Graph Coloring on a Coarse Grained Multiprocessor

(extended abstract)

Assefaw Hadish Gebremedhin[1], Isabelle Guérin Lassous[2],
Jens Gustedt[3], and Jan Arne Telle[1]

[1] Univ. of Bergen, Norway. {assefaw|telle}@ii.uib.no
[2] INRIA Rocquencourt, France. Isabelle.Guerin-Lassous@inria.fr
[3] LORIA & INRIA Lorraine, France. Jens.Gustedt@loria.fr

Abstract. We present the first efficient algorithm for a coarse grained multiprocessor that colors a graph G with a guarantee of at most $\Delta_G + 1$ colors.

1 Introduction and Overview

The problem of graph coloring is crucial both for the applications of graph algorithms to real world problems *and* for the domain of parallel graph algorithms itself. For the latter, graph colorings using a bounded number of colors are often used in a theoretical setting to ensure the independence of tasks that are to be accomplished on the vertices of a graph, i.e. since the color classes form independent sets that don't interact each one of them can be treated in parallel. For a long time, no efficient parallel implementation of a graph coloring heuristic with good speedups was known, see Allwright et al. (1995). However, in a recent result, Gebremedhin and Manne (1999a, 1999b) present an algorithm and an implementation for a *shared memory computer* that proves to be theoretically and practically efficient with good speedups.

In this paper we make this successful approach feasible for a larger variety of architectures by extending it to the more general setting of coarse grained multiprocessors (CGM), see Dehne et al. (1996). This model of parallel computation makes an abstraction of the interconnection network between the processors of a parallel machine (or network) and tries to capture 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 feasible, portable, predictable and efficient, see Guérin Lassous et al. (2000).

This paper is organized as follows. In the next section we review the coarse grained models of parallel computation and the basics of graph coloring heuristics. Then, we present our algorithm together with an average case analysis of its time and work complexity. Finally, we show how to handle high degree vertices and how to alter the algorithm to achieve the same good time and work complexity also in the worst-case.

1.1 Coarse Grained Models of Parallel Computation.

In recent years several efforts have been made to define models of parallel (or distributed) computation that are more realistic than the classical PRAM models. In contrast to the PRAM, these new models are *coarse grained*, i.e. they assume that the number of processors p and the size of the input N of an algorithm are orders of magnitudes apart, $p \ll N$. By that assumption these models map much better on existing architectures where in general the number of processors is at most some thousands and the size of the data that are to be handled goes into millions and billions.

This branch of research got its kick-off with Valiant (1990) introducing the so-called *bulk synchronous parallel machine*, BSP, and was refined in different directions for example by Culler et al. (1993) to LogP, and by Dehne et al. (1996) to CGM.

We place ourselves in the context of CGM which seems to be well suited for a design of algorithms that are not too dependent on an individual architecture. We summarize the assumptions of this model as follows:

- All algorithms perform in so-called supersteps. A superstep consists of one phase of interprocessor communication and one phase of local computation.
- All processors have the same size $M = O(N/p)$ of memory.
- The communication network between the processors can be arbitrary.

The goal when designing an algorithm in this model is to keep the individual workload, time for communication and idle time of each processor within $T/s(p)$ where T is the runtime of the best sequential algorithm on the same data and $s(p)$, the *speedup*, is a function that should be as close to p as possible. To be able to do so, it is considered as good idea to keep the number of supersteps of such an algorithm as low as possible, preferably $o(M)$.

The rationale for that is that for the communication time there are at least two invariants of the architecture that come into play: the *latency*, i.e. the minimal time a communication needs to *startup* before any data reach the other end, and the *bandwidth*, i.e. the overall throughput per time unit of the communication network for large chunks of data. In any superstep there are at most $O(p)$ communications for each processor and so a number of supersteps of $o(M)$ ensures that the latency can be neglected for the performance analysis of such an algorithm. The bandwidth restriction of a specific platform must still be observed, and here the best strategy is simply to reduce the communication volume as much as possible. See Guérin Lassous et al. (2000) for an introduction and overview on algorithms, code and experiments within the coarse grained setting.

