Practical Parallel Algorithms for Graph Coloring Problems in Numerical Optimization

Assefaw Hadish Gebremedhin

Thesis submitted in partial fulfillment of the
requirements for the degree of
Doctor Scientiarum

Department of Informatics
University of Bergen
Norway

February 2003

Acknowledgements

This work was financially supported by the University of Bergen through a research fellowship at the Department of Informatics.

I am indebted to many people for their contribution to the successful completion of this work.

I would first like to thank my advisor Professor Fredrik Manne for his support and guidance throughout the work with this thesis. Fredrik has always been encouraging, meticulous in reading my drafts, generous with his time, and his advices on both academic and non-academic matters have been invaluable.

I would also like to thank the other people whom I had the pleasure of working with on parts of this thesis. Thanks to Alex Pothen for facilitating my visit to Old Dominion University, Virginia, USA in the spring of 2001, for making the stay enjoyable, and for our continued collaboration since then. Thanks also to the Pothen family for their hospitality. I thank Jens Gustedt, Isabelle Guérin Lassous and Jan Arne Telle for our joint research work and friendship.

Thanks to friends and colleagues at the Department of Informatics for creating a warm and reliable working place. I also thank all my fellow countrymen living and studying in Bergen for their friendship and the memorable get-togethers.

I am grateful to my parents Hadish Gebremedhin and Alganesh Tsegay, my sisters Menen and Wintana, my brothers Theodros, Fasil, Alula and Eyob for their support and encouragement.

Finally, I thank my wife Tsigeweini for her love, support and patience. Without her constant encouragement and never-fading understanding, I could not have done this.

I dedicate this thesis to our little daughter Melody with love.

Bergen, February 2003

Assefaw Hadish Gebremedhin

1 Introduction

Partitioning a set of objects into groups according to certain rules is a fundamental process in mathematics and computer science. Often, the rules that determine whether or not a pair of objects can belong to the same group are conceptually simple. *Graph coloring* is one mathematical abstraction for such partitioning rules.

A graph represents binary relationships. It can be visualized as a diagram consisting of points called vertices, and lines called edges. A vertex represents some physical entity or abstract concept. An edge, which joins exactly two vertices, represents the relationship between them.

In its standard form, the graph coloring problem deals with assigning colors to the vertices of a graph such that adjacent vertices receive different colors and the number of colors used is minimized. This problem and its variants model a wide range of real-world problems in scientific, engineering and computing applications. For example, timetabling and scheduling, frequency assignment to mobile telephones, register allocation in compilers, printed circuit testing, computation of sparse Jacobian and Hessian matrices, partitioning of tasks in parallel computation, are just a few of the many areas in which some variant of the graph coloring problem is used as an effective model.

Among the examples mentioned above, computation of sparse derivative matrices is an application area considered in this study. The computation of Jacobian and Hessian matrices is a phase required in many numerical optimization algorithms. Automatic differentiation and finite differencing are two widely used techniques for computing, or estimating, the entries of these matrices. The computation involved in using the techniques can be made efficient by exploiting matrix structures such as sparsity and symmetry. This leads to several types of partitioning problems that can naturally be modeled using variants of the graph coloring problem.

This thesis is mainly about developing *algorithms* for solving different graph coloring problems. The coloring problems of our concern are all believed to be intractable. Therefore, we focus on finding polynomial-time algorithms that yield sub-optimal solutions. In particular, we emphasize greedy algorithms that give usable solutions quickly.

Like many other industrial problems, the graph coloring instances in our cases can be of very large size. One way of handling large-scale problems effectively is by using *parallel computers*. A parallel computer consists of several processors that work on different parts of a common problem. Today, parallel computers consisting of hundreds of processors are commer-

cially available. The basic idea behind parallel computation is to carry out several tasks simultaneously and thereby reduce execution time. A successful application of this idea requires, among other things, developing parallel algorithms—algorithms that handle the task-division, coordination, and communication among the processors. Most of the coloring algorithms developed in this thesis are parallel.

In general, the design and analysis of algorithms, sequential as well as parallel, presumes the existence of an underlying *computation model*. Unlike the sequential case, there is currently no single unifying standard model for parallel computation. The literature contains an abundant number of models. Among them, the Bulk Synchronous Parallel (BSP) model stands out for being proposed as a standard bridging model between hardware and software in parallel computation. The introduction of BSP stimulated several studies that lead to various adaptations of the model along different directions. The Coarse Grained Multicomputer (CGM) model is one such adaptation. Inspired by the BSP and CGM models, in this study, we propose a new model called the Parallel Resource-Optimal (PRO) model. The PRO model is proposed in an attempt to further narrow the gap between theory and practice in developing efficient and scalable parallel algorithms.

In summary, this thesis deals primarily with the design, analysis, and implementation of practical parallel algorithms for a variety of graph coloring problems. Besides the standard coloring problem, we consider several specialized variants that arise in the computation of sparse Jacobian and Hessian matrices. Secondarily, it is concerned with the development of a simple and effective parallel computation model suitable for theoretical as well as practical purposes. The thesis consists of the following five papers:

I. A.H. Gebremedhin and F. Manne. *Scalable Parallel Graph Coloring Algorithms*. Concurrency: Practice and Experience. 2000; 12:1131-1146.

II. A.H. Gebremedhin, F. Manne and A. Pothen. *Graph Coloring in Optimization Revisited*. January 2003. To be submitted.

III. A.H. Gebremedhin, F. Manne and A. Pothen. *Parallel Distance-k Coloring Algorithms for Numerical Optimization*. In B. Monien and R. Feldmann (Eds.): Proceedings of the 8th International Euro-Par Conference, Paderborn, Germany, August 2002, volume 2400 of Lecture Notes in Computer Science, pages 912-921. Springer-Verlag 2002.

IV. A.H. Gebremedhin, I.Guérin Lassous, J. Gustedt and J.A. Telle. *Graph Coloring on Coarse Grained Multicomputers.* Discrete Applied Mathematics, to appear.

A preliminary version appeared In U. Brandes and D. Wagner (Eds): Proceedings of WG 2000, 26th International Workshop on Graph-Theoretic Concepts in Computer Science, Konstanz, Germany, June 2000, volume 1928 of Lecture Notes in Computer Science, pages 184-195. Springer-Verlag 2000.

V. A.H. Gebremedhin, I.Guérin Lassous, J. Gustedt and J.A. Telle. *PRO: a Model for Parallel Resource-Optimal Computation.* In Proceedings of HPCS'02, 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, Canada, June 2002, pages 106-113. IEEE Computer Society Press 2002.

The versions of Papers I, IV and V included in this thesis are content-wise identical to the published versions. Paper III is slightly extended from its published form: it includes more experimental results and proofs that were omitted for space considerations. Paper II is planned to be submitted for publication; unlike the other papers, it is written with a broader audience in mind.

The topics addressed by Papers I–V can be classified into three groups:

- parallel algorithms for the standard graph coloring problem (Paper I),
- application of graph coloring in sparse Jacobian and Hessian computation (Papers II and III), and
- parallel computation models (Papers IV and V).

The purpose of this common introduction is to highlight the main results in each paper and to put the results in context. It is organized as follows. In Section 2 we discuss some parallel processing issues relevant for the subsequent presentation of the papers. In Sections 3–5, we consider the three topics listed above in their respective order. In particular, in each section, a concise summary of the corresponding paper(s) is given. Paper IV has aspects that make it relevant to both the first and the last topic. It is, however, presented under the last topic in Section 5. We conclude the common introduction in Section 6 with some remarks. The body of the thesis, Papers I–V, follows thereafter.

2 Multiprocessor parallel computing

The ever increasing need for faster solutions and for solving large scale problems is one of the major driving forces behind parallel computing. The primary objective here is to achieve increased computational speed by employing a number of processors concurrently.

Concurrent computation can be done at different levels. Our focus is on using a *multiprocessor parallel computer* where a number of processors communicate and cooperate to solve a *common* computational problem. Multiprocessor parallel computation involves three key ingredients: hardware, software (programming languages, operating systems and compilers), and parallel algorithms [18]. One can use these ingredients as basis for classifying the different kinds of *models* used in the context of parallel processing. Specifically, we identify three distinct, and yet closely related, categories of models called *parallel architecture*, *parallel programming*, and *parallel computation (algorithmic)* model. These categories are introduced merely to frame the discussion in the rest of the current section. The discussion focuses on issues relevant to the presentation of the thesis papers.

2.1 Parallel architecture models

Parallel computers are often divided into different *architectural* classes depending on such issues as *control mechanism* (SIMD and MIMD computers), *address space organization* (shared address space and message-passing architectures) and *interconnection network* (static and dynamic networks). More information on each of these classes can be found in books such as [9, 19].

2.2 Parallel programming models

From a *programming* point of view, shared address space[1] programming (SASP) and message-passing programming (MPP) are perhaps the two most widely used models in contemporary parallel computation.

In the SASP model, programs are viewed as a collection of processes accessing a central pool of shared variables. Data exchange among the processors is achieved by reading from, and writing to, the shared variables. This programming style is naturally suited to the shared-address-space architecture. In the MPP model, programs are viewed as a collection of processes with private local variables and the ability to exchange data via explicit mes-

[1] The phrases 'shared address space' and 'shared memory' are often used interchangeably, both in the literature and here in this thesis. There is, however, a difference in meaning: the former does not require memory to be physically shared.

sage passing. In this model, no variables are shared among processors. Each processor uses its local variables for computation, and whenever necessary, it sends data to, or receives data from, other processors. This programming model fits naturally to distributed-memory computers.

In using the SASP and MPP models, there is usually a trade-off in the choice to be made. MPP requires considerably more programming effort than SASP. This is mainly because MPP requires the data structures in a program to be explicitly partitioned whereas SASP does not. Moreover, SASP supports incremental parallelization of an existing sequential application whereas MPP does not. On the other hand, applications developed using MPP are likely to be more scalable than SASP based applications.

Currently, OpenMP [23] and MPI [21] are generally considered to be the standards for writing portable parallel programs using the SASP and MPP models, respectively. OpenMP, in which the parallel algorithms presented in Papers I and III are implemented, is a directive-based fork-join model for SASP. The fact that OpenMP can be used to write parallel programs relatively easily makes its usage appealing[2].

The experimental work reported in Papers I and III is performed on a Cray Origin 2000 [24], a machine that supports both SASP and MPP. In this machine, memory is physically distributed among the processors. However, for operational purposes, it behaves as a shared memory computer with the operating system taking care of maintaining memory consistency. This enables any processor to use the entire memory of the system.

2.3 Parallel computation models

A computation model is required to serve two major purposes [1]. First, it is used to *describe* a real entity, namely, a computer. As such, a computation model attempts to capture the essential features of a machine while ignoring less important details of its implementation. Second, it is used as a tool for analyzing problems and expressing *algorithms*. In this sense, the model is not necessarily linked to any real computer but rather to an understanding of *computation*.

In the realm of sequential computation, the Random Access Machine (RAM) is a standard model that has succeeded in achieving both of these purposes. It serves as an effective model for hardware designers, algorithm developers and programmers alike. When it comes to parallel computation, there is no such unifying standard model. This is mainly due to the complex

[2]OpenMP dedicated conference series such as EWOMP and WOMPAT could be cited as indicators for the increasing interest in using OpenMP.

set of issues inherent in parallel computation.

The natural parallel analogue of RAM, the PRAM, is an overly simplistic model for parallel computation. It is mainly of theoretical interest as it fails to capture the features of existing parallel computers. The Bulk Synchronous Parallel (BSP) model [26] is a relatively recent model, proposed to serve as a standard 'bridging model' between hardware (machine architecture) and software (algorithm design and programming). As opposed to the PRAM, parallel algorithms in the BSP model are organized in distinct *computation* and *communication* phases. Moreover, unlike the PRAM, the BSP model made parallel computation to be *coarse grained*, a concept we will make more precise shortly. Later, a variant of the BSP called the Coarse Grained Multicomputer (CGM) model was proposed [4, 11, 12]. The CGM model added a strictly coarse grained *communication* requirement to the BSP model. The CGM model uses fewer number of parameters than the BSP model, a feature that simplifies analysis to a certain extent.

In Paper V we propose a new parallel computation model called the Parallel Resource-Optimal (PRO) model. The PRO model inherits the advantages offered by the BSP and CGM models. It compromises between theoretical and practical considerations in the design of optimal and scalable parallel algorithms. Paper V is introduced in Section 5. However, since the BSP and CGM models are mentioned rather briefly in the paper, we discuss the key attributes of the two models in Sections 2.3.2 and 2.3.3 below. The two models are presented here as defined in [4].

We proceed in this section by first recapitulating the main features of the well known PRAM model, which is the model used in Papers I and III for theoretical runtime analysis. In the same section we include a discussion of a related complexity class. A good introduction to the PRAM model and parallel algorithm design using the model can be found in books such as [1, 16].

2.3.1 PRAM

The PRAM is an idealized parallel computation model. In its standard form, it consists of an arbitrarily large number of processors and a shared memory of unbounded size that is uniformly accessible to all processors. The processors share a common clock and operate in lockstep, but they may execute different instructions in each cycle. The PRAM is therefore a model for a synchronous shared memory MIMD computer. The PRAM can be classified into different subclasses depending on how simultaneous memory accesses are handled. For example, a subclass that allows simultaneous read

access, but not write access, to a single memory location is called concurrent read exclusive write (CREW) PRAM.

The PRAM is a model for *fine-grain* computation as it supposes that the number of processors can be arbitrarily large. Usually, it is assumed that the number of processors is a polynomial in the input size. However, practical parallel computation is typically *coarse-grain.* In particular, on most existing parallel machines, the number of processors is several orders of magnitude less than the input size.

Despite its serious deviation from the nature of real parallel computers, the PRAM model is sometimes used for theoretical runtime analysis. A PRAM-analysis gives an idea on the 'computational parallelism' rendered by an algorithm, leaving communication costs aside. Such an analysis should be supported by empirical results in order to make reasonable conclusions. In Papers I and III we use the PRAM model for theoretical analyses and support the claims with experimental results. It should, however, be noted that the algorithms in Paper I and III are not actually PRAM-algorithms. First, they do not assume lockstep synchronization. Second, they are developed in a coarse-grain setting.

The class NC On the PRAM model, a parallel algorithm is considered 'efficient' if its running time is polylogarithmic, i.e., $O(\log^k n)$ for some fixed constant k, while the number of processors it uses is polynomial in the input size n. The class of problems that can be solved by such efficient parallel algorithms is called NC [16]. By simulation, an NC-algorithm can be converted into a polynomial time sequential algorithm. Thus, the class NC is included in the class P, i.e., NC $\subseteq$ P. On the other hand, whether or not P $\subseteq$ NC is an open problem in complexity theory. The general belief is that P $\nsubseteq$ NC, and hence that there are problems in P that do not have NC-algorithms. The class of P-complete problems consists of the most likely candidates for such problems. Informally, a problem is said to be P-complete if an NC-algorithm for it implies that all problems in P have NC-algorithms.

The NC versus P-complete dichotomy on PRAM suffers from several drawbacks. Most importantly, the dichotomy excludes practical parallelizability in a coarse-grain setting and secondly it does not take work optimality into account. A more detailed discussion of these drawbacks is given in Paper V. Moreover, the experimental results reported in Papers I and III demonstrate the relevance of developing coarse-grain algorithms.

2.3.2 The BSP model

A BSP computer is a collection of processor/memory modules connected by a router that can deliver messages in a point to point fashion between the processors. A BSP-algorithm is divided into a sequence of *supersteps* separated by barrier synchronizations. A superstep has distinct *computation* and *communication* phases. In a computation phase the processors perform computation on data that exists locally at the beginning of the superstep. In a communication phase, data is exchanged among the processors via the router. The BSP model uses the four parameters, n, p, L, and g. Parameter n is the problem size, p is the number of processors, L is the minimum time between synchronization steps (measured in basic computation units), and g is the ratio of overall system computational capacity (number of computation operations) per unit time divided by the overall system communication capacity (number of messages of unit size that can be delivered by the router) per unit time.

In a superstep, a processor may send (and receive) at most h messages. Such a communication pattern is called an *h-relation* and the basic task of the router is to realize arbitrary h-relations.

2.3.3 The CGM model

The CGM model is a specialization of the BSP model in which the number of parameters involved is reduced to just two—p and n—making theoretical analysis relatively simpler. The CGM model is a collection of p processors, each with $O(n/p)$ local memory, interconnected by a router that can deliver messages in a point to point fashion. A CGM algorithm consists of an alternating sequence of *computation rounds* and *communication rounds* separated by barrier synchronizations. A computation round is equivalent to the computation phase of a superstep in the BSP model. A communication round consists of a single h-relation with $h \leq n/p$. In other words, all the information sent from one processor to another in one communication round is packed into one long message, thereby minimizing communication overhead.

The CGM model is used for designing parallel algorithms for coarse grain systems, in particular, in the case where $n \gg p$. Usually, a lower bound on n/p, e.g. $n/p \geq p^{\delta}$, for some $\delta > 0$, is required [4]. The value of δ depends on the nature of the problem for which a parallel algorithm is sought. The goodness of a CGM parallel algorithm is often measured by the number of communication rounds it uses [4].

3 Parallel algorithms for standard graph coloring - Paper I

In the simplest case, the graph coloring problem asks for an assignment of positive integers (called colors) to the vertices of a simple, connected, undirected graph such that no two adjacent vertices are assigned the same color and the total number of colors used is minimized.

The graph coloring problem is NP-hard. 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, it is known to be not approximable within $n^{1/7-\epsilon}$ for any $\epsilon > 0$ [3].

Despite these rather pessimistic theoretical results, *greedy* coloring heuristics are often found to be effective in practice. In some applications, these heuristics find colorings that are within small additive constants of the optimal coloring [6, 17]. The number of colors used by a greedy heuristic depends on the order in which the vertices are visited and the choice of color made at each step. If the smallest possible color is chosen at each step, the number of colors used by any such sequential greedy heuristic is at most $\Delta + 1$, where Δ is the maximum degree in the graph. For some applications, this bound can be quite acceptable, especially if the coloring is obtained quickly.

Such greedy heuristics are inherently sequential and consequently difficult to parallelize. Specifically, the problem of coloring the vertices of a graph in a *prespecified* order and using the smallest possible color at each step is known to be P-complete [13].

Paper I deals with parallelizing greedy heuristics using the shared-address-space programming model in a coarse-grain setting. Similar previous efforts focused on using the message-passing programming model on distributed-memory parallel computers. These efforts yielded no speedup [2, 17]. The algorithms presented in Paper I overcome this shortcoming.

In Paper I we present a new parallel algorithm that colors a graph $G = (V, E)$ using at most $\Delta + 1$ colors and has an expected parallel runtime $O(|E|/p)$ on a CREW PRAM for any number of processors p such that $p \leq |V|/\sqrt{2|E|}$.

The algorithm consists of three phases. The first two phases are parallel while the last one is sequential. In the first phase, the vertex set is simply equally partitioned among the p available processors. Each processor colors its block of vertices paying attention to already colored (local and non-local) vertices. In this scenario, a pair of adjacent vertices residing on different processors may be colored concurrently, and possibly get the same color, thereby leading to an inconsistency. In the second phase, inconsis-

tencies are detected in parallel. In the final sequential phase, the detected inconsistencies are rectified.

In the same paper we present a second algorithm that extends this idea to reduce the number of colors used. The reduction in the number of colors is obtained by dividing the first phase of the algorithm just described into two coloring steps. In the improved algorithm, the coloring in the first step is used to determine an ordering for a coloring in the second step.

Both algorithms are well suited for the shared address space programming model and are implemented using OpenMP. In agreement with the theoretical analyses, timing results obtained from experiments conducted on the Origin 2000 using a modest number of processors and graphs from finite element methods and eigenvalue computation show that the algorithms yield speedup.

4 Graph coloring and efficient computation of sparse derivative matrices

Computing a derivative matrix, i.e. a Jacobian or a Hessian, is a phase required in many numerical optimization algorithms. In large scale problems, this phase often constitutes an expensive part of the entire computation. Hence, efficient methods that exploit matrix structure such as sparsity and symmetry are often required. Finite difference and automatic differentiation (AD) are two widely used methods in the efficient determination of the elements of derivative matrices. In using these methods, the main goal is to minimize the number of function evaluations or AD passes required. The pursuit of this goal gives rise to several *matrix partitioning problems.*

Since the early 70's a considerable amount of research work has been done in the field of sparse derivative matrix computation [5, 6, 7, 8, 10, 14, 15, 20, 22, 25]. Early studies in this field, such as [10, 22, 25], used matrix oriented approaches. Later, following the pioneering work of Coleman and Moré [6], *graph theoretic* approaches proved to be advantageous in understanding, analyzing and solving the matrix problems. In particular, graph coloring—in its several variations—was found to be an effective model. However, existing algorithms and software for such coloring problems are developed for uniprocessor (sequential) computers. Currently, there is a need for parallel coloring algorithms that can be used to aid the solution of large-scale PDE-constrained optimization problems on parallel computers. Paper II is an expository work for, and Paper III is a first step towards, addressing this need in a flexible manner.

Matrix	Partition	Scheme	Entries	Prob.	Formulation	
Nonsym.	1d	direct	all	P1	D2 coloring	(II)
Sym.	1d	direct	all	P2	$D\frac{3}{2}$ coloring	[7]
Nonsym.	2d	direct	all	P3	$D\frac{3}{2}$ bicoloring	[8]
Sym.	1d	subst.	all	P4	acyclic coloring	[5]
Nonsym.	2d	subst.	all	P5	acyclic bicoloring	[8]
Nonsym.	1d	direct	some	P6	restricted D2 coloring	(II)
Sym.	1d	direct	some	P7	restricted $D\frac{3}{2}$ coloring	(II)
Nonsym.	2d	direct	some	P8	restricted $D\frac{3}{2}$ bicoloring	(II)
Sym.	1d	subst.	some	P9	not treated	
Nonsym.	2d	subst.	some	P10	not treated	

Table 1: Overview of partition/coloring problems considered in Paper II.

4.1 Unifying Framework – Paper II

In the literature, one finds several types of coloring problems characterizing different types of matrix partitioning problems. Understanding the similarities and differences among these problems greatly simplifies the development of algorithms—sequential or parallel—for solving them. Paper II aims at contributing to this end.

The nature of a partitioning problem in the context of efficient computation of sparse derivative matrices depends on the type of the underlying matrix and the scenario under which the matrix computation is carried out. Specifically, the particular problem depends on,

1. whether the matrix to be computed is *symmetric* or *nonsymmetric*,

2. whether a *one-dimensional* partition (involving only columns or rows) or a *two-dimensional* partition (involving both columns and rows) is used,

3. whether the evaluation scheme employed is *direct* (solves a diagonal system) or *substitution* based (solves a triangular system), and

4. whether *all* of the nonzero entries of the matrix or only *some* of them need to be determined.

The factors outlined above are mutually orthogonal to each other—potentially resulting in 2^4 different cases. Within our framework, ten of these correspond to practically meaningful partitioning problems whereas the remaining six do not. In Paper II, we use a unified graph theoretic framework to study eight of these problems; the remaining two are not considered due to their sophistication. The upper part of Table 1 gives an

overview of the problems considered in the paper and the respective coloring formulations used. The lower part shows the cases not treated in the paper.

Paper II is a review as well as research paper that integrates known and new formulations. In the paper we give the necessary background to motivate and formally introduce the matrix partitioning problems, and then develop their equivalent graph coloring formulations. Problems P1–P5 have been studied previously whereas P6–P8 are investigated for the first time. The motivation for introducing the new problems and the corresponding restricted coloring formulations is the fact that the computation of a specified subset of the nonzero entries of a matrix—for instance, for preconditioning purposes—can be carried out even more efficiently than the computation of all of the nonzero entries.

In formulating problems P1–P8 as coloring problems, we use a bipartite graph to represent a nonsymmetric matrix and an adjacency graph to represent a symmetric matrix. In the case of nonsymmetric matrices, bipartite graph based formulations are shown to be attractive in terms of flexibility and storage space requirement in comparison with known formulations based on *intersection* graphs.

The major contributions made in Paper II can be summarized as follows.

- We propose distance-2 coloring as an alternative, more flexible, formulation for problem P1 as compared to the distance-1 coloring formulation of Coleman and Moré [6] for the same problem.

- We expose the interrelationship among the coloring formulations of problems P1 through P5 and identify distance-2 coloring as as a *generic* model.

- For problems P3 and P5, based on the previously known relationship to graph bicoloring [8], we observe a connection to finding a vertex cover in a graph.

- Using the insight gained from the unified graph theoretic treatment, we develop several simple, sequential, heuristic algorithms for the coloring formulations of problems P1 through P5.

- We formulate the partitioning problems arising in the computation of a subset of the entries of a matrix (i.e. problems P6–P8) as restricted coloring problems.

4.2 Parallel Algorithms – Paper III

In Paper III, we develop several shared address space parallel algorithms for the coloring formulations of problems P1 and P2. The algorithms are straightforward extensions of the algorithm for the standard (distance-1) graph coloring problem presented in Paper I. However, in the cases where the graph to be colored is relatively dense, *randomization* is used as a supplementary tool to improve scalability. Our PRAM-analyses show that the algorithms give almost linear speedup for sparse graphs that are large relative to the number of processors. The algorithms are implemented using OpenMP and results from experiments conducted on the Origin 2000 using various large graphs show that the algorithms indeed yield reasonable speedup for a modest number of processors.

5 A new parallel computation model

5.1 Graph coloring on the CGM model – Paper IV

The aim of Paper IV is to adapt the shared address space algorithm for the standard graph coloring problem given in Paper I to the CGM model. The CGM model used is basically the same as the CGM model described in Section 2.3.3 except for the fact that the quality of an algorithm is measured not using communication rounds but rather using an explicit analysis of the overall computation and communication cost involved. Particularly, a CGM algorithm is considered efficient if the parallel runtime of the algorithm, taking both communication and computation costs into account, yields a speedup (relative to the complexity of the best known sequential algorithm) that is a linear function in the number of processors used.

In Paper IV we present such an efficient CGM algorithm that colors a connected graph $G = (V, E)$ using at most $\Delta + 1$ colors, where Δ is the maximum degree in G. The algorithm is given in two variants: *randomized* and *deterministic*. It is shown that on a p-processor CGM model, where $|E|/p > p^2$, the proposed algorithms have a parallel running time $O(|E|/p)$ and an overall (computation and communication) cost $O(|E|)$. These bounds correspond to the *average* and *worst* case for the randomized and deterministic versions, respectively.

In comparison with the original algorithm in Paper I, the CGM coloring algorithm in Paper IV is more complicated. Some of the factors that contribute to the complexity are distributed-memory related issues such as data distribution, communication overhead, and load balancing. In terms of the underlying algorithmic idea, a key difference between the algorithms

of Paper I and IV is that the latter uses *recursion*.

5.2 The PRO model – Paper V

The notion of 'efficiency' of a CGM algorithm as used in Paper IV motivated us to refine the definition of the model itself. One of the goals was to keep the number of parameters in the refined model as low as possible and at the same time make the algorithm design objective more precise. Another goal was to find a model that can serve as a design scheme for the algorithm developer, i.e., a model that identifies the features of a parallel algorithm that contribute to its practical scalability on different architectures. The PRO model is partly a result of this effort.

The novel idea behind the PRO model is the focus on relative resource optimality and the use of granularity as a quality measure. In this model, the design and analysis of a parallel algorithm is done relative to the complexity of a specific underlying sequential algorithm. The parallel algorithm is required to be time and space optimal with respect to the complexity of the reference sequential algorithm. As a consequence of its optimality, a PRO-algorithm yields linear speedup. The quality of a PRO-algorithm is measured by an attribute of the model called granularity function. This function, which is expressed in terms of the input size, shows the number of processors that can be employed without sacrificing optimality.

In Paper V, the PRO model is defined formally and it is systematically compared with a selection of other models, including BSP and CGM. The paper also presents PRO-algorithms for matrix multiplication and one-to-all broadcast to illustrate how the model is used in algorithm design and analysis.

6 Conclusion

The first algorithm introduced in Paper I, and later applied in Paper III, can be seen as a parallelization technique of general interest. The technique consists of (i) breaking up a given problem into p, not necessarily independent, subproblems of almost equal sizes, (ii) solving the p subproblems concurrently using p processors, and (iii) resolving any inconsistencies that may result due to the interdependence among the p subproblems sequentially. The technique assumes that resolving an inconsistency can be done locally. The success of the technique depends on the overall time spent on detecting and resolving inconsistencies. Papers I and III demonstrate the effectiveness of the technique in developing shared-address-space parallel al-

gorithms for various coloring problems. In these papers, we also showed different ways, including randomization, by which the technique can be enhanced in the context of coloring problems. In Paper IV we extended the basic idea behind the technique to develop an efficient CGM-algorithm for standard graph coloring. This algorithm suggests a way for extending the parallelization technique mentioned earlier. Specifically, if the inconsistencies that could result from (ii) can be determined a priori, then (iii) can be replaced by a recursive application of the technique.

In Paper II we revisited the role of graph coloring in modeling matrix partitioning problems that arise in numerical estimation of sparse Jacobian and Hessian matrices. We used a unified graph theoretic framework for dealing with the matrix problems in a flexible manner. As a result, we developed several simple and effective sequential heuristic algorithms. In Paper III, some of these algorithms were parallelized using the shared address space programming model. The work in Papers II and III can serve as basis for further design and implementation of parallel algorithms for the various coloring problems.

In Paper V we proposed PRO as a candidate model for the design and analysis of efficient, scalable and portable parallel algorithms. Verifying its utility needs further exploration, both theoretical and experimental.

For more specific directions for further work on each of the topics addressed in this thesis, the reader is referred to the concluding remarks given at the end of each paper.

References

[1] S. G. Akl. *Parallel Computation*. Prentice Hall, New Jersey, USA, 1997.

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

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

[4] E. Caceres, F. Dehne, A. Ferreira, P. Locchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *The 24th International*

Colloquium on Automata Languages and Programming, volume 1256 of *LNCS*, pages 390–400. Springer Verlag, 1997.

[5] T. F. Coleman and J. Cai. The cyclic coloring problem and estimation of sparse Hessian matrices. *SIAM J. Alg. Disc. Meth.*, 7(2):221–235, April 1986.

[6] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, 20(1):187–209, February 1983.

[7] T. F. Coleman and J. J. Moré. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Program.*, 28:243–270, 1984.

[8] T. F. Coleman and A. Verma. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.*, 19(4):1210–1233, July 1998.

[9] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture*. Morgan Kaufmann Publishers, San Francisco, California, 1999.

[10] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.

[11] F. Dehne. Coarse grained parallel algorithms. *Algorithmica Special Issue on "Coarse grained parallel algorithms"*, 24(3/4):173–176, 1999.

[12] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.

[13] R. Greenlaw, H.J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, New York, USA, 1995.

[14] S. Hossain and T. Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, 10:33–48, 1998.

[15] S. Hossain and T. Steihaug. Reducing the number of AD passes for computing a sparse Jacobian matrix. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors, *Authomatic Differentiation of Algorithms: From Simulation to Optimization*, pages 263 – 270. Springer, 2002.

[16] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[17] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, May 1993.

[18] H. F. Jordan and G. Alaghband. *Fundamentals of Parallel Processing.* Prentice Hall, New Jersey, USA, 2003.

[19] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., California, 1994.

[20] S. T. McCormick. Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem. *Math. Program.*, 26:153–171, 1983.

[21] MPI. Message-passing interface standard. *http://www.mpi-forum.org/.*

[22] G. N. Newsam and J. D. Ramsdell. Estimation of sparse Jacobian matrices. *SIAM J. Alg. Disc. Meth.*, 4:404–418, 1983.

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

[24] Parallab. High Performance Computing Laboratory. *http://www.parallab.uib.no/.*

[25] M. J. D. Powell and PH. L. Toint. On the estimation of sparse Hessian matrices. *SIAM J. Numer. Anal.*, 16(6):1060–1074, December 1979.

[26] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

Scalable parallel graph coloring algorithms

Assefaw Hadish Gebremedhin* Fredrik Manne

Abstract

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

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

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

1 Introduction

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

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

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

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

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

availability of shared memory computers where the entire memory can be accessed by any processor makes this argument less significant now.

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

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

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

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

2 Background

In this section we give a brief overview of previous work done on the development of fast coloring heuristics, both sequential and parallel. We begin

by introducing some graph notation used in this paper.

