Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 68 (2008) 515–535
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

A framework for scalable greedy coloring on distributed-memory parallel computers ☆

Doruk Bozdağ[a], Assefaw H. Gebremedhin[b], Fredrik Manne[c], Erik G. Boman[d,1], Umit V. Catalyurek[a,e,*]

[a] *Department of Electrical and Computer Engineering, The Ohio State University, USA*
[b] *Department of Computer Science, Old Dominion University, USA*
[c] *Department of Informatics, University of Bergen, Norway*
[d] *Department of Discrete Algorithms and Math, Sandia National Laboratories, USA*
[e] *Department of Biomedical Informatics, The Ohio State University, USA*

Received 23 September 2006; received in revised form 26 May 2007; accepted 6 August 2007
Available online 4 September 2007

Abstract

We present a scalable framework for parallelizing greedy graph coloring algorithms on distributed-memory computers. The framework unifies several existing algorithms and blends a variety of techniques for creating or facilitating concurrency. The latter techniques include exploiting features of the initial data distribution, the use of speculative coloring and randomization, and a BSP-style organization of computation and communication. We experimentally study the performance of several specialized algorithms designed using the framework and implemented using MPI. The experiments are conducted on two different platforms and the test cases include large-size synthetic graphs as well as real graphs drawn from various application areas. Computational results show that implementations that yield good speedup while at the same time using about the same number of colors as a sequential greedy algorithm can be achieved by setting parameters of the framework in accordance with the size and structure of the graph being colored. Our implementation is freely available as part of the Zoltan parallel data management and load-balancing library.

Keywords: Graph coloring; Parallel algorithms; Distributed-memory computers; Scientific computing; Experimental algorithmics

1. Introduction

In parallel scientific computing applications, computational dependency is often modeled using a graph. A vertex coloring of the graph is then used as a routine to identify independent subtasks that can be performed concurrently. Examples of cases where coloring is used for such a purpose include iterative solution of sparse linear systems [20], preconditioners [26], sparse tiling [27], and eigenvalue computation [24]. The computational (model) graph in such contexts is often already distributed among processors and therefore the coloring needs to be performed in parallel. The alternative approach of gathering the graph on one processor for a subsequent sequential coloring (by the same processor) is prohibitively time consuming or even infeasible due to memory constraints.

Theoretical results on graph coloring do not offer much good news: even approximating the chromatic number of a graph is known to be NP-hard [2]. For the computational graphs mentioned earlier as well as many other graphs that arise in practice, however, *greedy*, linear-time, serial coloring heuristics give solutions of acceptable quality and are often preferable to slower,

☆ The research was supported in part by the National Science Foundation under Grants #CNS-0643969, #CCF-0342615, #CNS-0426241, #CNS-0403342, NIH NIBIB BISTI #P20EB000591, Ohio Supercomputing Center #PAS0052, and by the Department of Energy's Office of Science through the CSCAPES SciDAC Institute.

* Corresponding author. Department of Biomedical Informatics, The Ohio State University, USA. Fax: +1 614 688 6600.
E-mail addresses: bozdagd@ece.osu.edu (D. Bozdağ), assefaw@cs.odu.edu (A.H. Gebremedhin), Fredrik.Manne@ii.uib.no (F. Manne), egboman@sandia.gov (E.G. Boman), umit@bmi.osu.edu (U.V. Catalyurek).

[1] 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.

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.08.002

iterative heuristics that may use fewer colors. An example of an application in which greedy graph coloring algorithms provide satisfactory solutions is the efficient computation of sparse derivative matrices [14].

This paper is concerned with the design and implementation of efficient, parallel, greedy coloring heuristics suitable for distributed-memory computers. The focus is on the realistic setting where the number of available processors is several orders of magnitude less than the number of vertices in the input graph. The goal is to develop parallel algorithms that satisfy the following two requirements simultaneously. First, the execution time of an implementation decreases with increasing number of processors. Second, the number of colors used by a parallel heuristic is fairly close to the number used by a serial heuristic.

Since greedy coloring heuristics are inherently sequential, the task at hand is difficult. A similar task had been the subject of several other studies, including the works of Gjertsen et al. [17,21]; Allwright et al. [1]; Finocchi et al. [9]; and Johansson [19]. Some of these studies reported discouraging speedup results [1,17,21], while others relied on only simulations without providing actual parallel implementations [9,19]. In contrast, Gebremedhin, Manne, Pothen, and Woods reported encouraging speedup results on shared-memory implementations of algorithms they developed in a series of interrelated works [12,13,15]. Gebremedhin et al. [11] extended one of these algorithms to the Coarse Grained Multicomputer model.

Building upon experiences from earlier efforts and introducing several new ideas, in this paper, we present a comprehensive description and evaluation of an efficient framework for parallelizing greedy coloring algorithms (preliminary results of this work have been presented in [4]). The basic features of the framework could be summarized as follows. Given a graph *partitioned* among the processors of a distributed-memory machine, each processor *speculatively* colors the vertices assigned to it in a series of rounds. Each round consists of a tentative coloring and a conflict-detection phase. The coloring phase in a round is further broken down into *supersteps* in which a processor first colors a pre-specified number $s \gg 1$ of its assigned vertices sequentially and then exchanges recent color information with other, relevant processors. In the conflict-detection phase, each processor examines those of its vertices that are colored in the current round for consistency and identifies a set of vertices that needs to be recolored in the next round to resolve any detected conflicts. The scheme terminates when no more conflicts remain to be resolved.

We implemented (using the message-passing library MPI) several variant algorithms derived from this framework. The various implementations are experimentally analyzed so as to determine the best way in which various parameters of the framework need to be combined in order to reduce both runtime and number of colors. With this objective in mind, we attempt to answer the following questions. How large should the superstep size s be? Should the supersteps be run synchronously or asynchronously? Should interior vertices be colored before, after, or interleaved with boundary vertices? How should a processor choose a color for a vertex? Should inter-processor communication be customized or broadcast-based?

The experiments are carried out on two different PC Linux clusters. The testbed consists of large-size synthetic graphs as well as real graphs drawn from various application areas. The computational results we obtained suggest that, for large-size, *structured* graphs (i.e., graphs where the percentage of boundary vertices in a given partition is fairly low), a combination of parameters in which

(1) a superstep size in the order of a thousand is used,
(2) supersteps are run asynchronously,
(3) each processor colors its assigned vertices in an order where interior vertices appear either strictly before or strictly after boundary vertices,
(4) a processor chooses a color for a vertex using a First-Fit (FF) scheme, and
(5) inter-processor communication is customized,

gives an overall best performance. Furthermore, the choice of the coloring order in (3) offers a trade-off between number of colors and execution time: coloring interior vertices first gives a faster and slightly more scalable algorithm whereas an algorithm in which boundary vertices are colored first uses fewer colors. The number of colors used even when interior vertices are colored first is fairly close to the number used by a sequential greedy algorithm. For *unstructured* graphs (i.e., graphs where the vast majority of the vertices are boundary), good performance is observed by using a superstep size close to a hundred in item (1), a broadcast-based communication mode in item (5), and by keeping the remaining parameters as in the structured case. For almost all of the test graphs used in our experiments, the variants of our framework with the aforementioned combination of parameters converged rapidly (within at most six rounds) and yielded fairly good speedup with increasing number of processors.

The remainder of this paper is organized as follows. In Section 2 we discuss relevant previous work on parallel (and distributed) coloring. In Section 3 we present the unifying parallelization framework proposed in this paper and discuss the several directions in which it can be specialized. In Section 4 we present extensive experimental results on our implementations of various specialized algorithms designed using the framework. We conclude in Section 5.

2. Background

2.1. Sequential coloring

A *coloring* of a graph is an assignment of positive integers (colors) to vertices such that every pair of adjacent vertices receives different colors. The graph coloring problem, whose objective is to minimize the number of colors used, is known to be NP-hard [10]. The current best known approximation ratio for the problem is $O(n\frac{(\log\log n)^2}{(\log n)^3})$, where n is the number of vertices in the graph. Moreover, the problem is known to be not approximable within $n^{1/7-\varepsilon}$ for any $\varepsilon > 0$ [2].

Despite these rather pessimistic results, greedy sequential coloring heuristics are quite effective in practice [6]. A greedy coloring heuristic iterates over the set of vertices in some order, at each step assigning a vertex the *smallest* permissible color (a FF strategy). Vertex ordering techniques that have proven to be effective in practice include Largest Degree First, Smallest Degree Last, Incidence Degree, and Saturation Degree ordering; see [14] for a review of these techniques. Any greedy heuristic that employs a FF coloring strategy uses no more (but often many fewer) than $\Delta + 1$ colors, where Δ is the maximum degree in the graph.

2.2. Related previous work on parallel coloring

A number of the existing parallel algorithms for greedy graph coloring rely on Luby's algorithm for computing a *maximal independent set* in parallel [23]. In Luby's algorithm, each vertex v in the input graph is assigned a random number $r(v)$, and an initially empty independent set I is successively populated in the following manner. In each iteration, if a vertex v *dominates* its neighborhood $N(v)$, i.e., if $r(v) > r(w)$ for all $w \in N(v)$, then the vertex v is added to the set I and v as well as the vertices in $N(v)$ are deleted from the graph. This procedure is then recursively applied on the graph induced by the remaining vertices, and the process terminates when the graph becomes empty, at which point I is a maximal independent set.

Luby's parallel algorithm for computing a maximal independent set can easily be turned into a parallel (or a distributed) coloring algorithm: Instead of removing a dominating vertex v (a vertex whose random value is larger than the value of any of its uncolored neighbors) and its neighbors $N(v)$ from the current (reduced) graph, assign the vertex v the smallest color that is not used by any of its already colored neighbors. If one proceeds in this manner, then a pair of adjacent vertices surely gets different colors, and once assigned, the color of a vertex remains unchanged during the course of the algorithm.

Each of the works of Jones and Plassmann [21], Gjertsen et al. [17], and Allwright et al. [1] is essentially a variation of this general scheme. In each algorithm, the input graph is assumed to be partitioned among the available processors, and each processor is responsible for coloring the vertices assigned to it. The interior vertices on each processor are trivially colored in parallel (using a sequential algorithm on each processor), and a variant of Luby's coloring algorithm is applied on the *interface graph*, the subgraph induced by the boundary vertices. A related algorithm that could be considered here is one in which the interface graph is built on and colored by one processor. In Section 3.4 we discuss this variation as well as a variant of the Jones–Plassmann algorithm [21] that we have implemented for the purposes of experimental comparison.

In the algorithms discussed in the previous paragraph, a pair of adjacent vertices is necessarily colored at different computational steps. Thus, once the random function that is used for determining dominating vertices has been set, the longest monotone path in the graph gives an upper bound on the parallel runtime of the algorithm. As the work of Johansson [19] shows, this inherent sequentiality can be overcome by using randomization in the selection of a color for a vertex. In particular, Johansson analyzed a distributed coloring algorithm where each processor is assigned exactly one vertex and the vertices are colored simultaneously by randomly choosing a color from the set $\{1, \ldots, \Delta + 1\}$, where Δ is the maximum degree in the graph. Since this may lead to an inconsistent coloring, the process is repeated recursively on the vertices that did not receive permissible colors.

More recently, Finocchi et al. [9] have performed extensive (sequential) simulations on a set of inter-related algorithms that each resemble Johansson's algorithm. In the basic variant of their algorithms, the upper bound on the range of permissible colors is initially set to be smaller than $\Delta + 1$ and is increased later only when needed. Another feature of their algorithm is that in a post-processing step at the end of each round, Luby's algorithm is applied on each subgraph induced by a color class. Specifically, the post-processing step is used to find a maximal independent set of vertices whose colors can be declared final.

2.3. Precursory work on parallel coloring

Each of the works surveyed in Section 2.2 has one or more of the following weaknesses: (i) no actual parallel implementation is given, (ii) the number of colors used, although bounded by $\Delta + 1$, is far from the number used by a sequential greedy algorithm, or (iii) an implementation yields poor parallel speedup on unstructured graphs. Overcoming these weaknesses had been the subject of several previous efforts in which at least a subset of the authors of the current paper have been involved.

Gebremedhin and Manne [12] developed a simple, shared-memory parallel coloring algorithm that gave fairly good speedup in practice. In their algorithm, the vertex set is equi-partitioned among the available processors and each processor colors its assigned set of vertices in a sequential fashion, at each step assigning a vertex the smallest color not used by any of its on- or off-processor neighbors; the assigned color information is immediately made available to other processors by writing to shared memory. Inconsistencies—which arise only when a pair of adjacent vertices residing on different processors is colored simultaneously—are then detected in a subsequent parallel phase and resolved in a final sequential phase.

