

AN EFFICIENT PARALLEL ALGORITHM THAT FINDS INDEPENDENT SETS OF GUARANTEED SIZE

MARK GOLDBERG[†] AND THOMAS SPENCER[‡]

Abstract. Every graph with n vertices and m edges has an independent set containing at least $n^2/(2m+n)$ vertices. We present a parallel algorithm that finds an independent set of this size and runs in $O(\log^3 n)$ time on a CRCW PRAM with $O((m+n)\alpha(m,n)/\log^2 n)$ processors, where $\alpha(n,m)$ is a functional inverse of Ackerman's function. The ideas used in the design of this algorithm are also used to design an algorithm that, with the same resources, finds a vertex coloring satisfying certain minimality conditions.

Key words. Turán's theorem, independent set, NC, graph, parallel computation, deterministic

AMS(MOS) subject classifications. 68Q22, 68R10, 68R05

1. Introduction. This paper presents a fast parallel algorithm that, given a graph G , finds an independent set of G whose size is bounded from below. The bound depends on the number n of vertices and number m of edges of G , and cannot be improved in these terms.

Since constructing a *maximum* independent set is NP-hard, it cannot be solved using a polynomial algorithm unless $P=NP$. Johnson [13] proved that if there is a polynomial-time algorithm that finds an independent set whose size is within a constant factor of optimal, then there is a polynomial approximation scheme for the maximum independent set problem, that is, an algorithm that finds an independent set whose size is within $(1-\epsilon)$ of optimal and whose running time is polynomial for any fixed $\epsilon > 0$. So far, nobody has devised such a polynomial approximation scheme, and it is somewhat unlikely that it exists. In particular, it seems to be unlikely that this approximation problem is in NC. In contrast to this, if we require that the algorithm only find a *maximal* independent set (the *MIS* problem), then the problem becomes polynomial. In fact, a maximal independent set can be easily found sequentially in a linear time. Karp and Wigderson [15] showed that *MIS* is in NC. Starting with their work, a number of parallel algorithms have been proposed to solve this problem [2], [10], [11], [17], [18]. Currently, the most efficient algorithm is presented in [11]; it runs in $O(\log^3 n)$ time on $O((n+m)/\log n)$ processors.

A common drawback of all NC-algorithms for *MIS* mentioned above is that occasionally they can find *too small* a set. Any graph with a large independent set and a vertex adjacent to all other vertices is a potential example of such a situation. This motivates the approach we take here; we require that the algorithm find a *sufficiently large* independent set. Besides the apparent theoretical interest in the question of whether this task can be accomplished in NC, it is conceivable that such an algorithm can be a useful subroutine for solving other problems. For example, in our earlier work [11], the crucial part of the algorithm for *MIS* is a subroutine which finds efficiently in $O(\log n)$ time a matching of sufficiently large, though not necessarily maximum, size. Notice that finding a maximum matching is not known to be in NC.

[†]Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 12180. goldberg@cs.rpi.edu. Supported in part by the National Science Foundation under grant CDA-8805910 and by the National Security Agency under grant MDA904-88-H-2037.

[‡]Department of Mathematics and Computer Science, University of Nebraska at Omaha, Omaha, NE, 68182. spencer@unocss.unomaha.edu. Supported in part by the National Science Foundation under grants CCR-8810609 and CDA-8805910 and by the University Committee on Research, University of Nebraska at Omaha. Most of this was done while the author was at the Department of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, 12180.

Our interpretation of a *sufficiently large* independent set is based on Turán's theorem [21]. It states that every graph with n vertices and m edges contains an independent set of size at least $\lceil n^2/(2m+n) \rceil$; this bound cannot be improved in terms of n and m .

The following sequential algorithm finds an independent set of this size [9].

```

 $I \leftarrow \emptyset$ ;
while  $G$  is not empty begin
   $v \leftarrow$  a vertex of minimum degree in  $G$ ;
   $I \leftarrow I \cup \{v\}$ 
  delete  $v$  and its neighbors from  $G$ ;
end;
```

In this paper, we present a parallel algorithm that finds an independent set of Turán's size in $O(\log^3 n)$ time on a CRCW PRAM with $O((n+m)\alpha(m,n)/\log^2 n)$ processors, where $\alpha(m,n)$ is the inverse of Ackerman's function. We assume that whenever several processors write to the same location at the same time, one of them succeeds. Notice that it was not known whether the problem of finding an independent set of Turán's size is in NC or even in RNC. The first parallel algorithm that finds an independent set of guaranteed size is due to Goldberg [8]. It finds an independent set of size at least $n^2/32m$ in $O(\log^2 n)$ time on an EREW PRAM with $O(n+m)$ processors, provided that $m \geq n/2$. An alternative approach is to delete all vertices of degree at least $4m/n$ and then to find a maximal independent set of the remaining graph. This independent set must contain at least $(n/2)/(4m/n+1) = n^2/(8m+2n)$ vertices.