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

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

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

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

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

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

Scheme 1

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

Figure 1: A parallel coloring heuristic

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

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

3 Block Partition Based Coloring Heuristics

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

3.1 The First Algorithm

The strategy we employ consists of three phases. In the first phase, the input vertex set V of the graph $G = (V, E)$ is partitioned into p blocks as $\{V_1, V_2, \ldots, V_p\}$ such that $|V_i| = n/p$, $1 \leq i \leq p$. The vertices in each block are then colored in parallel using p processors. The parallel coloring

comprises of n/p parallel steps with synchronization barriers at the end of each step. When coloring a vertex, *all* its previously colored neighbors, both the local ones and those found on other blocks, are taken into account. In doing so, two processors may simultaneously attempt to color vertices that are adjacent to each other. If these vertices are given the same color, the resulting coloring becomes invalid and hence we call the coloring obtained a *pseudo coloring*. In the second phase, each processor p_i checks whether vertices in V_i are assigned valid colors by comparing the color of a vertex against all its neighbors that were colored at the same parallel step in the first phase. This checking step is also done in parallel. If a conflict is discovered, one of the end points of the edge in conflict is stored in a table. Finally, in the third phase, the vertices stored in this table are colored sequentially. Algorithm 1 provides the details of this strategy and is given in Figure 2.

Algorithm 1

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

end

Figure 2: Block partition based coloring

3.1.1 Analysis

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

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

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

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

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

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

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

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

□

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

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

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

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

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

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

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

□

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

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

3.2 The Second Algorithm

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

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

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

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

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

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

Algorithm 2

$ImprovedBlockPartitionBasedColoring(G, p)$
begin
1. As Phase 1 of Algorithm 1
   {At this point we have the pseudo independent
   sets ColorClass(1) ... ColorClass(ColNum) }
2. for $k = ColNum$ down to 1 do
      Partition $ColorClass(k)$ into p equal blocks $V'_1 \dots V'_p$
      for $i = 1$ to p do in parallel
         for each $v_j \in V'_i$ do
            assign the smallest legal color to vertex v_j
         end-for
      end-for
   end-for
3. As Phase 2 of Algorithm 1
4. As Phase 3 of Algorithm 1

end

Figure 3: Modified block partition based coloring

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

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

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

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

due to the vertices of color class w (denoted by K') is bounded as follows.

$$E[K'] \leq \sum_{x\in V'} \frac{p(p-1)}{n(n'-1)} deg(x) \quad (6)$$

$$= \frac{p}{n}(p-1)\frac{\sum_{x\in V'} deg(x)}{n'-1} \quad (7)$$

$$\leq \frac{2p^2\Delta}{n} \quad (8)$$

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

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

□

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

4 Implementation Issues

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

4.1 Synchronization Overhead

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

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

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

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

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

4.2 OpenMP

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

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

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

5 Experimental Results

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

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

Table 1: Test Graphs.

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

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

5.1 Test Graphs

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

5.2 Discussion

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

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

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

Table 2: Experimental results for Algorithm 1.

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

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

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

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

Table 3: Experimental results for Algorithm 2.

values for the two largest graphs mrng3 and dense2.

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

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

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

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

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

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

6 Conclusion

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

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

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

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

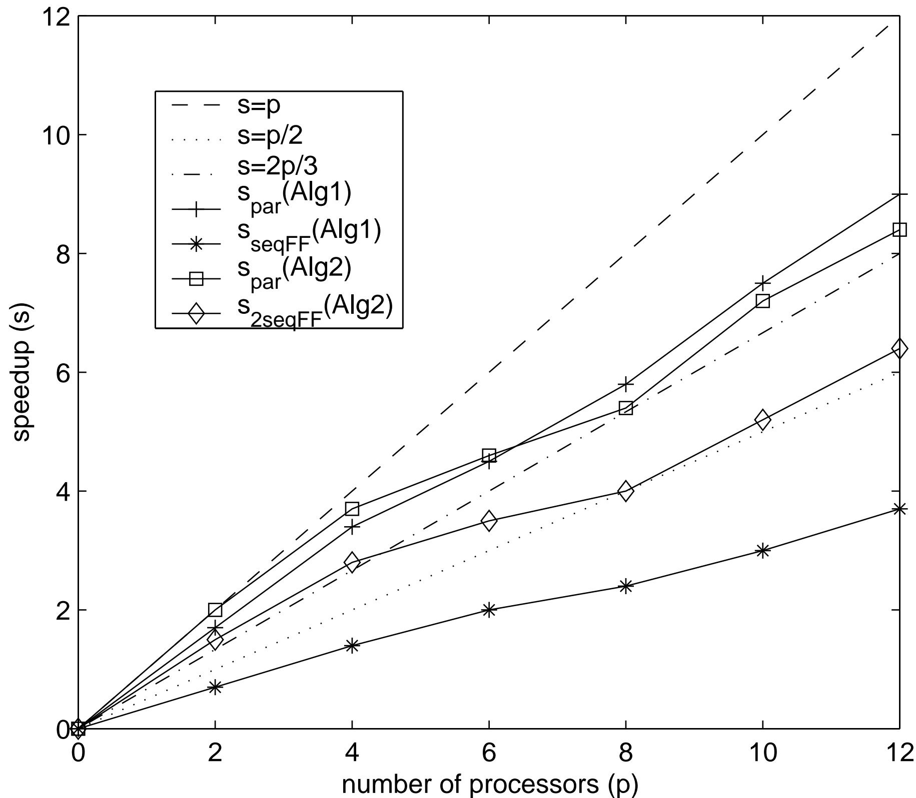

Figure 5: Comparison of speedup curves for dense2.

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

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

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

We believe that the general idea in these coloring heuristics of allowing

inconsistency for the sake of concurrency can be applied to develop parallel algorithms for other graph problems and we are currently investigating this in problems related to sparse matrix computations.

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

References

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

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

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

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

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

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

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

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

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

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

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

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

[13] G. Karypis. Private Communication.

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

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

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

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

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

Graph Coloring in Optimization Revisited

Assefaw Hadish Gebremedhin [*] Fredrik Manne Alex Pothen[†]

Abstract

We revisit the role of graph coloring in modeling a variety of matrix partitioning problems that arise in numerical determination of large sparse Jacobian and Hessian matrices. The problems considered in this paper correspond to the various scenarios under which a matrix computation, or estimation, may be carried out, i.e., the particular problem depends on whether the matrix to be computed is symmetric or nonsymmetric, whether a one-dimensional or a two-dimensional partition is to be used, whether a direct or a substitution based evaluation scheme is to be employed, and whether all nonzero entries of the matrix or only a subset need to be computed. The resulting complex partitioning problems are studied within a unified graph theoretic framework where each problem is formulated as a variant of a coloring problem. Our study integrates existing coloring formulations with new ones. As far as we know, the estimation of a subset of the nonzero entries of a matrix is investigated for the first time. The insight gained from the unified graph theoretic treatment is used to develop and analyze several new heuristic algorithms.

Key words: Sparsity, symmetry, Jacobians, Hessians, finite differences, automatic differentiation, matrix partitioning problems, graph coloring problems, NP-completeness, approximation algorithms

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

[†]Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162 USA. pothen@cs.odu.edu and ICASE, NASA Langley Research Center, Hampton, VA 23681-2199 USA. pothen@icase.edu

1 Introduction

Algorithms for solving nonlinear systems of equations and numerical optimization problems that rely on derivative information require the repeated estimation of Jacobian and Hessian matrices. Since this usually constitutes an expensive part of the entire computation, efficient methods for estimating these matrices via finite difference (FD) or automatic differentiation (AD) techniques are needed. It should be noted that FD techniques find an approximation whereas AD techniques enable exact (within the limits of machine precision) computation. For brevity, we use the term 'estimation' in referring to both cases. In applying AD and FD techniques, if the sparsity structure of the desired matrix is known a priori, or can be determined easily, the nonzero entries can be estimated efficiently. The objective in such an efficient estimation is to minimize the number of function evaluations or AD passes required.

This objective calls for a variety of *matrix partitioning problems*. The particular problem depends on whether the required matrix is symmetric or nonsymmetric, whether a one-dimensional partition (involving only columns or rows) or a two-dimensional partition (involving both columns and rows) is used, whether the entries are evaluated using a direct method or via substitution, and finally, whether all the nonzero entries of the matrix or only a subset of them need to be determined.

Several studies have demonstrated the usefulness of *graph coloring* in modeling the matrix partitioning problems of our concern [4, 7, 8, 9, 17, 18, 23]. However, these studies have been rather disintegrated: a typical study in this field focuses on one type of matrix, a specific numerical method, and a particular evaluation scheme. This has at least two consequences. First, the inherent similarity among the various partitioning problems gets obscured. Second, it makes the identification of a generic formulation difficult thereby hindering the development of algorithms and software in a flexible manner.

The main purpose of this paper is to study the various matrix partitioning problems within a *unified graph theoretic framework*. We consider eight different partitioning problems. For each matrix partitioning problem, we develop an equivalent *graph coloring formulation*. To our knowledge, the problems in partial matrix estimation, where a specified *subset* of the nonzero entries is to be determined, are studied for the first time. In full matrix estimation, the case in which all the nonzero entries are to be determined, we use known coloring formulations for all problems except for the estimation of a nonsymmetric matrix using a one-dimensional partition via a direct method. For this problem, we propose *distance-2 graph coloring* as

Matrix	1D Partition	2D Partition	Method
Jacobian	distance-2 coloring	distance-$\frac{3}{2}$ bicoloring	Direct
Hessian	distance-$\frac{3}{2}$ coloring	NA	Direct
Jacobian	NA	acyclic bicoloring	Substitution
Hessian	acyclic coloring	NA	Substitution

Table 1: Graph coloring formulations for estimating *all* nonzero entries of derivative matrices. The Jacobian and the Hessian are represented by their bipartite and adjacency graphs, respectively. NA stands for not applicable.

an alternative formulation to the known distance-1 coloring formulation [7]. The motivation for the distinction between full and partial matrix estimation is the fact that the computation in the latter case can be carried out even more efficiently.

Table 1 summarizes the coloring formulations used in this paper. The formulations for one-dimensional estimation of the Hessian are due to Coleman and Moré [8] and Coleman and Cai [4] and those for two-dimensional estimation of the Jacobian are due to Coleman and Verma [9].

In our work, for all the matrix partitioning problems, we rely on a *bipartite* graph representation for a nonsymmetric matrix and an *adjacency* graph representation for a symmetric matrix. We show that these representations are robust and flexible—they are decoupled from the eventual technique to be employed and the matrix entries to be computed. This is in contrast with the *column intersection* graph representation [7] of a nonsymmetric matrix which targets at determining *all* the nonzero entries of the matrix using a one-dimensional column partition. Moreover, the space required for storing the column intersection graph of matrix A is proportional to the number of non-zeros in $A^T A$, whereas the space required for storing the bipartite graph of A is proportional to the number of non-zeros in A.

Using our graph representations, we identify the distance-2 graph coloring problem as a unifying generic model for the various one-dimensional matrix partitioning problems, i.e., the coloring problem in a particular case is some relaxed variant of the distance-2 coloring problem. In the case of two-dimensional partitioning problems, we build upon the known relationship to graph *bicoloring* and observe a connection to finding a *vertex cover* in a graph.

All of the problems listed in Table 1 are known to be NP-hard [4, 8, 9, 21]. We use the insight gained from the interrelationship among these problems to develop several simple heuristic algorithms for finding sub-optimal solutions.

The rest of this paper is organized as follows. Section 2 is a detailed introduction to the partitioning problems that arise in full matrix estimation. In Section 3 we rigorously develop the equivalent graph problem formulations and discuss their interrelationship. Section 4 deals with the graph theoretic formulation of partitioning problems in partial matrix estimation. In Section 5 we compare formulations based on bipartite graph with formulations based on column intersection graph. In Section 6 we present and analyze various greedy heuristic algorithms. We conclude the paper in Section 7 with some remarks and point out avenues for further work.

2 Background

2.1 Finite Differences and Partitioning Problems

Given a continuously differentiable function $F : R^n \to R^m$, the **Jacobian** of F at the point x is the $m \times n$ matrix whose (i, j) entry $J(x)_{ij} = F'(x)_{ij} = \frac{\partial f_i}{\partial x_j}(x)$, where $f_1(x), f_2(x), \ldots, f_m(x)$ are the components of $F(x)$. Let A denote the Jacobian matrix $F'(x)$. An estimate for the jth column of A, denoted henceforth by a_j, can be obtained from the *forward difference* approximation,

$$Ae_j = a_j = \frac{\partial}{\partial x_j} F(x) \approx \frac{1}{h}[F(x + he_j) - F(x)], \quad 1 \le j \le n, \tag{1}$$

where e_j is the jth unit vector and h is a positive step length. Other finite difference approximations of higher order, such as *central differences*, could also be used to estimate A. In any case, if $F(x)$ is already evaluated, an approximation to a_j is obtained with one additional function evaluation. Thus, if each column of A is computed independently, n additional function evaluations will be required. However, by exploiting the sparsity structure of A, the required number of function evaluations can be reduced significantly. The sparsity structure of A is often easily available and the goal here is to exploit this to estimate the nonzero entries of A using as few function evaluations as possible under the assumption that evaluating $F(x)$ is more efficient than evaluating the components $f_i(x)$, $1 \le i \le m$, separately.

Given a twice continuously differentiable function $f : R^n \to R$, the **Hessian** of f at the point x is the $n \times n$ symmetric matrix whose (i, j) entry $H(x)_{ij} = \nabla^2 f(x)_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$. When ∇f is available, $\nabla^2 f$ can be approximated by applying Formula (1) to the function $F = \nabla f$. Again, we assume that evaluating the gradient $\nabla f(x)$ as a single entity is more desirable than evaluating the components $\partial_1 f(x), \ldots, \partial_n f(x)$ separately.

Let d_i be the binary vector obtained by adding some unit vectors e_j, $j \in \{1, 2, \ldots, n\}$, together. The problem of estimating a sparse derivative matrix, Jacobian or Hessian, using FD can then be stated as follows. Given the sparsity structure of a matrix A find vectors $d_1, d_2, \ldots, d_p$ such that the products $Ad_1, Ad_2, \ldots, Ad_p$ enable the determination of all the nonzero entries of A.

Specifying the vectors $Ad_1, Ad_2, \ldots, Ad_p$ gives rise to a system of linear equations where the unknowns are the nonzero elements of A. If the choice of the vectors d_i is such that the resulting system of equations can be ordered to a diagonal form, then we say that A is *directly* determined by the vectors d_i. If, on the other hand, the vectors d_i are chosen such that the system of equations can be ordered to a triangular form, then the unknowns can be determined via *substitution*.

In both a direct and a substitution based determination, minimizing the number of function evaluations corresponds to minimizing the number of vectors p. There is a trade-off in the choice of methods. A direct method is more restrictive and hence requires more function evaluations compared to a substitution method. On the other hand, a substitution method is subject to numerical instability, whereas a direct method is not. Moreover, in terms of parallel computation, direct methods offer straightforward parallelization since the estimates can be read off directly from each row of a matrix-vector product, whereas substitution methods have less parallelism since there are more dependencies among the computations required to obtain the matrix entries. In the next two paragraphs (Sections 2.1.1 and 2.1.2), we consider minimizing p in a direct and substitution based evaluation, respectively.

2.1.1 Direct estimation

In a direct determination of a matrix A, note that for each nonzero element a_{ij}, there is a vector d in the set $\{d_1, d_2, \ldots, d_p\}$ such that $a_{ij} = (Ad)_i$, where $(Ad)_i$ is the ith component of the vector Ad. Thus, each nonzero matrix element a_{ij} can be read off from some component of the vector Ad.

The problems that arise in the direct, efficient estimation of sparse Jacobian and Hessian matrices can thus be stated as follows.

Problem 2.1 *Given the sparsity structure of a general matrix $A \in R^{m \times n}$, find the fewest vectors $d_1, d_2, \ldots, d_p$ such that $Ad_1, Ad_2, \ldots, Ad_p$ determine A directly.*

Problem 2.2 *Given the sparsity structure of a symmetric matrix $A \in R^{n \times n}$, find the fewest vectors $d_1, d_2, \ldots, d_p$ such that $Ad_1, Ad_2, \ldots, Ad_p$ determine A directly.*

Curtis et al. [12] were the first to observe that, while using a direct method, a group of columns can be determined by one evaluation of Ad if no two columns in this group have a nonzero in the same row position. Such columns are *structurally orthogonal*, since their pairwise inner products are zero. Powell and Toint [26] later showed that, in the case of Hessian estimation, the number of function evaluations can be reduced further by considering symmetry. The following two notions define the underlying *partitions* used in these methods [8].

Definition 2.3 A partition of the columns of a matrix A is said to be *consistent* with a direct determination of A if whenever a_{ij} is a nonzero element of A then the group containing a_j has no other column with a nonzero in row i.

Definition 2.4 A partition of the columns of a symmetric matrix A is *symmetrically consistent* with a direct determination of A if whenever a_{ij} is a nonzero element of A then either the group containing a_j has no other column with a nonzero in row i, or the group containing a_i has no other column with a nonzero in row j.

Let $\{C_1, C_2, \ldots, C_p\}$ be a consistent partition. With each group C_k, associate a binary vector d_k having components $\delta_j = 1$ if a_j belongs to C_k, and $\delta_j = 0$ otherwise. Then,

$$Ad_k = \sum_{a_j \in C_k} a_j.$$

If $a_{ij} \neq 0$ and column $a_j \in C_k$, then $a_{ij} = (Ad_k)_i$. Thus, all the nonzero entries of A can be determined with p evaluations of Ad_k.

When A is symmetric, a symmetrically consistent partition is sufficient to determine A directly: if a_j is the only column in its group with a nonzero in row i then a_{ij} can be determined as discussed above; alternatively, if a_i is the only column in its group with a nonzero in row j then a_{ji} can be determined.

If a consistent partition (rather than a symmetrically consistent one) is used to compute a symmetric matrix A, the estimate for a_{ij} may actually be different from that of a_{ji} due to truncation error. Thus, using a symmetrically consistent partition to compute half of the nonzero elements of a matrix and determining the other half by symmetry is preferable both in terms of reducing computational work and ensuring that the computed matrix is indeed symmetric.

Using Definitions 2.3 and 2.4, Problems 2.1 and 2.2 can be restated as follows. In the remainder of this paper, we shall use the acronym MPP to refer to a matrix partitioning problem.

Problem 2.5 (*MPP1*) *Given the sparsity structure of a matrix $A \in R^{m \times n}$, find a consistent partition of the columns of A with the fewest number of groups.*

Problem 2.6 (*MPP2*) *Given the sparsity structure of a symmetric matrix $A \in R^{n \times n}$, find a symmetrically consistent partition of the columns of A with the fewest number of groups.*

2.1.2 Estimation via substitution

In estimating a matrix A via a substitution method, the vectors $d_1, \ldots, d_p$ are chosen such that the system of equations defined by the products $Ad_1, \ldots, Ad_p$ allows the determination of the unknowns via a substitution process. A partition suitable for a substitution method needs to fulfill more relaxed requirements compared to a direct method and hence results in smaller number of groups. In the FD context, this fact has been especially used when estimating a symmetric matrix since substitution can be effectively combined with the exploitation of symmetry [4]. For a nonsymmetric matrix, the advantage offered by a substitution method over a direct method is not so pronounced. An example of a substitution method for a nonsymmetric matrix can be found in [18].

In this paper, we concentrate on using a substitution method for a symmetric matrix. To illustrate the fact that a partition used in a substitution method requires fewer groups than that used in a direct method, consider the 4×4 symmetric matrix H shown in Figure 1.

$$\begin{bmatrix} x & x & & \\ x & x & x & \\ & x & x & x \\ & & x & x \end{bmatrix}$$

Figure 1: The structure of a 4×4 symmetric matrix

For matrix H, any partition consistent with a direct determination requires at least three groups. One such partition is $\{(h_1, h_4), (h_2), (h_3)\}$, where h_j is the jth column of H. However, if we do not insist on determining the elements directly, two groups would suffice. For example $\{(h_1, h_3), (h_2, h_4)\}$. The two matrix-vector products corresponding to the

two groups yield a system of eight equations involving the nonzero entries of H. Note that, due to symmetry, nonzero element h_{ij} can be identified with h_{ji}, and hence, there are effectively seven unknowns in the system. This system can be ordered to a triangular form and be solved via substitution.

In general, a partition of the columns of a symmetric matrix induces a substitution method if there is an ordering of the matrix unknowns such that all unknowns can be solved for, in that order, using symmetry and previously solved elements.

We now formally define such a partition and then state the corresponding partitioning problem.

Definition 2.7 A partition of the columns of a symmetric matrix A is said to be *substitutable* if there exists an ordering on the elements of A such that for every nonzero a_{ij}, either a_j is in a group where all the nonzeros in row i, from other columns in the same group, are ordered before a_{ij} or a_i is in a group where all the nonzeros in row j, from other columns in the same group, are ordered before a_{ij}.

Problem 2.8 (*MPP3*) *Given the sparsity structure of a symmetric matrix $A \in R^{n \times n}$, find a substitutable partition of the columns of A with the fewest number of groups.*

2.2 Automatic Differentiation and Partitioning Problems

Automatic differentiation (AD) is a chain rule based technique for evaluating the derivatives of functions defined by computer programs. The two basic modes of operation of AD, known as *forward* and *reverse* mode, correspond to a bottom-up and a top-down strategy of accumulating partial derivatives of elementary functions that define the computational scheme of the function to be differentiated. A treatment of the technical details of AD is beyond the scope of this paper, but the interested reader is referred to, for instance, the books [14] and [10].

What is of interest for us is that, as in the FD setting, the efficient computation of matrices using AD gives rise to partitioning problems in which structural orthogonality continues to be the partition-criterion. In particular, one can use the forward mode to compute a group of columns of a matrix A from the product Ad_1, where the vector d_1 has nonzeros in positions corresponding to columns of A that are structurally orthogonal. Furthermore, in the reverse mode, a group of structurally orthogonal rows of A can be computed from the vector-matrix product ${d_2}^T A$, where d_2 is

an appropriately defined vector. This means that one can potentially take advantage of the sparsity available in columns and in rows.

One way of exploiting sparsity in columns and rows is to *separately* partition the columns and rows of the matrix and use the partition which gives the minimum number of groups. For a symmetric matrix, a row partition is equivalent to a column partition, but for a nonsymmetric matrix, the two partitions may differ considerably. For example, consider an $n \times n$ matrix where all the entries on the diagonal and the first row are nonzero, and the rest of the matrix entries are all zero. For such a matrix structure, a column partition requires n groups whereas a row partition requires just two groups.

However, an approach based on a separate row and column partition is not always satisfactory. For example, consider an $n \times n$ matrix where all of the elements in the first row, first column, and the diagonal are nonzero and the rest of the entries are all zero. For such a structure, a row partition requires n groups and so does a column partition. However, using a *combined* row and column partition, three groups are enough to determine all the nonzero entries of the matrix. First, separately evaluate the entries in the first column and the first row (two groups). Then, since the remaining $(n-1) \times (n-1)$ matrix is diagonal, determine all entries by one evaluation. Thus, three groups (two column and one row) suffice to determine all the nonzero entries.

In Sections 2.2.1 and 2.2.2 we consider such a computation of a nonsymmetric matrix using the combined modes of AD via direct and substitution methods. We call a partition that involves both rows and columns a *two-dimensional* partition as opposed to a *one-dimensional* partition in which either only rows or only columns are involved. Note that a two-dimensional partition does not make sense for computing a symmetric matrix. In particular, a symmetry-exploiting one-dimensional partition is sufficient.

2.2.1 Direct computation

Consider the vectors $d_1, d_2, \ldots, d_p$ in Problem 2.1 as the p columns of the $n \times p$ matrix D. An alternative way of posing the problem would then be: given the structure of a matrix $A \in R^{m \times n}$, find a matrix $D \in R^{n \times p}$ with the least value of p such that the product AD determines A directly. By the same token, the problem that arises in the two-dimensional efficient direct computation of a Jacobian can be posed as follows.

Problem 2.9 *Given the sparsity structure of the matrix $A \in R^{m \times n}$, find matrices $D_1 \in R^{n \times p_1}$ and $D_2 \in R^{m \times p_2}$ such that AD_1 and $D_2^T A$ together determine A directly and the value $p = p_1 + p_2$ is minimized.*

Hossain and Steihaug [17] studied Problem 2.9 and reformulated it as a partitioning problem by using the notion of *consistent row-column partition* in which the *entire* set of rows and columns is partitioned into two respective set of groups. Coleman and Verma [9] also studied the same problem and identified a similar two-dimensional partition problem. Their notion of partition differs from that of Hossain and Steihaug in that it partitions only a *subset* of the columns and the rows of the matrix that suffice for the direct determination of the entries. The following concepts were used to formalize the requirements.

Definition 2.10 A *bipartition* of a matrix A is a pair (Π_C, Π_R) where Π_C is a column partition of a subset of the columns of A and Π_R is a row partition of a subset of the rows of A.

Definition 2.11 A bipartition (Π_C, Π_R) of a matrix A is *consistent with a direct determination* if for every nonzero entry a_{ij} of A, either column j is in a group of Π_C which has no other column having a nonzero in row i, or row i is in a group of Π_R which has no other row having a nonzero in column j.

The number of column and row groups in a consistent bipartition corresponds to the number of forward and reverse AD passes, respectively, required to compute the nonzero entries directly. To see this, observe that a nonzero a_{ij} can be determined either from a column group where column j is the only column with a nonzero in row i, or from a row group where row i is the only row with a nonzero in column j. Hence, assuming that the computational costs involved in the forward and reverse modes of AD are of the same order, in an efficient method, the value $|\Pi_C| + |\Pi_R|$ is required to be as small as possible[1].

Thus Problem 2.9 can be reformulated as follows.

Problem 2.12 (*MPP4*) *Given the sparsity structure of a matrix* $A \in R^{m \times n}$*, find a bipartition* (Π_C, Π_R) *of* A *consistent with a direct determination such that* $|\Pi_C| + |\Pi_R|$ *is minimized.*

2.2.2 Computation via Substitution

In using a substitution method in the AD context, the requirement on the bipartition can be relaxed so as to obtain fewer number of groups compared

[1] In this paper, we are concerned only with computational cost; however, in general, the forward mode requires less memory space than the reverse mode.

Matrix	1D Partition	2D Partition	Method
Jacobian	MPP1	MPP4	Direct
Hessian	MPP2	NA	Direct
Jacobian	NA	MPP5	Substitution
Hessian	MPP3	NA	Substitution

Table 2: Partitioning problems in estimating/computing *all* nonzero entries of derivative matrices using FD and AD. The entry NA denotes that the case is not applicable.

with that in a direct method. We state the following definition used by Coleman and Verma [9] to subsequently give the fifth matrix partitioning problem of our concern.

Definition 2.13 A bipartition (Π_C, Π_R) of a matrix A is *consistent with a determination by substitution* if there exists an ordering on the elements of A such that for every nonzero entry a_{ij}, either column j is in a group where all nonzeros in row i, from other columns in the group, are ordered before a_{ij} or row i is in a group where all the nonzeros in column j, from other rows in the group, are ordered before a_{ij}.

Problem 2.14 (*MPP5*) *Given the sparsity structure of a matrix $A \in R^{m \times n}$, find a bipartition (Π_C, Π_R) of A consistent with a determination by substitution such that $|\Pi_C| + |\Pi_R|$ is minimized.*

Problems MPP1 through MPP5 are summarized in Table 2. Note that the problems in Table 2 are formulated independent of the numerical technique used. For example, MPP1 could arise in using FD or only the forward mode of AD.

2.3 Other Methods

The matrix estimation methods considered in this paper rely on a one-dimensional or a two-dimensional *partition* (symmetrically) consistent with a direct or a substitution-based determination. However, approaches where this is not necessarily the case have also been suggested. For instance, direct methods that allow structurally non-orthogonal columns to reside in the same group and/or allow columns to reside in several groups have been suggested [8, 25, 26]. McCormick [23] gives a classification of direct methods for estimating a symmetric matrix, including those that do not necessarily rely on consistent partitions.

In another direction, Hossain [16] suggests a technique for a direct estimation of a Jacobian in which the rows are first grouped into blocks that define 'segmented' columns and then the segments are partitioned into groups each of which consists of structurally independent segments. It is shown that, for some matrix structures, such an approach may reduce the number of function evaluations compared with an approach that does not use segmentation.

3 Graph Formulations

Problems MPP1 through MPP5 are combinatorial and several previous studies have demonstrated the usefulness of graph theoretic approaches in analyzing and solving them [4, 7, 8, 9, 17, 23]. In this section, we integrate these approaches within a unified framework. In addition, we propose a new, more flexible, graph formulation for problem MPP1. The graph formulation of each matrix problem depends on the choice of the graph used to represent the matrix structure. In Section 3.2 we describe our graph representations which will be used to develop the equivalent graph problems in Sections 3.3 through 3.5. In Section 3.6 we show how the various graph problems obtained relate with one another. We start this section by defining some basic graph theoretic concepts. Other graph concepts will be defined later as required.

3.1 Basic Definitions

A *graph* G is an ordered pair (V, E) where V is a finite and nonempty set of *vertices* and E is a set of unordered pairs of distinct vertices called *edges*. If $(u, v) \in E$, vertices u and v are said to be *adjacent*; otherwise they are called *non-adjacent*. A *path* of *length* l in a graph is a sequence $v_1, v_2, \ldots, v_{l+1}$ of distinct vertices such that v_i is adjacent to v_{i+1}, for $1 \leq i \leq l$. Two distinct vertices are said to be *distance-k neighbors* if the shortest path connecting them has length *at most* k; otherwise they are called *non-distance-k neighbors*. The number of distance-k neighbors of a vertex u is referred to as the *degree-k* of u.

In a graph $G = (V, E)$, a set of vertices $C \subseteq V$ is said to *cover* a set of edges $F \subseteq E$ if for every edge $e \in F$, at least one of the endpoints of e is in C. If the set C covers the entire E, it is called a *vertex cover*. The set of vertices $I \subseteq V$ is called an *independent set* if no pair of vertices in I are adjacent to each other.

A graph $G = (V, E)$ is *bipartite* if its vertex set V can be partitioned into

two disjoint sets V_1 and V_2 such that every edge in E connects a vertex from V_1 to a vertex from V_2. We denote a bipartite graph by $G_b = (V_1, V_2, E)$.

3.2 Representing Matrix Structures Using Graphs

Bipartite graph Let A be an $m \times n$ matrix with rows $r_1, r_2, \ldots, r_m$ and columns $a_1, a_2, \ldots, a_n$. We define the bipartite graph $G_b(A)$ of A as $G_b(A) = (V_1, V_2, E)$ where $V_1 = \{r_1, r_2, \ldots, r_m\}$, $V_2 = \{a_1, a_2, \ldots, a_n\}$, and $(r_i, a_j) \in E$ whenever a_{ij} is a nonzero element of A, for $1 \leq i \leq m$, $1 \leq j \leq n$.

Note that $G_b(A)$ is a space-efficient representation for a nonsymmetric matrix A. To see this, notice that the number of vertices $|V_1| + |V_2| = m+n$, and the number of edges $|E| = nnz(A)$, where $nnz(A)$ is the number of nonzeros in A. Also, note that the graph can be constructed by reading off the entries of the matrix without any further computation.

Adjacency graph Let $A \in R^{n \times n}$ be a symmetric matrix with nonzero diagonal elements and let its columns be $a_1, a_2, \ldots, a_n$. The adjacency graph of A is $G(A) = (V, E)$ where $V = \{a_1, a_2, \ldots, a_n\}$, and $(a_i, a_j) \in E$ whenever a_{ij} is a nonzero element of A, for $1 \leq i \leq n$, $1 \leq j \leq n$, $i \neq j$. Note that $G(A)$ is a space-efficient, symmetry-exploiting, graph representation of the symmetric matrix A with no explicit representation for the edges corresponding to the nonzero diagonal elements. In particular, the number of vertices is n and the number of edges is $\frac{1}{2}(nnz(A) - n)$. This is in contrast to $2n$ and $nnz(A)$, respectively, had a bipartite graph representation been used.

For the symmetric matrix of our interest, the Hessian matrix, the assumption that the diagonals are nonzero is reasonable in many contexts. In particular, the Hessian is usually positive definite [4].

To simplify notation, we shall use a_i both when referring to the ith column of the matrix A and the corresponding vertex in $G(A)$ or $G_b(A)$.