Gebremedhin et al. [13] extended the algorithms in [12] to the distance-2 and star coloring problems, models that occur in the computation of Jacobian and Hessian matrices. The algorithms in [13] also used *randomized* techniques in selecting a color for a vertex to reduce the likelihood of conflicts. Gebremedhin et al. [15] enhanced these algorithms by employing graph partitioning tools to obtain more effective assignment of vertices to processors and by using vertex orderings to minimize cache misses in a shared-memory implementation.

In a different direction, Gebremedhin et al. [11] extended the algorithm in [12] to the Coarse Grained Multicomputer (CGM) model of parallel computation, thus making the algorithm feasible for distributed-memory architectures as well. The CGM model is a simplified version of the Bulk Synchronous

Parallel (BSP) model introduced by Valiant [3,32]. Besides paying attention to cost associated with non-local data access, the randomized as well as deterministic CGM-coloring algorithms proposed in [11] differed from the shared-memory algorithm in [12] in two ways. First, in the CGM-algorithms processors exchange information only after a group of vertices (rather than a single vertex) have been colored. Second, in the CGM algorithms potential conflicts are identified a priori and dealt with in a *recursive* fashion. The algorithms in [11] were not implemented, but we believe the first item would certainly translate into practical gain while the second item may not, since a recursive implementation is likely to be inefficient in practice.

3. A unifying framework

Building upon ideas and experiences gained from the efforts mentioned in Section 2.3, we have developed a framework for scalable parallel graph coloring on distributed-memory computers. In this section, we describe the framework in detail and show how it helps unify several existing algorithms.

3.1. The basic scheme

We assume that the input graph is *partitioned* among the p available processors in some reasonable way. Typically, this is achieved by employing a graph partitioning tool such as Metis [22]. The partitioning used (regardless of how it is achieved) classifies the vertices into two categories: interior and boundary. An interior vertex is a vertex all of whose neighbors are located on the same processor as itself whereas 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. Parallel coloring of the remainder of graph, however, requires interprocessor communication and coordination, and is a major issue in the scheme being described.

At the highest level, our scheme proceeds in *rounds*. Each round consists of a *tentative coloring* and a *conflict-detection* phase. In the spirit of the BSP model, the coloring phase of a round is organized as a sequence of *supersteps*,[2] where each superstep has distinct computation and communication subphases. Since the conflict-detection phase does not involve communication, it does not need to be organized in supersteps.

In every superstep, each processor colors a pre-specified number s of vertices in a sequential fashion, 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 recently colored 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 in the same superstep, they may receive the same color and cause a conflict.

[2] We use the word *superstep* in a looser sense than its usage in the BSP model.

Algorithm 1. A Scalable Parallel Coloring Framework. *Input*: graph $G = (V, E)$ and superstep size s. *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.

```
1:  procedure SPCFramework(G = (V, E), s)
2:      on each processor P_i, i ∈ I = {1, ..., p}
3:          for each bound. vtx v ∈ V_i' = {u|(u, v) ∈ E_i} do
4:              Assign v a random number r(v) generated using
                    v's ID as seed
5:          U_i ← V_i    ▷ U_i: current set of vtxs to be colored
6:          while ∃ j ∈ I, U_j ≠ ∅ do
7:              if U_i ≠ ∅ then
8:                  Partition U_i into ℓ_i subsets
                        U_{i,1}, U_{i,2}, ..., U_{i,ℓ_i}, each of size s
9:                  for k ← 1 to ℓ_i do
10:                     for each v ∈ U_{i,k} do
11:                         assign v a "permissible" color c(v)
12:                     Send colors of boundary vertices in U_{i,k}
                            to other processors
13:                     Receive color information from other
                            processors
14:                 Wait until all incoming messages are
                        successfully received
15:                 R_i ← ∅ ▷ R_i is a set of vtxs to be recolored
16:                 for each boundary vertex v ∈ U_i do
17:                     if ∃ (v, w) ∈ E_i where c(v) = c(w) and
                            r(v) < r(w) then
18:                         R_i ← R_i ∪ {v}
19:                 U_i ← R_i
20:     end on
21: end procedure
```

The purpose of the second phase of a round is to detect conflicts and accumulate on each processor a list of vertices to be recolored in the next round. Given a *conflict-edge*, only one of its two endpoints needs to be recolored to resolve the conflict. The vertex to be recolored is determined in a random fashion so that the workload in the next round is distributed more or less evenly among the processors; we shall shortly discuss how the random selection is done. The conflict-detection phase does not require communication since by the end of the tentative coloring phase every processor has gathered complete information about the colors of the neighbors of its vertices. The scheme terminates when every processor is left with an empty list of vertices to be recolored. Algorithm 1 outlines this scheme with some more details. The scheme is hereafter referred to as Scalable Parallel Coloring Framework (SPCFramework).

In each round of SPCFramework, the vertices to be recolored in the subsequent round are determined by making use of a uniformly distributed random function over boundary vertices, defined at the beginning of the scheme; interior vertices need not be considered since they do not cause conflicts. For each conflict-edge detected in a round, the vertex with the *lower*

random value is selected to be recolored while the other vertex retains its color (see the for-loop in Line 16 of SPCFRAMEWORK). Notice that the set of vertices that retain their colors in a round in this manner is exactly the set that would be obtained by running one step of Luby's algorithm (discussed in Section 2.2) on the set of vertices involved in conflicts.

In a distributed setting such as ours, the random numbers assigned to boundary vertices need to be computed in a careful way in order to avoid the need for communication between processors to exchange these values. The for-loop in Line 3 of SPCFRAMEWORK is needed for that purpose. There, notice that each processor P_i generates random values not only for boundary vertices in its own set V_i but also for adjacent boundary vertices residing on other processors. Since the (global) unique ID of a vertex is used in the random number generation, the random value computed for a vertex on one processor is the same as the value computed for the same vertex on another processor. Thus processors avoid inquiring each other of random values.

The coloring phase of a round is divided into supersteps (rather than communicating after a single vertex is colored) to reduce communication frequency and thereby the associated cost. However, the number of supersteps used (or, 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 likelihood of conflicts and hence the higher the number of rounds required. Choosing a value for the parameter s that minimizes the overall runtime is therefore a compromise between these two contradicting requirements. An optimal value for s would depend on such factors as the size and density of the input graph, the number of processors available, and the machine architecture and network.

3.2. Variations of the scheme

SPCFRAMEWORK has been deliberately presented in a general form. Here we discuss several ways in which it can be specialized.

3.2.1. Color selection strategies

In Line 11 of SPCFRAMEWORK, the choice of a color, *permissible* relative to the currently available color information, can be made in different ways. The strategy employed affects the number of colors used by the algorithm and 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. A coloring strategy typically reduces one of these quantities at the expense of the other. Here, we present two strategies, dubbed FF and *Staggered First Fit* (SFF).

In the FF strategy each processor P_i chooses the *smallest* permissible color from the set $\{1, \ldots, C_i\}$, where C_i, initially set to be one, is the current largest (local) color used. If no such color exists, C_i is incremented by one and the new value of C_i is chosen as a color. In contrast, the SFF strategy relies on an initial estimate K of the number of colors needed for the input graph, and each processor P_i chooses the *smallest* permissible color from the set $\{\lceil \frac{iK}{p} \rceil, \ldots, K\}$. If no such color exists, then the smallest permissible color in $\{1, \ldots, \lfloor \frac{iK}{p} \rfloor\}$ is chosen. If there still is no such color, the smallest permissible color greater than K is chosen. Notice that, unlike FF, the search for a color in SFF starts from different "base colors" for each processor. Therefore SFF is likely to result in fewer conflicts than FF. On a negative side, the fact that the search for a color starts from a base larger than one makes SFF likely to require more colors than FF.

The SFF strategy is similar to the strategy used in the algorithms of Finocchi et al. [9], specifically, to the variants they refer to as *Brooks–Vizing* coloring algorithms. The essential difference between their color choice strategy and SFF is that in their approach the color for a vertex is *randomly* picked out of an appropriate range, whereas in SFF the *smallest* available color in a similar range is searched for and chosen. Taking the comparison further, note that the formulation of SPCFRAMEWORK is general enough to encompass the (entire) algorithms of Finocchi et al. as well as the algorithm of Johansson [19]. Specifically, one arrives at the latter algorithms by letting the number of processors be equal to the number of vertices in the graph (which implies a superstep size $s = 1$) and by choosing the color of a vertex in Line 11 appropriately.

Other color selection strategies that can be used in SPCFRAMEWORK include: the *Least Used* strategy where the (locally) least used color so far is picked so that a more even color distribution is achieved, and the *randomized* methods of Gebremedhin et al. [13].

3.2.2. Coloring order

As mentioned earlier, the subgraphs induced by interior vertices are independent of each other and can therefore be colored concurrently without any communication. As a result, in the context of SPCFRAMEWORK, interior vertices can be colored *before*, *after* or *interleaved* with boundary vertices. SPCFRAMEWORK is presented assuming an interleaved order, wherein computational load is likely to be evenly balanced and communication is expected to be more evenly spaced out in time, avoiding congestion as a result. On the other hand, an interleaved order may involve communication of a higher number of messages of smaller size, which in turn degrades execution speed and scalability. Overall, a non-interleaved order, wherein interior vertices are colored strictly before or after boundary vertices, is likely to yield better performance. Furthermore, coloring interior vertices first is likely to produce fewer conflicts when used in combination with a FF coloring scheme, since the subsequent coloring of boundary vertices is performed with a larger spectrum of available colors. Coloring boundary vertices first might be advantageous when used together with the SFF color selection strategy.

3.2.3. Vertex ordering strategies

Regardless of whether interior and boundary vertices are colored in separate or interleaved order, SPCFRAMEWORK

provides another degree of freedom in the choice of the order in which vertices on each processor are colored in each round. Specifically, in Line 10 of SPCFRAMEWORK, the given order in which the vertices appear in the input graph, or any one of the effective (re)ordering techniques discussed in Section 2.1 could be used.

3.2.4. Synchronous vs. asynchronous supersteps

In SPCFRAMEWORK, 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 off-processor neighbors colored during the same superstep. The obvious disadvantage is that the barriers, in addition to the associated overhead, cause some processors to be idle while others are busy completing 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 be deferred to 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.

3.2.5. Inter-processor communication

Sending color information in Line 12 of SPCFRAMEWORK can be done in one of two ways. First, a processor may send the color of a boundary vertex v to another processor only if the latter owns at least one neighbor of the vertex v. Although this helps avoid unnecessary communication, the customization of messages incurs computational overhead. An alternative approach is for a processor to send the color information of all of its boundary vertices to every other processor without customizing the messages (broadcast). This approach might be more efficient when most of the boundary vertices have neighbors on a considerable portion of the processors in the system.

3.3. How then should all these various options be combined?

As we have been discussing in Section 3.2, there are as many as five axes along which SPCFRAMEWORK could be specialized. Each axis in turn offers at least two alternatives each having its own advantages and disadvantages. The final determination of how various options need to be put together to ultimately reduce both runtime and number of colors used is bound to rely on experimentation since it involves a complex set of factors, including the size and density of the input graph, the number of processors employed and the quality of the initial partitioning among the processors, and the specific characteristics of the platform on which the implementations are run. We have therefore experimentally studied the performance of several specialized implementations of SPCFRAMEWORK on two different platforms. The results will be presented in Section 4.

3.4. Related algorithms

For the purposes of comparison in our experiments, we have in addition implemented three other algorithms related to SPCFRAMEWORK. We briefly describe these algorithms in the sequel; in each case, the input graph is assumed to be already partitioned among the available processors.

3.4.1. Sequential Boundary Coloring Algorithm (SBC)

In this algorithm parallel processing is applied only on interior vertices, whereas the graph induced by boundary vertices is constructed on one dedicated processor and then colored sequentially by the same processor.

3.4.2. Sequential Conflict Resolution Algorithm (SCR)

This is a variant in which SPCFRAMEWORK is restricted to one round, and the conflicts detected at the end of the round are resolved sequentially on one processor. Here, only the graph induced by boundary vertices involved in conflicts is constructed on one dedicated processor and then colored sequentially by the same processor. Note that the algorithm of Gebremedhin and Manne [12] is a special case of SCR where the superstep size s is equal to identity.

3.4.3. Modified Jones and Plassmann Algorithm (JP)

This is a synchronous variant of the *Jones–Plassmann* (*JP*) algorithm first mentioned in Section 2.2. Given a graph partitioned among the available processors, we describe the steps applied on only boundary vertices; interior vertices are trivially colored in parallel in a final phase. In each round j of the algorithm, each processor P_i maintains a list D_i of dominating vertices—vertices that are assigned larger random numbers than their uncolored neighbors. The vertices in the list D_i are colored sequentially (by processor P_i) in the round j. Then, processor P_i sends the recent color information to processors who own vertices adjacent to elements of the list D_i, and receives corresponding information from other processors. For each off-processor vertex v whose color is received, the list D_i is updated in the following manner. If the coloring of the vertex v has made any uncolored on-processor neighbor vertex w of the vertex v dominate the uncolored neighbors of w, then the vertex w is included in the list D_i. In this way only vertices that are dominated by off-processor neighbors will be left uncolored at the end of the round j. The procedure is repeated in subsequent rounds until all boundary vertices are colored.