Our algorithm uses a graph partitioning subroutine that is of independent interest. In the graph partitioning problem, we are asked to divide the vertices of a given graph into two sets, A and A' , so that the number of edges joining a vertex in A to a vertex in A' is maximized; such edges are said to be cut by the partition (A, A') . Erdős [7] proved that every connected graph has a partition that cuts $\lceil m/2 + n/4 - 1/4 \rceil$ edges. His proof leads to a linear time sequential algorithm that appears to be hard to parallelize. For our purpose, we need a slightly better partition. We prove that the only connected graphs for which Erdős' bound cannot be improved are Δ -graphs.

We define Δ -graph to be either an isolated vertex or a connected graph in which every block is an *odd clique*¹. If a connected component C of a graph G is a Δ -graph, then C is a Δ -component of G . A graph that has no Δ -components is Δ -free. A connected graph G is called a *near Δ -graph* if exactly one block of G is an even clique, and all other blocks are odd cliques.

We call a partition (A, A') that cuts at least $\lceil m/2 + n/4 \rceil$ edges a *dividing partition*, provided that each part (A and A') contains at least one-fifth of the total vertices. It is easy to see that Δ -graphs do not have dividing partitions; we prove that all other graphs do. For every Δ -graph, our partition subroutine finds an optimal partition, and for every Δ -free graph, it finds a dividing partition; on a CRCW PRAM with $O((n+m)\alpha(m,n)/\log n)$ processors it runs in $O(\log^2 n)$ time.

An algorithm that finds a dividing partition can also be applied to the *weighted vertex coloring problem*. In the weighted vertex coloring problem, we assign a positive integer to each vertex of a given graph so that no adjacent vertices are assigned the same number and the sum of the numbers assigned is minimized. The sum is called the *weight* of the coloring. A coloring is called *minimum* if it is of the minimal possible weight. A coloring is *minimal* if, for each color k , every vertex of color k is adjacent to some vertex of each color less than k .

One can prove that any minimal coloring has weight at most $m+n$ and that every graph that is the union of disjoint cliques has a minimum weight coloring of weight

¹ The term *odd (even) clique* means a clique with an odd (even) number of vertices.

$m+n$. We call a coloring *light*, if its weight is at most $m+n$. A minimal coloring can be found sequentially in linear time. There are no previously known parallel algorithms that find light vertex colorings. We present a parallel algorithm that finds a light vertex coloring in $O(\log^3 n)$ time on an CRCW-PRAM with $O((n+m)\alpha(m,n)/\log^2 n)$ processors. Any coloring of this weight uses at most $\lceil\sqrt{2m}\rceil$ colors. Another parallel algorithm that colors graphs using $\lceil\sqrt{2m}\rceil$ colors is given in [8]; it runs in $O(\log^3 n)$ time on EREW PRAM and uses $O(m+n)$ processors.

The bottleneck in all three algorithms is finding the blocks of a graph. Tarjan and Vishkin proposed an algorithm that effectively reduces the block finding problem to the connected components finding problem [20]. If the connected components algorithm of Cole and Vishkin [5] is used, the block finding algorithm runs in $O(\log n)$ time on a CRCW PRAM with $O((n+m)\alpha(m,n)/\log n)$ processors. We call the resulting algorithm the CTV-algorithm. Using a more efficient algorithm or an algorithm for a different model of parallel computation (for example, see [12]) will lead to other results. If $T(n,m)$ and $P(n,m)$ are respectively the time and the number of processors required by the biconnected components algorithm, then the partitioning algorithm requires $O(T(n,m)\log n)$ time and $P(n,m)$ processors, the independent set algorithm and weighted vertex coloring algorithm take $O(T(n,m)\log^2 n)$ time and $P(n,m)/\log n$ processors.

Note that for the partitioning, we do not have sufficient resources to sort. This complicates several low-level subroutines. One such case is Step 4 of the procedure *DIVIDE*. If we were to use $\log n$ times more processors, Step 4 would be less complicated.

We follow the usual graph-theoretic terminology [3]. Our graphs are without loops or parallel edges. The vertices of a graph on n vertices are represented by integers $0, \dots, n-1$; the edges are given by a list of pairs $\{(i,j)\}$ where $0 \leq i < j \leq n-1$. The *decomposition tree* $T = T(G)$ of a graph G is defined to be the tree whose vertex set is comprised of the blocks and the cut vertices² of G ; two vertices of T are adjacent if one is a block and the other is a cut vertex belonging to the block. A *star* is a tree with at least two vertices; one of which, called *the center*, is adjacent to all of the other vertices.

The next section contains a description of *PARTITION*, a parallel algorithm that finds a dividing partition of a graph with no Δ -component. The third section describes applications of *PARTITION* to the problems of finding large independent sets in parallel and of finding weighted vertex colorings of small weight. The list processing steps that are used by *PARTITION* are described in section 4. Finally, suggestions are made for further work.