As a legacy from the PRAM model it is usually assumed that the number of supersteps should be polylogarithmic in p, but there seems to be no real world rationale for that. In fact, no relationship of the coarseness models to the complexity classes NC^k have been found, and algorithms that simply ensure a number of supersteps that are a function of p (and not of N) perform quite well in practice, see Goudreau et al. (1996).

To be able to organize the supersteps well it is natural to assume that each processor can keep a vector of p-sized data for each other processor and so the coarseness requirement translates into

$$p^2 < M \approx N/p \tag{1}$$

1.2 Graph Coloring.

A graph coloring is a labeling of the vertices of a graph $G = (V,E)$ with positive integers, the *colors*, such that no two neighbors obtain the same color. An important parameter of such a coloring C is the number of colors $\chi(C)$ that the coloring uses. This can be viewed equivalently as searching for a partition of the vertex set of the graph into *independent sets*. Even though coloring a graph with the fewest number of colors is an NP-hard problem, in many practical and theoretical settings a coloring using a bounded number of colors, possibly far from the minimum, may suffice. Particularly in many PRAM graph algorithms, a bounded coloring (resp. partition into independent sets) is needed as a subroutine. However, up to recently no reasonable coloring had shown to perform well in practical parallel settings.

One of the simplest but quite efficient sequential heuristics for coloring is the so-called *list coloring*. It iterates over $v \in V$ and colors v with the least color that has not yet been assigned to one of its neighbors. It is easy to see that this heuristic always uses at most $\Delta_G + 1$ colors, where $\Delta_G = \max_{v \in V}\{\text{degree of } v\}$. A parallelization of list coloring has been shown by Gebremedhin and Manne (1999b) to perform well both in theory and by experiment on shared memory machines. They show that distributing the vertices evenly among the processors and running list coloring concurrently on the processors, while checking for color compatibility with already colored neighbors, creates very few conflicts: the probability that two neighbors are colored at exactly the same instance of the computation is quite small. Their algorithm was proven to behave well on expectation and the conflicts could easily be resolved thereafter with very low cost.

1.3 The distribution of data on the processors

Even more than for sequential algorithms, a good organization of the data is crucial for the efficiency of parallel code. We will organize the graphs that we handle as follows:

- Every processor P_i is responsible for a specific subset U_i of the vertices. For a given vertex v this processor is denoted by P_v.
- Every edge $\{v,w\}$ is represented as arcs (v,w), stored at P_v, and (w,v), stored at P_w.
- For every 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 searching for P_w.
- The arcs are sorted lexicographically and stored as a linked list per vertex.

For convenience, we will also assume that the input size N for any graph algorithm is in fact equal to the amount of edges $|E|$. Up to a constant which corresponds to the encoding size of an edge this will always be achieved in graphs that don't have isolated vertices. If the input that we receive is not of the desired form it can be efficiently tranformed into one by the following steps:

- Generate two arcs for each edge as described above,
- Radix sort (see Guérin Lassous et al. (2000) for a CGM radix sort) the list of arcs such that each processor receives the arcs (v,w) if it is responsible for vertex w,
- Let every processor note its identity on these sibling arcs,
- Radix sort the list of arcs such that every processor receives its proper arcs (arcs (v,w) if it is responsible for vertex v).

On a somewhat simplified level, the parallelization of list coloring for shared memory machines as given by Gebremedhin and Manne (1999b) worked by tackling the list of vertices numbered from 1 to n in a 'round robin' manner: at a given time $1 \leq t \leq \frac{n}{p}$ processor P_i colors vertex $(i-1)\cdot\frac{n}{p}+t$. The shared memory assumptions ensure that P_i may access the color information of any vertex at unit cost of time. The only problems that can occur are with the neighbors that are in fact handled at exactly the same time. Gebremedhin and Manne (1999b) show that the number of such conflicts is small on