We claim that SPCFRAMEWORK uses fewer or at most as many rounds as the JP algorithm just described. Here is the argument. Suppose the two algorithms begin a particular round k with identical sets of uncolored vertices and random values. In SPCFRAMEWORK any vertex that needs to be recolored after the round k must have had an off-processor neighbor with the same color and with a larger random value. Clearly, in the JP algorithm, such a vertex would not have been colored in the round k. Thus the JP algorithm would require at least as many rounds as SPCFRAMEWORK. To see that SPCFRAMEWORK could use fewer rounds than the JP algorithm, consider a vertex v where

each off-processor neighbor with a larger random value (than that of v) receives a different color (from that of v). Then v can keep its color as final in SPCFRAMEWORK, whereas v would not have been colored in the JP algorithm.

4. Experiments

This section presents results from an experimental study of the various algorithms discussed in the previous section. The algorithms are implemented in C using the MPI message-passing library.

4.1. Experimental setup

Our experiments are conducted on two different platforms, each of which is a PC cluster running Linux. Table 1 gives the essential features of the platforms.

The testbed consists of very large-size *real* as well as *synthetic* graphs. The real graphs (21 in total) are obtained from various application areas, including molecular dynamics, finite element, structural engineering, civil engineering, and automotive and ship industry [12,29,28,31]. The first four columns in Table 2 display data concerning structural properties of

Table 1
The two platforms used in the experiments

Name	CPU	Memory	Cache size	Network	Bandwidth	Latency
I2	900 MHz Itanium 2	4 GB	512 KB	Myrinet 2000	240 MB/s	11 μs
P4	2.4 GHz Pentium 4	4 GB	1.5 MB	Infiniband	888 MB/s	7 μs

Table 2
Structural properties of the *real* test graphs used in the experiments, classified according to their *application* area and source

Name	$\|V\|$	Degree		Sequential FF	
		Max	Avg	Colors	T_{I2} (ms)
Molecular dynamics [28]					
popc-br-4	24,916	43	20	21	28
er-gre-4	36,573	42	25	19	48
apoa1-4	92,224	43	25	20	120
Finite elements [12]					
144	144,649	26	15	12	137
598a	110,971	26	13	11	98
auto	448,695	37	15	13	462
Linear car analysis [29]					
bmw3_2	227,362	335	49	48	561
bmw7st1	141,347	434	51	54	364
inline1	503,712	842	72	51	1818
Structural engineering [31]					
pwtk	217,918	179	52	48	564
nasasrb	54,870	275	48	41	130
ct20stif	52,329	206	51	49	132
Automotive [31]					
hood	220,542	76	48	42	541
msdoor	415,863	76	48	42	1020
ldoor	952,203	76	48	42	2362
Civil engineering [31]					
pkustk10	80,676	89	52	42	213
pkustk11	87,804	131	58	66	258
pkustk13	94,893	299	69	57	317
Ship section [29]					
shipsec1	140,874	101	54	48	382
shipsec5	179,860	125	55	50	491
shipsec8	114,919	131	57	54	322

Also shown is the number of colors used by and the runtime of a sequential First Fit algorithm run on a single node of the Itanium 2 cluster.

Table 3
Structural properties of the *synthetic* test graphs used in the experiments

Name	$\|V\|$	Degree		Sequential FF		
		Max	Avg	Colors	T_{I2} (ms)	T_{P4} (ms)
Clique based [16]						
cliq-1	40,000	44	40	41	127	33
cliq-2	40,000	61	47	42	103	40
cliq-3	40,000	81	59	43	87	47
Planar [25]						
plan-1	4,072,937	40	6	9	4780	3725
Random [18]						
rand-1	400,000	27	10	9	461	626
rand-2	400,000	61	30	15	1189	1750
rand-3	400,000	84	50	20	1865	2857
rand-4	400,000	112	70	25	2972	4074

Also shown is the number of colors used by and the runtime of a sequential First Fit algorithm run on a single node of each of the two test platforms.

Table 4
Configurations of SPCFRAMEWORK used in the experiments

Name	Color choice	Coloring order	Supersteps	Communication
FIAC	**F**irst Fit	**I**nterior first	**A**synchronous	**C**ustomized
FBAC	**F**irst Fit	**B**oundary first	**A**synchronous	**C**ustomized
FUAC	**F**irst Fit	**U**n-ordered	**A**synchronous	**C**ustomized
FISC	**F**irst Fit	**I**nterior first	**S**ynchronous	**C**ustomized
SIAC	**S**taggered FF	**I**nterior first	**A**synchronous	**C**ustomized
SBAC	**S**taggered FF	**B**oundary first	**A**synchronous	**C**ustomized
FIAB	**F**irst Fit	**I**nterior first	**A**synchronous	**B**roadcast
FBAB	**F**irst Fit	**B**oundary first	**A**synchronous	**B**roadcast
FIAC-block	**F**irst Fit	**I**nterior first	**A**synchronous	**C**ustomized

In each algorithm, vertices on each processor are colored in their *natural* order; "-block" indicates the initial data distribution is obtained via simple *block* partitioning, in all other cases graph partitioning is used.

these graphs and the source application areas. The last two columns show certain *computed* values—the number of colors used by and the runtime in milliseconds of a serial FF algorithm run on a single node of the Itanium 2 cluster; these values will be used as normalizing quantities in the presentation of relevant experimental results later in this section.

The synthetic graphs used in our experiment include instances drawn from three different classes: *random*, *planar* and *clique-based*. The random graphs belong to the Erdös–Renyi class and are generated using the GTgraph Synthetic Graph Generator Suite [18]. The planar graphs are maximally planar—the degree of every vertex is at least five—and are generated using the expansion method described in [25]; the generator is a part of the graph suite for the Second DIMACS Challenge [30]. Finally, the clique-based graphs are obtained using P. Ross's graph-coloring generator, which is a part of the Test Problem Generators for Evolutionary Algorithms [16]. Each of the clique-based graphs we generated consists of 1000 cliques of size 40 each, and the cliques are in turn interconnected by adding edges according to a probabilistic function. Table 3 displays structural information about the synthetic test graphs in a manner analogous to Table 2.

4.2. Algorithms compared

As discussed in Section 3.2, there are five orthogonal axes of specializations for SPCFRAMEWORK. Further, there are multiple options to be considered along each axis, making the possible combinations (*configurations*) exponentially many. We describe below the configurations we have considered in our experiments either for their potential performance worth or for benchmarking purposes.

For the initial *data distribution*, we consider two scenarios. In the first scenario, the input graph is *partitioned* among the processors using the software tool Metis [22] and its VMetis option. This option aims at minimizing both the number of boundary vertices and the communication volume. In the second scenario, the input graph is *block partitioned*, a case where the vertex set is simply equi-partitioned among the processors without any effort to minimize cross-edges.

For *vertex ordering*, we use the *natural* order in which vertices appear in the input graph. We forgo considering other vertex orderings, since our primary objective is to study issues that have relevance to parallel performance. In terms of *coloring order*, we consider three options: coloring interior vertices first,

coloring boundary vertices first or coloring in an un-ordered (interleaved) fashion. Each of the remaining three axes—the manner in which *a color is chosen for a vertex*, *supersteps are run*, and *inter-processor communication is handled*—offers two options. Table 4 summarizes the configurations we have considered in our experiments and the acronyms we use in referring to each.

In addition to the nine algorithms listed in Table 4, we have implemented and tested the three algorithms—Sequential Boundary Coloring (SBC), Sequential Conflict Resolution (SCR), and Modified Jones–Plassmann (JP)—discussed in Section 3.4.

Another configurable parameter of SPCFRAMEWORK is the size s of a superstep, the number of vertices sequentially colored by a processor before communication takes place. After several preliminary tests, we determined that a value of s close to a thousand is a good choice for cases where customized communication is preferable. Similarly, for cases where broadcast is a better choice, a value of s close to a hundred was found to be a good choice. For the results reported in the rest of this section, we use $s = 800$ for the former case and $s = 125$ for the latter.

4.3. Performance profiles

For much of our experimental analysis in this section, we use *performance profiles*, a generic tool introduced by Dolan and Moré [8] for comparing a set of *methods* over a large set of *test cases* with regard to a specific performance *metric*. The essential idea behind performance profiles (which we review below) is to use a cumulative *distribution function* for a performance metric, instead of, for example, taking averages or sum-totals over all the test cases.

Let S denote the set of methods (solvers) being compared, P denote the set of test cases (problems) used, and t denote the metric of interest, which is desired to be as small as possible. For a specific method $s \in S$ and a specific problem $p \in P$, let $t_{p,s}$ denote the quantity with regard to metric t used by method s in solving problem p. Similarly, let $t^*_{p,S}$ correspond to the *best* performance by any solver in S on the problem p. For example, if the performance metric of interest is runtime of algorithms, then $t_{p,s}$ is the time used by algorithm s in solving problem p, and $t^*_{p,S}$ is the time used by the *fastest* algorithm in S in solving the problem p. The ratio $t_{p,s}/t^*_{p,S}$ is known as the *performance ratio* of method s on problem p and is denoted by $r_{p,s}$. The lower the value of $r_{p,s}$, the closer method s is to the best method in S for solving problem p. In particular, $r_{p,s} = 1$ means that method s is the best solver in S for problem p. Now define a function ρ as follows:

$$\rho_s(\tau) = |P'|/|P| \quad \text{where } P' = \{p \in P : r_{p,s} \leqslant \tau\}.$$

The value $\rho_s(\tau)$ is then the probability for solver $s \in S$ that the performance ratio $r_{p,s}$ is within a factor of τ of the best possible ratio, and the function ρ_s is the cumulative distribution function for the performance ratio. In the performance profile *plots* presented in this section we label the axis corresponding to $\rho_s(\tau)$ as "Fraction of wins", to indicate that this quantity corresponds to the fraction of the test cases for which method s is a winner (among its competitors in S) considering a solution within a factor of τ of the best possible as the criterion for comparison.

4.4. Results and discussion: identifying the "best" algorithms

4.4.1. Fix color choice strategy and communication mode, vary other parameters

In our first set of experiments, we compare the performance (in terms of runtime and number of colors) of configurations of SPCFRAMEWORK obtained by fixing the color selection strategy to be FF and the communication mode to be Customized and by varying the remaining three parameters listed in Table 4.

Fig. 1 shows runtime performance profile plots (for various number of processors) of the resulting five algorithms, FIAC, FISC, FBAC, FUAC, and FIAC-block. Fig. 2 shows similar plots for number of colors, where, in addition to these five parallel algorithms, a sequential FF algorithm is included as a baseline. The test set in both figures is the set of application graphs listed in Table 2 and the experiments are conducted on the Itanium 2 cluster.

We illustrate how performance profiles are to be "read" using Fig. 1(c) as an example. In that example, one can see that for nearly every testcase, FIAC has the least runtime among the five algorithms being compared (see the values on the vertical line $\tau = 1$). In almost 90% of the cases, FISC is nearly equally fast. Suppose we are interested in identifying methods that would solve *every* testcase within a factor of 1.2 of the best time. Then FBAC would clearly join the company of FIAC and FISC. In general, Fig. 1 clearly shows that Algorithms FIAC and FISC are the fastest among the five for every value of p and are almost indistinguishable from each other. The two algorithms are very closely followed by FBAC.

In terms of number of colors, Fig. 2 shows that FBAC is the best candidate. Its performance is almost as good as the sequential algorithm. A major reason for this rather impressive performance is that algorithm FBAC has the flavor of a Largest Degree First sequential algorithm, since vertices of large degree are in general likely to be boundary vertices in a partition, and these are colored first in the FBAC configuration. In the sequential baseline algorithm, on the other hand, vertices are colored in their natural order. Further in Fig. 2, we observe that algorithms FUAC, FIAC, and FISC have fairly similar behavior with each other, and in general use 5–15% more colors than that used by the sequential algorithm. This in itself is a rather "good" performance considering the facts that the number of colors used by a sequential algorithm is very close to the average degree in the test graphs (see Table 2) and that existing parallel (distributed) coloring algorithms commonly try to only bound the number by maximum degree.

Based on observations made from Figs. 1 and 2, we pick the two algorithms FIAC and FBAC as the most preferred choices, which in turn offer a runtime-quality tradeoff. We use these algorithms in the rest of the experiments described in this section.