2. The Partitioning Algorithm. We use divide-and-conquer to design our partitioning algorithm, *PARTITION*.

Lemma 1. *Let (B_1, B_2) be a partition of G , and let (A_1, A'_1) and (A_2, A'_2) be dividing partitions of subgraphs $G[B_1]$ and $G[B_2]$ induced on B_1 and B_2 , respectively. Then either $\rho = (A_1 \cup A_2, A'_1 \cup A'_2)$ or $\sigma = (A_1 \cup A'_2, A'_1 \cup A_2)$ is a dividing partition of G .*

Proof. Since (A_1, A'_1) and (A_2, A'_2) are dividing partitions, each of the sets $A_1 \cup A_2$, $A'_1 \cup A'_2$, $A_1 \cup A'_2$, and $A'_1 \cup A_2$ contains at least one fifth of the vertices of G . Every edge which is cut by (B_1, B_2) is also cut by one of ρ or σ ; every edge cut by (A_1, A'_1) or (A_2, A'_2) is cut by both ρ and σ . Thus, the sum Σ of the numbers of edges cut by ρ and σ is at least $w + m_1 + n_1/2 + m_2 + n_2/2$, where w is the number of edges cut by (B_1, B_2) , m_i (respectively n_i) is the number of edges (respectively vertices) in $G[B_i]$ ($i = 1, 2$). If m and n are respectively the number of edges and the number of vertices

² Cut vertices are sometimes referred to as *articulation points* and blocks are sometimes referred to as *biconnected components*.

of G , then $m = m_1 + m_2 + w$ and $n = n_1 + n_2$. Thus, $\Sigma \geq m + n/2$ implying that at least one of ρ and σ is a dividing partition of G . ■

One can prove ³ that every Δ -free graph that does not contain a star as a connected component can be partitioned into two Δ -free subgraphs. With the use of this fact, every Δ -free graph can be split into disjoint stars. Since every star has a trivial dividing partition, we can obtain a dividing partition of the graph by repeated application of Lemma 1. For this approach to be effective in parallel, the partition into two Δ -free subgraphs must be quickly computable and must consist of two graphs of approximately equal size. Unfortunately, it is not clear how to do this. It is, however, possible to partition a Δ -free graph either into two approximately equally sized Δ -free subgraphs or into two parts; one of which is a Δ -free graph and the other consists of a “large” star and a number of Δ -components. Thus, instead of splitting Δ -free graphs into disjoint stars, we split them into disjoint subgraphs each containing a star large enough to guarantee the existence of a dividing partition.

The following two lemmas take care of the base case; that is, graphs that consist of stars possibly with some Δ -graphs. The processor counts given in these lemmas depend on the assumption that Δ -graphs are represented by a list of the vertices in each block and a list of the blocks that each cut vertex belongs to.

Lemma 2. Every connected Δ -graph G with n vertices and m edges has a partition (A, A') which cuts $\geq m/2 + n/4 - 1/4$ edges such that $|A| + 1 = |A'|$. Such a partition can be constructed in $O(\log n)$ time on an EREW PRAM with $O(n/\log n)$ processors, provided that the decomposition tree is given.

Proof. If G is an odd clique, then choosing A to be any $\lfloor n/2 \rfloor$ of the vertices gives the desired partition. If G has two or more blocks, the situation is more complicated. First, we partition each block b independently so that one part contains $\lfloor |b|/2 \rfloor$ vertices and the other contains $\lceil |b|/2 \rceil$ vertices. Moreover, we ensure that the parent of b is in the larger part. Next we use the Eulerian tour technique [20] to combine these partitions into a single partition of the required size. To do this, we give each edge in the decomposition tree between a block and its parent a weight of zero. Consider an edge e in the decomposition tree between a cut vertex v and its parent, a block b . Then b 's parent will be a cut vertex u . If, in the partition of b , u and v are in the same part, then the edge e has weight zero; otherwise it has weight one. (If the root of the decomposition tree is a block c , then one of the vertices in c is chosen arbitrarily to act as the parent of c in the determination of the weights of the edges between c and its children.) To determine which part each cut vertex v is in, we calculate, using the Eulerian tour technique, the weighted length of the path from the root of the decomposition tree to v . If this length is even, v is in A ; otherwise it is in A' . The non-cut vertices are assigned to the parts so that the partition of each block is respected. It is easy to prove by induction on the structure of the decomposition tree that this is the desired partition. ■

Lemma 3. Let G be formed by taking the union of a star S with s vertices and d Δ -components and possibly adding edges from vertices in the Δ -components to the center of S . Then, if $s - 2 \geq d$, G has a dividing partition. Furthermore, this partition can be found in $O(\log n)$ time on an EREW PRAM with $O(n/\log n)$ processors, provided that the blocks of the Δ -graphs are given.