Column intersection graph Our graph formulations for problems MPP1 through MPP5 are based on the bipartite and adjacency graph representations discussed above. However, in the literature, other graph representations have also been used. In particular, Coleman and Moré [7] used the column intersection graph $G_c(A)$ to represent a nonsymmetric matrix A. In $G_c(A) = (V, E)$, the columns of A constitute the vertex set V, and an edge (a_i, a_j) is in E whenever columns a_i and a_j have nonzero entries at the same row position, i.e. whenever a_i and a_j are not structurally orthogonal.

3.3 Distance-k Graph Coloring

A *distance-k p-coloring*, or (k,p)-coloring for short, of a graph $G = (V,E)$ is a mapping $\phi : V \to \{1, 2, \ldots, p\}$ such that $\phi(u) \neq \phi(v)$ whenever u and v are distance-k neighbors. The minimum possible value of p in a (k,p)-coloring of a graph G is called its *k-chromatic number*, and is denoted by $\chi_k(G)$. A (k,p)-coloring of $G = (V,E)$ is called *partial* if it involves only a subset of the vertices; in particular, a partial (k,p)-coloring of $G = (V,E)$ on W, $W \subset V$, is a mapping $\phi : W \to \{1, 2, \ldots, p\}$ such that $\phi(u) \neq \phi(v)$ whenever u and v are distance-k neighbors.

The *distance-k graph coloring problem* (DkGCP) asks for an optimal (k,p)-coloring of a graph: given a graph G and an integer k, find a (k,p)-coloring of G such that p is minimized.

Notice that a (k,p)-coloring of $G = (V,E)$ partitions the set V into p groups (called *color classes*) $U_1, U_2, \ldots, U_p$, where $U_i = \{u \in V : \phi(u) = i\}$. Each color class is a *distance-k independent set*, i.e., no pair of distinct vertices consists of distance-k neighbors. Noting this, a natural question arises—can the matrix partitioning problems MPP1 through MPP5 be related to the DkGCP for some value of k?

Recall that structural orthogonality is the criterion used in a consistent partition of the columns (or rows) of a matrix. The following two simple observations provide the graph theoretic equivalents of structural orthogonality in matrices.

Lemma 3.1 *Let $A \in R^{m \times n}$ be a matrix and $G_b(A) = (V_1, V_2, E)$ be its bipartite graph. Two columns (or rows) of A are structurally orthogonal if and only if the corresponding vertices in $G_b(A)$ are non-distance-2 neighbors.*

Proof: We prove the statement for columns; a similar argument can be used to prove the case for rows. Assume that vertices a_i and a_j in V_2 are non-distance-2 neighbors. Thus, by definition, there is no path a_i, r_k, a_j in G_b for any $r_k \in V_1$, $1 \leq k \leq m$. In terms of matrix A, this means that there is no $k \in [1,\ m]$ such that both a_{ki} and a_{kj} are nonzero. Hence, by definition, a_i and a_j are structurally orthogonal.

To prove the 'only if' part of the statement, assume that columns a_i and a_j are structurally orthogonal. Then, by definition, there is no $k \in [1,\ m]$ such that $a_{ki} \neq 0$ and $a_{kj} \neq 0$. This implies that there is no path a_i, r_k, a_j in $G_b(A)$, for any $1 \leq k \leq m$. Hence, by definition, a_i and a_j are non-distance-2 neighbors. $\square$

Similarly, one can prove the following statement for the case of a symmetric matrix and its adjacency graph representation.

Lemma 3.2 *Let $A \in R^{n\times n}$ be a symmetric matrix with nonzero diagonal elements and let $G(A) = (V, E)$ be its adjacency graph. Two columns in A are structurally orthogonal if and only if the corresponding vertices in $G(A)$ are non-distance-2 neighbors.*

Lemmas 3.1 and 3.2 provide a partial answer to our question regarding the relationship between the matrix partitioning problems and the DkGCP. As discussed in the forthcoming sections, distance-2 coloring is a generic model in efficient derivative matrix estimation using methods that rely on one-dimensional column or row partition.

3.4 Coloring Problems in Direct Methods

In this subsection we consider problems MPP1, MPP2, and MPP4.

By Lemma 3.1, finding a consistent partition of the columns of a matrix A is equivalent to finding a *partial* distance-2 coloring of $G_b(A) = (V_1, V_2, E)$ on V_2. The following result formalizes the equivalence.

Theorem 3.3 *Let A be a nonsymmetric matrix and $G_b(A) = (V_1, V_2, E)$ be its bipartite graph representation. A mapping ϕ is a partial distance-2 coloring of $G_b(A)$ on V_2 if and only if ϕ induces a consistent partition of the columns of A.*

In view of Theorem 3.3, Problem MPP1 is equivalent to the following graph coloring problem (GCP).

Problem 3.4 (*GCP1*) *Given the bipartite graph $G_b(A) = (V_1, V_2, E)$ representing the sparsity structure of a matrix $A \in R^{m\times n}$, find a partial (2,p)-coloring of $G_b(A)$ on V_2 with the least value of p.*

For matrices with a few dense rows, a row partition may yield fewer groups than a column partition. Consequently, the matrix problem one needs to solve is MPP1 applied on A^T. In such cases, our graph formulation becomes handy—the equivalent problem is to find a partial distance-2 coloring on the vertex set V_1.

By Lemma 3.2, finding a consistent partition of the columns of a symmetric matrix A is equivalent to finding a distance-2 coloring of the adjacency graph $G(A)$. This equivalence was in fact first observed by McCormick [23]. However, as has been stated earlier, the symmetry present in A can be

exploited to further reduce the number of groups (colors) required. Thus, we now consider the graph coloring formulation of the partitioning problem where A is symmetric (MPP2).

Consider a symmetric matrix A with nonzero diagonal elements and let a_{ij}, $i \neq j$, be any nonzero element in A. Further, let a_{ki}, $k \neq i, j$ and a_{jl}, $l \neq i, j, k$ be any other two nonzero elements. By Definition 2.4, in a symmetrically consistent partition of A,

- columns a_i and a_j should belong to two different groups (this is because both a_{ii} and a_{jj} are nonzero), and
- columns a_j and a_k should belong to two different groups, or columns a_i and a_l should belong to two different groups.

Coleman and Moré [8] gave an equivalent characterization of the aforementioned conditions in terms of a coloring of the associated adjacency graph. Specifically, they introduced the notion of *distance-$\frac{3}{2}$ coloring*[2], defined below.

Definition 3.5 A mapping $\phi : V \rightarrow \{1, 2, \ldots, p\}$ is a $(\frac{3}{2}, p)$*-coloring* of the graph $G = (V, E)$ if ϕ is $(1, p)$-coloring of G and every path of length three uses at least three colors.

The name 'distance-$\frac{3}{2}$ coloring' is chosen to reflect that it is in a sense 'in-between' distance-1 and distance-2 colorings. In particular, a distance-$\frac{3}{2}$ coloring is a relaxed distance-2 and a restricted distance-1 coloring. As an illustration, observe that a distance-1 coloring requires two colors for every path of length one, a distance-2 coloring requires three colors for every path of length two, and a distance-$\frac{3}{2}$ coloring is a distance-1 coloring further restricted to require three colors for every path of length three (see Figure 2). Note that a 4-cycle requires two, three, and four colors in a distance-1, a distance-$\frac{3}{2}$ and a distance-2 coloring, respectively.

The following theorem formalizes the connection between symmetrically consistent partition and distance-$\frac{3}{2}$ coloring. The result follows directly from the discussion that led to the definition of $(\frac{3}{2}, p)$-coloring.

Theorem 3.6 [Coleman and Moré [8]]
Let A be a symmetric matrix with nonzero diagonal elements and $G(A) = (V, E)$ be its adjacency graph representation. A mapping ϕ is a $(\frac{3}{2}, p)$-coloring of $G(A)$ if and only if ϕ induces a symmetrically consistent partition of the columns of A.

[2] Coleman and Moré used the term *path-coloring*.

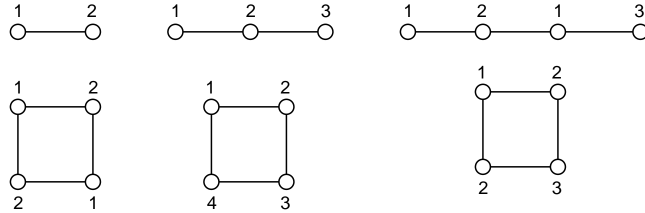

Figure 2: Distance-1, 2, and $\frac{3}{2}$ coloring of paths and a 4-cycle.

By Theorem 3.6, the following problem is equivalent to MPP2.

Problem 3.7 (*GCP2*) *Given the adjacency graph $G(A) = (V, E)$ representing the sparsity structure of a symmetric matrix $A \in R^{n\times n}$ with nonzero diagonal elements, find a $(\frac{3}{2}, p)$-coloring of $G(A)$ with the least value of p.*

Our next problem, MPP4, aims at finding a bipartition with the fewest number of groups consistent with a direct determination. When a nonsymmetric matrix A is represented by its bipartite graph $G_b(A) = (V_1, V_2, E)$, we have shown that a column partition consistent with a direct determination can be obtained by finding a partial distance-2 coloring of G_b on V_2. We now consider how this coloring has to be modified to capture a bipartition consistent with a direct determination. Notice that the coloring we are looking for should meet the following conditions.

- The sets V_1 and V_2 should use disjoint set of colors.
- Some vertices may not be involved in the determination of any nonzero entry of the underlying matrix. Such vertices are assigned a 'neutral' color (say color zero).
- Since every nonzero matrix entry has to be determined, for every edge in E, at least one of the endpoints has to be assigned a nonzero color.
- A nonzero matrix entry may be determined either from a positively colored column vertex or a positively colored row vertex. This suggests that the coloring condition sought here is some relaxation of the distance-2 coloring requirement imposed in the case of one-dimensional partition.

The following definition, introduced by Coleman and Verma [9], albeit using a different terminology, formalizes the conditions listed above. The subsequent theorem establishes the equivalence between the matrix and graph problems.

Definition 3.8 Let $G_b = (V_1, V_2, E)$ be a bipartite graph. A mapping $\phi : [V_1, V_2] \rightarrow \{0, 1, \ldots, p\}$ is a *distance-$\frac{3}{2}$ bicoloring* of G_b if the following conditions hold.

1. If $u \in V_1$ and $v \in V_2$, then $\phi(u) \neq \phi(v)$ or $\phi(u) = \phi(v) = 0$.
2. If $(u, v) \in E$, then $\phi(u) \neq 0$ or $\phi(v) \neq 0$.
3. If vertices u and v are adjacent to vertex w with $\phi(w) = 0$, then $\phi(u) \neq \phi(v)$.
4. Every path of three edges uses at least three colors.

Theorem 3.9 [Coleman and Verma[9]]
Let A be an $m \times n$ matrix and $G_b(A) = (V_1, V_2, E)$ be its bipartite graph. The mapping $\phi : [V_1, V_2] \rightarrow \{0, 1, \ldots, p\}$ is a distance-$\frac{3}{2}$ p-bicoloring if and only if ϕ induces a bipartition (Π_C, Π_R) of A, with $|\Pi_C| + |\Pi_R| = p$, consistent with a direct determination.

Thus MPP4 is equivalent to the following graph problem.

Problem 3.10 (*GCP4*) *Given the bipartite graph $G_b(A) = (V_1, V_2, E)$ representing the sparsity structure of an $m \times n$ matrix A, find a distance-$\frac{3}{2}$ p-bicoloring of $G_b(A)$ with the least value of p.*

3.5 Coloring Problems in Substitution Methods

In this subsection, we consider problems MPP3 and MPP5. To formulate MPP3 as a graph problem, we introduce the notion of *acyclic* coloring. Coleman and Cai [4] established the connection between acyclic[3] coloring and the estimation of a symmetric matrix using a substitution method. Acyclic coloring had been studied earlier by Grünbaum [15] in a different context.

Definition 3.11 *A mapping $\phi : V \rightarrow \{1, 2, \ldots, p\}$ is an acyclic p-coloring of a graph $G = (V, E)$ if ϕ is $(1, p)$-coloring and every cycle in G uses at least three colors.*

Theorem 3.12 [Coleman and Cai [4]]
Let A be a symmetric matrix with nonzero diagonal elements and $G(A) = (V, E)$ be its adjacency graph representation. A mapping ϕ is an acyclic p-coloring of $G(A)$ if and only if ϕ induces a substitutable partition of the columns of A.

[3] Coleman and Cai use the term cyclic coloring to refer to what is known as acyclic coloring in the graph theoretic literature.

Consider the 5×5 symmetric matrix and its adjacency graph depicted in Figure 3. An optimal acyclic coloring that uses three colors is shown. It can be verified that this induces a substitutable partition of the columns. By contrast, a symmetric partition consistent with a direct determination (a distance-$\frac{3}{2}$ coloring) would have required four groups (colors).

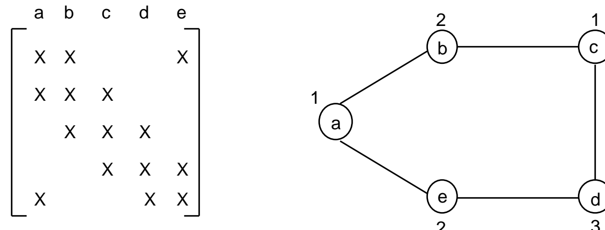

Figure 3: A 5×5 symmetric matrix and an acyclic coloring of its adj. graph

In view of Theorem 3.12, MPP3 is equivalent to the following graph problem.

Problem 3.13 (*GCP3*) *Given the adjacency graph $G(A) = (V, E)$ representing the sparsity structure of a symmetric matrix $A \in R^{n \times n}$ with nonzero diagonal elements, find an acyclic p-coloring of $G(A)$ with the least value of p.*

The relationship between bicoloring and bipartition, established by Theorem 3.9, coupled with that between acyclic coloring and substitutable partition, established by Theorem 3.12, suggests that 'acyclic bicoloring' might be the right graph model for MPP5. Coleman and Verma [9] showed that this was indeed the case.

Definition 3.14 Let $G_b = (V_1, V_2, E)$ be a bipartite graph. A mapping $\phi : [V_1, V_2] \to \{0, 1, \ldots, p\}$ is an *acyclic bicoloring* of G_b if the following conditions hold.

1. If $u \in V_1$ and $v \in V_2$, then $\phi(u) \neq \phi(v)$ or $\phi(u) = \phi(v) = 0$.
2. If $(u, v) \in E$, then $\phi(u) \neq 0$ or $\phi(v) \neq 0$.
3. If vertices u and v are adjacent to vertex w with $\phi(w) = 0$, then $\phi(u) \neq \phi(v)$.
4. Every cycle uses at least three colors.

Matrix	1D Partition	2D Partition	Method
Jacobian	distance-2 coloring	distance-$\frac{3}{2}$ bicoloring	Direct
Hessian	distance-$\frac{3}{2}$ coloring	NA	Direct
Jacobian	NA	acyclic bicoloring	Substitution
Hessian	acyclic coloring	NA	Substitution

Table 3: Graph coloring formulations for estimating *all* nonzero entries of derivative matrices. The Jacobian and the Hessian are represented by their bipartite and adjacency graphs, respectively. NA stands for not applicable.

Theorem 3.15 [Coleman and Verma [9]]
Let A be an $m \times n$ matrix and $G_b(A) = (V_1, V_2, E)$ be its bipartite graph. The mapping $\phi : [V_1, V_2] \rightarrow \{0, 1, \ldots, p\}$ is an acyclic p-bicoloring if and only if ϕ induces a bipartition (Π_C, Π_R) of A, with $|\Pi_C| + |\Pi_R| = p$, consistent with determination by substitution.

By Theorem 3.15, the following coloring problem is equivalent to MPP5.

Problem 3.16 (*GCP5*) *Given the bipartite graph $G_b(A) = (V_1, V_2, E)$ representing the sparsity structure of an $m \times n$ matrix A, find an acyclic p-bicoloring of $G_b(A)$ with the least value of p.*

Table 3 (the same table was given in Section 1) summarizes the graph coloring problems that arise in efficient derivative matrix estimation.

3.6 Distance-k Chromatic Numbers

In this subsection, we expose the inter-relationship among the various colorings introduced thus far and show that distance-2 coloring is the most *general* among them.

The *power* of a graph gives an alternative view to the DkGCP. The kth power of a graph $G = (V, E)$ is the graph $G^k = (V, F)$ where $(u, v) \in F$ if and only if u and v are distance-k neighbors in G. The following equivalence follows immediately.

Lemma 3.17 *Let G^k be the kth power of graph G. A mapping ϕ is a (k, p)-coloring of G if and only if it is a $(1, p)$-coloring of G^k.*

A particular implication of Lemma 3.17 is that distance-2 coloring of a graph is equivalent to distance-1 coloring of the square of the graph. This establishes the equivalence between GCP1, our bipartite graph based formulation of MPP1, and the distance-1 coloring formulation, which uses the

column intersection graph, suggested by Coleman and Moré [7]. Specifically, as has been shown in [7], the column intersection graph $G_c(A)$ of a matrix A is isomorphic to the adjacency graph of $A^T A$. We note that $G_c(A)$ is in fact the subgraph of $G_b(A)^2$ induced by the vertices in V_2. For a graph $G = (V, E)$, let the graph induced by $U \subseteq V$ be denoted by $G[U]$.

Lemma 3.18 *Let $G_b(A) = (V_1, V_2, E)$ and $G_c(A) = (V_2, E')$ be the bipartite and column intersection graphs of matrix A. Then, $G_c = {G_b}^2[V_2]$.*

Observe that the distance-1 neighbors of a vertex in a graph G form a *clique* in the square of the graph. A clique is a set of vertices in which the vertices are mutually adjacent to each other. This observation immediately provides a lower bound on $\chi_2(G)$, the 2-chromatic number of G. Let Δ denote the maximum degree-1 in G.

Lemma 3.19 *For any graph G, $\chi_2(G) \geq \Delta + 1$.*

Proof: Observe that the cardinality of a maximum clique in the square graph G^2 is $\Delta + 1$. □

Further, the conditions required by distance-1 coloring, acyclic coloring, distance-$\frac{3}{2}$ coloring, acyclic bicoloring, distance-$\frac{3}{2}$ bicoloring, and distance-2 coloring imply the following relationships among their respective chromatic numbers. The chromatic numbers for distance-k coloring (and bicoloring) of a general graph G (and bipartite graph G_b) are denoted by $\chi_k(G)$ (and $\chi_{kb}(G_b)$). Similarly the chromatic numbers for acyclic coloring (and bicoloring) of a graph G (and a bipartite graph G_b) are denoted by $\chi^a(G)$ (and $\chi^{ab}(G_b)$).

Theorem 3.20 *For a general graph $G = (V, E)$,*

$$\chi_1(G) \leq \chi^a(G) \leq \chi_{\frac{3}{2}}(G) \leq \chi_2(G) = \chi_1(G^2).$$

Proof: Observe that a distance-2 coloring is a distance-$\frac{3}{2}$ coloring; a distance-$\frac{3}{2}$ coloring is an acyclic coloring; and an acyclic coloring is a distance-1 coloring. □

Theorem 3.21 *For a bipartite graph $G_b = (V_1, V_2, E)$,*

$$\chi^{ab}(G_b) \leq \chi_{\frac{3}{2}b}(G_b) \leq min\{\chi_1({G_b}^2[V_1]), \chi_1({G_b}^2[V_2])\},$$

where $({G_b}^2[W])$ is the sub-graph of ${G_b}^2$ induced by W.

Proof: The first inequality is obvious. For the second inequality, observe that a partial distance-2 coloring on V_2 is a valid distance-$\frac{3}{2}$ bicoloring where all the vertices in V_1 are specified to be colored with 0. A similar argument, with the roles of V_1 and V_2 interchanged, can be used to complete the proof. □

In the context of matrix estimation using numerical methods, the implication of Theorem 3.21 is that, an optimal two-dimensional partition, irrespective of the structure of the matrix, yields fewer (or at most as many) groups compared to an optimal one-dimensional partition, and hence potentially results in a more efficient computation.

The results in this section show that distance-2 coloring is an archetypal model in the estimation of Jacobian and Hessian matrices using techniques that rely on one-dimensional partition via direct and substitution methods.

Distance-2 coloring has other applications. Examples include channel assignment [20] and facility location problems (see Chapter 5 in [27]). From a more theoretical perspective, distance-2 coloring for planar graphs has been studied in [2] and a similar study for chordal graphs is available in [1].

4 Partial Matrix Estimation

In many PDE constrained optimization contexts, the Jacobian or the Hessian is formed only for preconditioning purposes. For preconditioning, it is often common to compute a *subset* of the matrix elements. Computing a good preconditioner is critical for fast convergence to the solution. The recent survey article [19] discusses various applications where methods known as "Jacobian-free Newton-Krylov" are used. A basic ingredient in the use of these methods is an approximate computation of *some* elements of the Jacobian. Also, there are examples in which only certain elements of the Hessian need to be updated in an iterative procedure, while others do not because they are unlikely to change in value [3].

In this section we develop the graph coloring formulations of the partitioning problems that arise in the estimation of a *specified subset* of the nonzero entries of a matrix via direct methods. We call this *partial* matrix estimation as opposed to *full* matrix estimation, where all nonzero entries are required to be determined.

The coloring formulations in this section are new and more sophisticated than the coloring formulations in full matrix estimation. The motivation for developing the new graph formulations is that efficient partial matrix estimation can be used to further reduce the number of colors needed to

estimate the required elements. For example, if only the diagonal elements of a Hessian are needed, then we need only the *distance-1 coloring* of the adjacency graph, rather than the distance-$\frac{3}{2}$ coloring required for full matrix estimation.

The colorings defined in this section allow a vertex to have the color zero. A vertex with color zero signifies the fact that it would not be used to estimate any element of the matrix represented by the graph, i.e., columns or rows that correspond to the color zero are not used to estimate any elements in those columns or rows.

The rest of this section is organized in three parts. Each part deals with a scenario defined by the kind of matrix under consideration (symmetric or nonsymmetric) and the type of partition employed (one-dimensional or two-dimensional). In each case, the required entries are assumed to be determined using a direct method. The problems that correspond to estimation via substitution are not considered in this paper.

4.1 Nonsymmetric matrix, One-dimensional partition

Let $A \in R^{m \times n}$ be a nonsymmetric matrix, and S denote the set of nonzero elements of A required to be estimated. A partition $\{C_1, \ldots, C_p\}$ of a subset of the columns of A is *consistent with a direct determination of* S if for every $a_{ij} \in S$, column a_j is included in some group that contains no other column with a nonzero in row i.

Let $G_b(A) = (V_1, V_2, E)$ be the bipartite graph of A, and $F \subseteq E$ correspond to the elements of S. A mapping $\phi : V_2 \rightarrow \{0, 1, \ldots, p\}$ is a *distance-2 coloring of G_b restricted to F* if the following conditions hold for every $(v, w) \in F$, where $v \in V_1$, $w \in V_2$.

1. $\phi(w) \neq 0$, and
2. for every path (u, v, w), $\phi(u) \neq \phi(w)$.

Theorem 4.1 *The mapping ϕ is a distance-2 p-coloring of $G_b(A)$ restricted to F if and only if ϕ induces a column partition $\{C_1, \ldots, C_p\}$ consistent with a direct determination of S.*

Proof: Assume that ϕ is a distance-2 p-coloring of $G_b(A)$ restricted to F. We show that the groups $\{C_1, \ldots, C_p\}$ where $C_\alpha = \{a_j : \phi(a_j) = \alpha\}$, $1 \leq \alpha \leq p$, constitute a partition of a subset of the columns of A consistent with a direct determination of S. First, observe that by coloring condition 1, for every $a_{ij} \in S$ $((r_i, a_j) \in F)$, $\phi(a_j) \neq 0$ and thus column a_j belongs to group $C_{\phi(a_j)}$ and hence is involved in the partition. Assume now that the partition

induced by the coloring is not consistent with a direct determination of S. This occurs only if there exist nonzero elements a_{ij} and a_{ik}, $k \neq j$, such that $a_{ij} \in S$ and both a_j and a_k belong to group $C_{\alpha'}$ for some α', $1 \leq \alpha' \leq p$. But this contradicts coloring condition 2, and hence cannot occur.

Conversely, assume that the partition $C = \{C_1, \ldots, C_p\}$ is consistent with a direct determination of S. Construct a coloring ϕ of $G_b(A)$ as follows. $\phi(a_j) = \alpha$ if $a_j \in C_\alpha$, and $\phi(a_j) = 0$ if $a_j \notin C$. We claim that ϕ is a distance-2 p-coloring of $G_b(A)$ restricted to F. Each vertex in V_2 incident to an edge in F corresponds to a column with an entry in S and thus gets a nonzero color. Thus ϕ satisfies coloring condition 1. Consider any path (a_j, r_i, a_k) where $(r_i, a_j) \in F$. Note that such a path in $G_b(A)$ exists whenever entries a_{ij} and a_{ik} are nonzero. The partition condition implies that column a_k cannot be in the same group as a_j. Thus, by construction, $\phi(a_j) \neq \phi(a_k)$, satisfying coloring condition 2. $\square$

4.2 Symmetric matrix, One-dimensional partition

Let $A \in R^{n \times n}$ be a symmetric matrix with nonzero diagonal elements, and S denote the set of nonzero elements of A required to be estimated. A partition $\{C_1, \ldots, C_p\}$ of a subset of the columns of A is *symmetrically consistent with a direct determination of* S if for every $a_{ij} \in S$ at least one of the following two conditions are met.

1. The group containing a_j has no other column with a nonzero in row i.

2. The group containing a_i has no other column with a nonzero in row j.

Let $G(A) = (V, E)$ be the adjacency graph of A, $F_{od} \subseteq E$ correspond to the off-diagonal elements in S, and F_d correspond to the diagonal elements in S, i.e., $F_d = \{(u, u) : u \in U\}$ where $U \subseteq V$. Let $F = F_{od} \cup F_d$. A mapping $\phi : V \to \{0, 1, 2, \ldots, p\}$ is a *distance-$\frac{3}{2}$ coloring of G restricted to F* if the following conditions hold.

1. For every $(u, u) \in F_d$,

    1.1. $\phi(u) \neq 0$, and

    1.2. for every $(u, v) \in E$, $\phi(u) \neq \phi(v)$.

2. For every $(v, w) \in F_{od}$,

    2.1. $\phi(v) \neq \phi(w)$, and

    2.2. at least one of the following two conditions holds:

2.2.1. $\phi(v) \neq 0$ and for every path (v, w, x), $\phi(v) \neq \phi(x)$ or

2.2.2. $\phi(w) \neq 0$ and for every path (u, v, w), $\phi(u) \neq \phi(w)$.

Theorem 4.2 *The mapping ϕ is a distance-$\frac{3}{2}$ p-coloring of $G(A)$ restricted to F if and only if ϕ induces a column partition $\{C_1, \ldots, C_p\}$ symmetrically consistent with a direct determination of S.*

Proof: Assume that ϕ is a distance-$\frac{3}{2}$ p-coloring of $G(A)$ restricted to F. We show that the groups $\{C_1, \ldots, C_p\}$ where $C_\alpha = \{a_j : \phi(a_j) = \alpha\}$, $1 \leq \alpha \leq p$, constitute a partition of a subset of the columns of A symmetrically consistent with a direct determination of S.

By coloring conditions 1 and 2, for every $a_{ij} \in S$ ($(a_i, a_j) \in F$), at least one of a_i or a_j has a nonzero color and hence is involved in the partition $\{C_1, \ldots, C_p\}$. Let $a_{ij} \in S$ be a diagonal entry ($i = j$). Then, coloring condition 1 ensures that $\phi(a_i) \neq 0$ and that $\phi(a_i) \neq \phi(a_k)$ for every $(a_i, a_k) \in E$. Thus, by construction, column a_i belongs to group $C_{\phi(a_i)}$ and no column a_k, $k \neq i$ with $a_{ik} \neq 0$ is in $C_{\phi(a_i)}$. This clearly satisfies the partition condition. Let $a_{ij} \in S$ now be an off-diagonal entry ($i \neq j$). Assume without loss of generality that $\phi(a_j) \neq 0$. By condition 2.1, $\phi(a_j) \neq \phi(a_i)$. By condition 2.2.2, there is no path (a_k, a_i, a_j) in $G(A)$, for any $k \neq i, j$ such that $\phi(a_j) = \phi(a_k)$. The last two statements together imply that column a_j belongs to group $C_{\phi(a_j)}$ and that no column a_k, $k \neq j$ with $a_{ik} \neq 0$ is in $C_{\phi(a_j)}$. This satisfies partition condition 1. A similar argument applies to the case where $\phi(a_i) \neq 0$ which implies the satisfaction of the alternative partition condition.

To prove the converse, assume that the partition $C = \{C_1, \ldots, C_p\}$ is symmetrically consistent with a direct determination of S. Construct a coloring ϕ of $G(A)$ as follows. $\phi(a_j) = \alpha$ if $a_j \in C_\alpha$, and $\phi(a_j) = 0$ if $a_j \notin C$. We claim that ϕ is a distance-$\frac{3}{2}$ p-coloring of $G(A)$ restricted to F.

Consider a diagonal element $a_{ii} \in S$. The partition conditions ensure that a_i is in some group $C_{\alpha'}$ and that there is no column $a_k \in C_{\alpha'}$, $k \neq i$ such that $a_{ik} \neq 0$. Thus, by construction, $\phi(a_i) \neq 0$ and $\phi(a_i) \neq \phi(a_k)$ for every $(a_i, a_k) \in E$, satisfying coloring condition 1. Consider now the case where $a_{ij} \in S$ is an off-diagonal element. First, observe that since all diagonal elements are nonzero, a_i and a_j cannot belong to the same group. Thus $\phi(a_i) \neq \phi(a_j)$, satisfying coloring condition 2.1. Second, observe that there are two possibilities by which the partitioning conditions have been satisfied. We consider only one of these; the second one can be treated in a similar manner. Suppose a_j belongs to some group $C_{\alpha'}$ and that there is no other column $a_k \in C_{\alpha'}$, $k \neq j$ such that $a_{ik} \neq 0$. Thus, by construction,

$\phi(a_j) \neq 0$ and $\phi(a_j) \neq \phi(a_k)$ for every path (a_k, a_i, a_j) in $G(A)$, satisfying coloring condition 2.2.2. □

A special case of Theorem 4.2 is the problem of estimating *only* the diagonal elements of A, i.e., $F_d = \{(v, v) : v \in V\}$ and $F_{od} = \emptyset$. For this problem, condition 1 is the only applicable condition, and states that a distance-1 coloring of $G(A)$ is sufficient.

4.3 Nonsymmetric matrix, Two-dimensional partition

Let $A \in R^{m \times n}$ be a nonsymmetric matrix, and S denote the set of nonzero elements of A required to be estimated. A bipartition (Π_C, Π_R) of a subset of the columns and rows of A is *consistent with a direct determination of* S if for every $a_{ij} \in S$ at least one of the following conditions are met.

1. The group (in Π_C) containing column j has no other column having a nonzero in row i.

2. The group (in Π_R) containing row i has no other row having a nonzero in column j.

Let $G_b(A) = (V_1, V_2, E)$, and $F \subseteq E$ correspond to the elements in S. A mapping $\phi : [V_1, V_2] \rightarrow \{0, 1, \ldots, p\}$ is said to be a *distance-$\frac{3}{2}$ bicoloring of G_b restricted to F* if the following conditions are met.

1. Vertices in V_1 and V_2 receive disjoint colors, except for color 0; i.e., for every $u \in V_1$ and $v \in V_2$, either $\phi(u) \neq \phi(v)$ or $\phi(u) = \phi(v) = 0$.

2. At least one endpoint of an edge in F receives a nonzero color; i.e., for every $(v, w) \in F$, $\phi(v) \neq 0$ or $\phi(w) \neq 0$.

3. For every $(v, w) \in F$,

   3.1. if $\phi(v) = 0$, then, for every path (u, v, w), $\phi(u) \neq \phi(w)$,

   3.2. if $\phi(w) = 0$, then, for every path (v, w, x), $\phi(v) \neq \phi(x)$,

   3.3. if $\phi(v) \neq 0$ and $\phi(w) \neq 0$, then for every path (u, v, w, x), either $\phi(u) \neq \phi(w)$ or $\phi(v) \neq \phi(x)$.

Theorem 4.3 *The mapping ϕ is a distance-$\frac{3}{2}$ p-bicoloring of G_b restricted to F if and only if ϕ induces a bipartition (Π_C, Π_R), $|\Pi_C| + |\Pi_R| = p$, consistent with a direct determination of S.*

Proof: Let the construction of a partition given a coloring, and vice-versa, be done in a similar manner as in the proof of Theorem 4.1.