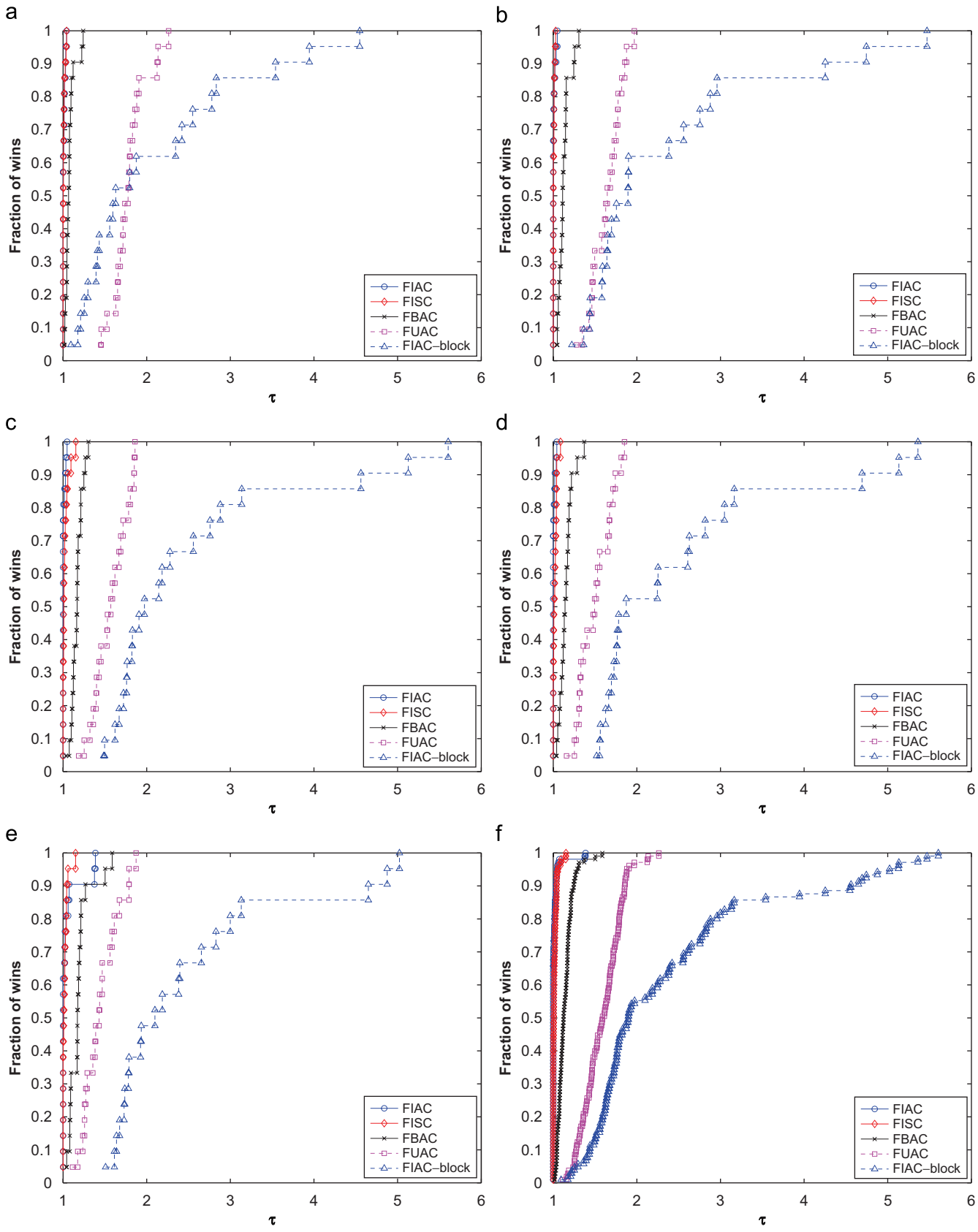


Fig. 1. *Algorithms compared*: five variants of SPCFRAMEWORK. *Plots*: performance profiles. *Metric*: runtime. *Test set*: (a) through (e), application graphs listed in Table 2; (f), set of all processor–graph pairs (p, G), for $p \in \{8, 16, 24, 32, 40\}$ and G drawn from Table 2. *Platform*: Itanium 2. $s = 800$. (a) $p = 8$, (b) $p = 16$, (c) $p = 24$, (d) $p = 32$, (e) $p = 40$, (f) $p \in \{8, 16, 24, 32, 40\}$.

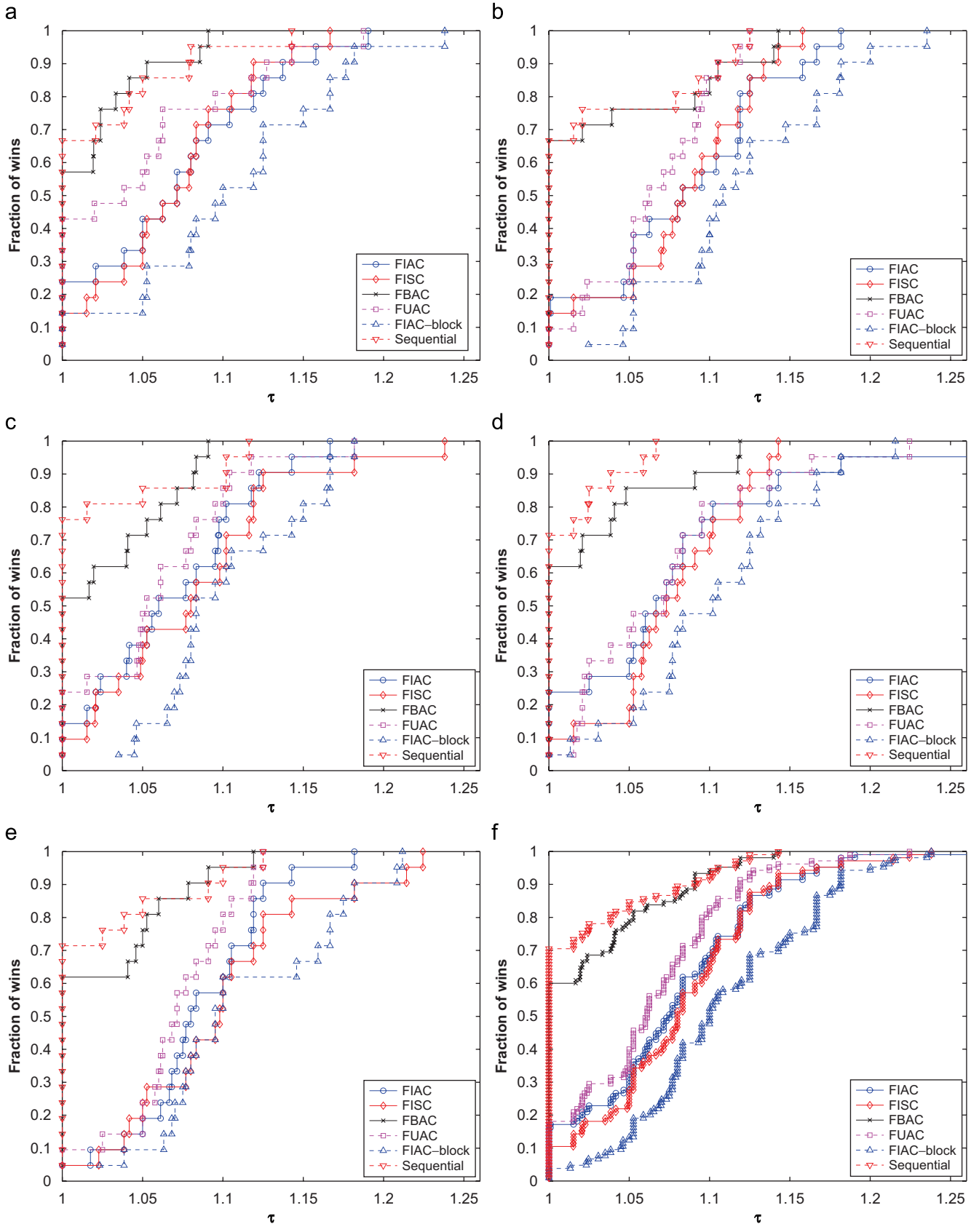


Fig. 2. *Algorithms compared*: five variants of SPCFRAMEWORK. *Plots*: performance profiles. *Metric*: number of colors. *Test set*: (a) through (e), application graphs listed in Table 2; (f), set of all processor–graph pairs (p, G), for $p \in \{8, 16, 24, 32, 40\}$ and G drawn from Table 2. *Platform*: Itanium 2. $s = 800$. (a) $p = 8$, (b) $p = 16$, (c) $p = 24$, (d) $p = 32$, (e) $p = 40$, (f) $p \in \{8, 16, 24, 32, 40\}$.

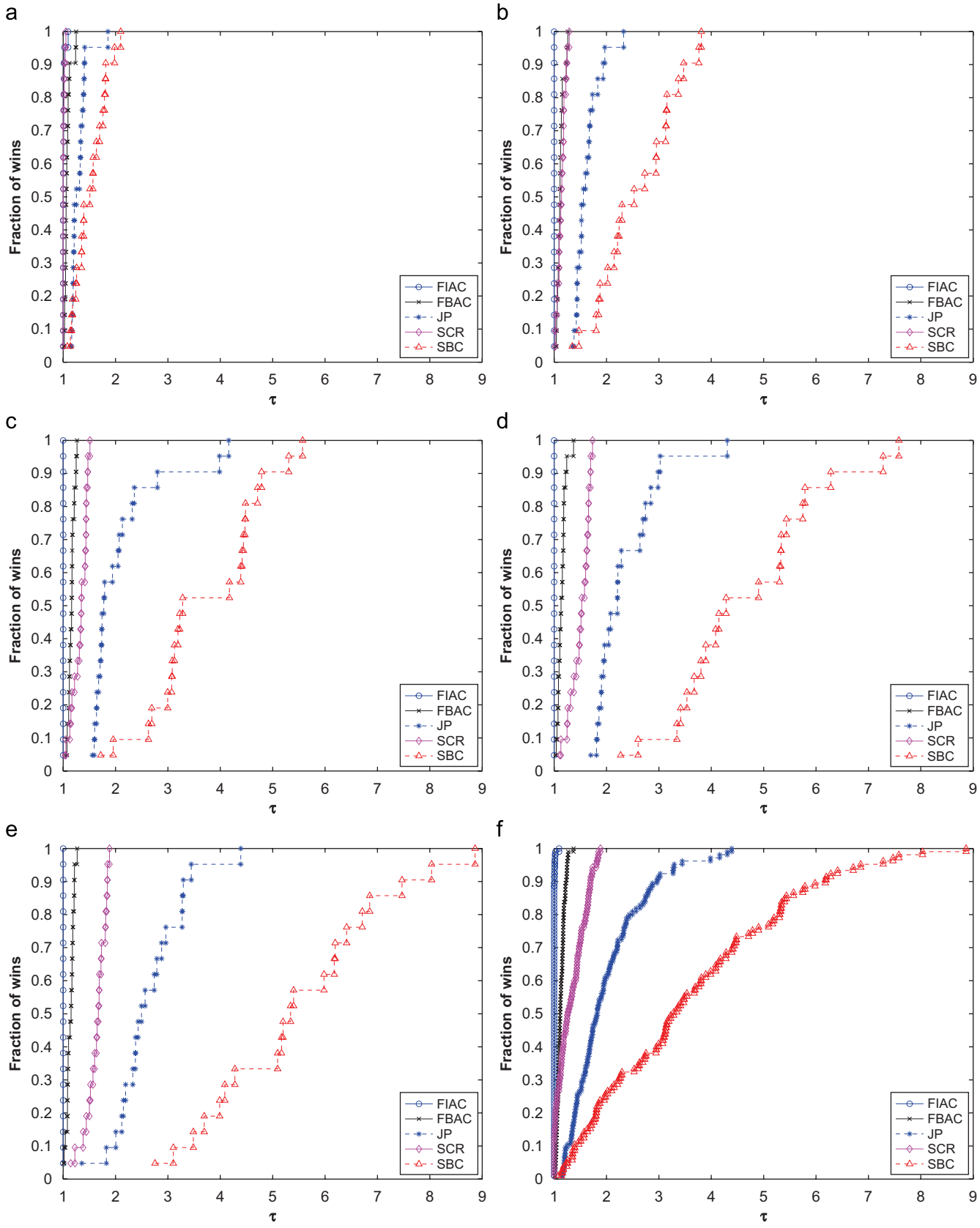


Fig. 3. *Algorithms compared*: two variants of SPCFRAMEWORK (FIAC and FBAC) and algorithms SCR, SBC and JP. *Plots*: performance profiles. *Metric*: runtime. *Test set*: (a) through (e), graphs listed in Table 2; (f), set of all processor–graph pairs (p, G), for $p \in \{8, 16, 24, 32, 40\}$ and G drawn from Table 2. *Platform*: Itanium 2. $s = 800$. (a) $p = 8$, (b) $p = 16$, (c) $p = 24$, (d) $p = 32$, (e) $p = 40$, (f) $p \in \{8, 16, 24, 32, 40\}$.