Proof. To find a dividing partition of G , we partition each Δ -component, as above, and then rename the parts, if necessary, so that the larger part is A' . If G contains d Δ -components with the total of m' edges and n' vertices, then the partition cuts $\geq m'/2 + n'/4 - d/4$ edges belonging to the blocks. We then place the center of the star in A or A' so as to cut at least half of the edges between the center of the star and

³ We omit this proof here.

the Δ -components. We place $\lceil (d + 3s - 2)/4 \rceil$ vertices other than the center into the part that does not contain the center; the remaining vertices of the star are placed to minimize the disbalance of the partition. Obviously, we can produce such a partition only if $\lceil (d + 3s - 2)/4 \rceil \leq s - 1$; this inequality is equivalent to the condition $s - 2 \geq d$. ■

From the proofs of the lemmas, one can easily extract a procedure for building a dividing partition of a graph described in lemma 3. We call this procedure \star *PARTITION*. Now we are ready to see how *DIVIDE* works. Given a Δ -free graph G , *DIVIDE* either produces a Δ -free partition or a partition (A, A') such that A is Δ -free and A' satisfies the assumptions of Lemma 3. Note that *DIVIDE* can treat each connected component of G separately; thus we can assume, without loss of generality, that G is connected. The procedure *DIVIDE* comprises the following four steps:

- Step 1.** Either find an induced star S with at least $n/3$ vertices such that $G - S$ has at most one isolated vertex, or create a partition (A, A') such that A and A' each contain at least $n/3$ vertices, $G[A']$ is connected, and $G[A]$ contains no isolated vertices. If the star is found, do step 2, otherwise do steps 3 and 4.
- Step 2.** Put S and all of the Δ -components of $G - S$ into A' . Put the rest of the graph into A . Return (A, A') .
- Step 3.** Move one vertex from each Δ -component of $G[A]$ to $G[A']$. Choose the vertices to be moved so that $G[A']$ stays connected.
- Step 4.** If $G[A']$ is a Δ -graph, move between one and three vertices from A' to A , so that the resulting partition is Δ -free. Return (A, A') .

Step 3 is fairly straightforward, but the other steps need to be explained in more detail.

Explanation of step 1. *DIVIDE* starts by finding a rooted spanning tree T of G and a vertex v with at least $2n/3$ descendants, none of which have $2n/3$ or more descendants. Next it rearranges T so that v is the root. It does this by reversing the direction of all the parent-child links between v and the root. That is, it makes v 's parent its child; its grandparent its grandchild; and so on. Let $D = V - \{v\}$ be the set of proper descendants of v . Then *DIVIDE* finds the connected components of $G[D]$ and counts the number of isolated vertices among them. If there are $n/3$ or more isolated vertices in $G[D]$, then v is the center of a big star S and the isolated vertices of $G[D]$ are the other vertices of S . Alternatively, suppose that there are fewer than $n/3$ isolated vertices in $G[D]$. In this case, *DIVIDE* looks for a Δ -free partition (A, A') of G . There are several subcases, depending on the sizes of the connected components of $G[D]$.

If $G[D]$ has a connected component C with at least $n/3$ and at most $2n/3$ vertices, then *DIVIDE* returns $(C, G - C)$.

Suppose that every connected component of $G[D]$ has less than $n/3$ vertices. In this case, *DIVIDE* makes a list of connected components of $G[D]$ that contain two or more vertices. Then, it calculates the minimum k so that the total size of the first k connected components in this list is at least $n/3$. These components become the set A and the rest of the graph becomes A' . *DIVIDE* returns the partition (A, A') .

Finally, if $G[D]$ has a connected component C with more than $2n/3$ vertices, it chooses A to be a subset of C . To construct this subset, it finds all the children of v that are in C and puts them in a list so that the children that are leaves of the spanning tree T of G are at the end of the list. If one of these children has at least $n/3$ descendants, that child and its descendants become A and the rest of the graph becomes A' . Otherwise, *DIVIDE* calculates the minimum k so that the subtrees rooted at the first k children of v together contain at least $n/3$ vertices. Let D' be the set of vertices of these subtrees. If $G[D']$ has no isolated vertices, *DIVIDE* returns $(D', G - D')$. If $G[D']$ contains isolated vertices, then the subtrees that were not included in D' all consist of a single vertex. Thus, *DIVIDE* can add a

subtree adjacent (in $G[D]$) to all of the isolated vertices of $G[D']$, while ensuring that $|D'| \leq 2n/3$.

Explanation of step 2. The hardest part of step 2 is finding the Δ -components of $G - S$. The connected components are found using the Cole-Vishkin algorithm. In section 4, we will see how to determine if a connected graph H is a Δ -graph in $O(\log n)$ time on a CRCW-PRAM with $O((n+m)\alpha(m,n)/\log n)$ processors.⁴

Explanation of step 4. The input to this step is a partition (A, A') with $G[A]$ Δ -free, $G[A']$ connected, and each part containing at least $2n/9$ vertices. If $G[A']$ is a Δ -graph, step 4 constructs the desired partition by transferring between one and three vertices from A' to A . (If $G[A']$ is not a Δ -graph, step 4 is not executed.) Step 4 comprises twelve sub-steps. Below we described these sub-steps and then prove their correctness. In section 4, we show that step 4 can indeed be executed on $O((n+m)\alpha(m,n)/\log n)$ processors in $O(\log n)$ time.