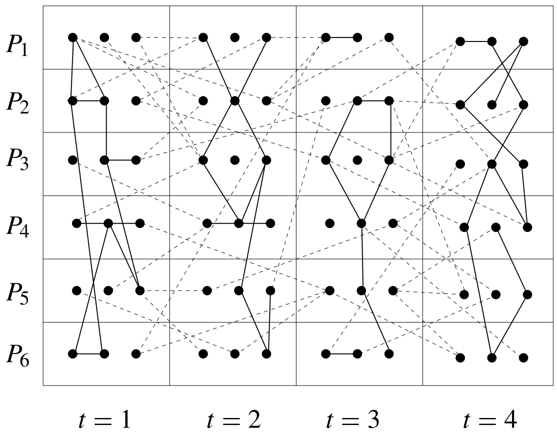

Fig. 1. Graph on 72 vertices distributed onto 6 processors and 4 timeslots.

expectation, and that they can easily be handled a posteriori. In a coarse grained setting we have to be more careful about the access to data that are situated on other processors.

2 The Algorithm

As the best sequential algorithm for an approximate graph coloring is linear in the size of the graph $|G|$, our aim is to design a parallel algorithm in CGM with a work (total number of local computations) per processor in $O(\frac{|G|}{p})$ and overall communication costs in $O(|G|)$.

Algorithm 1 colors any graph G such that $\Delta_G \leq M$ recursively. Section 3 shows how to get rid of high degree vertices. The first type of high degree vertices are the vertices v such that $deg(v) > \frac{N}{p}$. The second type of high degree vertices are the vertices v such that $\frac{N}{pk} < deg(v) \leq \frac{N}{p}$. Note that the first call to Algorithm 1 is applied on the whole graph G ($G' = G$).

In Algorithm 1, we group the vertices that are to be handled on each processor into k different *timeslots*, see the figure and **group vertices** in the algorithm. The number $1 < k \leq p$ of timeslots is a parameter of our algorithm. For each such timeslot we group the messages to the other processors together, see **send to neighbors** and **receive from neighbors**, and thus in general we produce few, even if possibly large, messages. Figure 1 shows a graph on 72 vertices distributed onto 6 processors and 4 timeslots.

A naive parallel list coloring will respect adjacencies between timeslots, but may violate adjacencies listed in bold, inside a timeslot. Our algorithm avoids this by coloring the bold subgraphs recursively until they are small enough to fit onto a single processor. In the recursive call, a vertex will belong to the same processor but probably to a different timeslot. Note that in general some processors may receive more vertices than others.

By proceeding like this, we may generate more conflicts than in the shared memory algorithm; we may accidentally color two neighbors in the same timeslot with the same

Algorithm 1: List coloring on a CGM with p processors

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 and lists F_v of forbidden colors for the vertices.

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

```
bottom                  if the size of G' is less than N/(kp^2) then solve the problem sequentially on processor P_1;
                        else
high degree                 Get rid of high degree vertices of second type (see Section 3);
group vertices              Each processor P_i groups its vertices U_i into k timeslots U_{i,t}, t = 1,...,k, such that
                            the degree sum in all timeslots is about the same;
                            For each vertex v denote the index of its timeslot by t_v;
                            for t = 1 to k do
                                foreach processor P_i do
identify conflicts                  Consider all arcs e = (v,w) with v ∈ 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;
resolve conflicts                   Recursively color the graph (⋃V_{S_i}, ⋃S_i);
color timeslot                      foreach uncolored vertex v with t_v = t do color v with least possible color;
send to neighbors                   foreach arc (v,w) with v ∈ U_i, t_v = t and t_w > t do collect the color of v for
                                    P_w in a send buffer;
                                    send out the tuples (w,color);
receive from neighbors              receive the colors from the other processors;
                                    foreach arc (v,w) with w ∈ U_i, t_v = t and t_w > t do add the color of v to F_w;
```

color. We find it convenient to resolve these conflicts *beforehand*, see **identify conflicts** and **resolve conflicts**, by a recursive call to the same algorithm. We must ensure that these recursive calls do not produce a blow-up in computation and communication, and in fact the main part of the technical details in the paper are dedicated to ensure this.