Assume that ϕ is a distance-$\frac{3}{2}$ bicoloring of $G_b(A)$ restricted to F. Let the induced bipartition be (Π_C, Π_R). Coloring condition 1 implies that (Π_C, Π_R) is a bipartition. By condition 2, for every $a_{ij} \in S$, either $a_j \in \Pi_C$ or $r_i \in \Pi_R$ (or both). Assume now that (Π_C, Π_R) is not consistent with a direct determination of S. This occurs only if one of the following cases hold for any $a_{ij} \in S$:

- $\phi(r_i) = 0, \phi(a_j) \neq 0$ and there exists a column a_k, $k \neq j$ with $a_{ik} \neq 0$ such that $\phi(a_j) = \phi(a_k)$. But this contradicts coloring condition 3.1, and hence cannot occur.

- $\phi(a_j) = 0, \phi(r_i) \neq 0$ and there exists a row r_l, $l \neq i$ with $a_{lj} \neq 0$ such that $\phi(r_i) = \phi(r_l)$. But this contradicts coloring condition 3.2, and hence cannot occur.

- $\phi(r_i) \neq 0, \phi(a_j) \neq 0$, and there exist column a_k, $k \neq j$ with $a_{ik} \neq 0$, and row r_l, $l \neq i$ with $a_{lj} \neq 0$ such that $\phi(a_j) = \phi(a_k)$ and $\phi(r_i) = \phi(r_l)$. But this contradicts coloring condition 3.3, and hence cannot occur.

Hence, (Π_C, Π_R) is consistent with a direct determination of S.

Conversely, assume that (Π_C, Π_R) is a bipartition consistent with a direct determination of S. Clearly, the constructed coloring ϕ satisfies conditions 1 and 2. To complete the proof, we show that ϕ also satisfies condition 3. Assume that ϕ violates condition 3. Then one of the following cases must have happened:

- There exists a path (a_k, r_i, a_j) for some $(r_i, a_j) \in F$ such that $\phi(r_i) = 0$ and $\phi(a_j) = \phi(a_k)$. But this implies that element a_{ij} cannot be determined directly, contradicting the assumption that (Π_C, Π_R) is consistent with a direct determination of S.

- There exists a path (r_i, a_j, r_l) for some $(r_i, a_j) \in F$ such that $\phi(a_j) = 0$ and $\phi(r_i) = \phi(r_l)$. Again this implies that element a_{ij} cannot be determined directly, a contradiction of our assumption.

- There exists a path (a_k, r_i, a_j, r_l) for some $(r_i, a_j) \in F$ such that $\phi(r_i) = \phi(r_l) \neq 0$ and $\phi(a_j) = \phi(a_k) \neq 0$. But this implies that element a_{ij} cannot be determined directly, contradicting the assumption.

□

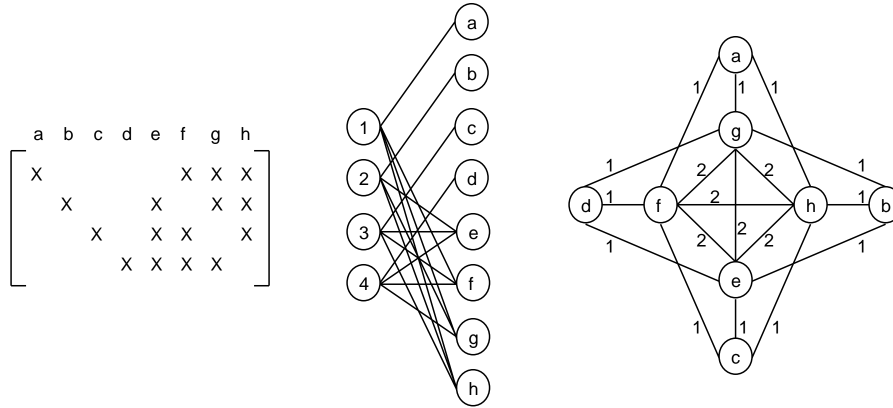

Figure 4: A matrix and its bipartite and intersection graph, resp.

5 Column Intersection vs. Bipartite Graph Formulation

In this section we compare our bipartite graph based formulations with formulations based on a column intersection graph.

Figure 4 depicts a matrix A, the corresponding bipartite graph $G_b(A) = (V_1, V_2, E)$, and the column intersection graph $G_c(A) = (V_2, E')$. Note that we have augmented the graph $G_c(A)$ with *edge weights*—$w(a_i, a_j)$ is the size of the intersection of the sets represented by vertices a_i and a_j. In terms of matrix A, $w(a_i, a_j)$ is the total number of rows where both columns a_i and a_j have nonzero entries. In the bipartite graph $G_b(A)$, it corresponds to the number of common neighbors of vertices a_i and a_j.

In Sections 5.1 to 5.4, we compare and contrast the two graph formulations in terms of flexibility, storage space requirement, ease of graph construction, and use of existing software.

5.1 Flexibility

Notice that the column intersection graph is a 'compressed' representation of the structure of the underlying matrix. Clearly, some information is lost in the compression process. In particular, given an edge in G_c between two columns, we cannot determine the row at which they share nonzero entries. By contrast, the bipartite graph is an equivalent representation of the structure of the matrix. This provides flexibility. As an illustration, notice that the bipartite graph can be used in a one-dimensional (row or column) partition as well as a two-dimensional (combined row and column) partition. The column intersection graph, on the other hand, is applicable only to a

column partition. Moreover, the graph formulations for partial matrix estimation problems would have required new types of 'intersection' graphs to be defined. In general, the advantage of the bipartite graph representation is that the representation is decoupled from the eventual technique to be employed and the matrix entries to be determined.

5.2 Storage Space Requirement

Although Lemma 3.18 correlates the bipartite graph of a matrix with its column intersection graph, one cannot immediately deduce that one graph is denser than the other. The density of the respective graph namely depends on the structure of the matrix. Here, we make a rough analysis to show that for sparse matrices of practical interest, the column intersection graph is likely to be denser than the bipartite counterpart.

Given a matrix A, the graph $G_b(A) = (V_1, V_2, E)$ and the weighted graph $G_c(A) = (V_2, E')$, for a vertex u in G_b, let $N_1(u) = \{v : (u, v) \in E\}$, $d_1(u) = |N_1(u)|$, and let the average degree-1 in the sets V_1 and V_2 of G_b be $\overline{\delta}_1(V_1)$ and $\overline{\delta}_1(V_2)$, respectively. Further, let $\overline{w}$ denote the *average edge weight* in G_c. Then,

$$\begin{aligned}
\sum_{e \in E'} w(e) &= \frac{1}{2} \cdot \sum_{u \in V_2} \sum_{v \in N_1(u)} (d_1(v) - 1) \\
|E'| \cdot \overline{w} &\approx \frac{1}{2} \cdot |V_2| \cdot \overline{\delta}_1(V_2) \cdot (\overline{\delta}_1(V_1) - 1) \\
&= \frac{1}{2} \cdot |E| \cdot (\overline{\delta}_1(V_1) - 1) \\
|E'| &= |E| \cdot (\frac{\overline{\delta}_1(V_1) - 1}{2\overline{w}})
\end{aligned}$$

Therefore, as long as $\frac{\overline{\delta}_1(V_1)-1}{2\overline{w}} > 1$, the column intersection graph is likely to have more edges (and hence requires more storage space) than the bipartite graph of the matrix.

5.3 Ease of Construction

The sparsity structure of matrix A directly, without any further computation, gives the corresponding bipartite graph $G_b(A)$. In principle, the data structure used to represent A can be used for implementing algorithms that use $G_b(A)$. By contrast, $G_c(A)$ has to be computed. As the following Lemma states, the time required for the computation of $G_c(A)$ is proportional to the number of edges in $G_c(A)$.

Lemma 5.1 *Given a graph $G_b(A) = (V_1, V_2, E)$, the time required for constructing $G_c(A)$ is $T_{const} = O(|V_2|(\overline{\delta}_1(V_2)(\overline{\delta}_1(V_1) - 1))$.*

It should however be noted that once the graph $G_c(A)$ is computed, a subsequent distance-1 coloring of $G_c(A)$ can be done faster than a distance-2 coloring of $G_b(A)$. As we shall show in Section 6, the overall time required for constructing and distance-1 coloring of $G_c(A)$ is of the same order as the time required for a direct distance-2 coloring of $G_b(A)$.

5.4 Use of Existing Software

Serial program packages that implement various practically effective distance-1 coloring heuristics exist [5, 6]. For matrix partitioning problems where a column intersection graph based formulation can be applied, these packages can be readily used. On the other hand, since distance-2 coloring is a prototypical model in our context, efficient programs, including parallel ones, for the distance-2 coloring problem need to be developed.

6 Distance-k Coloring Algorithms

In this section we present several fast algorithms for the various coloring problems considered in this paper. The algorithm design is greatly simplified by taking distance-2 coloring as a generic starting point.

For any fixed integer $k \geq 1$, DkGCP is NP-hard [21]. A proof sketch showing that the problem of finding a $(\frac{3}{2}, p)$-coloring with the minimum p is NP-hard, even if the graph is bipartite, is given in [8]. The problem of finding a distance-$\frac{3}{2}$ bicoloring with the fewest number of colors is also NP-hard [9]. Further, finding an acyclic p-coloring with the least value of p is NP-hard [4].

Since all the graph coloring problems of our concern are NP-hard, in practical applications, we are bound to rely on using *approximation algorithms* or *heuristics*. An algorithm $\mathcal{A}$ is said to be a γ-approximation algorithm for a minimization problem if its runtime is polynomial in the input size and if for every problem instance $\mathcal{I}$ with an optimal solution $OPT(\mathcal{I})$, the solution $\mathcal{A}(\mathcal{I})$ output by $\mathcal{A}$ is such that $\frac{\mathcal{A}(\mathcal{I})}{OPT(\mathcal{I})} \leq \gamma$. The *approximation ratio* $\gamma \geq 1$, and the goal is to make γ as close to unity as possible. If no such guarantee can be given for the quality of an approximate solution obtained by a polynomial time algorithm, the algorithm is usually referred to as a heuristic.

In the case of distance-1 coloring, there exist several, practically effective heuristics [7]. In this section we show that some of the ideas used in the

distance-1 coloring heuristics can be adapted to the cases considered in this paper by extending the notion of *neighborhood*. The algorithms we present are *greedy* in nature, i.e., the vertices of a graph are processed in some order and at each step a decision that looks best at the moment (and that will not be reversed later) is made.

In Section 6.1, we present a generic greedy distance-2 coloring algorithm and give a detailed analysis of its performance both in terms of computation time and number of colors used. In Sections 6.2 to 6.5, adaptations of this algorithm, tailored to the various coloring problems of our concern are presented.

Notations Some notations used in the rest of this section are first in order. Recall that for a vertex u in a graph G, a vertex $w \neq u$ is a *distance-k neighbor* of u if the shortest path connecting u and w has length $\leq k$. Let $N_k(u) = \{w : w \text{ is a distance-}k \text{ neighbor of } u\}$. Let $d_k(u) = |N_k(u)|$ denote the degree-k of u; Δ be the maximum degree-1 in G; and $\overline{\delta}_k = \frac{1}{|V|}\sum_{u \in V} d_k(u)$ denote the average degree-k in G.

Further, in a bipartite graph $G_b = (V_1, V_2, E)$, let the maximum degree-1 in the vertex sets V_1 and V_2 be denoted by $\Delta(V_1)$ and $\Delta(V_2)$, respectively. Similarly, let the average degree-k in the sets V_1 and V_2 be denoted by $\overline{\delta}_k(V_1)$ and $\overline{\delta}_k(V_2)$, respectively.

6.1 Distance-2 Coloring Algorithms

A simple approach for an approximate distance-2 coloring of a graph $G = (V, E)$ is to visit the vertices in some order, each time assigning a vertex the smallest color that is not used by any of its distance-2 neighbors. Note that the degree-2 of a vertex u in G is bounded by Δ^2, i.e., $d_2(u) \leq \sum_{w \in N_1(u)} d_1(w) \leq \Delta \cdot d_1(u) \leq \Delta^2$. Thus, since the vertices in G can always be distance-2 colored trivially using $|V|$ different colors, it is always possible to color a vertex using a value from the set $\{1, 2, \ldots, min\{\Delta^2 + 1, |V|\}\}$. Algorithm GreedyD2Coloring, outlined below, uses this as it colors the vertices of the graph in an arbitrary order. In the algorithm, color(v) is the color assigned to vertex v and forbiddenColors is a vector of size $C_{max} = \min\{\Delta^2 + 1, |V|\}$ used to mark the colors that cannot be assigned to a particular vertex. Specifically, forbiddenColors(c) $= v$ indicates that color c cannot be assigned to vertex v.

Lemma 6.1 *GreedyD2Coloring finds a distance-2 coloring in time* $O(|V|\overline{\delta}_2)$.

```
GreedyD2Coloring(G = (V, E))

for each v_i ∈ V do
    for each colored vertex u ∈ N_2(v_i) do
        forbiddenColors(color(u)) = v_i
    end-for
    color(v_i) = min{c : forbiddenColors(c) ≠ v_i}
end-for
```

Proof: We first show correctness. In step i of the algorithm, the color used by each of the distance-2 neighbors of vertex v_i is marked (using v_i) in the vector forbiddenColors. Thus, at the end of the inner for loop, the set of colors that are allowed for vertex v_i is the set of indices in forbiddenColors where the mark used is different from v_i. The minimum value in this set is thus the smallest allowable color for vertex v_i. Notice that the vector forbiddenColors does not need to be initialized at every step as the marker v_i is used *only* in step i.

Turning to complexity, note that marking the forbidden colors at step i of the algorithm takes $O(d_2(v_i))$ time. Finding the smallest allowable color to v_i can be done within the same order of time by scanning forbiddenColors sequentially until the first index c where a value other than v_i is stored is found. The total time is thus proportional to $\sum_{v \in V} d_2(v) = O(|V|\overline{\delta}_2)$. □

We now analyze the quality of the solution provided by GreedyD2Coloring. Let the number of colors used by GreedyD2Coloring on a graph $G = (V, E)$ be $\chi_2{}^{greedy}(G)$. Then recalling the lower bound given in Lemma 3.19, we get the following theorem and its corollary.

Theorem 6.2 $\Delta + 1 \leq \chi_2(G) \leq \chi_2{}^{greedy}(G) \leq min\{\Delta^2 + 1, |V|\}$.

Corollary 6.3 *GreedyD2Coloring is an $O(\sqrt{|V|})$-approximation algorithm.*

Proof: The approximation ratio γ is at most $\frac{1}{\Delta+1} \cdot min\{\Delta^2 + 1, |V|\}$. There are two possibilities to consider. In the first case $\Delta^2 + 1 < |V|$. This implies $\Delta = O(\sqrt{|V|})$ and $\gamma = \frac{\Delta^2+1}{\Delta+1} = O(\Delta) = O(\sqrt{|V|})$. In the second case $|V| < \Delta^2 + 1$. This implies $\Delta = \Omega(\sqrt{|V|})$ and $\gamma = \frac{|V|}{\Delta+1} = O(\sqrt{|V|})$. □

Note that for practical problems, such as problems that arise in solving PDEs using good finite element discretizations, $\Delta^2 + 1 \ll |V|$, making GreedyD2Coloring an $O(\Delta)$-approximation algorithm.

The actual number of colors used by GreedyD2Coloring depends on the order in which the vertices are visited. In GreedyD2Coloring, an arbitrary

ordering is assumed. A solution with fewer number of colors can be expected if a more elaborate ordering criterion is used. For example, the ideas in *largest degree first* and *incidence degree ordering* for distance-1 coloring [7] can be adapted to the distance-2 coloring case.

As a final remark on the complexity of GreedyD2Coloring, we show that the algorithm runs in linear time in the number of vertices for certain sparse graphs. Let $\delta_2 = \frac{1}{|V|} \sum_{u \in V} d_1(u)^2$ and let the *standard deviation* of degree-1 in $G = (V, E)$ be given by

$$\sigma^2 = \frac{1}{|V|} \sum_{v \in V} (d_1(v) - \overline{\delta}_1)^2.$$

Then,

$$\begin{aligned} |V|\sigma^2 &= \sum_{v \in V} d_1(v)^2 + \sum_{v \in V} \overline{\delta}_1^{\,2} - 2 \sum_{v \in V} d_1(v)\overline{\delta}_1 \\ &= |V|\delta_2 + |V|\overline{\delta}_1^{\,2} - 2\overline{\delta}_1 \sum_{v \in V} d_1(v) \\ &= |V|\delta_2 + |V|\overline{\delta}_1^{\,2} - 2|V|\overline{\delta}_1^{\,2} \\ &= |V|\delta_2 - |V|\overline{\delta}_1^{\,2} \end{aligned}$$

Rewriting we get,

$$\delta_2 = \overline{\delta}_1^{\,2} + \sigma^2.$$

Noting that $\delta_2 \geq \overline{\delta}_2$, we get the following corollary to Lemma 6.1.

Corollary 6.4 *GreedyD2Coloring has time complexity* $O(|V|(\overline{\delta}_1^{\,2} + \sigma^2))$.

Since $\overline{\delta}_1 = \frac{2|E|}{|V|}$, the complexity expression in Corollary 6.4 reduces to $O(\frac{|E|^2}{|V|})$ for graphs where $\sigma < \overline{\delta}_1$. In particular, for sparse graphs, where $|E| = O(|V|)$, the time complexity of GreedyD2Coloring becomes $O(|V|)$.

6.2 Partial Distance-2 Coloring Algorithms

Here we modify GreedyD2Coloring slightly to make it suitable for solving the partial distance-2 coloring problem GCP1 (our graph formulation of MPP1).

For any vertex $v \in V_2$, the number of vertices at distance *exactly* two units from v is at most $\Delta(V_2)(\Delta(V_1) - 1)$. Thus, vertex v can always be assigned a color from the set $\{1, 2, \ldots, C_{max}\}$, where $C_{max} = min\{\Delta(V_2)(\Delta(V_1) - 1)) + 1, |V_2|\}$. In Algorithm GreedyPartialD2Coloring, given below, the vector forbiddenColors is of size C_{max}.

The following result is straightforward.

GreedyPartialD2Coloring($G_b = (V_1, V_2, E)$)

```
for each v ∈ V2 do
    for each u ∈ N1(v) do
        for each colored vertex w ∈ N1(u) do
            forbiddenColors(color(w)) = v
        end-for
    end-for
    color(v) = min{c : forbiddenColors(c) ≠ v}
end-for
```

Lemma 6.5 *GreedyPartialD2Coloring has time complexity* $O(|V_2|\overline{\delta}_1(V_2)(\overline{\delta}_1(V_1) - 1))$.

As stated earlier, a distance-1 coloring formulation for MPP1 was provided in [7] using the column intersection graph. From Lemmas 5.1 and 6.5 and noting that greedy distance-1 graph coloring is linear in the number of edges, it follows that the time required for the construction of the column intersection graph plus the computation of a (greedy) distance-1 coloring is asymptotically the same as the time required for the direct computation of a (greedy) partial distance-2 coloring. This means the two formulations are (asymptotically) comparable in terms of overall computation time.

6.3 Distance-$\frac{3}{2}$ Coloring Algorithms

Recall that a distance-$\frac{3}{2}$ coloring, which was used to model MPP2, is a distance-1 coloring where every path of length three uses at least three colors. We propose two algorithms for this problem. In finding a valid color to assign a vertex, the first algorithm visits the distance-3 neighbors of the vertex while the second algorithm visits only the distance-2 neighbors. In both algorithms the vector forbiddenColors is of size $C_{max} = min\{\Delta^2 + 1, |V|\}$. GreedyD$\frac{3}{2}$ColoringAlg1 outlines the first algorithm.

Figure 5 graphically shows the decision made during one of the $|V|$ steps of GreedyD$\frac{3}{2}$ColoringAlg1. The root of the tree corresponds to the vertex v to be colored at the current step. The neighbors of v that are one, two, and three edges away are represented by the nodes at level u, w, and x, respectively. Each tree node corresponds to many vertices of the input graph. A darkly shaded node signifies that the vertex is already colored. The forbidden colors are marked by an f and '?' indicates that whether the color is forbidden or not depends on the color used at level x. The correspondence between the figure and Lines 1, 2 and 3 of the algorithm is

GreedyD$\frac{3}{2}$ColoringAlg1$(G = (V, E))$

```
for each v ∈ V do
    for each u ∈ N_1(v) do
        if u is colored                                    (1)
            forbiddenColors(color(u)) = v
        for each colored vertex w ∈ N_1(u) do
            if u is not colored                            (2)
                forbiddenColors(color(w)) = v
            else
                for each colored vertex x ∈ N_1(w), x ≠ u do
                    if (color(x) == color (u))             (3)
                        forbiddenColors(color(w)) = v
                        break
                    end-if
                end-for
            end-if
        end-for
    end-for
    color(v) = min{c : forbiddenColors(c) ≠ v}
end-for
```

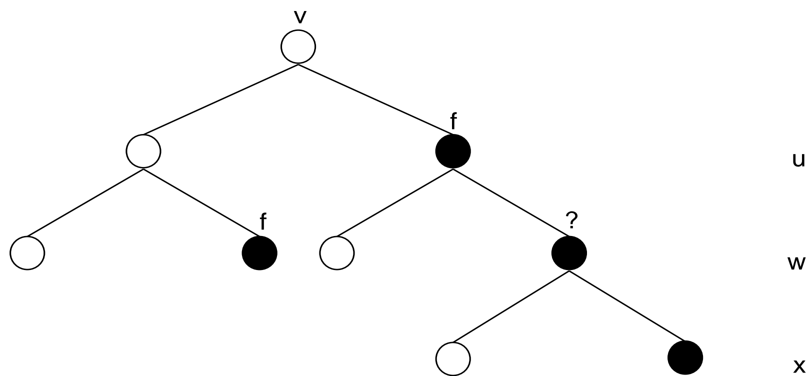

Figure 5: Visualizing a step in GreedyD$\frac{3}{2}$ColoringAlg1

obvious.

Notice that in Line 2 of the algorithm, the color of the vertex w in the path (v, u, w) where u is not yet colored is forbidden for vertex v. Later on, when vertex u is colored, the test in Line 1 ensures that u gets a color different from both v and w, making the path use three different colors. Had the requirement in Line 2 not been imposed, a situation in which a path (v, u, w, x) is two-colored could arise. Thus from its construction, the output of GreedyD$\frac{3}{2}$ColoringAlg1 is a valid distance-$\frac{3}{2}$ coloring. The amount of work done in each step of the algorithm is proportional to $d_3(v)$. Thus we get the following result.

Lemma 6.6 *GreedyD$\frac{3}{2}$ColoringAlg1 finds a distance-$\frac{3}{2}$ coloring in time $O(|V|\overline{\delta}_3)$.*

We shall now present the second distance-$\frac{3}{2}$ coloring algorithm in which the coloring at each step is obtained by considering only the distance-2 neighborhood (in contrast to distance-3 neighborhood of the previous case). The idea behind the algorithm (formulated in terms of matrices) was first suggested by Powell and Toint [26].

Recall that distance-$\frac{3}{2}$ coloring is a relaxed distance-2 coloring. As an illustration, suppose v, u, w, x is a path in a graph. A coloring ϕ in which $\phi(v) = \phi(w) = 2$, $\phi(u) = 1$ and $\phi(x) = 3$ is a valid distance-$\frac{3}{2}$ (but not distance-2) coloring on this path.

One way of relaxing the requirement for distance-2 coloring so as to obtain a distance-$\frac{3}{2}$ coloring is to let two vertices at distance of exactly two units from each other share a color as long as the vertex in between them has a color of lower value. More precisely, let v, u, w be a path in G and suppose v and u are colored and we want to determine the color of w. We allow $\phi(w)$ to be equal to $\phi(v)$ as long as $\phi(u) < \phi(v)$. To see that this coloring can always be extended to yield a valid distance-$\frac{3}{2}$ coloring, consider the path v, u, w, x, an extension of path v, u, w in one direction. Now, since $\phi(w) = \phi(v) > \phi(u)$, we cannot let $\phi(x)$ be equal to $\phi(u)$. Obviously, $\phi(x)$ should be different from $\phi(w)$, otherwise it will not be a valid distance-1 coloring. Thus the path v, u, w, x uses three colors, ϕ is a distance-1 coloring and therefore it is a valid distance-$\frac{3}{2}$ coloring. The algorithm that makes use of this idea is given in GreedyD$\frac{3}{2}$ColoringAlg2.

Clearly, the runtime of GreedyD$\frac{3}{2}$ColoringAlg2 is $O(|V|\overline{\delta}_2)$. Notice, however, that GreedyD$\frac{3}{2}$ColoringAlg1 may use smaller number of colors than GreedyD$\frac{3}{2}$ColoringAlg2. For instance, Figure 6 shows an example where the first algorithm uses three colors in coloring the vertices in their alphabetical order while the second one uses four in doing the same.

GreedyD$\frac{3}{2}$ColoringAlg2($G = (V, E)$)

```
for each v ∈ V do
    for each u ∈ N_1(v) do
        if u is colored
            forbiddenColors(color(u)) = v
        for each colored vertex w ∈ N_1(u) do
            if u is not colored
                forbiddenColors(color(w)) = v
            else
                if (color(w) < color(u))
                    forbiddenColors(color(w)) = v
            end-if
        end-for
    end-for
    color(v) = min{c : forbiddenColors(c) ≠ v}
end-for
```

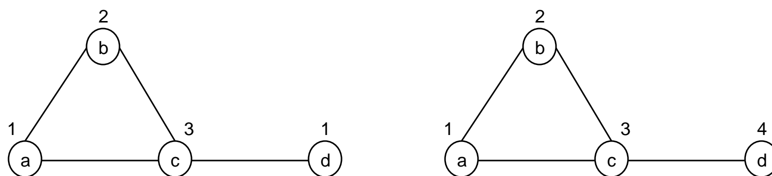

Figure 6: GreedyD$\frac{3}{2}$ColoringAlg1 vs. Alg2

6.4 Acyclic Coloring Algorithms

Here we present an effective algorithm called GreedyAcyclicColoring to find an approximate solution for the acyclic coloring problem (GCP3).

The basic idea in our algorithm is to detect and 'break' a two-colored cycle while an otherwise distance-1 coloring of the graph proceeds. Specifically, the algorithm colors the vertices of a graph $G = (V, E)$ while making a Depth First Search (DFS) traversal. Recall that a *back-edge* in the *DFS-tree* of an undirected graph defines a unique cycle. For more information on DFS, refer to the book [11].

In GreedyAcyclicColoring, the *DFS-tree* $T(G)$ of the graph G is implicitly constructed as the algorithm proceeds. Let $\phi(v)$ denote the color of vertex v and $s(v)$ denote the order in which v is first visited in the DFS traversal of G ($1 \leq s(v) \leq |V|$). The root r of $T(G)$ has $s(r) = 1$. Further, let $p(v)$ be a pointer to the *parent* of v in $T(G)$, and $l(v)$ be a pointer to the lowest ancestor of v in $T(G)$ such that $\phi(l(v)) \neq \phi(v)$ and $\phi(l(v)) \neq \phi(p(v))$.

The latter pointer will be used in the detection of a two-colored cycle. In particular, the algorithm proceeds in such a way that the path in $T(G)$ from v up to, but not including, $l(v)$ is two-colored. At the beginning of the algorithm, for every vertex u, $l(u)$ is set to point to null.

Consider the step in GreedyAcyclicColoring where vertex v is first visited. To start with, v is assigned the smallest color different from all of its distance-1 neighbors in G, including its parent $p(v)$ in $T(G)$. If there exists a back-edge $b = (v, w)$ in the current $T(G)$ such that $s(l(p(v))) < s(w)$ and $\phi(v) = \phi(p(p(v)))$, then this implies that the cycle corresponding to b is two-colored (see Figure 7 which shows a partial view of the DFS-tree at the step where vertex v is to be colored). To break the cycle, v is assigned a new color—the smallest color different from $\phi(p(v))$ and $\phi(p(p(v)))$—and $l(v)$ is set to point to $p(p(v))$. Otherwise, if no such back-edge exists, the color of v is declared final and $l(v)$ is updated in the following manner. If $\phi(v) \neq \phi(p(p(v)))$, then $l(v) = p(p(v))$; otherwise $l(v) = l(p(v))$.

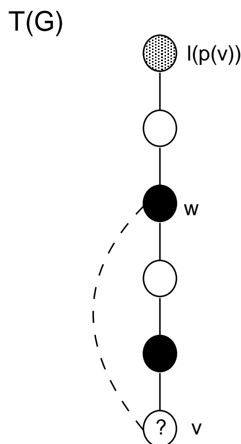

Figure 7: Visualizing a step in GreedyAcyclicColoring

Notice that the amount of work done in each DFS visit of a vertex v in the graph is proportional to the degree-1 of v. This makes GreedyAcyclicColoring an $O(|E|)$-time algorithm.

We note that Coleman and Cai [4] have proposed an algorithm for the acyclic coloring problem. The idea in their algorithm is to first transform a given graph $G = (V, E)$ to a 'completed' graph $G' = (V, E')$ such that a distance-1 coloring of G' is equivalent to an acyclic coloring of G, and then use a known distance-1 coloring heuristic on G'. The construction of G' is done in the following way. Start by setting $E' = E$; visit the vertices in V in a *predefined order*; at each step i, if vertex v_i is adjacent to vertices v_j and

v_k such that both v_j and v_k are ordered before v_i then add the edge (v_j, v_k) to E'.

Our approach differs from that of Coleman and Cai in at least two ways. First, the graph G' used in the latter approach may require substantially more storage space than the original graph G used in our approach. Second, an edge in $E' \setminus E$ in the latter approach may actually be redundant. For example, a distance-1 coloring of an odd-length cycle in G uses at least three colors and hence is a valid acyclic coloring whereas the Coleman and Cai approach adds one redundant edge to the cycle.

6.5 Bicoloring Algorithms

Here we consider problems GCP4 and GCP5, introduced in Sections 3.4 and 3.5, respectively.

Recall that in a distance-$\frac{3}{2}$ p-bicoloring, some of the vertices are assigned the neutral color 0. We make the following crucial observation which helps us identify a possible set of such vertices. The observation is a direct consequence of Condition 2 of Definition 3.8.

Observation 6.7 *Let $G_b = (V_1, V_2, E)$ be a bipartite graph and $\phi : [V_1, V_2] \to \{0, 1, \ldots, p\}$ be a distance-$\frac{3}{2}$ bicoloring of G_b. Then,*

- *the set $C = \{v : \phi(v) \neq 0\}$ is a **vertex cover** in G_b, and*
- *the set $I = \{v : \phi(v) = 0\}$ is an **independent set** in G_b.*

One consequence of Observation 6.7 is that $|I| + |C| = |V_1| + |V_2|$. Thus, minimizing the cardinality of the vertex cover C corresponds to maximizing the cardinality of the independent set I.

6.5.1 An algorithm for GCP4

In light of Observation 6.7, we suggest GCP4Algorithm as a scheme for solving the coloring problem GCP4.

GCP4Algorithm$(G_b = (V_1, V_2, E))$

1. Find a vertex cover C in G_b.
2. Assign the vertices in the set $I = (V_1 \cup V_2) \setminus C$ the color 0.
3. Color the vertices in C such that the result is a distance-3/2 bicoloring of G_b.

In Step 1 of GCP4Algorithm, any vertex cover can be used. However, the choice of the vertex cover affects the subsequent coloring in Step 3, both in terms of number of colors used and coloring time spent. To reduce the coloring time in Step 3, the size of the vertex cover should be minimized. Furthermore, minimizing the potential number of colors to be used imposes an additional requirement: the vertex cover should include those vertices from V_1 and V_2 with relatively high number of distance-1 neighbors. (Recall the introductory discussion in Section 2.2 used to motivate the need for a two-dimensional partition: matrices with a few dense rows and columns benefit from a two-dimensional partition.)

In a bipartite graph, a minimum cardinality vertex cover can be obtained via finding a *maximum matching* in polynomial time [22, 28]. In fact, it can be computed practically in effectively linear time in the number of edges [24].

Once Steps 1 and 2 are carried out, Step 3 can be done by a suitable adaptation of GreedyD$\frac{3}{2}$ColoringAlg1. GreedyD$\frac{3}{2}$BiColoring, given here only pictorially, is such an adaptation. One of the differences between the coloring and bicoloring algorithms is that in the latter case, two disjoint set of colors are used in coloring the vertices in V_1 and V_2 of the bipartite graph $G_b = (V_1, V_2, E)$. Another difference is that at a step of the bicoloring algorithm where v is colored, a vertex within the distance-3 neighborhood of v may be one of *three* types: it is colored with a positive value, it is colored with 0, or it is not yet colored. The choice of color for v thus needs to consider these three options.