- 4.1 Find the connected components of $G[A]$ and the blocks of $G[A]$ and $G[A']$.
- 4.2 If one of the connected components of $G[A]$ is not a near Δ -graph, then find a vertex $x \in A'$ adjacent to this connected component and transfer x from A' to A .
- 4.3 Else, if there is a vertex $x \in A'$ that is adjacent to a vertex $v \in A$ that is not in an even clique, then transfer x from A' to A .
- 4.4 *[If step 4 gets to this point, each connected component of $G[A]$ must be a near Δ -graph. Moreover, all vertices in A that are adjacent to a vertex in A' are in even blocks of $G[A]$.]* Else, find a block B of the original graph G that is not an odd clique.
- 4.5 If B contains no vertices from A' , find a vertex $x \in A'$ that is adjacent to the connected component of $G[A]$ that contains B and transfer it to A .
- 4.6 Else, if B contains exactly one vertex x from A' , transfer x from A' to A .
- 4.7 Else, if B contains two vertices that are not in the same block of $G[A']$, then find such two vertices x and y such that no block of $G[A']$ contains both of them and there is a block of $G[A]$ containing vertices adjacent to x and y ; transfer x and y from A' to A . *[At this point, every connected component of $G[A]$ containing vertices in B must be adjacent to two or more vertices in A' , so x and y exist.]*
- 4.8 *[If step 4 gets to this point, $C' = B \cap A'$ is a single block of $G[A']$.]* Else, if $|B \cap A| = 1$, then find $v = B \cap A$ and $x \in B \cap A'$, such that (v, x) is an edge, and transfer x from A' to A .
- 4.9 Let $C' = A' \cap B$. If $B \cap A$ contains vertices only from a single block of $G[A]$ and $|B \cap A| > 1$, then find three vertices $x, y, z \in C'$ such that x and y are each adjacent to some vertex in $B \cap A$ and z is not adjacent to all vertices in $B \cap A$. Transfer the vertices x, y and z from A' to A . *[At this point, the vertex z exists, since B contains an odd number of vertices, so it is not a clique.]*
- 4.10 *[If step 4 gets to this point, B contains vertices from multiple blocks of A .]* Else, if every vertex of $C' = B \cap A'$ is adjacent to every vertex of $B \cap A$, then take any three vertices in C' and move them in A .
- 4.11 Else, find a vertex $x \in C'$ and an even block C_0 , $(C_0 \cap B \neq \emptyset)$ such that x is not adjacent to all vertices in C_0 . If $C_0 \cap B$ consists of one vertex u only, then let y and z be any two vertices in C' adjacent to u . Move x, y , and z from A' to A .

⁴ The obvious algorithm would be to find the blocks of H and then check if they are odd cliques by seeing whether the vertices have the “right” degrees. This algorithm does not work with the resources stated because there does not seem to be any way to produce, for each vertex, a list that contains exactly once each block that this vertex belongs to (see section 4.)

4.12 Let $y, z \in C'$ be vertices adjacent to some vertex in C_0 . If y or z is x , replace x with some vertex (other than y or z) in C' . Move x, y , and z from A' to A .

The next three lemmas prove the correctness of step 4; the first two of them are quite obvious so we omit the proofs to them. For all the lemmas, partition (A, A') satisfies the conditions of the input to step 4.

Lemma 4. *Moving any odd number of vertices from some block of A' to A , or moving any two non-adjacent vertices from A' to A makes $G[A']$ a Δ -free graph. ■*

Lemma 5. *If for some set R , $G[R] \cup x$ is a Δ -graph, then every component of $G[R]$ is a near Δ -graph. ■*

Lemma 6. *Step 4 produces a new partition (A, A') for which both $G[A]$ and $G[A']$ are Δ -free.*

Proof. Since G is connected, every component C of $G[A]$ contains vertices adjacent to some vertices in A' , therefore sub-step 4.2 can indeed find a required vertex $x \in A'$. By Lemma 4, moving x into A makes $G[A']$ Δ -free and by Lemma 5, it does not create Δ -components in $G[A]$, provided the premise of 4.2 or 4.3 holds. Thus, if the algorithm does not halt after completion of 4.3, then:

- (a) every component of $G[A]$ is a near Δ -graph; and
- (b) for every edge (u, x) with $u \in A$ and $x \in A'$, u belongs to an even block.