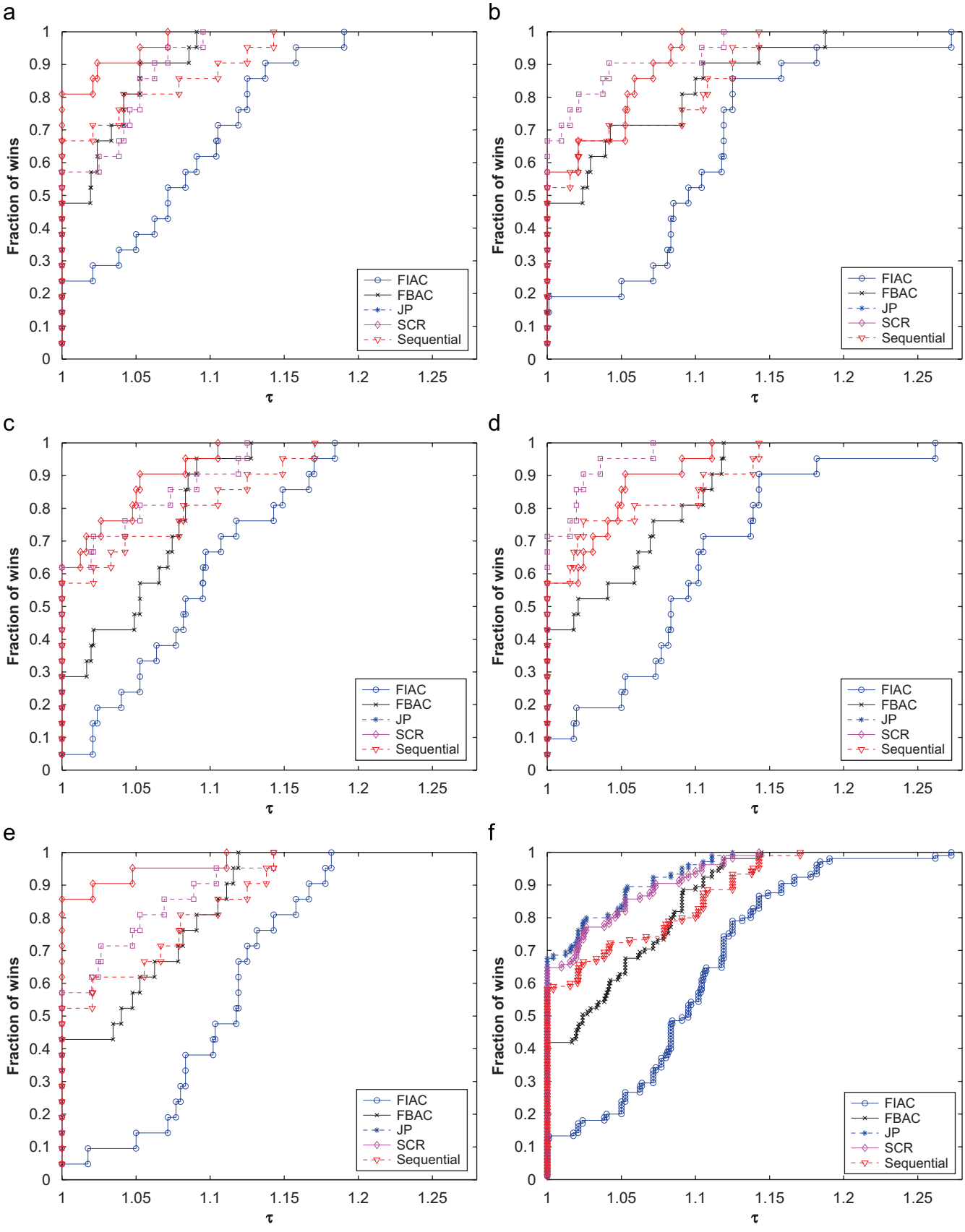


Fig. 4. *Algorithms compared*: two variants of SPCFRAMEWORK (FIAC and FBAC) and algorithms SCR, JP and Sequential. *Plots*: performance profiles. *Metric*: number of colors. *Test set*: (a) through (e), graphs listed in Table 2; (f), set of all processor–graph pairs (p, G), for $p \in \{8, 16, 24, 32, 40\}$ and G drawn from Table 2. *Platform*: Itanium 2. $s = 800$. (a) $p = 8$, (b) $p = 16$, (c) $p = 24$, (d) $p = 32$, (e) $p = 40$, (f) $p \in \{8, 16, 24, 32, 40\}$.

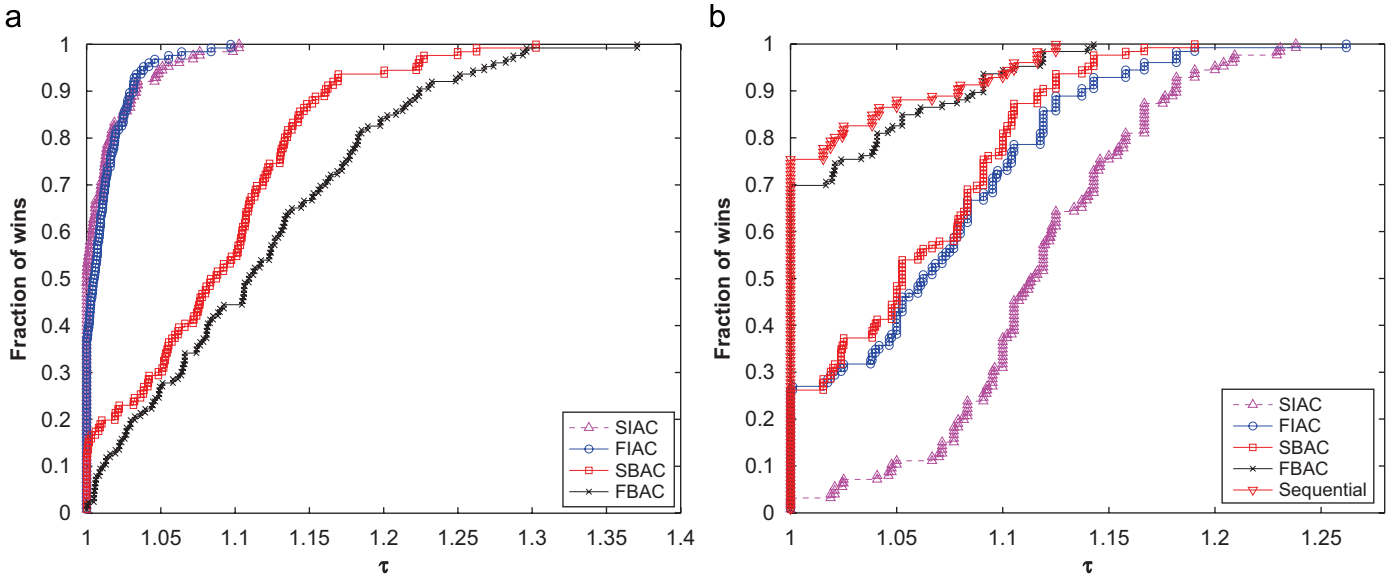


Fig. 5. *Algorithms compared*: variants xIAC and xBAC of SPCFRAMEWORK where the color selection strategy x is either F (First Fit) or S (Staggered FF). *Plots*: performance profiles. *Test set*: application graphs listed in Table 2. *Platform*: Itanium 2, $p = 40$, $s = 800$. (a) Runtime, (b) number of colors.

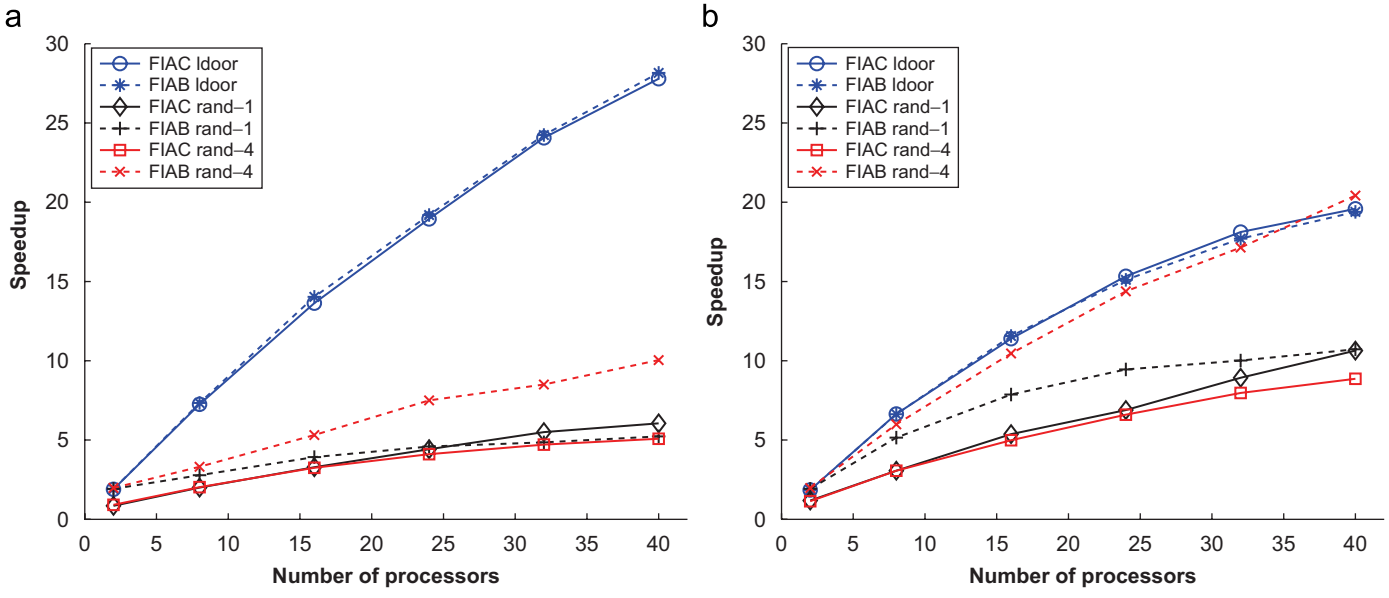


Fig. 6. *Algorithms compared*: variant FIAx of SPCFRAMEWORK where the communication mode x is either Customized (C) or Broadcast-based (B). *Plots*: standard. *Metric*: speedup. For FIAC, $s = 800$; for FIAB, $s = 125$. (a) Itanium 2, (b) Pentium 4.

4.4.2. Comparison with related algorithms

In our second set of experiments, whose results are summarized in the performance profile plots shown in Figs. 3 and 4, we compare FIAC and FBAC with the three other algorithms in our study: JP, SCR, and SBC. A more elaborate set of experimental data comparing these algorithms is available in Tables A1–A5 in the Appendix.

The conclusion to be drawn from the runtime plots in Fig. 3 is straightforward: algorithms FIAC and FBAC significantly outperform their competitors, which could further be ranked in a decreasing order of execution speed as SCR, JP, and SBC. The underlying reasons for this phenomenon are obvious, but it is worth pointing out that the relative slowness of the JP algorithm is due to the many rounds it requires.