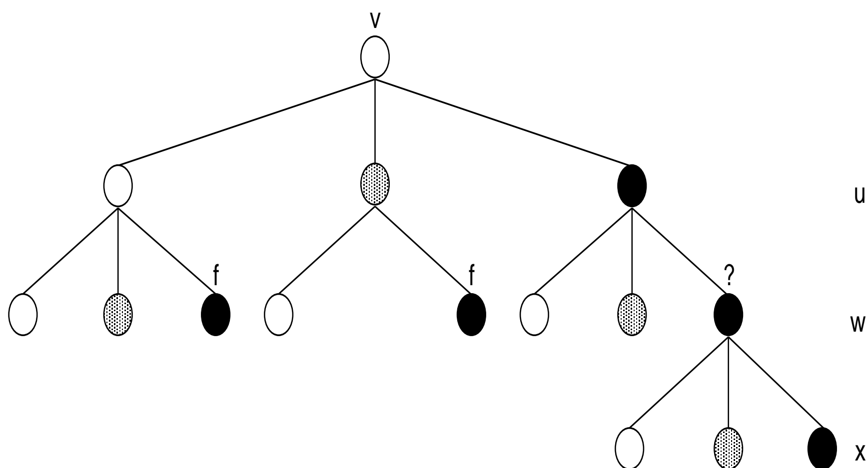

Figure 8: Visualizing a step in GreedyD$\frac{3}{2}$BiColoring

Figure 8 shows a visual presentation of the ith step of GreedyD$\frac{3}{2}$BiColoring. Note the similarity with Figure 5. Notice also that, in choosing a color for vertex v in $G_b = (V_1, V_2, E)$, since the colors for vertices in V_1 and V_2 are drawn from two disjoint sets, we need only consider colors of vertices two edges away from v. In the figure, dark shaded nodes correspond to colored vertices, light shaded nodes show vertices with color zero, and unshaded nodes correspond to uncolored vertices. Observe that the node with color 0 at level u has only two children. The colors of the vertices in the nodes marked by an f indicate forbidden colors and whether the color at the node marked by '?' is forbidden or not depends on the color used at node x: if $\phi(u) = \phi(x)$, $\phi(w)$ is forbidden, otherwise, it is not.

The time complexity of GreedyD$\frac{3}{2}$BiColoring is $O((|V_1| + |V_2|)\overline{\delta}_3)$, which is also the overall time complexity of GCP4Algorithm assuming that step 1 is done using a greedy algorithm that is linear in the number of edges.

Notice that a partial distance-2 coloring of G_b on V_2 is a just special case of the scheme GCP4Algorithm. To see this, consider the trivial choice of vertex cover $C = V_2$ in Step 1. This implies that, in Step 2, the vertices in the set $I = V_1$ will be colored with zero. By Condition 3 of Definition 3.8, vertices adjacent to a vertex colored with zero are required to be assigned different colors. Thus, the result is effectively a partial distance-2 coloring of G_b on V_2.

We note that Hossain and Steihaug [17] and Coleman and Verma [9] have each proposed an algorithm for GCP4. These algorithms can be interpreted in light of GCP4Algorithm. The algorithm of Hossain and Steihaug (HS) *implicitly* finds a vertex cover while the coloring of the graph proceeds. Using our terminology, the vertices that remain uncolored at the end of the HS-algorithm form an independent set in the graph and can thus be assigned the neutral color 0.

The algorithm of Coleman and Verma uses a preprocessing step to identify the rows and columns of the underlying matrix that eventually need to be colored with positive values. The preprocessing step uses a non-straightforward matrix-based procedure. It appears that the procedure effectively produces a small sized vertex cover (however, this is not clearly stated in the paper). After the preprocessing step, a certain 'column intersection' graph, adapted to the distance-$\frac{3}{2}$ bicoloring requirements, is constructed to finally use known distance-1 coloring heuristics on the resulting graph.

6.5.2 An algorithm for GCP5

Similarly, by virtue of Observation 6.7, the approach we suggest for solving the acyclic bicoloring problem is given in GCP5Algorithm.

GCP5Algorithm($G_b = (V_1, V_2, E)$)

1. Find a vertex cover C in G_b.
2. Assign the vertices in the set $I = (V_1 \cup V_2) \setminus C$ the color 0.
3. Color the vertices in C such that the result is an acyclic bicoloring of G_b.

7 Conclusion

We have studied the efficient estimation of sparse Jacobian and Hessian matrices using FD and AD techniques. We considered methods that rely on a one-dimensional as well as a two-dimensional partition to be used in an evaluation based on a direct or a substitution scheme. We introduced partial matrix estimation problems in distinction from full matrix estimation problems. In doing so, we developed a unified graph theoretic framework to cope with a variety of complex matrix partitioning problems.

At the basis of our graph problem formulations lies a robust graph representation of the sparsity structure of a matrix: a nonsymmetric matrix is represented by its bipartite graph and a symmetric matrix by its adjacency graph.

We showed that the distance-2 graph coloring problem is a generic model for the various one-dimensional matrix partitioning problems.

Our unified graph theoretic approach enabled us to provide some fresh insight into the matrix problems and as a result we developed several simple and effective algorithms. Our emphasis has been on greedy algorithms. Other algorithmic techniques need to be explored in the future. For example, it could be interesting to find a distance-2 coloring algorithm that uses asymptotically the same time as the greedy algorithm discussed in this paper and balances the number of vertices in each color class. Finding a random color, rather than the smallest color, from an allowable set could be an idea to consider in this regard.

In the case of two-dimensional partition problems, based on the known relationship to graph bicoloring, we argued that finding a 'small'-size vertex cover as a preprocessing step contributes to making the overall computation

more efficient. A more precise characterization of the 'optimum' vertex cover required is a worthwhile issue.

We have not developed any special algorithms for the restricted coloring problems arising in partial matrix estimation. The ideas used in our algorithms for the coloring problems in full matrix estimation can be adapted to the restricted cases by observing the particular coloring conditions.

In general, most of the algorithms in the literature for solving the coloring problems considered in this paper rely on first transforming the input graph $G = (V, E)$ to some denser graph $G' = (V, E')$, $E' \supseteq E$, such that a distance-1 coloring of G' is equivalent to the particular coloring problem on G. In contrast, the algorithms proposed in this paper solve the particular coloring problem directly on G. As has been argued, the main advantages offered by our approach are the possibility to mix-and-match methods, less storage space requirement, and ease of developing flexible software.

One of the motivations for the current study has been the need for the development of parallel algorithms for solving partitioning problems in large-scale PDE-constrained optimization contexts. In a recent work [13], we have shown some parallel algorithms (using the shared-memory programming model) for the distance-2 and distance-$\frac{3}{2}$ coloring problems. Our results, theoretical as well as experimental, were promising. We believe that this study lays a foundation for further work on the development and implementation of not only shared-memory but also distributed-memory parallel algorithms.

Acknowledgement We thank Trond Steihaug for interesting discussions.

References

[1] G. Agnarsson, R. Greenlaw, and M. M. Halldórsson. On powers of chordal graphs and their colorings. *Congress Numerantium*, 100:41–65, 2000.

[2] G. Agnarsson and M. M. Halldórsson. Coloring powers of planar graphs. In *11th. Ann. ACM-SIAM symp. on Discrete Algorithms*, pages 654–662, 2000.

[3] O. Axelsson and U. Nävert. On a graphical package for nonlinear partial differential equation problems. In B. Gilchrist, editor, *Proceedings of IFIP Congress 77*, Information Processing, pages 103–108. North-Holland, 1977.

[4] T. F. Coleman and J. Cai. The cyclic coloring problem and estimation of sparse Hessian matrices. *SIAM J. Alg. Disc. Meth.*, 7(2):221–235, April 1986.

[5] T. F. Coleman, B. Garbow, and J. J. Moré. Software for estimating sparse Jacobian matrices. *ACM Trans. Mathematical Software*, 10:329–347, 1984.

[6] T. F. Coleman, B. Garbow, and J. J. Moré. Software for estimating sparse Hessian matrices. *ACM Trans. Mathematical Software*, 11:363–377, 1985.

[7] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, 20(1):187–209, February 1983.

[8] T. F. Coleman and J. J. Moré. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Program.*, 28:243–270, 1984.

[9] T. F. Coleman and A. Verma. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.*, 19(4):1210–1233, July 1998.

[10] G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann. *Automatic Differentiation of Algorithms: From Simulation to Optimization*. Springer, New York, 2002.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Massachusetts Institute of Technology, 1990.

[12] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.

[13] A.H. Gebremedhin, F. Manne, and A. Pothen. Parallel distance-k coloring algorithms for numerical optimization. In B. Monien and R. Feldmann, editors, *Euro-Par 2002 Parallel Processing*, volume 2400 of *LNCS*, pages 912–921. Springer Verlag, 2002.

[14] A. Griewank and G. F. Corliss. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* SIAM, Philadelphia, 1991.

[15] B. Grünbaum. Acyclic colorings of planar graphs. *Israel J. Math.*, 14:390–408, 1973.

[16] S. Hossain. On the computation of sparse Jacobian matrices and Newton steps. Technical Report 146, Department of Informatics, University of Bergen, Norway, March 1998.

[17] S. Hossain and T. Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, 10:33–48, 1998.

[18] S. Hossain and T. Steihaug. Reducing the number of AD passes for computing a sparse Jacobian matrix. In G. Corliss, C. Faure, A. Griewank, L. Hascoét, and U. Naumann, editors, *Authomatic Differentiation of Algorithms: From Simulation to Optimization*, pages 263 – 270. Springer, 2002.

[19] D. A. Knoll and D. E. Keyes. Jacobian-free Newton-Krylov methods: A survey of appoaches and applications. Available at http://www.math.odu.edu/~ keyes/.

[20] S. O. Krumke, M. V. Marathe, and S.S. Ravi. Approximation algorithms for channel assignment in radio networks. In *Dial M for Mobility*, Dallas, Texas, October 1998.

[21] Y. Lin and S. S. Skiena. Algorithms for square roots of graphs. *SIAM J. Disc. Math.*, 8:99–118, 1995.

[22] L. Lovász and M.D. Plummer. *Matching Theory*. North-Holland, Amsterdam, 1986.

[23] S. T. McCormick. Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem. *Math. Program.*, 26:153–171, 1983.

[24] R. Motwani. Average-case analysis of algorithms for matchings and related problems. *Journal of the ACM*, 41:1329–1356, 1994.

[25] G. N. Newsam and J. D. Ramsdell. Estimation of sparse Jacobian matrices. *SIAM J. Alg. Disc. Meth.*, 4:404–418, 1983.

[26] M. J. D. Powell and P. L. Toint. On the estimation of sparse Hessian matrices. *SIAM J. Numer. Anal.*, 16(6):1060–1074, December 1979.

[27] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[28] L. A. Wolsey. *Integer Programming*. Wiley-Interscience Publication, New York, USA, 1998.

Parallel Distance-k Coloring Algorithms for Numerical Optimization

Assefaw Hadish Gebremedhin* Fredrik Manne Alex Pothen‡

Abstract

Matrix partitioning problems that arise in the efficient estimation of sparse Jacobians and Hessians can be modeled using variants of graph coloring problems. In a previous work [7], we argue that *distance-2* and *distance-*$\frac{3}{2}$ graph coloring are robust and flexible formulations of the respective matrix estimation problems. The problem size in large-scale optimization contexts makes the matrix estimation phase an expensive part of the entire computation both in terms of execution time and memory space. Hence, there is a need for both shared- and distributed-memory parallel algorithms for the stated graph coloring problems. In the current work, we present the first practical shared address space parallel algorithms for these problems. The main idea in our algorithms is to randomly *partition* the vertex set equally among the available processors, let each processor *speculatively* color its vertices using information about already colored vertices, detect eventual conflicts in parallel, and finally re-color conflicting vertices sequentially. *Randomization* is also used in the coloring phases to further reduce conflicts. Our PRAM-analysis shows that the algorithms should give almost linear speedup for sparse graphs that are large relative to the number of processors. Experimental results from our OpenMP implementations on a Cray Origin2000 using various large graphs show that the algorithms indeed yield reasonable speedup for modest numbers of processors.

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. {assefaw, fredrikm}@ii.uib.no

†Computer Science Department, Old Dominion University, Norfolk, VA 23529 USA; CSRI, Sandia National Labs, Albuquerque NM 87185 USA; and ICASE, NASA Langley Research Center, Hampton, VA 23681-2199 USA. pothen@cs.odu.edu

‡This author's research was supported by NSF grant DMS-9807172, DOE ASCI level-2 subcontract B347882 from Lawrence Livermore National Lab; and by DOE SCIDAC grant DE-FC02-01ER25476.

1 Introduction

Numerical optimization algorithms that rely on derivative information often need to compute the Jacobian or Hessian matrix. Since this is an expensive part of the computation, efficient methods for estimating these matrices via finite differences (FD) or automatic differentiation (AD) are needed. It is known that the problem of minimizing the number of function evaluations (or AD passes) required in the computation of these matrices can be formulated as variants of graph coloring problems [1, 2, 3, 9, 13]. The particular coloring problem differs with the optimization context: whether the Jacobian or the Hessian matrix is to be computed; whether a direct or a substitution method is employed; and whether only columns, or only rows, or both columns and rows are to be used to evaluate the matrix elements. In addition, the type of coloring problem depends on the kind of graph used to represent the underlying matrix. In [7], we provide an integrated review of previous works in this area and identify the *distance-2* (D2) graph coloring problem as a unifying, generic, and robust formulation. The D2-coloring problem has also noteworthy applications in other fields such as channel assignment [11] and facility location problems [14].

Large-scale PDE-constrained optimization problems can be solved only with the memory and time resources available on parallel computers. In these problems, the variables defined on a computational mesh are already distributed on the processors, and hence parallel coloring algorithms are needed for computing, for instance, the Jacobian. It turns out that the problems of efficiently computing the Jacobian and Hessian can be formulated as the D2- and D$\frac{3}{2}$-coloring problems, respectively. The latter coloring problem is a relaxed variant of the former, and will be described in Section 2.2. In these formulations, the bipartite graph associated with the rows and columns of the matrix is used for the Jacobian; in the case of the Hessian matrix, the adjacency graph corresponding to the symmetric matrix is used.

In this paper, we present several new deterministic as well as probabilistic parallel algorithms for the D2- and D$\frac{3}{2}$-coloring problems. Our algorithms are practical and effective, well suited for shared address space programming, and have been implemented in C using OpenMP primitives. We report results from experiments conducted on a Cray Origin 2000 using large graphs that arise in finite element methods and in eigenvalue computations.

In the sequel, we introduce the graph problems in Section 2, present the algorithms in Section 3, discuss our experimental results in Section 4, and conclude the paper in Section 5.

2 Background

2.1 Matrix Partition Problems

An essential component of the efficient estimation of a sparse Jacobian or Hessian using FD or AD is the problem of finding a suitable *partition* of the columns and/or the rows of the matrix. The particular partition chosen defines a system of equations from which the matrix entries are determined. A method that utilizes a diagonal system is called a *direct* method, and one that uses a triangular system is called a *substitution* method. A direct method is more restrictive but the computation of matrix entries is straightforward and numerically stable. A substitution method, on the other hand, is less restrictive but it may be subject to approximation difficulties and numerical instability. Moreover, direct methods offer more parallelism than substitution methods. In this paper, we focus on direct methods that use column partitioning.

A partition of the columns of a nonsymmetric matrix A is said to be *consistent* with the direct determination of A if whenever a_{ij} is a non-zero element of A then the group containing column j has no other column with a non-zero in row i [1]. Similarly, a partition of the columns of a symmetric matrix A is called *symmetrically consistent* with the direct determination of A if whenever a_{ij} is a non-zero element of A then *either* (i) the group containing column j has no other column with a non-zero in row i, or (ii) the group containing column i has no other column with a non-zero in row j [2]. From a given (symmetrically) consistent partition $\{C_1, C_2, \ldots, C_\rho\}$ of the columns of A, the nonzero entries can be determined with ρ function evaluations (matrix-vector products).

Thus we have the following two problems of interest. Given the sparsity structure of a nonsymmetric $m \times n$ matrix A, find a consistent partition of the columns of A with the fewest number of groups. We refer to this problem as NONSYMCOLPART. The second problem of our interest, which we call SYMCOLPART, states: given the sparsity structure of a symmetric $n \times n$ matrix A, find a symmetrically consistent partition of the columns of A with the fewest number of groups.

2.2 Graph Problems

In a graph, two distinct vertices are said to be *distance-k neighbors* if the shortest path connecting them consists of *at most* k edges. The number of distance-k neighbors of a vertex u is referred to as the *degree-k* of u and is denoted by $d_k(u)$. A *distance-k ρ-coloring* (or (k, ρ)-coloring for

short) of a graph $G = (V, E)$ is a mapping $\phi : V \rightarrow \{1, 2, \ldots, \rho\}$ such that $\phi(u) \neq \phi(v)$ whenever u and v are distance-k neighbors. We call a mapping $\phi : V \rightarrow \{1, 2, \ldots, \rho\}$ a $(\frac{3}{2}, \rho)$-coloring of a graph $G = (V, E)$ if ϕ is a $(1, \rho)$-coloring of G and every path containing three edges uses at least three colors. Notice that a $(\frac{3}{2}, \rho)$-coloring is a restricted $(1, \rho)$-coloring, and a relaxed $(2, \rho)$-coloring, and hence the name. For instance, consider a path u, v, w, x in a graph. The assignment $2, 1, 2, 3$ to the respective vertices is a valid D1- and D$\frac{3}{2}$-coloring, but not a valid D2-coloring. The *distance-k graph coloring problem* asks for a (k, ρ)-coloring of a graph with the least possible value of ρ.

Let A be an $m \times n$ rectangular matrix with rows $r_1, r_2, \ldots, r_m$ and columns $a_1, a_2, \ldots, a_n$. We define the *bipartite graph* of A as $G_b(A) = (V_1, V_2, E)$ where $V_1 = \{r_1, r_2, \ldots, r_m\}$, $V_2 = \{a_1, a_2, \ldots, a_n\}$, and $(r_i, a_j) \in E$ whenever $a_{ij} \neq 0$, for $1 \leq i \leq m$, $1 \leq j \leq n$. When A is an $n \times n$ symmetric matrix with non-zero diagonal elements, the *adjacency graph* of A is defined to be $G_a(A) = (V, E)$ where $V = \{a_1, a_2, \ldots, a_n\}$ and $(a_i, a_j) \in E$ whenever a_{ij}, $i \neq j$, is a non-zero element of A. Note that the non-zero diagonal elements of A are not explicitly represented by edges in $G_a(A)$. In our work, we rely on the bipartite and adjacency graph representations. However, in the literature, a nonsymmetric matrix A is often represented by its *column intersection graph* $G_c(A)$. In this representation, the columns of A constitute the vertex set, and an edge (a_i, a_j) exists whenever the columns a_i and a_j have non-zero entries at the same row position (i.e., a_i and a_j are not structurally orthogonal). As argued in [7], the bipartite graph representation is more flexible and robust than the 'compressed' column intersection graph.

Coleman and Moré [1] showed that problem NONSYMCOLPART is equivalent to the D1-coloring problem when the matrix is represented by its column intersection graph. We have shown [7] that the same problem is equivalent to a partial D2-coloring when a bipartite graph is used and discussed the relative merits of the two approaches. The word 'partial' reflects the fact that only the vertices corresponding to the columns need to be colored.

McCormick [13] showed that the approximation of a Hessian using a direct method is equivalent to a D2-coloring on the adjacency graph of the matrix. One drawback of McCormick's formulation is that it does not exploit symmetry. Later Coleman and Moré [2] addressed this issue and showed that the resulting problem (SYMCOLPART) is equivalent to the D$\frac{3}{2}$-coloring problem.

3 Parallel Coloring Algorithms

The distance-k coloring problem is NP-hard for any fixed integer $k \geq 1$ [12]. A proof-sketch showing that D$\frac{3}{2}$-coloring is NP-hard is given in [2]. Furthermore, Lexicographically First $\Delta+1$ Coloring (LFC), the polynomial variant of D1-coloring in which the vertices are given in a predetermined order and the question at each step is to assign the vertex the smallest color not used by any of its neighbors, is P-complete [8]. The practical implication of this is that designing efficient fine-grained parallel algorithm for LFC is hard.

In practice, greedy sequential D1-coloring heuristics are found to be quite effective [1]. In a recent work [6], we have shown effective methods of parallelizing such greedy algorithms in a coarse-grained setting. Here, we extend this work to develop parallel algorithms for the D2- and D$\frac{3}{2}$-coloring problems.

Jones and Plassmann [10] describe a parallel distributed memory D1-coloring algorithm that uses randomization to assign priorities to the vertices, and then colors the vertices in the order determined by the priorities. There is no speculative coloring in their algorithm. It was reported that the algorithm slows down as the number of processors is increased. Finocchi et al. [5] suggest a parallel D1-coloring algorithm organized in several rounds; in each round, currently uncolored vertices are assigned a tentative pseudo-color without consulting their neighbors mapped to other processors; in a conflict resolution step, a maximal independent set of vertices in each color class is assigned these colors as final; the remainder of the vertices are uncolored, and the algorithm moves into the next round. However, they do not give any implementation and we believe that this algorithm incurs too many rounds, each with its synchronization and communication steps, for it to be practical on large graphs.

Our algorithm (in its generic form) may be viewed as a compromise between these algorithms, where we permit speculative coloring, but limit the number of synchronization steps to two in the whole algorithm. However, our current algorithms rely on the shared address space programming model; we will adapt our algorithms to distributed memory programming models in future work. We are unaware of any previous work on parallel algorithms for D2- and D$\frac{3}{2}$-coloring problems.

3.1 Generic Greedy Parallel Coloring Algorithm (GGPCA)

The steps of our generic parallel coloring algorithm can be summarized as shown below; refer to [6] for a detailed discussion of the D1-coloring case. Let $G = (V, E)$ be the input graph and p be the number of processors.

Phase 0: *Partition*

Randomly partition V into p equal blocks $V_1 \dots V_p$. Processor P_i is responsible for coloring the vertices in block V_i.

Phase 1: *Pseudo-color*

for $i = 1$ **to** p **do in parallel**

  for each $u \in V_i$ **do**

    assign the smallest available color to u, paying attention to already colored vertices (both local and non-local).

—barrier synchronize—

Phase 2: *Detect conflicts*

for $i = 1$ **to** p **do in parallel**

  for each $u \in V_i$ **do**

    check whether the color of u is valid. If the colors of u and v are the same for some 'neighbor' v of u then

      $L_i = L_i \cup min\{u, v\}$

—barrier synchronize—

Phase 3: *Resolve conflicts*

Color the vertices in the conflict list $L = \cup L_i$ sequentially.

In Phase 0, the vertices are randomly partitioned into p equal blocks each of which is assigned to some processor. In Phase 1, the processors color the vertices in their respective blocks in parallel. When two 'neighbor' vertices reside on different processors, the two processors could color both simultaneously, possibly assign them the same value, and cause a conflict. The purpose of Phase 2 is to detect and store any such conflict vertices which are subsequently re-colored sequentially in Phase 3.

3.2 Simple Distance-2 (SD2) Coloring Algorithm

The meaning of 'available' color in GGPCA depends on the required coloring. In the case of a D2-coloring, a vertex is assigned the smallest color not used by any of its distance-2 neighbors. Let Δ denote the maximum degree-1 in the graph $G = (V, E)$. It can easily be verified that, in a D2-coloring, a vertex can always be assigned one of the colors from the set $\{1, 2, \dots, \Delta^2 + 1\}$. Moreover, since the distance-1 neighbors of a vertex are

distance-2 neighbors with each other, the 2-chromatic number, i.e., the least number of colors required in a D2-coloring, is at least $\Delta+1$. Thus, the greedy approach is an $O(\Delta)$-approximation algorithm. We refer to the variant of GGPCA that applies to D2-coloring as Algorithm SD2.

Note that the sequential time complexity of greedy D2-coloring is $O(\Delta^2|V|)$. The following results show that the number of conflicts discovered in Phase 2 of Algorithm SD2 is often small for sparse graphs, making the algorithm scalable when the number of processors $p = O(\sqrt{\frac{|V|^2}{\Delta|E|}})$. The proofs are straightforward extensions of our proofs for the D1-coloring case given in [6] and hence are omitted here. One only needs to observe that the degree-2 and degree-1 of a vertex u are related by $d_2(u) \le \Delta d_1(u)$. Let $\overline{\delta} = 2|E|/|V|$ denote the average degree-1 in G.

Lemma 1 *The expected number of conflicts created at the end of Phase 1 of Algorithm SD2 is at most* $\approx \frac{\Delta\overline{\delta}(p-1)}{2}$.

Theorem 2 *On a CREW PRAM, Algorithm SD2 distance-2 colors the input graph consistently in expected time* $O(\Delta^2(\frac{|V|}{p} + \Delta\overline{\delta}p))$ *using at most* $\Delta^2 + 1$ *colors.*

Corollary 3 *When* $p = O(\sqrt{\frac{|V|^2}{\Delta|E|}})$, *the expected runtime of Algorithm SD2 is* $O(\frac{\Delta^2|V|}{p})$.

The number of conflicts predicted by Lemma 1 is an overestimate. The analysis assumes that whenever two distance-2 neighbor vertices are colored simultaneously, they are assigned the same color, thereby resulting in a conflict. However, a more involved probabilistic analysis that takes the distribution of colors used into account may provide a tighter bound. Besides, the actual number of conflicts in an implementation could be significantly reduced by choosing a random color from the allowable set, instead of the smallest one as given in Phase 1 of GGPCA.

3.3 Improved Distance-2 (ID2) Coloring Algorithm

The number of colors used in Algorithm SD2 can be reduced using a 'two-round-coloring' strategy. The underlying idea in the D1-coloring case is due to Culberson [4] and was used in our parallel algorithms for D1-coloring [6]. In Lemma 4 we extend the result to the D2-coloring case; the proof is basically a reproduction of Culberson's proof for the distance-1 coloring case. The greedy sequential algorithm referred to in the lemma is one that

visits the vertices of a graph $G = (V, E)$ in some order (a permutation of $\{1, 2, \ldots, |V|\}$), each time assigning a vertex the *smallest* allowed color. In particular, the first vertex to be visited is assigned color 1.

Lemma 4 *Let ϕ be a distance-2 coloring of a graph G using α colors, and π a permutation of the vertices such that if $\phi(v_{\pi(i)}) = \phi(v_{\pi(l)}) = c$, then $\phi(v_{\pi(j)}) = c$ for $i < j < l$. Applying the greedy sequential distance-2 coloring algorithm to G where the vertices have been ordered by π will produce a coloring ϕ' using α or fewer colors.*

Proof: The proof is a simple induction showing that the first i color classes[1] listed in the permutation will be colored with i or fewer colors. Clearly, the first color class listed will be colored with color 1. Suppose some element of the ith class requires color $i+1$. This means that it must have a distance-2 neighbor of color i. But by induction the vertices in the 1st to the $(i-1)$th classes used no more than $i-1$ colors. Thus, the conflict has to be with a member of its own color class, but this contradicts the assumption that ϕ is a valid distance-2 coloring. □

The idea in Lemma 4 is that if the greedy coloring algorithm is re-applied on a graph, with the vertices belonging to the same color class (in the original coloring) listed consecutively, then the new coloring obtained is better or at least as good as the original. One ordering (among many) that satisfies this condition, with a good potential for reducing the number of colors used, is to list the vertices consecutively for each color class in the reverse order of the introduction of the color classes. Based on Lemma 4, we modify Algorithm SD2 and introduce an additional parallel coloring phase between Phases 1 and 2. Algorithm ID2 below outlines the resulting 4-phase algorithm.

Phases 0 and 1. Same as Ph. 0 and 1 of GGPCA.
(Let s be the number of colors.)
Phase 2. **for** $k = s$ **downto** 1 **do**
  Partition ColorClass(k) into p equal blocks $V'_1, \ldots, V'_p$
  for $i = 1$ **to** p **do in parallel**
    for each $u \in V'_i$ **do**
      assign the smallest available color to vertex u.
  —*barrier synchronize*—
Phases 3 and 4. Same as Phases 2 and 3 of GGPCA, respectively.

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

In Phase 2, most of the vertices in a color class are at a distance greater than two edges from each other, and the exceptions arise from the conflict vertices colored incorrectly in Phase 1. Since the number of such conflict vertices from Phase 1 is low, the number of conflict vertices at the end of the re-coloring phase will be even lower. Phases 3 and 4 are included to detect and resolve any eventual conflicts not resolved in Phase 2.

3.4 Simple Distance-$\frac{3}{2}$ (SD$\frac{3}{2}$) Coloring Algorithm

Recall that a D$\frac{3}{2}$-coloring is a *relaxed* D2-coloring (see the example in Section 2.2). One way of relaxing the requirement for D2-coloring in GGPCA so as to obtain a valid D$\frac{3}{2}$-coloring is to let two vertices at a distance of *exactly* two edges from each other share a color as long as the vertex in between them is already colored with a (color of) lower value. We refer to the variant of GGPCA that employs this technique to achieve a distance-3/2 coloring as Algorithm SD$\frac{3}{2}$.

Note that both Algorithms ID2 and SD$\frac{3}{2}$ take asymptotically the same time as Algorithm SD2.

3.5 Randomization

The potential scalability of GGPCA depends on the number of conflicts discovered in Phase 2, since these are resolved sequentially in Phase 3. For dense graphs the number of conflicts could be large enough to destroy the scalability of the algorithm. To overcome this problem, we use randomization as a means for reducing the number of conflicts. In the randomized variants of Algorithms SD2, SD$\frac{3}{2}$, and ID2, a vertex is assigned the next available color with probability q, where $0 < q \leq 1$. The first attempt is made with the smallest available color, i.e., the color is chosen with probability q. An attempt is said to be successful if the vertex is assigned a color. If an attempt is not successful, then the next available color is tried with probability q, and so on, until the vertex gets a color. Algorithms SD2, SD$\frac{3}{2}$, and ID2, can be seen as the deterministic variants where $q = 1$. We refer to the randomized versions of the respective algorithms as RSD2, RSD$\frac{3}{2}$, and RID2.

Let u and v be two vertices with the same (infinite) set of allowable colors. If u and v are colored concurrently, it can be shown that the probability that u and v get the same color is $q/(2-q)$. This shows how randomization leads to a reduction in the number of conflicts; the lower the value of q, the lower chance for a conflict to arise. It should however be noted that a 'low' value

Set	Problem	$\lvert V\rvert$	$\lvert E\rvert$	Δ	δ	$\overline{\delta}$
Set I (FE)	mrng2	1,017,253	2,015,714	4	2	4
	fe144	144,649	1,074,393	26	4	15
	m14b	214,765	1,679,018	40	4	16
Set II (EV)	ev01	10,134	1,318,579	634	93	260
	ev02	19,845	3,353,890	749	78	338

Table 1: Test Graphs: the last three columns list the max., min., and average degree-1, respectively.

of q may result in an increase in the number of colors used. Choosing the right value for q thus becomes a design issue.

4 Experimental Results and Discussion

Our test bed consists of graphs that arise from finite element methods and from eigenvalue computations [6]. Table 1 provides the test graphs' structural information: columns $|V|$ and $|E|$ give the number of vertices and edges of each graph, respectively, and the maximum, minimum, and average degree-1 of each graph is given under columns Δ, δ, and $\overline{\delta}$, respectively.

Table 2 through 7 provide coloring and timing information of the different algorithms. The number of blocks (processors) is given in column p. Column χ_i gives the number of colors used at the end of Phase i of the corresponding algorithm. The number of conflicts that arise in Phase 1 (or 2) are listed under the column labeled K (or K_2). The time in milliseconds required by the different phases are listed under T_1, T_2, T_3, T_4; column T_{tot} gives the total time used. The last two columns display speedup with respect to two different references: S_{seq} lists the speedup obtained in comparison with the runtime of a pure sequential version, i.e., a version with no conflict detection phase, and S_{par} displays speedup obtained by taking the runtime of a parallel algorithm on one processor as a reference.

Since the deterministic algorithms gave acceptable results on the relatively sparse graphs of Set I, the randomized variants were run only on the graphs from Set II.