The analysis of steps 4.5 and 4.6 is also simple. If $|B \cap A'| = 0$, then B is the even block of one of the components of $G[A]$, a vertex x can be found, and moving it to A makes $G[A']$ Δ -free. Furthermore, moving x to A does not create a Δ -component in $G[A]$, since x can only be adjacent to one vertex in B , say u , which implies that x and u make an even block in $G[A]$. Similarly, if $|B \cap A'| = 1$, then step 4 halts after execution of 4.6 (Lemma 4 and the fact that B is not an odd block). Let us now assume that the premise of 4.7 holds. We first prove that B contains two vertices, x and y , that are not in the same block of $G[A']$, but are adjacent to vertices of the same block of $G[A]$. Indeed, otherwise for every even block B_e of $G[A]$, all vertices in A' adjacent to B_e belong to the same block of $G[A']$, implying that every cut-vertex of $G[A']$ is also a cut vertex of G . Obviously, this implication is eliminated by the premise of 4.7.

If step 4 does not halt after 4.7, then in addition to (a) and (b), the partition satisfies two more conditions:

- (c) the intersection $B \cap A'$ is a single block C' of $G[A']$; and
- (d) for every even block C of $G[A]$, $|B \cap C| = 0$, or 1, or $|C|$.

Both properties follow from the fact we already used: if a vertex is adjacent to two vertices in a block, then it is itself in the block.

If v is the only vertex of $B \cap A$, (sub-step 4.8) then moving to A any $x \in B \cap A'$ adjacent to v completes step 4. Indeed, since B is a block of G , v must be the only vertex in $A \cap B$ adjacent to x .

Let us now assume that step 4 does not halt after 4.8 and the premise of 4.9 holds. Since B is not an odd clique, it is obvious that the required three vertices exist and their transferring to A yields a Δ -free partition.

If 4.10 is executed, $G[A']$ becomes Δ -free by Lemma 4, and $G[A]$ is Δ -free since the vertices from different even blocks are not adjacent to each other.

Finally, reasoning similar to the above proves that, if step 4 attempts 4.11, then the resulting $G[A]$ and $G[A']$ are Δ -free. ■

Now we are ready to prove that the procedure *DIVIDE* produces the required partition efficiently.

Theorem 1. *If G has n vertices and $n \geq 27$, then *DIVIDE* finds a Δ -free partition of G such that each part has at least $2n/9$ vertices or *DIVIDE* finds a partition where one part satisfies the requirements of Lemma 3 and the other is Δ -free and has at*

most $2n/3$ vertices. Furthermore, *DIVIDE* finds this partition in $O(\log n)$ time on an CRCW PRAM with $O((n+m)\alpha(m,n)/\log n)$ processors.

Proof. First, let us see that *DIVIDE* produces an acceptable partition. If step 2 is executed, $G - S$ contains at most $\lfloor 2n/9 \rfloor + 1$ Δ -components, since $G - S$ contains at most $2n/3$ vertices and at most one of the Δ -components of $G - S$ has fewer than three vertices. Therefore, A' satisfies the requirements of Lemma 3.

If steps 3 and 4 are executed, then after step 3, A contains at least $2n/9$ vertices, since each Δ -component of $G[A]$ has at least three vertices, and $|A| \geq n/3$. Furthermore, $G[A]$ is Δ -free and $G[A']$ is connected. Thus, step 4 produces a Δ -free partition. Therefore, *DIVIDE* produces an acceptable partition.

Steps 3 and 4 can be done in $O(\log n)$ time on a CRCW PRAM with $O((n+m)/\log n)$ processors. Therefore, *DIVIDE* finds an acceptable partition in $O(\log n)$ time using $O((n+m)\alpha(m,n)/\log n)$ processors in the CRCW model. ■

Finally, we describe the algorithm *PARTITION* which delivers a dividing partition for every Δ -free graph.

```
function PARTITION(G);
begin
  DIVIDE(G, G1, G2);
  /* G1 and G2 are the resulting parts and G1 is a
     Δ-free graph */
  (A1, A'1) ← PARTITION(G1);
  if G2 is not Δ-free, then
    (A2, A'2) ← *PARTITION(G2)
  else (A2, A'2) ← PARTITION(G2);
  ρ ← (A1 ∪ A2, A'1 ∪ A'2); σ ← (A1 ∪ A'2, A'1 ∪ A2);
  return ρ or σ, whichever is bigger;
end;
```

The correctness of *PARTITION* follows from Lemmas 1-6. Putting them together with Theorem 1, we get

Theorem 2. *A dividing partition can be found in $O(\log^2 n)$ time on an CRCW PRAM with $O((m+n)\alpha(m,n)/\log n)$ processors.*

3. Applications. The algorithm that finds a dividing partition can be used to find an independent set of a graph G with n vertices and m edges that contains at least $n^2/(2m+n)$ vertices. It turns out that if (G_1, G_2) is a dividing partition of G , then one of the parts has an independent set of the necessary size. If, however, G has a Δ -component, then G may not have a dividing partition. Thus, *IND*, an algorithm that finds an independent set of Turán's size, must treat Δ -components as a special case. The treatment is based on the following:

Lemma 7. *Let G be a Δ -graph and let I contain one non-cut vertex from each block that has one. Also, let G' be the result of deleting I and its neighborhood from G . Then n' , the number of vertices of G' , satisfies $n' \leq n/3$. Furthermore, if I' is an independent set of G' such that*

$$|I'| \geq \frac{n'^2}{2m' + n'},$$

where m' is the number of edges of G' , then $I \cup I'$ is an independent set of G such that

$$|I \cup I'| \geq \frac{n^2}{2m + n}.$$

Proof. If G is a single clique, the lemma is obvious. Alternatively, let us assume that G has at least two blocks.

To prove that $n' \leq n/3$, we consider the decomposition tree of G . Note that any vertex in a block containing a noncut vertex will not be in G' . Let b be the number of blocks containing a noncut vertex. Since each block of G has at least three vertices, there are at least $2b + 1$ vertices of G that are not in G' . Allocate each cut vertex to one of its children in the decomposition tree. Then each block has at most one cut vertex allocated to it. Each block that contains only cut vertices has at least two children. Therefore, there are at most b vertices in blocks containing only cut vertices and at most b vertices in G' . Thus, G' has at most $n/3$ vertices.

Since I contains only noncut vertices and at most one vertex from each block, I is independent. Furthermore, since G' contains no neighbors of I , $I \cup I'$ is independent.

It remains to estimate the size of $I \cup I'$. Suppose that one vertex of I and its neighborhood is deleted, leaving G'' with n'' vertices and m'' edges. It is enough to show that

$$\frac{n^2}{2m + n} \leq 1 + \frac{n''^2}{2m'' + n''}.$$

The following computation that appeared in [9] does this. Suppose that the vertex to be deleted has degree d . Then all of its neighbors have degrees of at least d , so $n'' = n - d - 1$ and $m'' \leq m - d(d + 1)/2$. Note that

$$\begin{aligned} 1 + \frac{n''^2}{2m'' + n''} &\geq 1 + \frac{(n - d - 1)^2}{2\left(m - \frac{d(d+1)}{2}\right) + n - d - 1} \\ &= 1 + \frac{(n - d - 1)^2}{2m - (d + 1)^2 + n} \\ &= \frac{2m + n + n^2 - 2n(d + 1)}{2m + n - (d + 1)^2}. \end{aligned}$$

Now let us consider

$$1 + \frac{n''^2}{2m'' + n''} - \frac{n^2}{2m + n}.$$

Simplifying, we see that

$$\begin{aligned} &1 + \frac{n''^2}{2m'' + n''} - \frac{n^2}{2m + n} \\ &= \frac{2m + n + n^2 - 2n(d + 1)}{2m + n - (d + 1)^2} - \frac{n^2}{2m + n} \\ &= \frac{(2m + n)^2 - 2n(d + 1)(2m + n) + n^2(d + 1)^2}{(2m + n)(2m + n - (d + 1)^2)} \\ &= \frac{(2m + n - n(d + 1))^2}{(2m + n)(2m + n - (d + 1)^2)} \\ &\geq 0. \end{aligned}$$

This completes the proof. \blacksquare

To find a large independent set of a Δ -graph, *IND* computes I and G' as suggested in Lemma 7. If G' is empty, I is the desired set. If G' is a Δ -graph, *IND* repeats the process. Finally, if G' is neither empty nor a Δ -graph, *IND* calls itself recursively to find I' .

The algorithm *IND*, given below, implements this idea.

function *IND*(G);

```

begin
  J ← ∅;
  foreach connected component D of G do begin
    if D is a Δ-component do begin
      while D is a nonempty Δ-graph do begin
        I ← one noncut vertex from each block of D that has one;
        J ← J ∪ I;
        Delete J and the neighborhood of J from D;
      end;
      J ← J ∪ IND(D);
    /* J is a big independent set of the Δ-components */
    end;
    else /* D is not a Δ-components */
      Delete the Δ-components of G;
      if G is empty then I' := ∅
    else begin
      (G1, G2) ← a dividing partition of G;
      ni ← the number of vertices of Gi (i = 1, 2);
      mi ← the number of edges of Gi (i = 1, 2);
      if n12/(2m1 + n1) ≥ n22/(2m2 + n2)
        then I' := IND(G1)
      else I' := IND(G2); end
    return J ∪ I';
  end;

```

Theorem 3. *The procedure IND finds an independent set of size $\geq n^2/(2m+n)$ of a graph G with n vertices and m edges in $O(\log^3 n)$ time on an CRCW PRAM with $O((n+m)\alpha(m,n)/\log^2 n)$ processors.*