In the recursive calls we 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 is initially empty. An important issue for the complexity bounds will be that we will only add individual forbidden colors to F_v as the knowledge about them 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 **resolve conflicts** are not issued synchronized between the different processors: it is not necessary (nor desired) that the processors start recursion at exactly the same moment in time. When the calls reach the communication parts of the algorithm during recursion, they will be synchronized automatically when waiting for the data of each other.[1]

Another issue that we have to face is the possible degree variation among the vertices. Whereas for the shared memory algorithm, different degrees of the vertices that are handled in parallel just causes a slight asynchronicity of the execution of the algo-

[1] In fact, when writing this the authors got more and more convinced of the importance of asynchronicity between those of us working in Europe and the one visiting Australia.

rithm, in a CGM setting it might result in a severe imbalance of charge and even in a memory overflow of individual processors. We will see in Section 3 how to handle this.

2.1 An Average Case Analysis

To get a high level understanding of the algorithm, let us first assume that we deal with the best of all worlds, that is:

- we never have high degree vertices of first and second type (d is a high degree if $d > \frac{M'}{k}$),
- the events of having an edge between a pair of vertices are all random and totally independent of each other.

In Section 3, we show how to handle high degree vertices and in Section 4 we show what processing we have to add if we don't assume randomness for the edges.

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

Proof. The expected size of the degree sums of the timeslots $U_{i,t}$ is the same. Since there are k timeslots, when fixing the timeslot of v the probability that w is in the same timeslot on its processor is $\frac{1}{k}$. □

Lemma 2. *The expected size of all subgraphs over all $t = 1,\dots,k$ in* ***resolve conflicts*** *is N/k.*

Proof. Every processor has M edges so each timeslot is expected to contain M/k edges, M/k^2 of which may create conflicts. So on each processor, considering all the timeslots, we expect M/k conflict edges and in total N/k. □

Lemma 3. *For any value $1 < k \leq p$ the expected number of supersteps is linear in p.*

Proof. Each call can initiate k recursive calls. The maximal recursion depth of our algorithm is the minimum value d such that $N/k^d \leq M = N/p$, i.e. $k^d \geq p$, i.e. $d = \lceil \log_k p \rceil$. The total number of supersteps in each call is $c \cdot k$, for some constant c, one for each timeslot plus some to get rid of high degree vertices. Then, the total number of supersteps on recursion level i is given by $c \cdot k^i$ and so the total number of supersteps is

$$\sum_{i=1}^{\lceil \log_k p \rceil} c \cdot k^i \approx c \cdot k^{\log_k p} = c \cdot p \tag{2}$$

□

Lemma 4. *Besides the cost for* ***bottom*** *and* ***high degree****, for any value of $1 < k \leq p$ the expected work and communication per processor is $O(M)$.*

Proof. First we show that the work and communication that any processor has to perform is a function in the number of its edges, i.e. M.

Inserting new forbidden colors into an unsorted list F_v can be done in constant time per color. Any edge adds an item to the list of forbidden colors of one of its end-vertices at most once, so 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.

Algorithm 2: Solve the problem sequentially on processor P_1

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.

collect colors **foreach** *processor* P_i **do**
- Let $U_i' = U_i \cap V'$ be the vertices that are stored on P_i;
- For each $v \in U_i'$ let $d(v)$ be the degree of v in G';
- Compute a sorted list A_v of the least $d(v)+1$ allowed colors for v;

Communicate E' and all lists A_v to P_1;

solve sequentially **for** *processor* P_1 **do**
- Collect the graph G' together with the lists A_v;
- Color G' sequentially;
- Send the resulting colors back to the corresponding processors;

retransmit colors **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;

To find the least available color in **color timeslot** we then have to process the list as a whole. This happens only once for each vertex v, and so one might hope to get away with just sorting the list F_v. But sorting here can be too expensive, comparison sort would impose a time of $M \log M$ whereas counting sort would lead to $|F_v| + M$ per vertex.

But nevertheless the work for this can be bound as follows. Each processor maintains a Boolean vector *colors* that is indexed with the colors and that will help to decide for a vertex v on the least color to be taken. Since we got rid of high degree vertices we know that no list F_v will be longer than M/k and so a length of $M/k+1$ suffices for *colors*.