The experimental results show that Algorithm SD2 uses many fewer colors than the bound $\Delta^2 + 1$ and that Algorithm ID2 reduces the number of colors by up to 10% compared to SD2. The advantage of exploiting symmetry in Problem SYMCOLPART can be seen by comparing the number of colors used in SD2 and SD$\frac{3}{2}$; the latter can be as much as 37% fewer than the former. In general, the number of colors used in our deterministic algorithms increases only slightly with increasing p. For the randomized

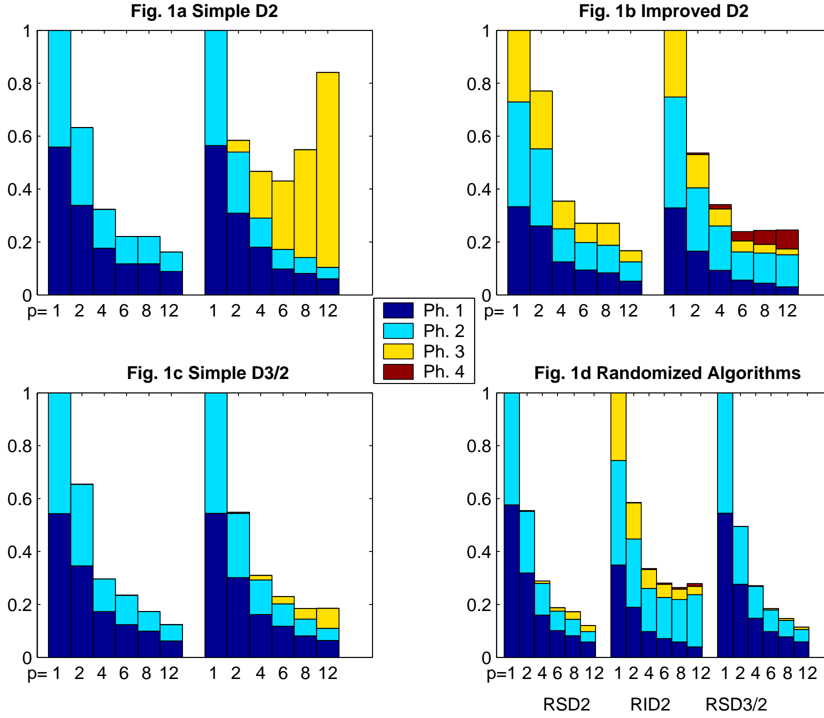

Figure 1: **a-c**: Relative performance of deterministic algorithms on graphs fe144 (left-half on each sub-figure) and ev02 (right-half). Fig. 1d: Relative performance of randomized algorithms on ev02.

algorithms, the number of colors in some cases even decreases as p increases, but we think this is a random phenomenon.

Based on the results displayed in Tables 2 through 7, Fig. 1 shows how the different phases of the algorithms scale as more processors are employed for two representative graphs: fe144 from Set I and ev02 from Set II. For graph fe144, the time used in resolving conflicts sequentially is negligible compared to the overall time. Moreover, it can be seen that Phases 1 and 2 of Algorithms SD2 and SD$\frac{3}{2}$ scale rather well on this graph as the number of processors is increased. However, Phase 2 of Algorithm ID2 does not scale as well. This is due to the existence of many color classes with few vertices which entails extra synchronization and communication overhead in the parallel re-coloring phase.

The picture for the more dense graph ev02 is different: the time elapsed in Phase 3 of Algorithm SD2 is significant and increases as the number of processors is increased. The situation is somewhat better for SD3/2 and ID2. Note that ev02 is about 100 times denser than fe144 (where density $= \frac{|E|}{|V|^2}$), and the results in Tables 2 to 7 and Fig. 1 agree well with the results

in Corollary 3: when the average degree is high, we lose scalability.

Fig. 1d shows how using probabilistic algorithms solves the problem of high number of conflicts for graph ev02. The improvement in scalability comes at the expense of increased number of colors used (see Table 2 to 7). We have experimented using different values for q and found good results when $q = 1/20$ for RSD2, and $q = 1/6$ for RID2 and RSD$\frac{3}{2}$.

It should be noted that the speedups observed in Fig. 1 are all relative to the respective parallel algorithm run with $p = 1$ (see S_{par} in the various tables). A comparison against a sequential version with no conflict detecting and resolving phase would yield less speedup (see S_{seq} in the tables). In particular, relative to a sequential version, the ideal speedup obtained by Algorithms SD2 and ID2 is roughly $\frac{1}{2}p$ and $\frac{2}{3}p$, respectively.

Our algorithms in general did not scale well beyond around 16 processors. We believe this is due to, among other things, the relatively high cost associated with non-local physical memory accesses. It would be interesting to see how this affects the behavior of the algorithms on different parallel platforms.

5 Conclusion

We have presented several simple and effective parallel approximation algorithms as well as results from OpenMP-implementations for the D2- and D$\frac{3}{2}$-coloring problems. The number of colors produced by the algorithms in the case where $p = 1$ is generally good as it is typically off from the lower bound $\Delta + 1$ of SD2 by a factor much less than the approximation ratio Δ. As more processors are employed, the algorithms provide reasonable speedup while maintaining the quality of the solution. In general, our deterministic algorithms seem to be suitable for sparse graphs and the probabilistic variants for more dense graphs. We believe the functionality provided by our algorithms is useful for many large-scale optimization codes, where parallel speedups while desirable, are not paramount, as long as running times for coloring are low relative to the other steps in the optimization computations. The three sources of parallelism in our algorithms – partitioning, speculation, and randomization – can be exploited in developing distributed parallel algorithms, but the algorithms would most likely differ significantly from the shared memory variants presented here.

References

[1] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, 20(1):187–209, February 1983.

[2] T. F. Coleman and J. J. Moré. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Program.*, 28:243–270, 1984.

[3] T. F. Coleman and A. Verma. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput.*, 19(4):1210–1233, July 1998.

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

[5] I. Finocchi, A. Panconesi, and R. Silvestri. Experimental analysis of simple, distributed vertex coloring algorithms. In *Proceedings of the Thirteenth ACM-SIAM Symposium on Discrete Algorithms (SODA 02)*, San Francisco, CA, 2002.

[6] A. H. Gebremedhin and F. Manne. Scalable parallel graph coloring algorithms. *Concurrency: Pract. Exper.*, 12:1131–1146, 2000.

[7] A. H. Gebremedhin, F. Manne, and A. Pothen. Graph coloring in optimization revisited. Technical Report 226, University of Bergen, Dept. of Informatics, Norway, January 2002. Available at: http://www.ii.uib.no/publikasjoner/texrap/.

[8] R. Greenlaw, H.J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory.* Oxford University Press, New York, 1995.

[9] A.K.M S. Hossain and T. Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, 10:33–48, 1998.

[10] M. T. Jones and P. E. Plassmann. A parallel graph coloring heuristic. *SIAM J. Sci. Comput.*, 14(3):654–669, May 1993.

[11] S. O. Krumke, M. V. Marathe, and S. S. Ravi. Approximation algorithms for channel assignment in radio networks. In *Dial M for Mobility, 2nd International Workshop on Discrete Algorithms and Methods for*

Mobile Computing and Communications, Dallas, Texas, September 30 – October 1 1998.

[12] Y. Lin and S. S. Skiena. Algorithms for square roots of graphs. *SIAM J. Disc. Math.*, 8:99–118, 1995.

[13] S. T. McCormick. Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem. *Math. Program.*, 26:153–171, 1983.

[14] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001. Chapter 5.

Problem	p	χ_1	χ_3	K	T_1	T_2	T_3	T_{tot}	S_{par}	S_{seq}
mrng2	1	12	12	0	11.5	11	0	22.5	1	0.5
mrng2	2	12	12	1	6.5	6.4	0	12.9	1.7	0.9
mrng2	4	12	12	9	4.2	4	0	8.3	2.7	1.4
mrng2	6	12	12	9	2.8	2.8	0	5.7	4	2
mrng2	8	12	12	9	2.5	2.5	0	5	4.5	2.3
mrng2	12	13	13	16	1.5	1.5	0	3	7.5	3.8
fe144	1	41	41	0	3.8	3	0	6.8	1	0.6
fe144	2	40	40	3	2.3	2.0	0	4.3	1.6	0.9
fe144	4	41	41	13	1.2	1	0	2.2	3.1	1.7
fe144	6	41	41	25	0.8	0.7	0	1.5	4.5	2.5
fe144	8	43	43	20	0.8	0.7	0	1.5	4.5	2.5
fe144	12	42	43	110	0.6	0.5	0	1.1	6.2	3.5
m14b	1	42	42	0	5.6	4.8	0	10.4	1	0.5
m14b	2	43	43	0	3.5	3.1	0	6.6	1.6	0.9
m14b	4	44	44	24	1.8	1.6	0	3.4	3.1	1.7
m14b	6	43	43	16	1.7	1.4	0	3.1	3.4	1.8
m14b	8	44	46	28	1.2	1	0	2.2	4.7	2.6
m14b	12	43	44	70	0.9	0.8	0	1.8	5.8	3.1
ev01	1	3393	3393	0	38.2	28.9	0	67.1	1	0.6
ev01	2	3298	3497	1514	18.7	14.1	7.6	40.4	1.7	0.9
ev01	4	3150	3598	4555	10.3	7	23.4	40.8	1.7	0.9
ev01	6	3215	3725	7786	7.2	5.3	41.3	54	1.2	0.7
ev01	8	2902	3755	10113	5.7	3.9	52	61.6	1.1	0.6
ev01	12	2922	3900	17872	3.9	3	95	102	0.7	0.4
ev02	1	4260	4260	0	144	111	0	255	1	0.6
ev02	2	4212	4353	1012	78.8	58.9	11.2	148.9	1.7	1
ev02	4	4184	4633	4050	46	28	45	119	2.1	1.2
ev02	6	4172	4660	6206	25	18.8	66	110	2.3	1.3
ev02	8	4152	4870	8873	20.6	15.4	104	140	1.8	1
ev02	12	4036	5168	16394	15.5	11	188	214.7	1.2	0.7

Table 2: Performance of Algorithm SD2.

Problem	p	χ_1	χ_2	χ_4	K_1	K_2	T_1	T_2	T_3	T_4	T_{tot}	S_{par}	S_{seq}
mrng2	1	12	11	11	0	0	9.6	11	8.9	0	29.6	1	0.7
mrng2	2	12	11	11	1	0	4.8	5.7	4.7	0	15.3	1.9	1.3
mrng2	4	13	10	10	0	0	3.6	5.4	2.8	0	11.8	2.5	1.7
mrng2	6	12	10	10	6	0	2.8	3.8	2.4	0	9	3.3	2.3
mrng2	8	12	11	11	9	0	2	2.7	2	0	6.7	4.4	3.1
mrng2	12	12	10	10	20	0	1.5	2	1.5	0	5	6	4.1
fe144	1	41	37	37	0	0	3.2	3.8	2.6	0	9.6	1	0.7
fe144	2	41	38	38	0	0	2.5	2.8	2.1	0	7.4	1.3	1
fe144	4	41	37	37	18	0	1.2	1.2	1	0	3.4	2.8	2.1
fe144	6	41	38	38	39	0	0.9	1	0.7	0	2.6	3.7	2.7
fe144	8	41	37	37	55	0	0.8	1	0.8	0	2.6	3.7	2.7
fe144	12	41	38	38	78	0	0.5	0.7	0.4	0	1.6	6	4.4
m14b	1	42	41	41	0	0	6	8.8	5	0	19.8	1	0.7
m14b	2	43	41	41	0	0	3.9	4.9	3.4	0	12.3	1.6	1.2
m14b	4	43	41	41	26	0	2	2.9	1.7	0	6.6	3	2.2
m14b	6	43	41	41	17	0	1.6	1.5	1	0	4.1	4.8	3.6
m14b	8	44	41	41	31	0	1.2	1.3	1	0	3.6	5.5	4.1
m14b	12	44	41	41	45	0	0.9	1	0.7	0	2.7	7.3	5.5
ev01	1	3393	3148	3148	0	0	38.5	44.5	28.5	0	111.5	1	0.7
ev01	2	3301	3132	3290	1687	316	18.4	24.7	13.4	1.8	58.4	1.9	1.4
ev01	4	3159	3091	3263	4171	863	10	19	7.3	4.6	41	2.7	2
ev01	6	3220	3116	3371	7913	1475	7.8	17.5	5.4	8.5	39.2	2.8	2.1
ev01	8	2927	3041	3468	10932	2675	5.3	15.3	4	14.3	39	2.9	2.1
ev01	12	2906	3071	3481	18264	3040	3.8	17	2.8	16.4	40.6	2.7	2
ev02	1	4260	4016	4016	0	0	150	191	115	0	456	1	0.7
ev02	2	4210	4024	4085	1019	207	75.2	109.2	57.5	2.7	244.7	1.9	1.4
ev02	4	4194	4023	4226	4154	644	42.3	76.5	29.2	7.6	155.7	2.9	2.7
ev02	6	4186	4031	4247	6196	1354	25.4	48.6	19	16	109	4.2	3.1
ev02	8	4138	4013	4349	10732	2007	20	52	15	24	111	4.1	3.1
ev02	12	4030	4008	4455	18392	2819	14	55.2	10	32.7	112.5	4.1	3

Table 3: Performance of Algorithm ID2.

Problem	p	χ_1	χ_3	K	T_1	T_2	T_3	T_{tot}	S_{par}	S_{seq}
mrng2	1	10	10	0	9.3	9	0	18.3	1	0.5
mrng2	2	10	10	1	5.2	4.8	0	10	1.8	0.9
mrng2	4	10	10	6	3.3	2.7	0	6	3	1.6
mrng2	6	10	10	4	2.5	2.5	0	5	3.7	1.9
mrng2	8	11	11	12	2.3	2.3	0	4.6	4	2
mrng2	12	11	11	10	1.7	1.7	0	3.4	5.4	2.7
fe144	1	35	35	0	4.4	3.7	0	8.1	1	0.5
fe144	2	36	36	0	2.8	2.5	0	5.3	1.5	0.8
fe144	4	35	35	6	1.4	1	0	2.4	3.3	1.8
fe144	6	36	36	15	1	0.9	0	1.9	4.3	2.3
fe144	8	35	35	11	0.8	0.6	0	1.5	5.4	2.9
fe144	12	36	36	39	0.5	0.5	0	1	8.1	4.4
m14b	1	34	34	0	5.4	4.6	0	10	1	0.5
m14b	2	37	37	0	2.9	2.6	0	5.5	1.8	1
m14b	4	37	37	3	1.9	1.6	0	3.5	2.9	1.5
m14b	6	39	39	7	1.6	1.3	0	2.9	3.5	1.9
m14b	8	37	37	10	1.2	1.1	0	2.3	4.3	2.3
m14b	12	38	38	16	1	0.8	0	1.8	5.6	3
ev01	1	2148	2148	0	35.5	29	0	64.5	1	0.6
ev01	2	1969	2012	135	19.2	15.4	0.7	35.3	1.8	1
ev01	4	1915	2070	366	9.5	7.6	1.9	19	3.4	1.9
ev01	6	2429	2618	538	7.7	6	2.8	16.5	3.9	2.2
ev01	8	1822	2286	1031	5.6	4.3	6.1	16	4	2.2
ev01	12	2103	2687	1424	4	3	7.5	14.5	4.4	2.5
ev02	1	2697	2697	0	134.4	112.6	0	247	1	0.5
ev02	2	2626	2653	83	74.4	60.1	1	135.5	1.8	1
ev02	4	2590	2736	340	40	32.2	4.4	76.5	3.2	1.8
ev02	6	2584	2818	595	29	21	6.7	56.7	4.4	2.4
ev02	8	2626	2967	960	20	15.6	10	45.5	5.4	3
ev02	12	2536	3049	1569	15.7	11.4	18.6	45.7	5.4	3

Table 4: Performance of Algorithm SD$\frac{3}{2}$.

Problem	p	χ_1	χ_3	K	T_1	T_2	T_3	T_{tot}	S_{par}	S_{seq}
ev01	1	4077	4077	0	38	28	0	66	1	0.6
ev01	2	4084	4088	59	20	15	0.3	35.2	1.9	1.1
ev01	4	4043	4051	139	10	7.8	0.8	19	3.5	2
ev01	6	4132	4144	572	7	5.5	3.5	16	4.1	2.4
ev01	8	3998	4015	388	6	4.3	2	12	5.5	3.2
ev01	12	4019	4042	587	4	2.8	3.2	10	6.6	3.8
ev02	1	5564	5564	0	148.3	109	0	257.3	1	0.6
ev02	2	5581	5584	43	82	60	0.6	143	1.8	1
ev02	4	5518	5525	187	41	31	2.3	75	3.4	2
ev02	6	5553	5562	273	26	19	3.3	49	5.2	3
ev02	8	5576	5583	558	21	16	7.4	44	5.8	3.4
ev02	12	5514	5536	522	15	10	6	31	8.3	4.8

Table 5: Performance of Algorithm RSD2, $q = 1/20$.

Problem	p	χ_1	χ_2	χ_4	K_1	K_2	T_1	T_2	T_3	T_4	T_{tot}	S_{par}	S_{seq}
ev01	1	3719	3169	3169	0	0	36.6	42.5	27	0	106.1	1	0.7
ev01	2	3706	3161	3173	171	21	19	27	15	0.1	61	1.7	1.3
ev01	4	3648	3183	3217	483	152	10	21	7.5	0.8	40	2.7	2
ev01	6	3736	3205	3326	998	193	8	21	5.3	1	35	3	2.3
ev01	8	3601	3167	3236	1272	342	5.7	19	4	2	30	3.5	2.6
ev01	12	3628	3195	3291	1857	447	3.7	18.7	2.6	2.2	27	3.9	2.9
ev02	1	4919	4024	4024	0	0	132.7	150	97.3	0	380	1	0.7
ev02	2	4871	4027	4066	208	43	72	98	52	0.5	223	1.7	1.3
ev02	4	4860	4025	4082	526	127	37	62	27	1.5	128	3	2.2
ev02	6	4865	4040	4084	787	138	27	59	19	1.7	108	3.5	2.6
ev02	8	4869	4032	4100	1577	207	22	61	15	2.3	101	3.8	2.8
ev02	12	4839	4031	4104	1953	320	15	75	12	4	106	3.6	2.7

Table 6: Performance of Algorithm RID2, $q = 1/6$.

Problem	p	χ_1	χ_3	K	T_1	T_2	T_3	T_{tot}	S_{par}	S_{seq}
ev01	1	2392	2392	0	37	30	0	67	1	0.6
ev01	2	2242	2242	23	18	15	0.1	33	2	1.1
ev01	4	2189	2190	46	9.7	7.8	0.3	18	3.7	2
ev01	6	2582	2583	55	7.6	5.4	0.3	13	5.1	2.8
ev01	8	2170	2225	138	5.8	4.8	0.7	11	6.1	3.4
ev01	12	2496	2528	191	3.8	3	1.3	8	8.4	4.6
ev02	1	3142	3142	0	140	117	0	257	1	0.5
ev02	2	3049	3049	18	71	56	0.2	127	2	1.1
ev02	4	3076	3080	51	38	31	0.6	69	3.7	2
ev02	6	3018	3060	123	25	21	1.4	48	5.3	2.9
ev02	8	3106	3125	155	20	16	1.7	37	6.9	3.8
ev02	12	2995	3081	214	15	12	2.4	29	8.8	4.8

Table 7: Performance of Algorithm RSD$\frac{3}{2}$, $q = 1/6$.

Graph Coloring on Coarse Grained Multicomputers*

Assefaw Hadish Gebremedhin† Isabelle Guérin Lassous‡
Jens Gustedt§ Jan Arne Telle

Abstract

We present an efficient and scalable Coarse Grained Multicomputer (CGM) coloring algorithm that colors a graph G with at most $\Delta + 1$ colors where Δ is the maximum degree in G. This algorithm is given in two variants: *randomized* and *deterministic*. We show that on a p-processor CGM model the proposed algorithms require a parallel time of $O(\frac{|E|}{p})$ and a total work and overall communication cost of $O(|E|)$.These bounds correspond to the average case for the randomized version and to the worst case for the deterministic variant.

key words: graph algorithms, parallel algorithms, graph coloring, Coarse Grained Multicomputers.

---

*Research supported in part by The Aurora Programme, a France-Norway Collaboration Research Project of The Research Council of Norway, The French Ministry of Foreign Affairs and The Ministry of Education, Research and Technology.

†Department of Informatics, University of Bergen, 5020 Bergen, Norway. {assefaw, telle}@ii.uib.no

‡INRIA Rocquencourt, France. Isabelle.Guerin-Lassous@inria.fr

§LORIA & INRIA Lorraine, France. Jens.Gustedt@loria.fr

1 Introduction

The graph coloring problem deals with the assignment of positive integers (colors) to the vertices of a graph such that adjacent vertices do not get the same color and the number of colors used is minimized. A wide range of real world problems, among others, timetabling and scheduling [20], frequency assignment [8], register allocation [2], and efficient estimation of sparse matrices in optimization [4], have successfully been modeled using the graph coloring problem. Besides modeling real world problems, graph coloring plays a crucial role in the field of parallel computation. In particular, when a computational task is modeled using a graph where the vertices represent the subtasks and the edges correspond to the relationship among them, graph coloring is used in dividing the subtasks into independent sets that can be performed concurrently.

The graph coloring problem is known to be NP-hard [9], making heuristic approaches inevitable in practice. There exist a number of sequential graph coloring heuristics that are quite effective in coloring graphs encountered in practical applications. See [4] for some of the popular heuristics. However, due to their inherent sequential nature, these heuristics are difficult to parallelize. In fact, coloring the vertices of a graph in a given order where each vertex is assigned the smallest color that has not been given to any of its neighbors is shown to be P-complete [12]. Consequently, parallel graph coloring heuristics different from the effective sequential coloring heuristics had to be suggested. One of the important contributions in this regard is the parallel maximal independent set finding algorithm of Luby [21] and the coloring algorithm based on it. Subsequently, Jones and Plassmann [16] improved Luby's algorithm and in addition used *graph partitioning* as a means to achieve a distributed memory coloring heuristic based on explicit message-passing. Unfortunately, Jones and Plassmann did not get any speedup from their experimental studies. Later, Allwright et al. [1] performed a comparative study of the implementations of the Jones-Plassmann algorithm and a few other variations and reported that none of the algorithms included in the study yielded any speedup. The justification for the use of these parallel coloring heuristics has been the fact that they enabled solving large-scale problems that could not otherwise fit onto the memory of a sequential machine.

Despite these discouraging experiences, Gebremedhin and Manne [10] recently proposed a *shared memory* parallel coloring algorithm that yields good speedup. Their theoretical analysis using the PRAM model shows that the algorithm is expected to provide an almost linear speedup and

experimental results conducted on the Origin 2000 supercomputer using graphs that arise from finite element methods and eigenvalue computations validate the theoretical analysis.

The purpose of this paper is to make this successful approach feasible for a larger variety of architectures by extending it to the Coarse Grained Multicomputer (CGM) model of parallel computation [6]. The CGM model makes an abstraction of the interconnection network among the processors of a parallel computer (or network of computers) and captures the efficiency of a parallel algorithm using only a few parameters. Several experiments show that the CGM model is of practical relevance: implementations of algorithms formulated in the CGM model in general turn out to be portable, predictable, and efficient [13, 14].

In this paper we propose a CGM coloring algorithm that colors a graph G with at most $\Delta + 1$ colors where Δ is the maximum degree in G. The algorithm is given in two variants: one randomized and the other deterministic. We show that the proposed algorithms require a parallel time of $O(\frac{|E|}{p})$ and a total work and overall communication cost of $O(|E|)$. These bounds correspond to the average case for the randomized version and to the worst case for the deterministic variant.

The remainder of this paper is organized as follows. In Section 2 we review the CGM model of parallel computation and the graph coloring problem. In Section 3 we discuss a good data organization for our CGM algorithms and present the randomized variant of the algorithm along with its various subroutines. In Section 4 we provide an average case analysis of the randomized algorithm's time and work complexity. In Section 5 we show how to de-randomize our algorithm to achieve the same good time and work complexity also in the worst case. Finally, in Section 6 we give some concluding remarks.

2 Background

2.1 Coarse grained models of parallel computation

In the last decade several efforts have been made to define models of parallel (or distributed) computation that are more realistic than the classical PRAM models; see [7] or [19] for an overview of PRAM models. In contrast to the PRAM models that suppose that the number of processors p is polynomial in the input size N, the new models are *coarse grained*, i.e., they assume that p and N are orders of magnitude apart. Due to this assumption, the coarse grained models map much better on existing architectures

where in general the number of processors is in the order of hundreds and the size of the data to be handled could be in the order of billions.

The introduction of the *Bulk Synchronous Parallel* (BSP) bridging model for parallel computation by Valiant [24] marked the beginning of the increasing research interest in coarse grained parallel computation. The BSP model was later modified along different directions. For example, Culler et al. [5] suggested the LogP model as an extension of Valiant's BSP model in which asynchronous execution was modeled and a parameter was added to better account for communication overhead. In an effort to define a parallel computation model that retains the advantages of coarse grained models while at the same time is simple to use (involves few parameters), Dehne et al. [6] suggested the CGM model.

The CGM model considered in this paper is well suited for the design of algorithms that are not too dependent on a particular architecture and our basic assumptions of the model are listed below.

- The model consists of p processors and all the processors have the same size $M = O(N/p)$ of memory, where N is the input size.
- An algorithm on this model proceeds in so-called *supersteps*. A superstep consists of one phase of local computation and one phase of interprocessor communication.
- The communication network between the processors can be arbitrary.

The goal when designing an algorithm in this model is to keep the sum total of the computational cost per processor, the overall communication cost, and idle time of each processor within $T/s(p)$, where T is the runtime of the best sequential algorithm on the same input, and the *speedup* $s(p)$ is a function that should be as close to p as possible.

To achieve this, it is desirable to keep the number of supersteps of such an algorithm as low as possible, preferably within $O(M)$. The rationale here lies in the fact that, among others, the *message startup-cost* and the *bandwidth* of an architecture determine the communication overhead. In each superstep, a processor may need to do at most $O(p)$ communications and hence a number of supersteps of $O(M)$ ensures that the total startup-cost is at most $O(Mp) = O(N)$ and therefore lies within the complexity bound of the overall computational cost we anticipate for such an algorithm. The bandwidth restriction of a specific platform must still be observed, and here the best strategy is to reduce the communication volume as much as possible. See [13] for an overview of algorithms, implementations and experiments on the CGM model.

As a legacy from the PRAM model, it is usually assumed that the num-

ber of supersteps should be polylogarithmic in p. However, the assumption seems to have no practical justification. In fact, there is no known relationship between the coarse grained models and the complexity classes NC^k. In practice, algorithms that simply ensure a number of supersteps that is a function of p (but not of N) perform quite well [11].

To be able to organize the supersteps well, we assume that each processor can store a vector of size p for every other processor. Thus, the following inequality is assumed throughout this paper,

$$p^2 < M. \tag{1}$$

2.2 Graph coloring

A graph coloring is a labeling of the vertices of a graph $G = (V, E)$ with positive integers, called *colors*, such that adjacent vertices do not obtain the same color. It can equivalently be viewed as searching for a partition of the vertex set of the graph into *independent sets*. The primary objective in the graph coloring problem is to minimize the number of colors used. Even though coloring a graph with the fewest number of colors is an NP-hard problem, in many applications coloring using a bounded number of colors, possibly far from the minimum, may suffice. Particularly in many parallel graph algorithms, a bounded coloring (partition into independent sets) is needed as a subroutine. For example, graph coloring is used in the development of a parallel algorithm for computing the eigenvalues of certain matrices [22] and in parallel partial differential equation solvers [1].

One of the simplest and yet quite effective sequential heuristics for graph coloring is the *greedy* algorithm that visits the vertices of the graph in some order and in each visit assigns a vertex the smallest color that has not been used by any of the vertex's neighbors. It is easy to see that, for a graph $G = (V, E)$, such a greedy algorithm always uses at most $\Delta+1$ colors, where $\Delta = \max_{v \in V}\{\text{degree of } v\}$. In Greenlaw et al. [12], a restricted variant of the greedy algorithm in which the ordering of the vertices is predefined, and the algorithm is required to respect the given order, is termed as *Lexicographically First* $\Delta + 1$*-coloring* (LF$\Delta + 1$-coloring). We refer to the case where this restriction is absent and where the only requirement is that the resulting coloring uses at most $\Delta + 1$ colors, simply as $\Delta + 1$*-coloring*.

LF$\Delta+1$-coloring is known to be P-complete [12]. But for special classes of graphs, some NC algorithms have been developed for it. For example, Chelbus et al. [3] show that for *tree structured* graphs LF$\Delta + 1$-coloring is in NC. In the absence of the lexicographically first requirement, a few NC algorithms for general graphs have been proposed. Luby [21] has given

an NC algorithm for $\Delta + 1$-coloring by reducing the coloring problem to the maximal independent set problem. Moreover, Karchmer and Naor [17], Karloff [18], and Hajnal and Szemerédi [15] have each presented different NC algorithms for Brook's coloring (a coloring that uses at most Δ colors for a graph whose chromatic number is bounded by Δ). Earlier, Naor [23] had established that coloring planar graphs using five colors is in NC.

However, all of these NC coloring algorithms are mainly of theoretical interest as they require a polynomial number of processors, whereas, in reality, one has only a limited number of processors on a given parallel computer. In this regard, Gebremedhin and Manne [10] have recently shown a practical and effective shared memory parallel $\Delta + 1$-coloring algorithm. They show that distributing the vertices of a graph evenly among the available processors and coloring the vertices on each processor concurrently, while checking for color compatibility with already colored neighbors, creates very few conflicts. More specifically, the probability that a pair of adjacent vertices are colored at exactly the same instance of the computation is quite small. On a somewhat simplified level, the algorithm of Gebremedhin and Manne works by tackling the list of vertices numbered from 1 to n in a 'round robin' manner. At a given time t, where $1 \leq t \leq r$ and $r = \lceil \frac{n}{p} \rceil$, processor P_i colors vertex $(i-1) \cdot r + t$. The shared memory assumptions ensure that P_i may access the color information of any vertex at unit cost of time. Adjacent vertices that are in fact handled at exactly the same time are the only causes for concern as they may result in conflicts. Gebremedhin and Manne show that the number of such conflicts is small on expectation, and that conflicts can easily be resolved *a posteriori*. Their analysis of the resulting algorithm using the PRAM model shows that the algorithm colors a general graph $G = (V, E)$ with $\Delta + 1$ colors in expected time $O(|E|/p)$ when the number of processors p is such that $p \leq |V|/\sqrt{2|E|}$.

However, in a distributed memory setting, the most common case in our target model CGM, one has to be more careful about access to data located on other processors.

3 A CGM $\Delta + 1$-coloring algorithm

We start this section by discussing how we distribute the input graph among the available processors for our CGM $\Delta + 1$-coloring algorithms. Then, the randomized variant of our algorithm is presented in a top-down fashion, starting with an overview and filling the details as the presentation proceeds.

3.1 Data distribution

In general, a good data organization is crucial for the efficiency of a distributed memory parallel algorithm. For our CGM-coloring algorithm in particular, the input graph $G = (V, E)$ is organized in the following manner.

- Each processor P_i ($1 \leq i \leq p$) is responsible for a subset U_i of the vertices ($V = \bigcup_{i=1}^{p} U_i$). With a slight abuse of notation, the processor hosting a vertex v is denoted by P_v.
- Each edge $e = \{v, w\} \in E$ is represented as arcs (v, w) stored at P_v, and (w, v) stored at P_w.
- For each arc (v, w) processor P_v stores the identity of P_w and thus the location of the arc (w, v). This is to avoid a logarithmic blow-up due to the search for P_w.
- The arcs are sorted lexicographically and stored as a linked list per vertex.

In this data distribution, we require that the degree of each vertex be less than $D = \lceil \frac{N}{p} \rceil$, where $N = |E|$. Vertices with degree greater than D are treated in a separate preprocessing step.

If the input of the algorithm is not of the desired form, it can be efficiently transformed into one by carrying out the following steps.

- Generate two arcs for each edge as described above,
- Radix sort (see [13] for a CGM radix sort) the list of arcs such that each processor receives the arc (v, w) if it is responsible for vertex w,
- Let every processor note its identity on these arcs,
- Radix sort the list of arcs such that every processor receives its proper arcs (arc (v, w) if it is responsible for vertex v).

3.2 The algorithm

As the time complexity of sequential $\Delta + 1$-coloring is linear in the number of edges $|E|$, our aim is to design a parallel algorithm in CGM with $O(\frac{|E|}{p})$ work per processor and $O(|E|)$ overall communication cost. In an overview, our CGM coloring algorithm consists of two phases, an *initial* and a main *recursive* phase; see Algorithm 1.