In terms of number of colors, Fig. 4 shows that, in general, algorithms JP and SCR use (slightly) fewer colors than both FBAC and FIAC. Notice that the difference in the quality of the solution obtained by the various algorithms here is fairly marginal: the τ values for more than 90% of the test cases range between 1 and 1.15, indicating that the worst algorithm (FIAC)

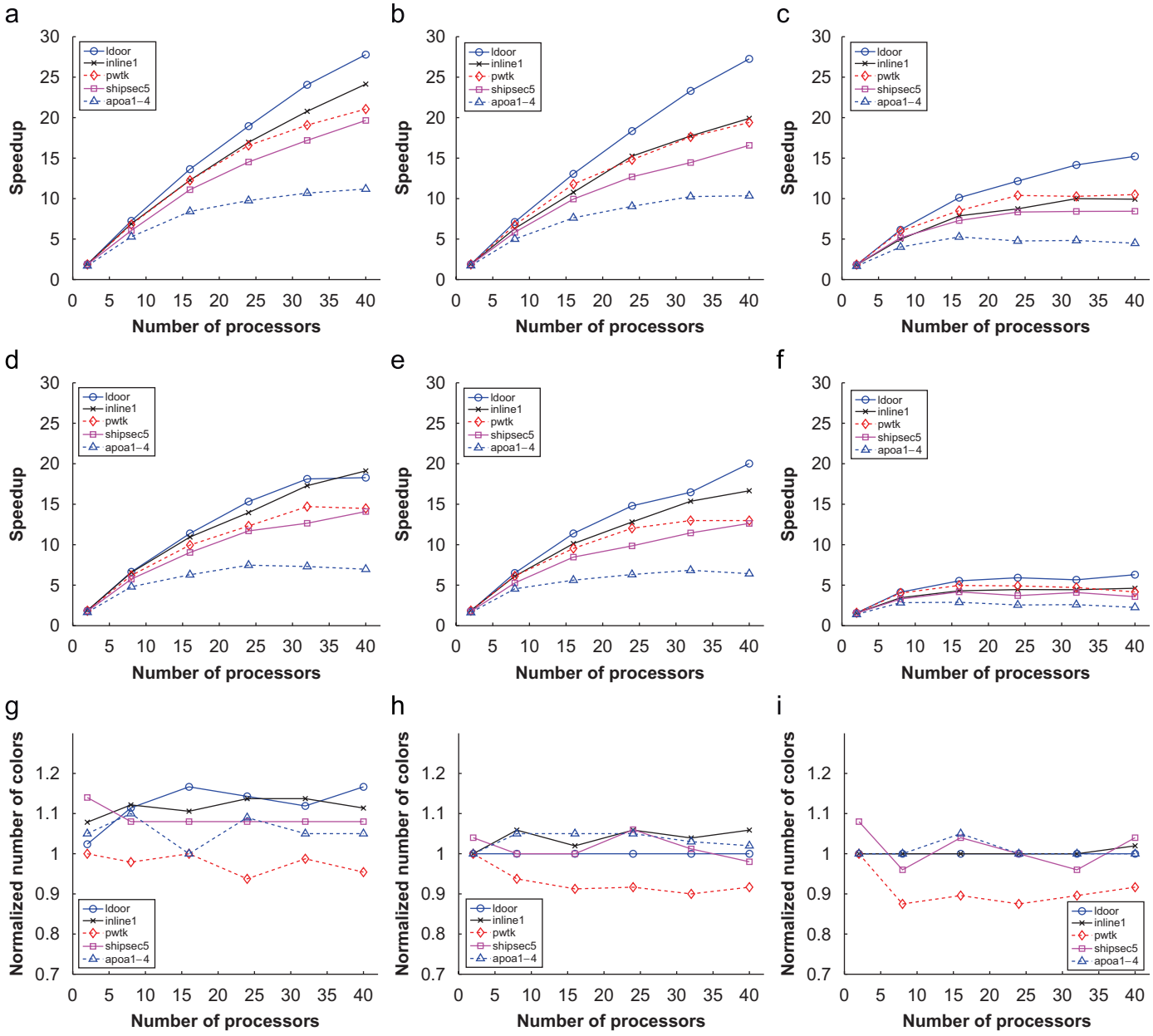


Fig. 7. Performance on select application graphs using $s = 800$. (a) FIAC on I2, (b) FBAC on I2, (c) JP on I2, (d) FIAC on P4, (e) FBAC on P4, (f) JP on P4, (g) FIAC, (h) FBAC, (i) JP.

uses in the worst case, only 15% more colors than that used by the best algorithm (JP).

4.4.3. *Effects of color selection*

The purpose of our third set of experiments is to compare the FF and SFF color selection strategies while using the xIAC and xBAC configuration, i.e., algorithms FIAC, SIAC, FBAC and SBAC. For SFF, we used the number of colors obtained by running the sequential FF algorithm as our estimate for K. Fig. 5 shows performance profile plots corresponding to these experiments.

As can be seen in Fig. 5(a), there is little difference in runtime between FF and SFF when used in combination with an interior-first coloring order (xIAC), but with boundary-first ordering (xBAC), SFF is slightly faster. This is expected since SFF is likely to involve fewer conflicts than FF. Fig. 5(b) shows, again as expected, that SFF uses more colors than FF in both cases.

4.4.4. *Effects of communication mode*

In the fourth set of experiments we consider the FIAC and FIAB configurations in order to compare the two communication modes, Customized and Broadcast. We consider three

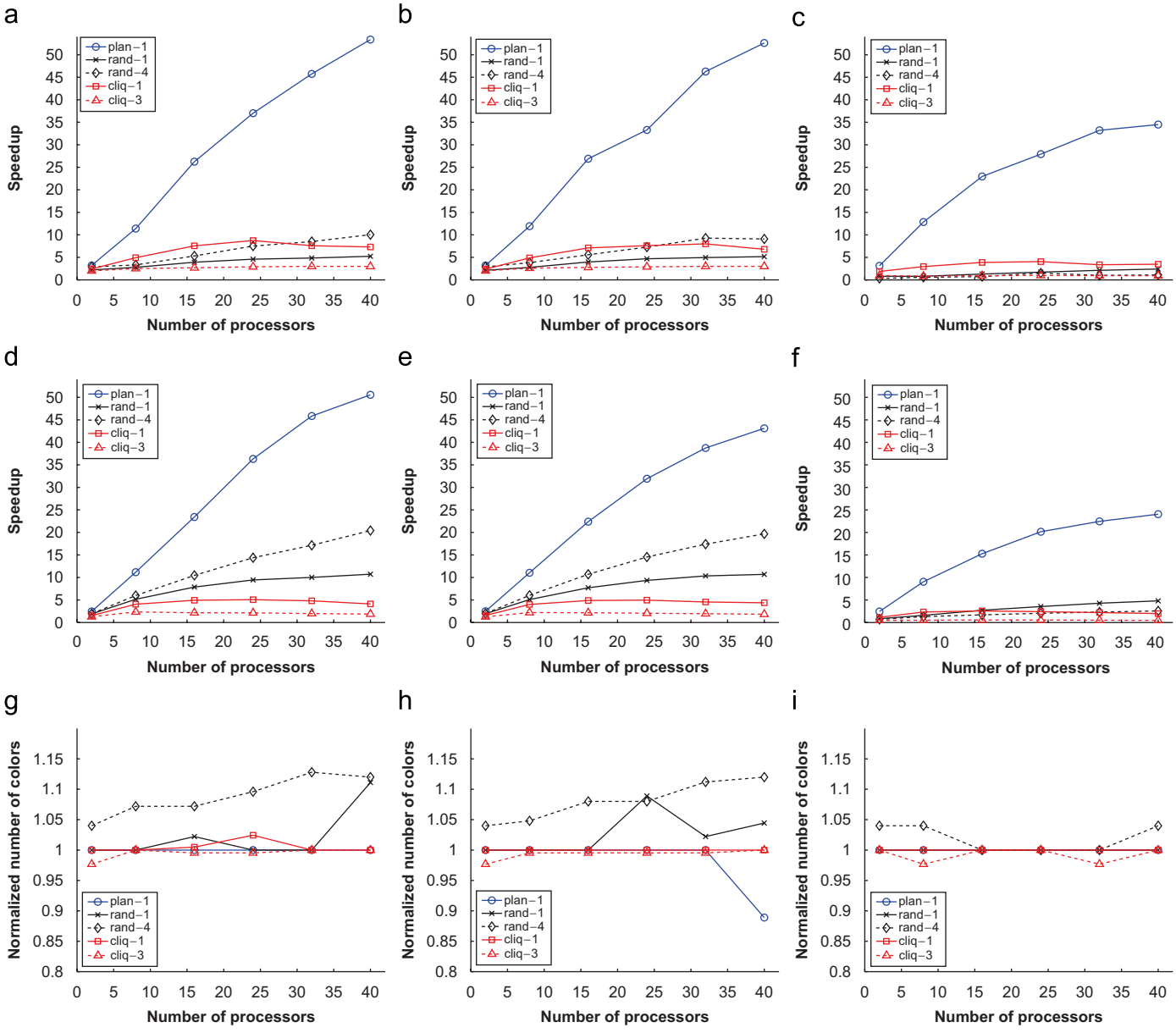


Fig. 8. Performance on select synthetic graphs using $s = 125$. (a) FIAB on I2, (b) FBAB on I2, (c) JP on I2, (d) FIAB on P4, (e) FBAB on P4, (f) JP on P4, (g) FIAB, (h) FBAB, (i) JP.

graphs: the largest application graph ldoor from Table 2 and the two random graphs rand-1 and rand-4 (which are of varying density) from Table 3. These examples are chosen to be representatives of *well partitionable* (requiring low communication) and *poorly partitionable* (requiring high communication) graphs. As can be seen in the tables in the Appendix, the ratio of boundary vertices to total vertices for $p = 24$ (say) is only 4% for ldoor whereas it is nearly 100% for the random graphs.

Fig. 6 shows speedup plots for algorithms FIAC and FIAB on the two test platforms Itanium 2 and Pentium 4. One can see that the difference in performance between FIAB and FIAC is negligible for the well-partitionable graph ldoor, whereas FIAB is in general better than FIAC for the random graphs, the difference being more pronounced for the denser graph rand-4.

4.5. Results and discussion: scalability

From the four set of experiments we have discussed thus far, we can fairly conclude that algorithms FIAC and FBAC are the best choices (offering runtime-quality tradeoff) for well-partitionable graphs, while the corresponding choices for poorly partitionable graphs are FIAB and FBAB. The results we presented so far showed how the various algorithms *compared* against each other, but not the performance of each considered separately. We now present results where the latter aspect

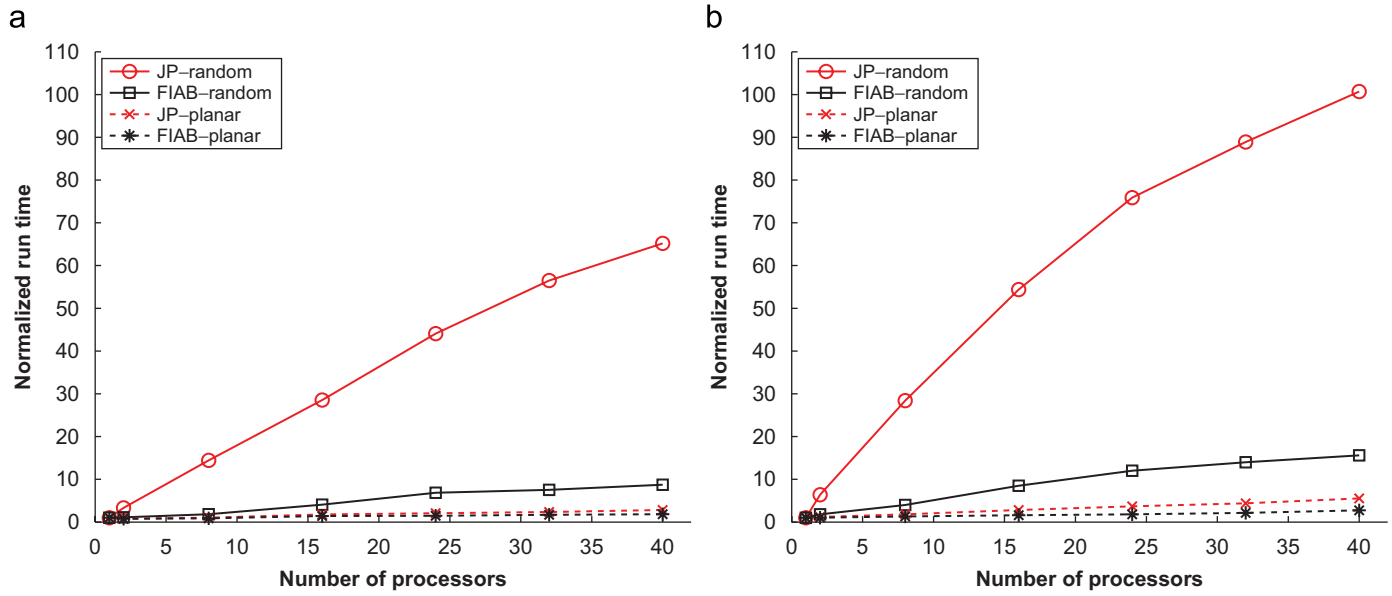


Fig. 9. Weak scaling of algorithms FIAB and JP on select synthetic graphs, $s = 125$. (a) Itanium 2, (b) Pentium 4.

is addressed for the configurations FIAC, FBAC, FIAB and FBAB.

4.5.1. Strong scalability

Application graphs: Figs. 7(a) through (c) show strong scalability results regarding algorithms FIAC, FBAC, and JP on five test graphs drawn from Table 2. The test graphs are selected from five different application areas, and each graph is the largest in its group. The test platform used is Itanium 2. A similar set of results for the Pentium 4 platform is shown in Figs. 7(d)–(f). Figs. 7(g)–(i) show the number of colors used by the three algorithms, each normalized with respect to the number obtained using a sequential FF algorithm.