Later, when relaxing this condition in Section 3 we will need at most p colors for vertices of degree greater than N/p and ensure to add no more than $\Delta'+1$ colors, where Δ' is the maximum degree among the remaining vertices ($\Delta' \leq M$).

In total this means that we have at most $p+M+1$ colors and so our vector *colors* still fits on a processor. This vector is initialized once with all values “true”. Then when processing a vertex v we run through its list of forbidden colors and set the corresponding items of *colors* to “false”. After that, we look for the first item in *colors* that still is true and choose that color for v. Then, to revert the changes we run through the list again and set all values to “true”. This then clearly needs at most a time of $p+M+1$ plus the sizes of the list, so $O(M)$ time in total.

As seen above, on any processor the total fraction of edges going into recursion is expected to be M/k, so the Main Theorem for divide and conquer algorithms shows that the total costs are $O(M)$. □

2.2 The bottom of recursion

At first one might be tempted to think that the bottom of the recursion should easily stay within the required bounds and only communicates as much data as there are edges in

Algorithm 3: Compute the allowed colors A_v of a vertex v.

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 $true$.

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

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

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

the corresponding subgraph. But such an approach doesn't count for the lists F_v of forbidden colors that the vertices might already have collected during higher levels of recursion. The size of these lists may actually be too large and their union might not fit on a single processor. To take care of that situation we proceed in three steps, see Algorithm 2.

In the first step **collect colors**, for each vertex $v \in V'$ we produce a short list of *allowed* colors. In fact, the idea is that when we color the vertex later on we will not use more than its degree $+1$ colors so a list of $d(v)+1$ allowed colors suffices to take all restrictions of forbidden colors into account. With the same trick as in the previous section, we can get away with a computation time of $|F_v| + d(v)$ to compute the list A_v, see Algorithm 3. It is also easy to see that we can use this trick again when we sequentially color the graph on P_1. We summarize:

Lemma 5. *The bottom of the recursion for graph $G' = (V', E')$ with lists of forbidden colors F_v can be done in a constant number of communication steps with overall work that is proportional to $|G'|$ and the lists F_v and with a communication that is proportional to $|G'|$.*

Note that there will be $k^{\lceil \log_k p \rceil}$ calls to **bottom**, therefore P_1 handles at most $k^{\lceil \log_k p \rceil} \frac{N}{kp^2}$ edges and $k^{\lceil \log_k p \rceil} \frac{N}{kp^2} \leq k^{1+\log_k p} \frac{N}{kp^2} \leq kp \frac{N}{kp^2} = M$. That implies a total time for **bottom** of $O(M)$.

3 Getting Rid of High Degree Vertices

Line **group vertices** of Algorithm 1 groups the vertices into $k \leq p$ timeslots of about equal degree sum. Such a grouping would not be possible if the variation in the degrees of the vertices is too large. For example, if we have one vertex of very large degree, it would always dominate the degree sum of its timeslot and we cannot achieve a balance. So we will ensure that the degrees of all vertices is fairly small, namely smaller than M/k. Observe that this notion of 'small' depends on the input size M and thus the property of being of small degree may change during the course of the algorithm. This is why we have to have the line **high degree** in every recursive call and not only for the top level call. On the other hand this choice of M/k will leave us enough freedom to choose k in the range of $2, \ldots, p$ as convenient.

We distinguish two different kinds of high degree vertices. The **first type** we have to handle in a preprocessing step that is only done once on the top level of recursion. These are vertices for which the degree is greater than $M = N/p$. In fact, these vertices can't have all their arcs stored at one processor alone. Clearly, overall we can have at most

Algorithm 4: Get rid of high degree vertices of second type.

foreach *processor* P_i **do**
- find all $v \in U_i$ with degree higher than M'/k (Note: all degrees 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 $k' \leq k$ 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 U_i;

for $t = 1$ ***to*** k' **do**
- **foreach** *processor* P_i **do** Communicate $E_{t,i}$ to processor P_1;
- **for** *processor* P_1 **do**
  - Receive $E_t = \bigcup_{1 \leq i \leq p} E_{t,i}$ and denote by $G_t = (W_t, E_t)$ the induced subgraph of high degree vertices of timeslot t;
  - Solve the problem for G_t sequentially, see Algorithm 2;

p such vertices, otherwise we would have more than N edges total. Thus the subgraph induced by these vertices has at most p^2 edges. Because of (1) this induced subgraph fits on processor P_1 and a call to Algorithm 2 in a preprocessing step will color it.