In the initial phase, the subgraph induced by the vertices with degree greater than $\lceil \frac{N}{p} \rceil$ is colored sequentially on one of the processors. Clearly, there are at most p such vertices since otherwise we would have more than N edges in total. Thus the subgraph induced by these vertices has at most p^2 edges. Since p^2 is assumed to be less than M, the induced subgraph fits

Algorithm 1: $\Delta + 1$-coloring on a CGM with p processors

Input: Base graph $G = (V, E)$, the subgraph H induced by vertices of degree greater than $D = \lceil N/p \rceil$, the lists F_v of forbidden colors of vertices $v \in V$.

Output: A valid coloring of $G = (V, E)$ with at most $\Delta + 1$ colors.

initial phase Sequential$\Delta + 1$Coloring(H,$\{F_v\}_v$) (see Algorithm 3);

main phase ParallelRecursive$\Delta + 1$Coloring(G,$\{F_v\}_v$) (see Algorithm 2);

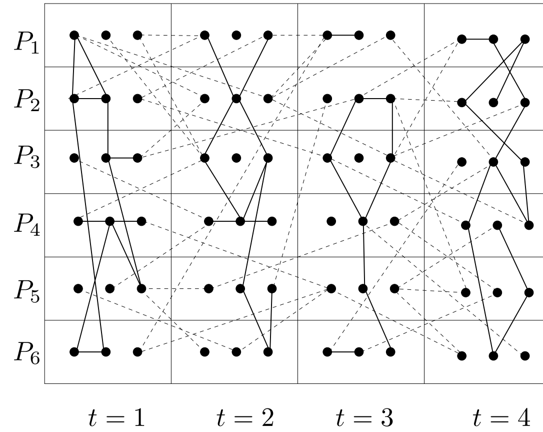

Figure 1: Graph on 72 vertices distributed onto 6 processors and 4 timeslots.

on a single processor (say P_1) and a call to Algorithm 3 colors it sequentially. Algorithm 3 is also used in a situation other than coloring such vertices. We defer the discussion on the details of Algorithm 3 to Section 3.2.1 where the situation that calls for its second use is presented.

The main part of Algorithm 1 is the call to Algorithm 2 which recursively colors any graph G such that the maximum degree $\Delta \leq M$. The basic idea of the algorithm is based on placing the vertices residing on each processor into different *timeslots*. The assignment of timeslots to the vertices gives rise to two categories of edges. The first category consists of edges which connect vertices having the same timeslot. We call these edges *bad* and all other edges *good*. Figure 1 shows an example of a graph distributed on 6 processors and 4 timeslots in which bad edges are solid and good edges are dashed.

In a nutshell, Algorithm 2 proceeds timeslot by timeslot where in each timeslot the graph defined by the bad edges and the vertices incident on them is identified and the algorithm is called recursively with the identified graph as input while the rest of the input graph is colored concurrently.

In Algorithm 2, while partitioning the vertices into k timeslots, where

Algorithm 2: Parallel Recursive $\Delta + 1$-coloring

Input: Subgraph $G' = (V', E')$ of a base graph $G = (V, E)$ with M' edges per processor such that $\Delta_{G'} \leq M'$, M the initial input size per processor, lists F_v of forbidden colors for the vertices.

Output: A valid coloring of G' with at most $\Delta_G + 1$ colors.

base case **if** *($|G'| < \frac{N}{kp^2}$)* **then** Sequential$\Delta + 1$Coloring(G',$\{F_v\}_v$) (see Algorithm 3);

else

- **high degree** HandleHighDegreeVertices(G',$\{F_v\}_v$,$2k$) (see Algorithm 5);
- **group vertices** **foreach** P_i **do**
  - Let $U_{i,t}$ for $t = 1, \ldots, k$ be result of the call GroupVerticesIntoTimeslots(V',k), (see Algorithm 6);
  - For each vertex v denote the index of its timeslot by t_v;
  - **foreach** *arc* (v, w) **do** collect the timeslot t_v in a send buffer for P_w;
  - Send out the tuples (w,t_v);
  - Receive the timeslots from the other processors;
- **for** $t = 1$ ***to*** k **do**
  - **foreach** *processor* P_i **do**
    - **identify conflicts** Consider all arcs $e = (v, w)$ with $v \in U_{i,t}$ and $t_v = t_w = t$;
    - Name this set S_i and consider the vertices V_{S_i} that have such an arc;
  - **balance load** G_{rec} =Balance($\bigcup_{i=1}^{p} V_{S_i}, \bigcup_{i=1}^{p} S_i$) (see Algorithm 7);
  - **recurse** ParallelRecursive$\Delta + 1$Coloring(G_{rec},$\{F_v\}_v$);
  - **foreach** *processor* P_i **do**
    - **color vertex** **foreach** *uncolored vertex* v *with* $t_v = t$ **do** Color v with least legal color;
    - **send messages** **foreach** *arc* (v, w) *with* $v \in U_i$*,* $t_v = t$ *and* $t_w > t$ **do** Collect the color of v in a send buffer for P_w;
    - Send out the tuples (w,color of v);
    - **receive messages** Receive the colors from the other processors;
    - **foreach** *received tuples (w, color of v)* **do** add color of v to F_w;

$1 < k \leq p$, we would like to achieve as even a distribution as possible. The call to Algorithm 6 in line **group vertices** does this by using the degree of each vertex as a criterion. This randomized algorithm is presented in Section 3.2.2 where the issue of load balancing is briefly discussed. Prior to calling Algorithm 6, vertices with 'high degrees' that would otherwise result in an uneven load balance are treated separately; see line **high degree**. The algorithm for treating high degree vertices, Algorithm 5, is presented in Section 3.2.2.

Notice that an attempt to concurrently color vertices incident on a bad edge may result in an inconsistent coloring (conflict). In a similar situation, Gebremedhin and Manne, in their shared memory formulation, tentatively allow such conflicts and resolve eventual conflicts in a later sequential phase. The success of their approach lies in the fact that the expected size of the edges in conflict is relatively small. In our case, we deal with the potential conflicts *a priori*. We first identify the subgraphs that could result in conflict and then color these subgraphs in parallel recursively until their union is small enough to fit onto the memory of a single processor. See lines **identify conflicts** and **recurse** in Algorithm 2. Note that, in general, some processors may receive more vertices than others. We must ensure that these recursive calls do not produce a blow-up in computation and communication. In order to ensure that the subgraph that goes into recursion is evenly distributed among the processors, a call to Algorithm 7 is made at line **balance load**. Algorithm 7 is discussed in Section 3.2.2.

In the recursive calls one must handle the restrictions that are imposed by previously colored vertices. We extend the problem specification and assume that a vertex v also has a list F_v of forbidden colors that initially is empty. An important issue for the complexity bounds is that a forbidden color is added to F_v only when the knowledge about it arrives on P_v. The list F_v as a whole will only be touched once, namely when v is finally colored.

Observe also that the recursive calls in line **recurse** need not be synchronized. In other words, it is not necessary (nor desired) that the processors start recursion at exactly the same moment in time. During recursion, when the calls reach the communication phase of the algorithm, they will automatically be synchronized in waiting for data from each other.

Clearly, the subgraph defined by the good edges and their incident vertices can be colored concurrently by the available processors. In particular, each processor is responsible for coloring its set of vertices as shown in line **color vertex** of Algorithm 2. In determining the least available color to a vertex, each processor maintains a Boolean vector *Bcolors*. This vector is indexed with the colors and initialized with all values set to "true". Then

Algorithm 3: Sequential $\Delta + 1$Coloring

Input: M the initial input size per processor, subgraph $G' = (V', E')$ of a base graph $G = (V, E)$ with $|E'| \leq M$ and lists F_v of forbidden colors for the vertices.

find allowed **foreach** *processor* P_i **do**
- Let $U_i' = U_i \cap V'$ be the vertices of G' that are stored on P_i;
- For each $v \in U_i'$ let $d(v)$ be the degree of v in G';
- A_v = ComputeAllowedColors(v,$d(v)$,$\{F_v\}_v$) (see Algorithm 4);

Communicate E' and all lists A_v to P_1;

color sequentially **for** *processor* P_1 **do**
- Collect the graph G' together with the lists A_v;
- Color each vertex in G' with least available color;
- Send the resulting colors back to the corresponding processors;

communicate **foreach** *processor* P_i **do**
- Inform all neighbors of U_i' of the colors that have been assigned;
- Receive the colors from the other processors and update the lists F_v accordingly;

when processing a vertex v, the entries of $Bcolors$ corresponding to v's list of forbidden colors are set to "false". After that, the first item in $Bcolors$ that still is true is looked for and chosen as the color of v. Then, the vector is reset by assigning all its modified values the value "true" for future use.

After a processor has colored a vertex, it communicates the color information to processors hosting a neighbor. In each timeslot the messages to the other processors are grouped together, see **send messages** and **receive messages**. This way at most $p - 1$ messages are sent per processor per timeslot.

3.2.1 The base case

The base case of the recursion is handled by a call to Algorithm 3 (see **base case** in Algorithm 2). Note that the sizes of the lists F_v of forbidden colors that the vertices might have collected during higher levels of recursion may actually be too large and their union might not fit on a single processor. To handle this situation properly, we proceed in three steps as shown in Algorithm 3. Notice that Algorithm 3 is the same routine called in the initial phase of Algorithm 1.

In the step **find allowed**, for each vertex $v \in V'$ a short list of *allowed* colors A_v is computed. Observe that a vertex v can always be colored using one color from the set $\{1, 2, \ldots, d(v)+1\}$, where $d(v)$ is the degree of v. Hence a list of $d(v) + 1$ allowed colors suffices to take all restrictions of forbidden

Algorithm 4: Compute Allowed Colors

Input: v together with its actual degree $d(v)$ and its (unordered) list F_v of forbidden colors; A Boolean vector *colors* with all values set to *true*.

Output: a sorted list A_v of the least $d(v)+1$ allowed colors for v

foreach $c \in F_v$ **do** Set $colors[c] = false$;

for $(c = 1; |A_v| < d(v); ++c)$ **do** **if** $colors[c]$ **then** $A_v = A_v + c$;

foreach $c \in F_v$ **do** Set $colors[c] = true$;

colors into account. Using a similar technique as described in **color vertex** of Algorithm 2, we can obtain a sorted list A_v of allowed colors for v in time proportional to $|F_v| + d(v)$. This is done by the call to Algorithm 4 in line **find allowed**. Then in the step **color sequentially**, the vertices of the input graph are colored sequentially using their computed lists of allowed colors. In the final step **communicate**, the color information of the vertices is communicated.

3.2.2 Load balancing

In this section we address the issue of load balancing. In Algorithm 2, three matters that potentially result in an uneven load balance are (i) high variation in the degrees of the vertices, (ii) high variation in the sum of the degrees in the timeslots, and (iii) the recursive calls on the subgraphs that go into recursion. The following three paragraphs are concerned with these points.

Handling high degree vertices Whereas for the shared memory algorithm differences in degrees of the vertices that are colored in parallel just causes a slight asynchrony in the execution of the algorithm, in a CGM setting it might result in a severe load imbalance and even in memory overflow of a processor.

Line **group vertices** of Algorithm 2 groups the vertices into $k \leq p$ timeslots of about equal degree sum. If the variation in the degrees of the vertices is too large, such a grouping would not be even. For example, if we have one vertex of very large degree, it would always dominate the degree sum of its time slot thereby creating imbalance. So, we have to make sure that the degree of each vertex is fairly small, namely smaller than $\lceil M'/q \rceil$ where q is a parameter of Algorithm 5. Observe that the notion of 'small' degree depends on the input size M' and thus may change during the course of the algorithm. This is why we need to have the line **high degree** in every

Algorithm 5: Handle High Degree Vertices

Input: Subgraph $G' = (V', E')$ of a base graph $G = (V, E)$ with M' edges per processor such that $\Delta_{G'} \leq M'$, lists F_v of forbidden colors for the vertices and a parameter q.

foreach *processor* P_i **do**
- find all $v \in U_i$ with degree higher than M'/q (Note: all degrees are less than N/p);
- send the names and the degrees of these vertices to P_1;

for *processor* P_1 **do**
- Receive lists of high degree vertices;
- Group these vertices into $k' \leq q$ timeslots $W_1, ..., W_{k'}$ of at most p vertices each and of a degree sum of at most $2N/p$ for each timeslot;
- Communicate the timeslots to the other processors;

foreach *processor* P_i **do**
- Receive the timeslots for the high degree vertices in U_i;
- Communicate these values to all the neighbors of these vertices;
- Receive the corresponding information from the other processors;
- Compute $E_{t,i}$ for $t = 1, \ldots, k'$ where one endpoint is in $W_t \cup U_i$;

for $t = 1$ ***to*** k' **do**
- Let $E_t = \bigcup_{1 \leq i \leq p} E_{t,i}$ and denote by $G_t = (W_t, E_t)$ the induced subgraph of high degree vertices of timeslot t;
- Sequential$\Delta + 1$Coloring(G_t,$\{F_v\}_v$) (see Algorithm 3);

recursive call and not only at the top level call. Note that q is a multiple of k, the number of timeslots of Algorithm 2.

Thus, the high degree vertices that we indeed have to treat in each recursive call are those vertices v with $\lceil M'/q \rceil < deg(v) \leq M'$. Such vertices are handled using Algorithm 5, which essentially divides the set of high degree vertices into $k' \leq q$ timeslots and colors each of the subgraphs induced by these timeslots sequentially.

Grouping vertices into timeslots Algorithm 6 partitions the vertices into k timeslots. It does so by first dividing the set of vertices into groups of size k and then distributing the vertices of each group into the distinct timeslots. Observe that no communication is required during the course of this algorithm.

The partition obtained with this algorithm is relatively balanced.

Lemma 1 *On each processor P, the difference of the degree sums of the vertices in any two timeslots is at most the maximum degree over all vertices*

Algorithm 6: Group Vertices Randomly into Timeslots

```
Input: V' the set of vertices, k
foreach processor P_i do
    Radix sort its vertices according to their descending degrees;
    Let v_1, ..., v_s be this order of the vertices;
    for i = 0, ..., ⌈s/k⌉ - 1 do
        Let j_1, ..., j_k be a random permutation of the values 1, ..., k;
        Assign v_{ik+1}, ..., v_{(i+1)k} to timeslots j_1, ..., j_k respectively;
```

that P holds.

Proof: Since the vertices are considered in descending order of their degrees, the difference in degree sums between two timeslots is maximized when one of the timeslots always receives the vertex with the highest degree in the group and the other the smallest. In group i, the vertex of highest degree is v_{ik+1} and the one of smallest degree is $v_{(i+1)k}$. Thus we can estimate the difference as follows:

$$\sum_{i=0}^{\lceil \frac{s}{k} \rceil - 1} deg(v_{ik+1}) - \sum_{i=0}^{\lceil \frac{s}{k} \rceil - 1} deg(v_{(i+1)k}) \leq \sum_{i=0}^{\lceil \frac{s}{k} \rceil - 1} deg(v_{ik+1}) - \sum_{i=0}^{\lceil \frac{s}{k} \rceil - 2} deg(v_{(i+1)k+1})$$

which is in turn bounded by $deg(v_1)$, where v_1 has the maximum degree over all vertices that P holds.

□

From Lemma 1 and from the fact that we do not have high degree vertices, it follows that the sum of the degrees of the vertices in any timeslot is between $\frac{M'}{2k}$ and $\frac{3M'}{2k}$.

Balancing during recursion In Algorithm 2, unless proper attention is paid, the edges of the subgraph that goes into recursion may not be evenly distributed among the processors. To address this, we suggest an algorithm that ensures that G_{rec}, the graph that goes into recursion in Algorithm 2, is evenly distributed among the processors. See Algorithm 7.

Obviously Algorithm 7 runs in time proportional to the input size on each processor and has a constant number of supersteps.

4 Average case analysis

In this section we provide an average case analysis of Algorithm 2. In Section 5 we show how to replace the randomized algorithm, Algorithm 6,

Algorithm 7: Balance

Input: Graph $G' = (V', E')$, such that each $v \in V'$ has $deg_{G'}(v) \leq |E'|/p$.

Output: A redistribution of V' and E' on the processors such that each processor handles no more than $M' = 2|E'|/p$ edges.

Initialize a distributed array Deg indexed by V' that holds the degrees of all vertices;
Do a prefix sum on Deg and store this sum in a similar array Pre;
foreach *processor* P_i **do**
  foreach $v \in V' \cap U_i$ **do**
    Let $j \in \{1, ..., p\}$ be such that $jM' \leq Pre[v] < (j+1)M'$;
    Send v and its adjacent edges to processor P_j;
foreach *processor* P_i **do**
  receive the corresponding vertices and edges

by a deterministic one. Every lemma in this section refers to Algorithm 2 unless stated otherwise.

Lemma 2 *For any edge $\{v, w\}$, the probability that $t_v = t_w$ is at most $\frac{1}{k}$.*

Proof: Consider Algorithm 6. We distinguish between two cases. The first is the case where v and w reside on different processors. In this case, the choices for the timeslots of v and w are clearly independent, implying that the probability that w is in the same timeslot as v is $\frac{1}{k}$.

The same argument applies for the case where v and w reside on the same processor but are not processed in the same group. Whenever they are in the same group, they are never placed into the same timeslot. Therefore, the overall probability is bounded by $\frac{1}{k}$. □

Lemma 3 *The expected sum total of the number of edges of all subgraphs going into recursion in* ***recurse*** *is at most $\frac{|E'|}{k}$.*

Proof: The expected total number of edges going into recursion is equal to the expected total number of bad edges. The latter is in turn equal to $\sum_{e \in E'} prob(e \text{ is bad})$, which by Lemma 2 can be bounded by $\frac{|E'|}{k}$. □

Lemma 4 *The expected overall size of the subgraphs at the ith recursion level is at most N/k^i, with at most M/k^i per processor.*

Proof: Notice that the choices of timeslots between two successive recursion levels may not be independent. However, the dependency that may occur actually reduces the number of bad edges even more. This can be seen from a similar argument as that of Lemma 2: vertices that are in the same group of the degree sequence in Algorithm 6 are forced to be separated into two

different timeslots. For all others, the choices are again independent.

Thus, the total number of edges going into recursion can be immediately bounded by N/k^i. The fact that it is also balanced across the processors is due to Algorithm 7. ◻

Lemma 5 *The expected sum total of the sizes of all the subgraphs handled by any processor during Algorithm 2 (including all recursions) is $O(M)$.*

Proof: By Lemma 4, the expected sum of the sizes of these graphs is bounded by

$$\sum_{i=0}^{\infty} k^{-i} M = \frac{k}{k-1} M \leq 2M, \tag{2}$$

for all $k \geq 2$. Thus, the total expected size in all the steps per processor is $O(M)$. ◻

Lemma 6 *For any $1 < k \leq p$, the expected number of supersteps is at most quadratic in p.*

Proof: The expected recursion depth of our algorithm is the minimum value d such that $N/k^d \leq M = N/p$, which implies $k^d \geq p$, i.e. $d = \lceil \log_k p \rceil$. The total number of supersteps in each call (including the supersteps in Algorithm 5) is $c \cdot k$, for some constant $c \geq 1$. The constant c captures the following supersteps:

- some to handle high degree vertices,
- one to propagate the chosen timeslots,
- some to balance the edges inside each timeslot,
- one to propagate the colors for each timeslot.

Thus, the total number of supersteps on recursion level i is $c \cdot k^i$ and the expected number of supersteps is bounded as follows

$$\sum_{i=1}^{\lceil \log_k p \rceil} c \cdot k^i \leq c \cdot k^{\log_k p + 1} = c \cdot k \cdot p. \tag{3}$$

◻

Lemma 7 *The expected overall work involved in* ***base case*** *is $O(M)$.*

Proof: Algorithm 3 on input $G' = (V', E')$ and lists of forbidden colors F_v has overall work and communication cost proportional to $|G'|$ and the size of the lists F_v.

There are $k^{\lceil \log_k p \rceil}$ expected calls to Algorithm 3 in Algorithm 2; therefore P_1 is expected to handle $k^{\lceil \log_k p \rceil} \frac{N}{kp^2}$ edges and $k^{\lceil \log_k p \rceil} \frac{N}{kp^2} \leq k^{1+\log_k p} \frac{N}{kp^2} \leq kp \frac{N}{kp^2} = M$. This implies an expected work and communication cost of $O(M)$ for **base case**. ◻

Lemma 8 *The expected overall work per processor involved in* ***high degree*** *is $O(M)$.*

Proof: In Algorithm 2, in the first call to Algorithm 5 ($M' = M$), every processor holds at most $q = 2k$ high degree vertices (i.e vertices v of degree $deg_v(G)$ such that $\frac{N}{pq} < deg_v(G) \leq \frac{N}{p}$). Otherwise, it would hold more than $(M/q) \cdot q = M$ edges. So, overall, there are at most $p \cdot q$ such vertices for the first level of recursion. Processor P_1 distributes these $O(p^2)$ vertices onto k' timeslots such that each timeslot has a degree sum of at most $2N/p = 2M$. Thus, each timeslot induces a graph of expected size $2M/k'$. Subsequently, sequential $\Delta+1$-coloring is called for the subgraph induced by each timeslot, for total work $O(M) = O(M')$.

By induction we see that in the ith level of recursion, if a vertex v is of high degree, its degree $deg_v(G_{rec})$ has to be $\frac{M'}{q} < deg_v(G_{rec}) \leq M'$. Using the same argument as the one above, it can be shown that the total work to handle these vertices is $O(M')$.

From Lemma 5, the total expected work in all the steps per processor is $O(M)$. □

Lemma 9 *The expected overall work per processor involved in* ***group vertices*** *is $O(M)$.*

Proof: Observe that the radix sort can be done in $O(M')$, since the sort keys are less than M'. The random permutations can easily be computed locally in linear time.

Again, Lemma 5 proves the claim. □

Theorem 1 *For any $1 < k \leq p$, the expected work, communication, and idle time per processor of Algorithm 2 is within $O(M)$. In particular, the expected total run-time per processor is $O(M)$.*

Proof: From Lemma 6 we see that the expected number of supersteps is $O(p^2)$; hence by inequality (1) the expected communication overhead generated in all the supersteps is $O(M)$.

We proceed by showing that the work and communication that a processor has to perform in Algorithm 2 is a function of the number of edges on that processor, i.e. M. Inserting a new forbidden color into an unsorted list F_v can be done in constant time. Since an edge contributes an item to the list of forbidden colors of one of its incident vertices at most once, the size of such a list is bounded by the degree of the vertex. Thus, the total size of these lists on any of the processors will never exceed the input size M' (recall that vertices of degree greater than $\frac{N}{p}$ have been handled in the preprocessing step).

As discussed in Section 3.2, a Boolean vector $Bcolors$ is used in determining the color to be assigned to a vertex. In the absence of high degree vertices no list F_v will be longer than $\frac{M'}{q}$ and hence the size of $Bcolors$ need not exceed $\frac{M'}{q} + 1$. Even when this restriction is relaxed, as shown in Section 3.2.2, we need at most p colors for vertices of degree greater than N/p and need not add more than $\Delta' + 1$ colors, where Δ' is the maximum degree among the remaining vertices ($\Delta' \leq M'$). Overall, this means that we have at most $p + M' + 1$ colors and hence the vector $Bcolors$ still fits on a single processor. So, $Bcolors$ can be initialized in a preprocessing step in time $O(M')$.

After that, coloring any vertex v can be done in time proportional to the size of A_v, which is bounded by the degree of v. Thus, the overall time spent per processor in coloring vertices is $O(M')$. By Lemma 5, the expected total time (including recursions) per processor is $O(M)$.

Lemmas 7, 8, and 9 show that the contributions of **base case**, **high degree**, and **recurse** in Algorithm 2 are within $O(M)$ per processor, proving the claim on the total amount of work per processor.

As for processor idle time, observe that the bottleneck in all the algorithms as presented is the sequential processing of parts of the graphs by processor 1. Since the total run time (of Algorithm 3) on processor 1 is expected to be $O(M)$, the same expected bound holds for the idle time of the other processors. □

5 An add-on to achieve a good worst case behavior

So far, for a possible implementation of our algorithm, we have some degree of freedom in choosing the number of timeslots k. If our goal is just to get results based on *expected* values as shown in Section 4, we can avoid recursion by choosing $k = p$ and by replacing the recursive call in Algorithm 2 by a call to Sequential$\Delta + 1$Coloring$(G_{rec}, \{F_v\}_v)$ (Algorithm 3). We can do this since by Lemma 4 the expected size of G_{rec} is N/k which in this case means $N/p = M$, implying that G_{rec} fits on one processor. The resulting algorithm would have cp supersteps, for some integer $c > 1$; see Lemma 6.

To get a deterministic algorithm with a good worst case bound we choose the other extreme, namely $k = 2$, and replace the call to the randomized Algorithm 6, in line **group vertices** of Algorithm 2, by a call to Algorithms 8 and 9. This will enable us to bound the number of edges that go into recursion. We need to distinguish between two types of edges: *internal* and *external* edges. Internal edges have both of their endpoints on the same

Algorithm 8: Deterministically group the vertices on processor P_i into $k = 2$ buckets.

```
HandleHighDegreeVertices(G,{F_v}_v,8) (see Algorithm 5);
initialize bucket[0] and bucket[1] to empty;
foreach vertex v do
    determine the number of edges connecting v to bucket[0] and
    bucket[1], resp;
    insert v in the bucket to which it has the least number of edges;
    if this bucket is full then
        put the remaining vertices in the other bucket;
        return ;
return ;
```

processor while external edges have their endpoints on different processors.

First we argue that internal edges are handled by the call to Algorithm 8. Then we show that external edges are handled by the subsequent call to Algorithm 9. For internal edges, the following two points need to be observed.

1. The vertices are grouped into two timeslots of about equal degree sum.
2. Most of the internal edges are good.

To achieve the first goal, Algorithm 8 first calls Algorithm 5 to get rid of vertices with degree $\geq M/8$. The constant 8 is somewhat arbitrary and could be replaced by any constant $k' > 2$ depending on the needs of an implementation. Algorithm 8 groups the vertices of internal edges according to (1) and (2) above into *bucket*[0] and *bucket*[1] that will form the two timeslots. A bucket is said to be *full* when the degree sum of its vertices becomes greater than $M/2$.

Proposition 1 *Suppose $\gamma_i M$ of the edges on processor P_i are external ($0 \leq \gamma_i M \leq M$). Then after an application of Algorithm 8 at least $(\frac{1}{4} - \frac{\gamma_i}{2})M$ of the edges on P_i are good internal edges and each bucket has a degree sum of at most $\frac{5M}{8}$.*

Proof: Considering the fact that each vertex is of degree less than $M/8$, the claim for the degree sum is immediate.

To see the lower bound on the number of good internal edges, consider the bucket B that became full. The vertices in B have a degree sum of at least $M/2$ and at least $(\frac{1}{2} - \gamma_i)M$ of these edges are internal. We claim that at least half of these internal edges are good.

For the following argument, suppose that an edge is considered only when its second endpoint is placed into a bucket. We distinguish between

Algorithm 9: Determine an ordering on the $k = 2$ buckets on each processor.

```
foreach processor P_i do
    foreach edge (v, w) do inform the processor of w about the bucket of v;
    for s = 1 ... p do
        for r, r' = 0, 1 do set m_is^{rr'} = 0;
    foreach edge (v, w) do add 1 to m_is^{rr'}, where P_s is the processor
        of w and r and r' are the buckets of v and w;
    Broadcast all values m_is^{rr'} for s = 1, ..., p to all other processors;
    inv[1] = false;
    for s = 2 to p do
        A^|| = 0; A = 0;
        for s' < s do
            if ¬inv[s'] then
                A^|| = A^|| + m_ss'^00 + m_ss'^11;
                A = A + m_ss'^01 + m_ss'^10
            else
                A^|| = A^|| + m_ss'^01 + m_ss'^10;
                A = A + m_ss'^00 + m_ss'^11
        if A < A^|| then inv[s] = true;
        else inv[s] = false;
```

two types of internal edges. *Early* edges join vertices both of which have been put into a bucket before the bucket was full, while edges for which at least one of the endpoints was placed thereafter are called *late* edges .

First, observe that until one of the two buckets becomes full, both buckets have more good internal edges than bad internal edges. So, at least one half of the early edges are good. But, notice that all the late edges that have one endpoint in B are also good. This is the case since the second endpoint of a late edge is never placed in B.

Therefore, overall, there are at least $\frac{1}{2}(\frac{1}{2} - \gamma_i)M$ good internal edges. □

To handle the external edges we add a call to Algorithm 9 right after the call to Algorithm 8. This algorithm counts the number $m_{is}^{rr'}$ of edges between all possible pairs of buckets on different processors, and broadcasts these values to all processors. Then a quick iterative algorithm is executed on each processor to ascertain as to which of the processor's two buckets represents the first and second timeslot.

After having decided the order in which the buckets are processed on

processors $P_1, \ldots, P_{i-1}$, we compute two values for processor P_i: $A^{||}$ the number of external bad edges if we would keep the numbering of the buckets as the timeslot numbering, and A the corresponding number if we would interchange them. Depending on which value is less, the order of the two buckets of processor P_i is kept as-is or exchanged.

Using the same type of argument as the one above, we get the following remark.

Remark 10 *Algorithm 9 ensures that overall at least $\frac{1}{2}$ of the external edges are good.*

Note that the above statement is true for the whole edge set, but not necessarily for the set of edges on each processor.

Proposition 2 *Algorithms 8 and 9 run with linear work and communication and in a constant number of supersteps. The assignment of timeslots they make is such that at least $\frac{1}{4}$ of the edges are good.*

Proof: For the number of good edges, let γ_i be the fraction of external edges of processor P_i. The total amount of good edges can now be bounded from below by

$$\frac{1}{2}\left(\sum_{i=1}^{p} \gamma_i\right) M + \sum_{i=1}^{p}\left(\frac{1}{4} - \frac{\gamma_i}{2}\right) M = \sum_{i=1}^{p} \frac{M}{4} = \frac{pM}{4} = \frac{N}{4}. \tag{4}$$

For the complexity claim, observe that because of the load balancing done in line **balance load** of Algorithm 2 (see Section 3.2.2) all processors hold the same amount (up to a constant factor) M' of edges. □

This implies that the recursion depth is $\lceil \log_{\frac{4}{3}} p \rceil$ (which is greater than $\lceil \log_2 p \rceil$ of the average case). Moreover, more edges go into recursion here than in the average case and therefore the work and total communication costs are slightly greater than the costs for the average case, but still within $O(M)$.

6 Conclusion

We have presented a randomized as well as a deterministic Coarse Grained Multicomputer coloring algorithm that colors the vertices of a general graph G using at most $\Delta + 1$ colors, where Δ is the maximum degree in G. We showed that on a p-processor CGM model our algorithms require a parallel time of $O(\frac{|G|}{p})$ and a total work and overall communication cost of $O(|G|)$. These bounds correspond to the average case for the randomized version and to the worst case for the deterministic variant. To the best of our knowledge, our algorithms are the first parallel coloring algorithms with good speedup for a large variety of architectures.

In light of the fact that LF$\Delta+1$-coloring is P-complete, a CGM LF$\Delta+1$-coloring algorithm, if found, would be of significant theoretical importance. Brent's scheduling principle shows that anything that works well on PRAM should, in principle, also work well on CGM or similar models (although the constants that are introduced might be too big to be practical). But the converse may not necessarily be true. There are P-complete problems (problems where we can't expect exponential speedup on PRAM) that have polynomial speedup [25]. It can be envisioned that such problems may as well have efficient CGM algorithms. In this regard, our CGM $\Delta+1$-coloring algorithm might be a first step towards a CGM LF$\Delta+1$-coloring algorithm.

We also believe that, in general, designing a parallel algorithm on the CGM model is of practical relevance. In particular, we believe that the algorithms presented in this paper should have efficient and scalable implementations (i.e. implementations that yield good speedup for a wide range of N/p).

Such implementations would also be of interest in other contexts. One example is the problem of finding *maximal independent sets*. Notice that in our algorithms the vertices colored by the least color always constitute a maximal independent set in the input graph. In general, considering $G_{\geq i}$, the subgraph induced by the color classes $i, i+1, \ldots$, we see that color class i always forms a maximal independent set in the graph $G_{\geq i}$.

Acknowledgements

We are grateful to the anonymous referees for their helpful comments and pointing out some errors in an earlier version of this paper. Isabelle Guérin Lassous and Jens Gustedt would like to acknowledge that part of this work was accomplished within the framework of the "opération CGM" of the parallel research center *Centre Charles Hermitte* in Nancy, France.