The results in Fig. 7 show that both FIAC and FBAC scale well for these graphs in both platforms. In general we observe higher speedup for sparser graphs. The JP algorithm showed some speedup, albeit small, largely due to the well-partitionability of the graphs. We also observe that the number of colors used by the three algorithms is fairly close the number used by a sequential algorithm.

Synthetic graphs: Fig. 8 shows a set of figures similar to those shown in Fig. 7, except now that the algorithms considered are FIAB, FBAB and JP and the test graphs are five synthetic graphs drawn from Table 3. One of the five graphs, planar-1, is extremely well partitionable (almost all of its vertices are interior in a given partition, see Table A4). The opposite is true for the other four graphs, rand-1, rand-4, cliq-1 and cliq-3.

The strong scalability results in Figs. 8(a) through (f) agree well with one's expectation: good speedup on well-partitionable graphs and poor speedup on poorly partitionable graphs. The fact that both FIAB and FBAB actually yielded some reasonable speedup (especially on the P4 platform) for the random and clique-based graphs, however, is worth emphasizing: for such graphs the speedup stems solely from the "core" algorithm in SPCFRAMEWORK, without any contribution from parallelization due to partitioning.

The speedup results in Figs. 7 and 8 also show how the nature of a platform affects the performance of the algorithms. For poorly partitionable graphs, where relatively high interprocessor communication is needed due to the large number of boundary vertices, the Pentium 4 cluster with its lower latency and higher network bandwidth, gave better scalability. For well-partitionable graphs, on the other hand, although linear or even super-linear speedup can be achieved on both platforms, the Itanium 2 cluster was observed to be a better environment, primarily due to its larger cache.

4.5.2. Weak scalability

Our final experimental result is on weak scalability, where both problem size and number of processors are increased in proportion so as to result in nearly constant runtime. Fig. 9 shows weak scaling result on random and planar graphs. (The complete experimental data are available in Table A5.)

From Fig. 9 we observe that runtime grows slowly with increasing number of processors for both JP and FIAB on planar graphs, but there is a dramatic difference for random graphs. While FIAB takes only slightly more than a constant time on planar graphs, the runtime for JP grows almost linearly with number of processors. This demonstrates that our framework (FIAB in particular) is clearly more scalable than JP on random graphs. This is due to the large number of boundary vertices, where JP performs poorly due to the many rounds it requires.

5. Conclusion

We have developed an efficient and scalable parallel graph coloring algorithmic framework suitable for distributed-memory computers. The framework is tunable to suit the structure of the graph being colored and the specifics of the hardware being used.

The scalability of several algorithms developed using the framework has been experimentally demonstrated on platforms of modest size processors. For almost all graphs tested in our experiments, the variants of our framework yielded fairly good speedup with increasing number of processors and the number of colors used was close to the number used by a sequential greedy algorithm. Furthermore, we observed that customizing inter-processor communication and using a superstep size in the order of a thousand improved the scalability of the algorithms for structured graphs. For unstructured graphs, good performance was observed by using a broadcast-based communication mode and using a superstep size close to a hundred. The computational results we obtained also suggest that, coloring interior vertices first gives a faster and slightly more scalable algorithm whereas an algorithm in which boundary vertices are colored first uses fewer colors.

Our implementation has been incorporated into Zoltan, a freely available parallel data management and load-balancing library [7]. We plan to extend our work to other coloring problems such as the distance-2 graph coloring and the hypergraph coloring problems, which arise in the efficient computation of derivative matrices. A preliminary work we have done on parallel distance-2 coloring is available in [5].

Acknowledgments

We thank the anonymous reviewers for their valuable comments that helped us improve the paper significantly.

Appendix A.

A set of experimental data comparing FIAC and FBAC with the three other algorithms in our study: JP, SCR, and SBC is shown in Tables A1–A5.

Table A1
Comparison of algorithms for the application graphs on the Itanium 2 cluster using $s = 800$ and $p = 8$

Name	Speedup					Colors				Rounds			$\frac{\lvert V_B\rvert}{\lvert V\rvert}$ (%)	$\frac{\text{time on } V_B}{\text{total time}}$ (%)		
	FIAC	FBAC	JP	SCR	SBC	FIAC	FBAC	JP	SCR	FIAC	FBAC	JP		FIAC	FBAC	JP
popc-br-4	4.5	3.7	2.5	4.6	2.6	21	20	19	20	4	4	22	22	56	56	67
er-gre-4	4.7	4.4	3.3	4.7	2.2	22	19	19	19	4	5	21	27	56	56	66
apoa1-4	5.3	5.0	4.0	5.0	2.9	21	20	20	20	5	5	22	18	43	43	53
144	5.8	5.6	5.1	6.0	4.3	13	12	12	12	4	4	14	10	34	33	39
598a	6.2	5.9	5.2	6.1	4.6	12	12	11	11	4	4	13	9	30	30	38
auto	6.4	6.3	5.2	7.0	5.6	13	13	13	13	4	5	14	6	28	27	39
bmw3_2	6.6	6.3	5.6	6.6	4.9	51	48	48	48	5	5	30	6	20	20	26
bmw7st1	6.5	6.1	5.4	6.6	4.7	54	53	52	54	5	6	31	6	21	23	31
inline1	6.9	6.3	5.0	6.8	5.6	58	52	51	51	5	5	45	4	16	19	25
pwtk	6.8	6.7	6.0	6.8	5.8	45	43	42	45	5	6	30	3	15	12	22
nasasrb	6.0	5.6	4.5	6.1	4.0	38	40	40	40	5	5	30	10	29	29	43
ct20stif	5.4	4.3	3.8	5.4	3.1	49	50	48	49	5	5	39	15	43	50	55
hood	6.9	6.8	5.8	7.0	6.0	45	42	42	42	5	5	29	3	14	12	23
msdoor	7.1	6.8	6.0	7.2	6.3	42	42	42	42	5	5	29	2	11	12	16
ldoor	7.3	7.1	6.1	7.3	6.8	47	43	43	42	5	5	29	2	8	8	21
pkutsk10	6.1	5.6	4.6	6.0	3.6	50	46	45	46	5	7	39	12	30	33	45
pkutsk11	6.0	5.7	4.5	5.9	3.3	66	66	66	66	5	5	43	13	35	35	48
pkutsk13	5.8	5.2	4.2	5.6	2.9	57	57	57	57	4	5	46	16	38	41	52
shipsec1	6.2	6.1	5.1	6.1	4.0	53	50	48	50	6	6	36	10	28	27	39
shipsec5	6.1	5.8	5.2	6.1	3.9	54	48	49	51	5	5	36	10	29	28	36
shipsec8	6.0	5.7	4.8	6.0	3.7	54	50	48	50	5	6	39	13	33	32	44

$|V_B|$ denotes the total number of boundary vertices after partitioning.

Table A2
Comparison of algorithms for the application graphs on the Itanium 2 cluster using $s = 800$ and $p = 24$

Name	Speedup					Colors				Rounds			$\frac{\lvert V_B\rvert}{\lvert V\rvert}$ (%)	$\frac{\text{time on } V_B}{\text{total time}}$ (%)		
	FIAC	FBAC	JP	SCR	SBC	FIAC	FBAC	JP	SCR	FIAC	FBAC	JP		FIAC	FBAC	JP
popc-br-4	6.1	4.8	1.5	4.3	1.9	20	20	20	19	4	5	22	37	82	81	81
er-gre-4	7.4	6.1	3.2	4.9	1.6	20	20	21	20	4	5	20	43	81	83	91
apoa1-4	9.8	9.0	4.8	6.8	2.2	21	20	21	20	5	5	25	32	71	68	83
144	12.3	10.4	7.2	8.5	3.9	13	13	13	12	4	5	15	18	58	61	73
598a	11.8	9.7	6.8	8.2	3.9	12	12	11	12	4	5	13	17	61	64	76
auto	15.2	13.4	9.3	11.9	5.6	14	13	13	13	4	5	15	13	49	51	65
bmw3_2	16.1	14.4	9.6	13.3	5.3	49	49	48	48	5	5	31	11	38	38	59
bmw7st1	14.9	12.9	3.6	12.8	4.6	49	52	48	54	5	6	34	13	44	47	48
inline1	17.0	15.2	8.7	14.8	5.4	57	52	51	52	6	6	58	10	35	36	64
pwtk	16.6	14.8	10.4	12.6	5.4	45	43	41	44	5	6	32	11	36	38	58
nasasrb	11.8	10.1	5.0	8.2	2.7	45	41	39	38	5	5	32	25	63	63	83
ct20stif	10.4	8.4	3.7	7.7	2.3	54	51	47	48	5	6	43	27	69	66	84
hood	17.2	15.6	10.0	15.3	6.5	43	42	42	42	5	6	31	8	32	34	58
msdoor	18.0	16.8	11.2	17.1	9.2	46	42	42	42	5	5	30	5	27	28	52
ldoor	19.0	18.3	12.2	18.3	11.0	49	42	42	42	6	6	32	4	22	21	48
pkutsk10	13.0	11.1	6.3	9.6	2.9	48	45	44	47	5	5	33	23	58	58	76
pkutsk11	12.5	10.7	7.0	9.3	2.2	66	65	62	61	5	5	48	26	64	64	83
pkutsk13	12.3	9.9	7.5	9.2	2.3	57	57	57	57	5	6	49	29	64	65	82
shipsec1	13.8	11.8	7.7	9.4	3.1	50	48	47	47	5	6	36	21	53	54	71
shipsec5	14.5	12.7	8.3	10.2	3.5	53	52	49	48	5	6	37	18	49	51	67
shipsec8	13.2	11.3	6.2	9.0	2.8	55	53	47	49	5	5	40	24	57	58	90

$|V_B|$ denotes the total number of boundary vertices after partitioning.

Table A3
Comparison of algorithms for the application graphs on the Itanium 2 cluster using $s = 800$ and $p = 40$

Name	Speedup					Colors				Rounds			$\frac{\lvert V_B\rvert}{\lvert V\rvert}$ (%)	$\frac{\text{time on } V_B}{\text{total time}}$ (%)		
	FIAC	FBAC	JP	SCR	SBC	FIAC	FBAC	JP	SCR	FIAC	FBAC	JP		FIAC	FBAC	JP
popc-br-4	5.4	4.5	1.2	3.0	1.5	22	21	19	20	4	5	24	46	91	90	97
er-gre-4	6.8	6.5	2.1	3.9	1.3	21	20	20	18	5	5	24	51	90	89	97
apoa1-4	11.2	10.3	4.5	5.9	1.8	21	20	20	20	5	5	22	38	81	78	91
144	13.1	12.0	6.2	7.8	3.3	13	12	12	12	5	5	17	22	75	74	87
598a	11.6	10.5	5.4	7.1	3.3	13	12	11	11	5	5	15	22	78	78	88
auto	19.6	16.5	14.4	12.2	4.8	14	13	13	13	4	5	16	16	62	64	76
bmw3_2	21.8	18.6	9.2	14.5	4.3	48	48	48	48	5	6	34	14	52	51	76
bmw7st1	19.2	15.2	7.5	12.6	3.6	54	48	49	53	5	7	35	19	59	63	81
inline1	24.1	19.9	9.9	16.7	4.5	57	55	51	51	5	6	57	14	48	51	77
pwtk	21.1	19.4	10.4	12.6	4.1	47	44	42	44	6	6	34	16	54	55	78
nasasrb	13.2	10.9	4.6	7.2	2.1	43	40	38	39	5	6	33	32	77	77	94
ct20stif	11.8	10.3	3.6	7.2	1.8	52	51	48	48	5	5	43	36	81	77	94
hood	23.6	20.2	10.8	17.1	5.5	45	42	42	42	5	6	30	11	46	49	74
msdoor	26.1	23.9	10.9	21.3	8.4	47	42	42	42	5	5	40	7	37	38	72
ldoor	27.8	27.3	15.2	24.3	10.1	47	42	42	42	6	5	31	5	32	31	61
pkutsk10	14.7	14.6	5.0	9.4	2.5	48	47	44	48	6	5	39	28	73	68	89
pkutsk11	14.8	13.7	4.3	8.8	1.8	64	60	58	62	6	5	55	33	77	72	91
pkutsk13	15.9	13.8	4.9	9.3	1.8	58	57	57	57	5	5	54	37	75	72	90
shipsec1	17.9	15.2	6.5	9.7	2.6	53	50	45	49	5	6	44	26	66	66	85
shipsec5	19.7	16.6	8.4	10.7	2.9	54	53	49	50	5	6	38	23	61	62	81
shipsec8	16.6	14.7	6.0	9.2	2.2	54	52	50	51	6	5	42	29	70	68	87

$|V_B|$ denotes the total number of boundary vertices after partitioning.