The **second type** of high degree vertices, that we indeed have to treat in each recursive call, are those vertices v with $N/(pk) = M'/k < deg(v) \leq M' = N/p$, see Algorithm 4. Every processor holds at most k such vertices, otherwise it would hold more than $(M'/k) \cdot k = M'$ edges. So in total there are at most $p \cdot k$ such vertices.

It is again easy to see that the probability for a vertex v to become a high degree vertex of second type is small: even if it actually has M'/k edges, with high probability only M'/k^2 have their other endpoint in the same timeslot, and so v will not become of high degree on the next recursion level. So on expectation Algorithm 4 will only contribute little to the total number of supersteps, workload and communication.

4 An add-on to achieve a good worst-case behavior

So far for a possible implementation of our algorithm we have a degree of freedom in the number of timeslots k. If we are heading for just a guarantee on expectation as shown above we certainly would not like to bother with recursion and can choose $k = p$. This gives an algorithm that has $3p$ supersteps.

To give a deterministic algorithm with a worst case bound we choose the other extreme, namely $k = 2$. This enables us to bound the number of edges that go into the recursion. We have to distinguish two different types of edges: edges that have both

Algorithm 5: Determine an ordering on the $k = 2$ timeslots on each processor.

```
foreach processor P_i do
    foreach edge (v,w) do inform the processor of w about the timeslot 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 timeslots 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;
```

endpoints on the same processor, *internal* edges, and those that have them on different processors, *external* edges.

Here we only describe how to handle external edges. In fact, the ideas for handling internal edges during the partition of the vertices into the two different timeslots are quite similar, but we omitted them for this short abstract.

To handle the external edges we add a call to Algorithm 5 after **group vertices** in Algorithm 1. This algorithm counts the number $m_{is}^{rr'}$ of edges between all possible pairs of timeslots on different processors, and broadcasts these values to all processors. Then a quick iterative algorithm is executed in parallel on all processors that decides on the processors for which the roles of the two timeslots are interchanged.

After having decided whether or not to interchange the roles of the timeslots on processors $P_1, \ldots, P_{i-1}$ we compute two values for processor P_i: $A^{||}$ the number of edges that would go into recursion if we would keep the role of the two timeslots, and A the same number if we would interchange the role of the two timeslots.

References

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

[Culler et al., 1993] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., and von Eicken, T. (1993). LogP: Towards a Realistic Model of Parallel Computation. In *Proceeding of 4-th ACM SIGPLAN Symp. on Principles and Practises of Parallel Programming*, pages 1–12.

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

[Gebremedhin and Manne, 1999a] Gebremedhin, A. H. and Manne, F. (1999a). Parallel graph coloring algorithms using OpenMP (extended abstract). In *First European Workshop on OpenMP*, pages 10–18, Lund, Sweden.

[Gebremedhin and Manne, 1999b] Gebremedhin, A. H. and Manne, F. (1999b). Scalable, shared memory parallel graph coloring heuristics. Technical Report 181, Department of Informatics, University of Bergen, 5020 Bergen, Norway.

[Goudreau et al., 1996] Goudreau, M., Lang, K., Rao, S., Suel, T., and Tsantilas, T. (1996). Towards efficiency and portability: Programming with the BSP model. In *8th Annual ACM symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 1–12.

[Guérin Lassous et al., 2000] Guérin Lassous, I., Gustedt, J., and Morvan, M. (2000). Feasability, portability, predictability and efficiency: Four ambitious goals for the design and implementation of parallel coarse grained graph algorithms. Technical report, INRIA.

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