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, Allwrignt 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 number 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);

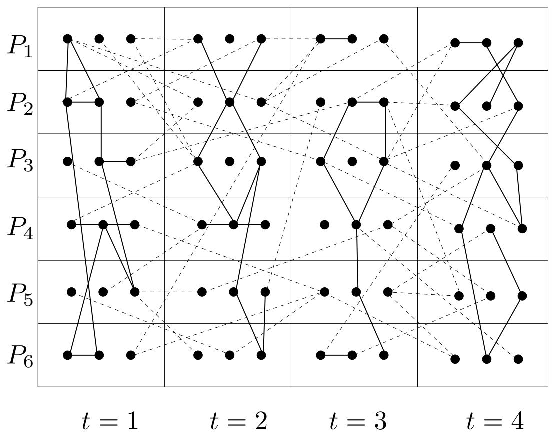

Figure 1: Graph on 72 vertices distributed onto 6 processors and 4 timeslots.

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 $\Delta_{G'} \le M'$, 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 $v_1, \ldots, v_s$ be this order of the vertices;
  for $i = 0, \ldots, \lceil \frac{s}{k} \rceil - 1$ **do**
    Let $j_1, \ldots, j_k$ be a random permutation of the values $1, \ldots, k$;
    Assign $v_{ik+1}, \ldots, v_{(i+1)k}$ to timeslots $j_1, \ldots, j_k$ 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.