	140 (144)
	shipsec1	2.09%	5	22.2	134 (126)	1.79%	4	23.2	126 (126)
	shipsec5	1.06%	6	22.4	150 (140)	1.06%	4	23.1	144 (141)
	bmw7st_1	2.44%	12	20.2	435 (435)	1.95%	4	23.7	435 (408)
	bmw3.2	1.51%	12	24.3	336 (336)	1.17%	4	27.3	336 (204)
	inline_1	1.31%	14	22.2	843 (843)	0.63%	4	24.2	843 (843)
	hood	1.67%	5	30.8	113 (103)	1.73%	5	27.8	112 (79)
	msdoor	0.85%	5	29.0	116 (105)	0.89%	4	28.9	108 (98)
	ldoor	0.45%	4	36.5	117 (112)	0.46%	4	33.5	111 (91)
	pkustk10	3.78%	5	17.4	133 (126)	3.02%	5	21.1	123 (114)
pkustk11	3.84%	5	16.8	209 (198)	3.02%	4	18.4	182 (174)	
pkustk13	4.53%	6	16.1	306 (303)	2.19%	5	17.8	282 (279)	
96	nasasrb	10.17%	23	6.7	276 (276)	7.90%	9	12.2	131 (100)
	ct20stif	14.00%	15	9.2	210 (210)	11.26%	6	15.8	156 (210)
	pwtk	2.02%	4	43.6	180 (180)	2.19%	5	38.4	180 (180)
	shipsec8	4.43%	5	28.4	166 (150)	4.53%	5	28.1	138 (144)
	shipsec1	3.60%	6	29.6	141 (126)	2.64%	5	31.7	133 (126)
	shipsec5	2.33%	4	34.5	154 (140)	2.40%	4	35.0	132 (141)
	bmw7st_1	5.59%	18	16.5	435 (435)	4.08%	5	28.8	399 (408)
	bmw3.2	3.10%	11	25.8	350 (336)	2.44%	6	33.0	330 (204)
	inline_1	2.22%	15	35.3	843 (843)	1.46%	5	34.1	789 (843)
	hood	2.65%	6	39.4	115 (103)	2.84%	4	39.4	105 (79)
	msdoor	1.65%	6	46.1	118 (105)	1.65%	4	46.2	130 (98)
	ldoor	0.78%	4	60.6	121 (112)	0.79%	4	58.6	111 (91)
	pkustk10	8.32%	8	22.2	130 (126)	6.19%	5	24.9	118 (114)
pkustk11	8.55%	6	20.7	198 (198)	6.44%	5	24.4	189 (174)	
pkustk13	10.18%	8	21.2	314 (303)	4.90%	6	24.6	275 (279)	