Table A4
Comparison of algorithms for the synthetic graphs using $s = 125$

p	Name	$\frac{\lvert V_B\rvert}{\lvert V\rvert}$ (%)	Colors			Speedup on Itanium 2			Speedup on Pentium 4		
			FIAC	FBAC	JP	FIAC	FBAC	JP	FIAC	FBAC	JP
8	rand-1	99.5	9	9	9	2.8	2.8	0.8	5.1	5.1	1.6
	rand-2	100.0	16	16	15	3.1	3.1	0.5	5.7	5.6	1.3
	rand-3	100.0	21	21	21	3.2	3.1	0.5	5.8	5.8	1.3
	rand-4	100.0	27	26	26	3.4	3.8	0.6	6.0	6.0	1.3
	plan-1	0.0	9	9	9	11.4	11.9	12.8	11.2	11.1	9.0
	cliq-1	64.8	41	41	41	4.9	4.9	3.0	4.1	4.0	2.3
	cliq-2	100.0	42	42	42	2.4	2.4	1.0	2.1	2.0	0.6
	cliq-3	100.0	43	43	42	2.5	2.6	0.8	2.3	2.2	0.5
24	rand-1	99.8	9	10	9	4.6	4.7	1.7	9.4	9.3	3.5
	rand-2	100.0	16	16	15	6.2	6.2	1.1	12.5	12.2	2.5
	rand-3	100.0	22	22	21	6.3	6.4	1.4	12.5	13.0	2.2
	rand-4	100.0	27	27	25	7.5	7.3	1.5	14.4	14.5	2.0
	plan-1	0.1	9	9	9	37.0	33.3	27.9	36.3	31.9	20.2
	cliq-1	60.3	42	41	41	8.7	7.6	4.1	5.1	5.0	2.4
	cliq-2	100.0	42	42	42	2.9	2.9	1.3	2.1	2.0	0.7
	cliq-3	100.0	43	43	43	2.9	2.9	1.0	2.1	2.0	0.5
40	rand-1	99.8	10	9	9	5.2	5.2	2.4	10.7	10.7	4.8
	rand-2	100.0	17	17	15	7.2	6.7	1.4	16.5	16.3	3.3
	rand-3	100.0	23	23	20	8.2	8.0	1.1	18.4	18.5	2.8
	rand-4	100.0	28	28	26	10.0	9.1	1.1	20.4	19.7	2.6
	plan-1	0.2	9	8	9	53.4	52.6	34.5	50.6	43.1	24.1
	cliq-1	63.3	41	41	41	7.3	6.8	3.5	4.1	4.4	2.0
	cliq-2	100.0	42	42	42	2.9	2.9	1.1	1.8	1.8	0.6
	cliq-3	100.0	43	43	43	3.0	3.0	0.9	1.9	1.8	0.5

$|V_B|$ denotes the total number of boundary vertices after partitioning.

Table A5
Weak scaling on synthetic graphs using $s = 125$

Name	Degree		Seq. colors	p	$\lvert\overline{V_I}\rvert$	$\lvert V_B\rvert$	Colors			Runtime on Itanium 2			Runtime on Pentium 4		
	Max	Avg					FIAC	FBAC	JP	FIAC	FBAC	JP	FIAC	FBAC	JP
rand-5	80	50	20	1	10,000	0	20	20	20	26	26	26	10	10	10
rand-6	79	50	20	2	0	20,000	20	20	20	29	29	86	18	17	64
rand-7	82	50	20	8	0	80,000	22	22	21	48	80	375	40	46	284
rand-8	91	50	21	16	0	160,000	22	22	20	106	110	742	85	88	544
rand-9	86	50	21	24	0	240,000	22	22	21	178	142	1146	120	121	759
rand-10	86	50	20	32	0	320,000	23	22	21	196	193	1469	140	140	889
rand-11	84	50	20	40	0	400,000	23	23	20	227	232	1695	156	155	1007
plan-2	35	6	8	1	102,093	0	8	8	8	49	49	49	28	28	28
plan-3	29	6	8	2	101,988	74	8	8	8	38	38	39	29	31	33
plan-4	32	6	9	8	102,248	1,521	9	9	9	43	62	50	37	39	51
plan-5	37	6	8	16	101,448	2,811	8	8	8	69	69	87	45	51	79
plan-6	40	6	9	24	101,522	4,539	9	9	9	72	76	102	51	61	103
plan-7	43	6	9	32	101,394	6,347	9	9	9	83	83	116	60	71	123
plan-8	40	6	9	40	101,655	6,750	9	8	9	90	91	139	77	86	155

$|\overline{V_I}|$ is the average number of internal vertices on each processor, and $|V_B|$ is the total number of boundary vertices after partitioning.

References

[1] J. Allwright, R. Bordawekar, P.D. Coddington, K. Dincer, C. Martin, A comparison of parallel graph coloring algorithms, Technical Report, NPAC Technical Report, SCCS-666, Northeast Parallel Architectures Center at Syracuse University, 1994.

[2] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, Complexity and Approximation, Springer, Berlin, 1999.

[3] R.H. Bisseling, Parallel Scientific Computation: A Structured Approach Using BSP and MPI, Oxford, 2004.

[4] E.G. Boman, D. Bozdağ, U. Catalyurek, A.H. Gebremedhin, F. Manne, A scalable parallel graph coloring algorithm for distributed memory computers, in: Proceedings of Euro-Par 2005, Lecture Notes in Computer Science, vol. 3648, Springer, Berlin, 2005, pp. 241–251.

[5] D. Bozdağ, U. Catalyurek, A.H. Gebremedhin, F. Manne, E.G. Boman, F. Özgüner, A parallel distance-2 graph coloring algorithm for distributed memory computers, in: Proceedings of HPCC 2005, Lecture Notes in Computer Science, vol. 3726, Springer, Berlin, 2005, pp. 796–806.

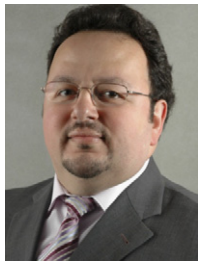
[6] T.F. Coleman, J.J. More, Estimation of sparse Jacobian matrices and graph coloring problems, SIAM J. Numer. Anal. 1 (20) (1983) 187–209.
[7] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, C. Vaughan, Zoltan data management services for parallel dynamic applications, Comput. Sci. Eng. 4 (2) (2002) 90–97.
[8] E.D. Dolan, J.J. Moré, Benchmarking optimization software with performance profiles, Math. Programming 91 (2) (2002) 201–213.
[9] I. Finocchi, A. Panconesi, R. Silvestri, Experimental analysis of simple, distributed vertex coloring algorithms, Algorithmica 41 (1) (2004) 1–23.
[10] M.R. Garey, D.S. Johnson, Computers and Intractability, Freeman, New York, 1979.
[11] A.H. Gebremedhin, I. Guérin-Lassous, J. Gustedt, J.A. Telle, Graph coloring on coarse grained multicomputers, Discrete Appl. Math. 131 (1) (2003) 179–198.
[12] A.H. Gebremedhin, F. Manne, Scalable parallel graph coloring algorithms, Concurrency: Practice and Experience 12 (2000) 1131–1146.
[13] A.H. Gebremedhin, F. Manne, A. Pothen, Parallel distance-k coloring algorithms for numerical optimization, in: Proceedings of Euro-Par 2002, Lecture Notes in Computer Science, vol. 2400, Springer, Berlin, 2002, pp. 912–921.
[14] A.H. Gebremedhin, F. Manne, A. Pothen, What color is your jacobian? Graph coloring for computing derivatives, SIAM Rev. 47 (4) (2005) 629–705.
[15] A.H. Gebremedhin, F. Manne, T. Woods, Speeding up parallel graph coloring, in: Proceedings of Para 2004, Lecture Notes in Computer Science, vol. 3732, Springer, Berlin, 2005, pp. 1079–1088.
[16] Genetic algorithms (evolutionary algorithms): repository of test problem generators ⟨http://www.cs.uwyo.edu/∼ wspears/generators.html⟩.
[17] R.K. Gjertsen Jr., M.T. Jones, P. Plassmann, Parallel heuristics for improved, balanced graph colorings, J. Par. and Dist. Comput. 37 (1996) 171–186.
[18] Gtgraph: a suite of synthetic graph generators ⟨http://www-static.cc.gatech.edu/∼kamesh/GTgraph/⟩.
[19] Ö. Johansson, Simple distributed $\delta + 1$-coloring of graphs, Inform. Process. Lett. 70 (1999) 229–232.
[20] M. Jones, P. Plassmann, Scalable iterative solution of sparse linear systems, Parallel Comput. 20 (5) (1994) 753–773.
[21] M.T. Jones, P. Plassmann, A parallel graph coloring heuristic, SIAM J. Sci. Comput. 14 (3) (1993) 654–669.
[22] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20(1).
[23] M. Luby, A simple parallel algorithm for the maximal independent set problem, SIAM J. Comput. 15 (4) (1986) 1036–1053.
[24] F. Manne, A parallel algorithm for computing the extremal eigenvalues of very large sparse matrices, in: Proceedings of Para 1998, Lecture Notes in Computer Science, vol. 1541, Springer, Berlin, 1998, pp. 332–336.
[25] C.A. Morgenstern, H.D. Shapiro, Heuristics for rapidly four-coloring large planar graphs, Algorithmica 6 (6) (1991) 869–891.
[26] Y. Saad, ILUM: a multi-elimination ILU preconditioner for general sparse matrices, SIAM J. Sci. Comput. 17 (1996) 830–847.
[27] M.M. Strout, L. Carter, J. Ferrante, J. Freeman, B. Kreaseck, Combining performance aspects of irregular Gauss–Seidel via sparse tiling, in: W. Pugh, C.-W. Tseng (Eds.), LCPC, Lecture Notes in Computer Science, vol. 2481, Springer, Berlin, 2002, pp. 90–110.
[28] M.M. Strout, P.D. Hovland, Metrics and models for reordering transformations, in: Proceedings of the Second ACM SIGPLAN Workshop on Memory System Performance (MSP), 2004, pp. 23–34.
[29] Test data from the Parasol project ⟨http://www.parallab.uib.no/projects/parasol/data/⟩.
[30] The Second DIMACS Challenge ⟨http://mat.gsia.cmu.edu/challenge.html⟩.
[31] University of Florida matrix collection ⟨http://www.cise.ufl.edu/research/sparse/matrices/⟩.
[32] L. Valiant, A bridging model for parallel computation, Comm. ACM 33 (8) (1990) 103–111 bSP.

Doruk Bozdağ is a graduate student in the Department of Electrical and Computer Engineering at The Ohio State University. His research interests include scheduling algorithms for multiprocessor systems, parallel graph algorithms and high-performance computing. He received his MS in Electrical and Computer Engineering from The Ohio State University in 2005 and BS in Electrical and Electronic Engineering and BS in Physics from Boğaziçi University, Turkey, in 2002.

Assefaw H. Gebremedhin is a research scientist at the Department of Computer Science and Center for Computational Sciences at Old Dominion University, VA, USA. He is a member of the Combinatorial Scientific Computing and Petascale Simulations (CSCAPES) Institute, which is sponsored by the US Department of Energy through its Scientific Discovery through Advanced Computing (SciDAC) program. Assefaw received his PhD in Computer Science from the University of Bergen, Norway, in 2003. His academic background also includes a BSc degree in Electrical Engineering. His research interests are in the areas of algorithmics, parallel computing, combinatorial scientific computing, automatic differentiation, and software development.

Fredrik Manne is a full professor at the Department of Informatics, University of Bergen, Norway. He also received his PhD from the same university in 1993. Previous to joining the University of Bergen as a faculty member he worked in the oil industry as well as at Parallab, the supercomputing centre of the University of Bergen. His main research area is in parallel and distributed computing with a particular emphasis on problems arising in combinatorial scientific computing.

Erik G. Boman is a scientist at Sandia National Laboratories, NM, USA. He received a PhD in Scientific Computing and Computational Mathematics from Stanford University (1999) and also holds a Cand.Scient. (MS) degree in Informatics from the University of Bergen, Norway, (1992). His current research interests are in combinatorial scientific computing and algorithms for parallel computing. He has extensive experience in algorithms for partitioning and load balancing, and is a co-author of the Zoltan software toolkit. He has also published several journal papers in numerical linear algebra and preconditioning methods.



Umit V. Catalyurek is an associate professor in the Department of Biomedical Informatics at The Ohio State University, and has a joint faculty appointment in the Department of Electrical and Computer Engineering. His research interests include combinatorial scientific computing, grid computing, and runtime systems and algorithms for high-performance and data-intensive computing. He received his PhD, MS and BS in Computer Engineering and Information Science from Bilkent University, Turkey, in 2000, 1994 and 1992, respectively.