References

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

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

[3] B. Chelbus, K. Diks, W. Rytter, and T. Szymacha. Parallel complexity of lexicographically first problems for tree-structured graphs. In A. Kreczmar and G. Mirkowska, editors, *Mathematical Foundations of Computer Science 1989*, volume 379 of *LNCS*, pages 185–195. Springer-Verlag, 1989.

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

[5] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceeding of the fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, CA, 1993.

[6] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.

[7] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Tenth ACM Symposium on Theory of Computing*, pages 114–118, San Diego, CA, 1978.

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

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

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

[11] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *8th Annual ACM symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 1–12, Padua, Italy, June 1996.

[12] R. Greenlaw, H.J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, New York, USA, 1995.

[13] I. Guérin Lassous, J. Gustedt, and M. Morvan. Feasability, portability, predictability and efficiency: Four ambitious goals for the design and

implementation of parallel coarse grained graph algorithms. Technical Report 3885, INRIA, 2000.

[14] I. Guérin Lassous, J. Gustedt, and M. Morvan. Handling graphs according to a coarse grained approach: Experiments with MPI and PVM. In Jack Dongarra, Péter Kacsuk, and N. Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *LNCS*, pages 72–79. Springer Verlag, 2000.

[15] P. Hajnal and E. Szemerédi. Brooks coloring in parallel. *SIAM Journal on Discrete Mathematics*, 3(1):74–80, 1990.

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

[17] M. Karchmer and J. Naor. A fast parallel algorithm to color a graph with D colors. *Journal of Algorithms*, 9(1):83–91, 1988.

[18] H. J. Karloff. An NC algorithm for Brook's theorem. *Theoretical Computer Science*, 68(1):89–103, 16 October 1989.

[19] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science Volume A: Algorithms and Complexity*, pages 869–942. Elsevier, 1990.

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

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

[22] F. Manne. A parallel algorithm for computing the extremal eigenvalues of very large sparse matrices (extended abstract). In *Para98*, volume 1541 of *LNCS*, pages 332–336. Springer-Verlag, 1998.

[23] J. Naor. A fast parallel coloring of planar graphs with five colors. *Information Processing Letters*, 25(1):51–53, 20 April 1987.

[24] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[25] J. S. Vitter and R. A. Simons. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Transactions on Computers*, C-35(5):403–418, 1986.

PRO: a Model for Parallel Resource-Optimal Computation *

Assefaw Hadish Gebremedhin[†] Isabelle Guérin Lassous[‡]
Jens Gustedt[§] Jan Arne Telle

Abstract

We present a new parallel computation model that enables the design of resource-optimal scalable parallel algorithms and simplifies their analysis. The model rests on the novel idea of incorporating relative optimality as an integral part and measuring the quality of a parallel algorithm in terms of granularity.

Key words: Parallel computers, Parallel models, Parallel algorithms, Complexity analysis

---

*Research supported by IS-AUR 02-34 of The Aurora Programme, a France-Norway Collaboration Research Project of The Research Council of Norway, The French Ministry of Foreign Affairs and The Ministry of Education, Research and Technology.

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

[‡]LIP & INRIA Rhone-Alpes, France. Isabelle.Guerin-Lassous@inria.fr

[§]LORIA & INRIA Lorraine, France. gustedt@loria.fr

1 Introduction

One of the challenges in parallel processing is the development of a general purpose and effective model of parallel computation. Unlike the realm of sequential computation, where the Random Access Machine (RAM) has successfully served as a standard computational model, no such single unifying model exists in the field of parallel computation. From an algorithmic point of view, the performance of a sequential algorithm is adequately evaluated using its execution time making the RAM powerful enough for analysis and design. On the other hand, the performance evaluation of a parallel algorithm involves several metrics, the most important of which are *speedup*, *optimality* (or *efficiency*), and *scalability*. Speedup and optimality are *relative* in nature as they are expressed with respect to some sequential algorithm. The notion of relativity is also relevant from a practical point of view. A parallel algorithm is often not designed from scratch, but rather starting from a sequential algorithm.

We believe that a parallel computation model should incorporate the most important performance evaluation metrics of parallel algorithms as the RAM does for sequential algorithms. In light of this, the objective of the current work is to develop a model that simplifies the design and analysis of *resource-optimal scalable* parallel algorithms.

In an interesting survey paper [21], Maggs *et al.* suggest that an ideal parallel computation model be designed within "the philosophy of *simplicity* and *descriptivity* balanced with *prescriptivity*". The Parallel Resource-Optimal (PRO) computation model proposed here is developed within this spirit. The key features of the PRO model that distinguish it from existing parallel computation models are *relativity*, *resource-optimality*, and a new quality measure referred to as *granularity*.

Relativity pertains to the fact that the design and analysis of a parallel algorithm in PRO is done relative to the time and space complexity of a *specific* sequential algorithm. Consequently, the parameters involved in the analysis of a PRO-algorithm are the number of processors p, the input size n, and the time and space complexity of the reference sequential algorithm A_{seq}.

A PRO-algorithm is required to be both time- and space-optimal (hence resource-optimal). A parallel algorithm is said to be time- (or work-) optimal if the overall computation and communication cost involved in the algorithm is proportional to the time complexity of the sequential algorithm used as a reference. Similarly, it is said to be space-optimal if the overall memory space used by the algorithm is of the same order as the memory usage of

the underlying sequential version. As a consequence of its time-optimality, a PRO-algorithm always yields *linear speedup* relative to the reference sequential algorithm; *i.e.*, the ratio between the sequential and parallel runtime is a linear function of p.

The *quality* of a PRO-algorithm is measured by the range of values p can assume while linear speedup is maintained. This range is captured by an attribute of the model called the granularity function $\text{Grain}(n)$. In other words, a PRO-algorithm with granularity $\text{Grain}(n)$ is required to be fully *scalable* for all values of p such that $p = O(\text{Grain}(n))$. The granularity function $\text{Grain}(n)$ determines the quality of one PRO-algorithm over another relative to the same sequential time and space complexity. The higher the function value $\text{Grain}(n)$ the better the algorithm. Note that since optimality (consequently linear speedup) is 'hard-wired' into the model, the runtime cannot be a quality measure for a PRO algorithm. However, in a sense, the time and space complexity of the reference sequential algorithm A_{seq} can also be seen as a quality measure of the PRO-algorithm. This means that the selection of the reference sequential algorithm is of significant importance.

The rest of the paper is organized as follows. In Section 2 we give an overview of existing parallel computation models and highlight their limitations. In Section 3 the PRO model is presented in detail and in Section 4 it is compared with a selection of existing parallel models. In Section 5 we illustrate how the model is used in design and analysis using the matrix multiplication problem as an example. In Section 6 we give a PRO-algorithm for one-to-all broadcast, as an example of a primitive communication routine found in a potential PRO library. Finally, we conclude the paper in Section 7 with some remarks.

2 Existing models and their limitations

There exists a plethora of parallel computation models in the literature. On the theoretical end, we find the Parallel Random Access Machine (PRAM) model [8, 17] which in its simplest form posits a set of p processors, with global shared memory, executing the same program in lockstep. In this model, every processor can access any memory location at unit cost of time regardless of the memory location. This assumption is in obvious disagreement with the reality of practical parallel computers.

However, despite its serious limitation of being an 'idealized' model of parallel computation, the standard PRAM model still serves as a theoretical framework for investigating the maximum possible computational parallelism in a given task. Specifically, on this model, the NC versus P-complete

dichotomy [14] is used to reflect the ease/hardness of finding a parallel algorithm for a problem. Recall that NC denotes the class of problems which have PRAM-algorithms with polylogarithmic runtime and polynomial number of processors in the input size. A problem is said to be P-complete if an NC-algorithm for it would imply that all polynomial time sequential problems have NC-algorithms. The problem of whether or not $P = NC$ has long been an open problem.

The NC versus P-complete dichotomy has its own practical limitations. First, P-completeness does not depict a full picture of non-parallelizability since the runtime requirement for an NC parallel algorithm is so stringent that the classification is confined to the case where up to polynomial number of processors in the input size is available (fine-grained setting). For example, there are P-complete problems for which less ambitious, but still satisfactory, runtime can be obtained by parallelization in PRAM [23]. In a fine-grained setting, since the number of processors p is a function of the input size n, it is customary to express speedup as a function of n. Thus the speedup obtained using an NC-algorithm is sometimes referred to as exponential. In a coarse-grained setting, *i.e.*, the case where n and p are orders of magnitude apart, speedup is expressed as a function of only p and some recent results [4, 7, 9, 15] show that this approach is practically relevant. Second, an NC-algorithm is not necessarily work-optimal, and thus not resource-optimal considering runtime and memory space as resources that one wants to use efficiently. Third, even if we restrict ourselves to work-optimal NC-algorithms and apply Brent's scheduling principle, which says an algorithm in theory can be simulated on a machine with fewer processors by only a constant factor more work, implementations of PRAM algorithms often do not reflect this optimality in practice [6]. This is mainly because the PRAM model does not account for non-local memory access (communication), and a Brent-type simulation relies heavily on cheap communication.

To overcome the defects of the PRAM related to its failure of capturing real machine characteristics, the advocates of shared memory models propose several modifications to the standard PRAM model. In particular, they enhance the standard PRAM model by taking practical machine features such as memory access, synchronization, latency and bandwidth issues into account. Pointers to the PRAM family of models can be found in [21].

Critics of shared memory models argue that the PRAM family of models fail to capture the nature of existing parallel computers with *distributed* memory architectures. Examples of distributed memory computational models suggested as alternatives include the Postal Model [2] and the Block

Distributed Memory (BDM) model [18]. Other categories of parallel models such as low-level, hierarchical memory, and network models are briefly reviewed in [21].

A more recent category of parallel models is that of 'bridging' models, a notion popularized by Valiant with his introduction of the Bulk Synchronous Parallel (BSP) model [22]. The BSP model is a distributed memory coarse-grained model in which parallel computation proceeds as a sequence of barrier synchronized supersteps where local computation and communication are distinct rather than intermingled phases. Culler *et al.* [5] extended the BSP model by allowing asynchronous execution and better accounting for communication overhead. Their model is coined LogP, an acronym for the four parameters involved. A common feature of the BSP, LogP, and other related models is their lack of simplicity: each model involves relatively many parameters making analysis and design of algorithms cumbersome.

The Coarse Grained Multicomputer (CGM) model [4, 7] was later proposed in an effort to retain the advantages of BSP while keeping the model simple (making the number of parameters fewer). The BSP and its special case CGM have been the primary inspirations for our model. Thus, we believe that many optimal CGM and BSP algorithms can easily be adapted to PRO.

The PRO model attempts to partially address the limitations of existing parallel models highlighted in the foregoing discussion and compromises between theoretical and practical considerations. One of its advantages from a theoretical point of view is that it is a step forward towards the identification of the class of problems for which 'good' parallel algorithms exist in a more realistic (practical) way than the existing NC versus P-complete classification.

Our main goal in suggesting the PRO model is to enable the development of scalable and resource-optimal parallel algorithms and to simplify their analysis. The model identifies the salient features of a parallel algorithm that make its practical scalability and optimality highly likely. In this regard, it can be considered as a set of 'guidelines' for the algorithm designer in the quest for developing scalable and efficient parallel algorithms. Hence, PRO can be seen as a mix of a parallel computation model and a parallel algorithm design scheme which makes it biased towards the software side in its role as a bridging model.

3 The PRO model

The PRO model is an algorithm *design* and *analysis tool* used to deliver a practical, optimal, and scalable parallel algorithm relative to a specific sequential algorithm whenever this is possible. Let $\mathrm{Time}(n)$ and $\mathrm{Space}(n)$ denote the time and space complexity of a specific sequential algorithm for a given problem with input size n. The PRO model is defined to have the following attributes.

Machine The underlying machine is assumed to consist of p processors with $M = O(\frac{\mathrm{Space}(n)}{p})$ private memory each, interconnected by some communication network (or shared memory) that can deliver messages in a point-to-point fashion. A message can consist of several machine words.

Coarseness We assume that $p \leq M$, *i.e.*, the size of the local memory of each processor is big enough to store p words.

Execution For any value $p = O(\mathrm{Grain}(n))$, a PRO algorithm,

- consists of $O(\frac{\mathrm{Time}(n)}{p^2})$ *supersteps*. A superstep consists of a local computation phase and an interprocessor communication phase. In particular, in each superstep, each processor
  - sends at most one message to every other processor,
  - sends and receives at most M words in total, and pays a unit of time per word sent and received,
  - performs local computation, and pays a unit of time per operation,
- has parallel runtime $\mathrm{Time}(n, p) = O(\frac{\mathrm{Time}(n)}{p})$.

Note that the *granularity* function $\mathrm{Grain}(n)$ is a quality measure of a PRO-algorithm.

As discussed in the LogP paper [5], technological factors are forcing parallel systems to converge towards systems formed by a collection of essentially complete computers connected by a robust communication network. The *machine* model assumption of PRO is consistent with this convergence and maps well on several existing parallel computer architectures. The memory requirement $M = O(\frac{\mathrm{Space}(n)}{p})$ ensures that the space utilized by the underlying sequential algorithm is uniformly distributed among the p processors. Since we may, without loss of generality, assume that $\mathrm{Space}(n) = \Omega(n)$, the implication is that the private memory of each processor is large enough

to store its 'share' of the input and any additional space the sequential algorithm might require. When $\text{Space}(n) = \Theta(n)$, note that the input data must be uniformly distributed on the p processors. In this case the machine model assumption of PRO is similar to the assumption in the CGM model [7].

The *coarseness* assumption $p \leq M$ is consistent with the structure of existing parallel machines and machines to be built in the foreseeable future. The assumption is required to simplify the implementation of collecting messages (from possibly all other processors) on a single processor.

The *execution* of a PRO-algorithm consists of a sequence of *supersteps* (or rounds). The *length* of (time spent in) a superstep on each processor is determined by the sum of the time used for communication and the time used for local computation. The length of a superstep s in the parallel algorithm seen as a whole, denoted by $\text{Time}_s(n, p)$, is the maximum over the lengths of the superstep on all processors. We can conceptually think as if the supersteps are synchronized by a barrier set at the end of the longest superstep across the processors. However, note that in PRO the processors are not in reality required to synchronize at the end of each superstep. The parallel runtime $\text{Time}(n, p)$ of the algorithm is the sum of the lengths of all the supersteps. Notice that the hypothetical barriers result in only a constant factor more time compared with an analysis that does not assume the barriers.

In PRO, since a processor sends at most one message to every other processor in each superstep, each processor is involved in at most $2(p - 1)$ messages per superstep. Therefore, the requirement $\text{Steps} = O(\frac{\text{Time}(n)}{p^2})$ on the number of supersteps implies that the overall time paid per processor for *communication overhead* and *latency* is $O(\text{Time}(n)/p)$ and hence can be neglected from the analysis since our goal is to achieve an $O(\text{Time}(n)/p)$ parallel runtime. Notice that the bandwidth restriction of the underlying architecture which in turn contributes to the communication cost is accounted for since each processor pays a unit of time per word sent and received. This is not an unrealistic assumption noting that the network throughput (accounted in machine words) on modern architectures such as high performance clusters is relatively close to the CPU frequency and to the CPU/memory bandwidth.

The condition $\text{Time}(n, p) = O(\frac{\text{Time}(n)}{p})$ requires that a PRO-algorithm be optimal and yield linear speedup relative to the sequential algorithm used as a reference. This requirement ensures the potential practical use of the parallel algorithm.

Observation 1 *A PRO algorithm relative to a sequential algorithm with runtime $O(Time(n))$ and space requirement $O(Space(n))$ has maximum granularity $Grain(n) = O(min\{\sqrt{Space(n)}, \sqrt{(Time(n)}\}) = O(\sqrt{Space(n)})$. A PRO algorithm that achieves this is said to have optimal grain.*

Observation 1 is due to the limit on the memory size of each processor, the coarseness assumption, and the bound on the number of supersteps. The limit on the size of the private memory of each processor ($M = O(\frac{\text{Space}(n)}{p})$) together with the coarseness assumption $p \leq M$ imply $p = O(\sqrt{\text{Space}(n)})$. The fact that the number of supersteps of a PRO-algorithm should be $\text{Steps} = O(\text{Time}(n)/p^2)$, gives $p = O(\sqrt{(\text{Time}(n)/\text{Steps})})$ upon resolving and we clearly have $\text{Steps} \geq 1$. Finally, note that $\text{Time}(n) \geq \text{Space}(n)$, since an algorithm has to at least read the input.

Since a PRO-algorithm yields linear speedup for any $p = O(\text{Grain}(n))$, a result like Brent's scheduling principle is implicit for these values of p. But Observation 1 shows that we cannot start with an arbitrary number of processors and efficiently simulate on a fewer number. So Brent's scheduling principle does not hold with full generality in the PRO model, which is in accordance with practical observations.

The design of a PRO-algorithm may sometimes involve subroutines for which there do not exist sequential counterparts. Examples of such tasks include communication primitives such as broadcasting, data (re)-distribution routines, and load balancing routines. Such routines are often required in various parallel algorithms. With a slight abuse of notation, we call such parallel routines PRO-algorithms if the overall computation and communication cost is linear in the input size to the routines.

4 Comparison with other models

In this section we compare the PRO model with PRAM, QSM, BSP, LogP, and CGM. Our tabular format for comparison is inspired by a similar presentation in [13], where the Queuing Shared Memory (QSM) model is proposed. The columns of Table 1 are labeled with the names of the selected models in our comparison and some relevant features of a model are listed along the rows.

The synchrony assumption of the model is indicated in the row labeled *synch.* Lock-step indicates that the processors are fully synchronized at each step (of a universal clock), without accounting for synchronization. Bulk-synchrony indicates that there can be asynchronous operations between synchronization barriers. The row labeled *memory* shows how the model views

	PRAM [8]	QSM [13]	BSP [22]	LogP [5]	CGM [4]	PRO
synch.	lock-step	bulk-synch.	bulk-synch.	asynch.	asynch.	asynch.
memory	sh.	sh.	dist.	dist.	priv.	priv.
commun.	SM	SM	MP	MP	MP/SM	MP/SM
parameters	n	p, g, n	p, g, L, n	p, g, l, o, n	p, n	p, n, A_{seq}
granularity	fine	fine	coarse	fine	coarse	Grain(n)
speedup	NA	NA	NA	NA	NA	$\Theta(p)$
optimal	NA	NA	NA	NA	NA	rel. A_{seq}
quality	time	time	time	time	rounds	Grain(n)

Table 1: Comparison of parallel computational models

the memory of the parallel computer: sh. indicates globally accessible shared memory, dist. stands for distributed memory and priv. is an abstraction for the case where the only assumption is that each processor has access to private (local) memory. In the last variant the whole memory could either be distributed or shared. The row labeled *commun.* shows the type of interprocessor communication assumed by the model. Shared memory (SM) indicates that communication is effected by reading from and writing to a globally accessible shared memory. Message-passing (MP) denotes the situation where processors communicate by explicitly exchanging messages in a point-to-point fashion. The MP abstraction hides the details of how the message is routed through the interprocessor communication network.

The parameters involved in the model are indicated in the row labeled *parameters*. The number of processors is denoted by p, n is the input size, A_{seq} is the reference sequential algorithm, l is the communication cost (latency), L is a single parameter that accounts for the sum of latency (l) and the cost for a barrier synchronization, g is the bandwidth gap, and o is the overhead associated with sending or receiving a message. Note that the machine characteristics l and o are are taken into account in PRO, even though they are not explicitly used as parameters. Latency is taken into consideration since the length of a superstep is determined by the sum of the computational and communication cost. Communication overhead is hidden by the PRO-requirement that states Steps $= O(\frac{\text{Time}(n)}{p^2})$.

The row labeled *granularity* indicates whether the model is fine-grained, coarse-grained or a more precise measure is used. We say that a model is coarse-grained if it applies to the case where $n \gg p$ and call it fine-grained if it relies on using up to a polynomial number of processors in the input size. In PRO granularity is exactly the quality measure Grain(n), and appears as one of the attributes of the model.

The rows labeled *speedup* and *optimal* indicate the speedup and resource optimality requirements imposed by the model. Whenever these issues are not directly addressed by the model or are not applicable, the word 'NA' is

used. Note that these requirements are 'hard-wired' in the model in the case of PRO. The label 'rel. A_{seq}' means that the algorithm is optimal relative to the time and space complexity of A_{seq}. We point out that the goal in the design of algorithms using the CGM model [7, 4] is usually stated as that of achieving optimal algorithms, but the model *per se* does not impose an optimality requirement.

The last row indicates the *quality* measure of an algorithm designed using the different models. For all other models except CGM and PRO, the quality measure is running time. In CGM, the number of supersteps (rounds) is usually presented as a quality measure. In PRO the quality measure is granularity, one of the features that make PRO fundamentally different from all existing parallel computation models.

5 Algorithm example: matrix multiplication

In this section we illustrate how the PRO model is used, by starting from a given sequential algorithm and then designing and analyzing a parallel algorithm relative to it. We use the standard matrix multiplication algorithm with three nested for-loops as an example. This example is chosen for its simplicity and since our objective at this stage is to illustrate the use of a new model rather than solving a "difficult" problem.

Consider the problem of computing the product C of two $m \times m$ matrices A and B (input size $n = m^2$). We want to design a PRO-algorithm relative to the standard sequential matrix multiplication algorithm which has $\text{Time}(n) = O(n^{\frac{3}{2}})$ and $\text{Space}(n) = O(n)$.

We assume that the input matrices A and B are distributed among the p processors $P_0, ..., P_{p-1}$ so that processor P_i stores rows (respectively columns) $\frac{m}{p} \cdot i + 1$ to $\frac{m}{p} \cdot (i+1)$ of A (respectively B). The output matrix C will be row-partitioned among the p processors in a similar fashion. Notice that with this data distribution each processor can, without communication, compute a block of $\frac{m^2}{p^2}$ of the $\frac{m^2}{p}$ entries of C expected to reside on it. In order to compute the next block of $\frac{m^2}{p^2}$ entries, processor P_i needs the columns of matrix B that reside on processor P_{i+1}. In each superstep the processors in the PRO algorithm will therefore exchange columns in a round-robin fashion and then each will compute a new block of results. Note that each column exchanged in a superstep constitutes one single message. Note also that the initial distribution of the rows of matrix A remains unchanged. In Algorithm 1, we have organized this sequence of computation and communication steps in a manner that meets the requirements of the

Algorithm 1: Matrix multiplication

Input: Two $m \times m$ matrices A and B. The rows (columns) of A (B) are divided into m/p contiguous blocks, and stored on processors $P_0, P_1, \dots P_{p-1}$ respectively

Output: The product matrix C where the rows are stored in contiguous blocks across the p processors

for *superstep* $s = 1$ *to* p **do**
- **foreach** *processor* P_i **do**
  - P_i computes the local sub-matrix product of its rows and current columns;
  - $P_{(i+1) mod\ p}$ sends its current block of columns to P_i;
  - P_i receives a new current block of columns from $P_{(i+1) mod\ p}$;

PRO model.

Algorithm 1 has p supersteps (Steps $= p$). In each superstep, the time spent in locally computing each of the m^2/p^2 entries is $\Theta(m)$ resulting in local computing time $\Theta(m^3/p^2) = \Theta(n^{\frac{3}{2}}/p^2)$ per superstep. Likewise, the total size of data (words) exchanged by each processor in a superstep is $\Theta(m^2/p) = \Theta(n/p)$. Thus, the length of a superstep s is $\text{Time}_s(n,p) = \Theta(n^{\frac{3}{2}}/p^2 + n/p)$. Note that for $p = O(\sqrt{n})$, $\text{Time}_s(n,p) = \Theta(n^{\frac{3}{2}}/p^2)$. Hence, for $p = O(\sqrt{n})$, the overall parallel runtime of the algorithm is

$$\text{Time}(n,p) = \sum_{\text{Steps}} \Theta(n^{\frac{3}{2}}/p^2) = \Theta(n^{\frac{3}{2}}/p) = \Theta(\text{Time}(n)/p). \qquad (1)$$

Noting that $\text{Space}(n) = \Theta(n)$, we see that the memory restriction of the PRO model is respected, *i.e.*, each processor has enough memory size to handle the transactions. In order to be able to neglect communication overhead, the condition on the number of supersteps, which in this case is just p, should be met. In other words, we need $p = O(\text{Time}(n)/p^2) = O(n^{\frac{3}{2}}/p^2)$, which is true for $p = O(\sqrt{n})$. Thus the granularity function of the PRO-algorithm is $\text{Grain}(n) = \sqrt{n}$.

In summary,

Lemma 1 *Multiplication of two m by m matrices has a PRO-algorithm with $Grain(n) = m$ relative to a sequential algorithm with $Time(n) = m^3$ and $Space(n) = m^2$ (input size $n = m^2$).*

From Observation 1, we note that Algorithm 1 achieves optimal granularity. Note that on a relaxed model, where the assumption that $p \leq M$ is not present, the strong regularity of matrix multiplication and the exact

knowledge of the communication pattern allows for algorithms that have an even finer granularity than m. For example, a systolic matrix multiplication algorithm has a granularity of m^2. However, PRO is intended to be applicable for general problems and practically relevant parallel systems.

6 Communication primitive example: one-to-all broadcast

A good parallel computation model should have a selection of algorithms for primitive communication tasks available in its algorithm design toolbox. The PRO model is intended to meet this demand, but for lack of space we give only one example.

In this section we illustrate how the PRO model allows optimal one-to-all broadcasting among its processors. Since there is no sequential basis algorithm in this case, we want an algorithm whose overall communication and computation cost is linear in the input and output sizes. More precisely, we consider the situation where the input consists of a vector of size m on a single processor and the output should be a copy of this vector on each of the p processors, and we want an algorithm that achieves this in $O(m)$ time using $O(m)$ memory on each processor. See Algorithm 2.

Algorithm 2: One-to-All Broadcast

Input: A vector V of size m on processor P_0
Output: A copy of V on each processor

S1 P_0 divides V into p equal sized parts;
P_0 sends the ith part of V to processor P_i, for each $0 < i \leq p$;
foreach *processor* P_i, $i > 0$ **do** processor P_i receives the ith part from P_0;

S2 **foreach** *processor* P_i **do**
  P_i sends out the ith part to P_j, for each $j \neq i$ and $0 < j \leq p$.
foreach *processor* $P_j, j \neq 0$ **do**
  P_j receives the ith part from P_i, for each $i \neq j$ and $0 < i \leq p$

Lemma 2 *PRO Algorithm 2 implements a one-to-all broadcast of m memory words in two supersteps using $O(m)$ time and $O(m)$ space per processor, for any number of processors $p \leq m$.*

Proof: First, we note that the algorithm correctly broadcasts the desired vector V, while observing the space restriction, in two supersteps. We turn to the timing. In step **S1** processor P_0 in total sends out $(p-1)m/p$ words

and each of the other processors receives a message of size m/p. In step **S2** processor P_i in total sends out $\frac{p-2}{p}m$ words. Processor P_j, $j \neq 0$, in total receives $\frac{p-1}{p}m$ words.

The total time is dominated by the communication which is

$$(p-1)m/p + m/p + \frac{p-2}{p}m + \frac{p-1}{p}m = \quad (2)$$

$$m/p(p+p-2+p-1) < 3m \quad (3)$$

for total time $O(m)$ as claimed. □

7 Conclusion

We have introduced a new parallel computation model (called PRO) that enables the development of efficient scalable parallel algorithms and simplifies the complexity analysis of such algorithms.

The distinguishing feature of the PRO model is the novel focus on relativity, resource-optimality, and a new quality measure (granularity). In particular, the model requires a parallel algorithm to be both time- and space-optimal relative to an underlying sequential algorithm. Having optimality as a built-in requirement, the quality of a PRO-algorithm is measured by the maximum number of processors that could be used while the optimality of the algorithm is maintained.

The focus on relativity has theoretical as well as practical justifications. From a theoretical point of view, the performance evaluation metrics of a parallel algorithm includes speedup and optimality, both of which are always expressed relative to some sequential algorithm. Moreover, there is an inherent asymmetry between sequential and parallel computation. A parallel algorithm would always imply a sequential algorithm, whereas the converse is usually not true. Thus, in a sense, it is natural to think of an underlying sequential algorithm whenever one speaks of a parallel algorithm. From a practical point of view, one notes that the development of a parallel algorithm is often built on some known sequential algorithm.

The fact that optimality is incorporated as a requirement in the PRO model enables one to concentrate only on parallel algorithms that are practically useful.

However, the PRO model is not just a collection of some 'ideal' features of parallel algorithms, it is also a means to achieve these features. In particular, the attributes of the model capture the salient characteristics of a parallel algorithm that make its practical optimality and scalability highly likely.

In this sense, it can also be seen as a parallel algorithm design scheme. Moreover, the simplicity of the model eases analysis.

We believe that the PRO model is a step forward towards the identification of problems for which 'practically good' parallel algorithms exist. Much work remains to be done, and we hope that other members of the research community will join in. As a first item on the agenda, the PRO model needs to be tested for compatibility with already existing practical parallel algorithms.

Acknowledgments We are grateful to the anonymous referees for their helpful comments.

References

[1] A. G. Alexandrakis, A. V. Gerbessiotis, D. S. Lecomber, and C. J. Siniolakis. Bandwidth, space and computation efficient PRAM programming: The BSP approach. In *Proceedings of the SUP'EUR '96 Conference, Krakow, Poland*, September 1996.

[2] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the Postal Model for message passing systems. In *The 4th annual ACM symposium on parallel algorithms and architectures*, pages 13–22, July 1992.

[3] R. P. Brent. The parallel evaluation of generic arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[4] E. Caceres, F. Dehne, A. Ferreira, P. Locchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In *The 24th International Colloquium on Automata Languages and Programming*, volume 1256 of *LNCS*, pages 390–400. Springer Verlag, 1997.

[5] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on principles and practice of parallel programming, San Diego, CA*, May 1993.

[6] F. Dehne. Coarse grained parallel algorithms. *Algorithmica Special Issue on "Coarse grained parallel algorithms"*, 24(3/4):173–176, 1999.

[7] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400, 1996.

[8] S. Fortune and J. Wyllie. Parallelism in random access machines. In *10th ACM Symposium on Theory of Computing*, pages 114–118, May 1978.

[9] A. H. Gebremedhin, I. Guérin Lassous, J. Gustedt, and J. A. Telle. Graph coloring on a coarse grained multiprocessor. In Ulrik Brandes and Dorothea Wagner, editors, *WG 2000*, volume 1928 of *LNCS*, pages 184–195. Springer-Verlag, 2000.

[10] A. V. Gerbessiotis, D. S. Lecomber, C. J. Siniolakis, and K. R. Sujithan. PRAM programming: Theory vs. practice. In *Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing, Madrid, Spain*. IEEE Computer Society Press, January 1998.

[11] A. V. Gerbessiotis and C. J. Siniolakis. A new randomized sorting algorithm on the BSP model. Technical report, New Jersey Institute of Technology, 2001.

[12] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[13] P. B. Gibbons, Y. Matias, and V. Ramachandran. Can a Shared-Memory Model Serve as a Bridging Model for Parallel Computation? *Theory of Computing Systems*, 32(3):327–359, 1999.

[14] R. Greenlaw, H.J. Hoover, and W. L. Ruzzo. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, New York, 1995.

[15] I. Guérin Lassous, J. Gustedt, and M. Morvan. Handling graphs according to a coarse grained approach: Experiments with MPI and PVM. In Jack Dongarra, Péter Kacsuk, and N. Podhorszki, editors, *7th European PVM/MPI Users' Group Meeting*, volume 1908 of *LNCS*, pages 72–79. Springer Verlag, 2000.

[16] K. Hawick et al. High performance computing and communications glossary. see `http://nhse.npac.syr.edu/hpccgloss/`.

[17] J. Jájá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

[18] J. JáJá and K. W. Ryu. The Block Distributed Memory model. *IEEE Transactions on Parallel and Distributed Systems*, 8(7):830–840, 1996.

[19] R. M. Karp and V. Ramachandran. Parallel Algorithms for Shared-Memory Machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, Algorithms and Complexity, pages 869–941. Elsevier Science Publishers B.V., Amsterdam, 1990.

[20] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science*, 71(1):95–132, march 1990.

[21] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: A survey and synthesis. In *28th HICSS*, volume 2, pages 61–70, January 1995.

[22] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[23] J. S. Vitter and R. A. Simons. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Transactions on Computers*, C-35(5):403–418, 1986.