Proof. The first thing to notice is that IND can afford to treat the Δ -components separately, since, if $n_1 + n_2 = n$ and $m_1 + m_2 = m$, then

$$\lceil n_1^2/(2m_1 + n_1) \rceil + \lceil n_2^2/(2m_2 + n_2) \rceil \geq \lceil n^2/(2m + n) \rceil.$$

It remains to be seen that IND finds an independent set of Turán's size on the Δ -free part of the graph. Suppose that this part has n vertices and m edges, and that G_1 (G_2) has n_1 (n_2) vertices and m_1 (m_2) edges. Since (G_1, G_2) is a dividing partition, $n_1 + n_2 = n$ and $m_1 + m_2 \leq m - m/2 - n/4$. By induction, IND returns a set of size at least $\max(n_1^2/(2m_1 + n_1), n_2^2/(2m_2 + n_2))$. Thus, if IND does not return a big enough set, then

$$\frac{n_i^2}{2m_i + n_i} < \frac{n^2}{2m + n}, \quad (i = 1, 2)$$

so

$$(2m + n)n_i^2 < (2m_i + n_i)n^2.$$

Adding the previous inequalities for $i = 1$ and $i = 2$ together, we get

$$(2m + n)(n_1^2 + n_2^2) < (2m_1 + n_1 + 2m_2 + n_2)n^2.$$

Since the partition is dividing, we have

$$\begin{aligned} (2m + n)(n_1^2 + n_2^2) &< (2(m_1 + m_2) + n)n^2 \\ &\leq (2(m - \frac{m}{2} - \frac{n}{4}) + n)n^2 \\ &= (m + \frac{n}{2})n^2. \end{aligned}$$

This is equivalent to

$$2(n_1^2 + n_2^2) < (n_1 + n_2)^2, \quad \text{so} \quad (n_1 - n_2)^2 < 0,$$

which is impossible. Thus, *IND* returns a big enough independent set.

The most time consuming part of *IND* is finding a dividing partition of G . Recall that *PARTITION* runs in $O(\log^2 n)$ time, so *IND* runs in $O(\log^3 n)$ time, since each recursive call is made on a graph with at most $4/5$ as many vertices and at most half as many edges as the original graph. Furthermore, since there is only one recursive call, the technique of Brent [4] can be used to reduce the processor count by a factor of $\log n$, to $O((n+m)\alpha(m,n)/\log^2 n)$. ■

The other application of *PARTITION* is finding a light coloring. The algorithm follows the same outline as *IND* except that both parts of the partition need to be considered. To combine the colorings, we require that each part use different colors. It turns out that this gives a light coloring.

```

procedure COLOR( $G$ );
begin
  for each connected component  $C$  of  $G$  begin
    if  $C$  is an isolated vertex then color it 1
    else begin
      if  $C$  is a  $\Delta$ -graph then  $\star$ PARTITION( $G, G_1, G_2$ )
      else PARTITION( $G, G_1, G_2$ )
      /*( $G_1, G_2$ )  $\leftarrow$  the resulting partition of  $C$ ;*/
      COLOR( $G_1$ ); COLOR( $G_2$ );
       $c \leftarrow$  the number of colors used for  $G_1$ ;
      increase by  $c$  every color used for  $G_2$ ;
      compute the sizes of the color classes of the
        resulting coloring of  $C$  and sort them in
        nonincreasing order;
      renumber the color classes according to the new
        order;
      end;
    end;
  end;
end;

```

Note that the partitions of the connected components C into G_1 and G_2 eliminate at least half of the edges. Furthermore, since isolated vertices are part of the base case, we can assume that $n \leq 2m$. Thus, at each level of the recursion, the total problem size $(n+m)$ decreases by a constant fraction, so the same analysis as for *IND* shows that *COLOR* runs in $O(\log^3 n)$ time on a CRCW PRAM with $O((n+m)\alpha(m,n)/\log^2 n)$ processors. It is less obvious that *COLOR* finds a light coloring.

Theorem 4. *The procedure COLOR finds a light coloring.*

Proof. The proof is by induction on the progress of the algorithm. If G is an isolated vertex, *COLOR* produces a light coloring. Note that if two graphs have light colorings, these light colorings combine in the obvious way to form a light coloring of their union. Therefore, we can assume, without loss of generality, that G has a single connected component.

Let n , n_1 , and n_2 be the number of vertices of G , G_1 and G_2 , respectively, and let m , m_1 , and m_2 be the number of edges of G , G_1 and G_2 , respectively. If (G_1, G_2) is a dividing partition, $n_1 + n_2 = n$ and $m - m/2 - n/4 \geq m_1 + m_2$, so $m/2 \geq m_1 + m_2 + n/2$. Alternatively, if C is a Δ -graph, then $m - m/2 - n/4 + 1/4 = m_1 + m_2$, so $m/2 = m_1 + m_2 + n/2 - 1/2$. Note that Δ -graphs have an odd number of vertices, so, in either event, $m \geq m_1 + m_2 + \lfloor n/2 \rfloor$.

By induction, *COLOR* produces light colorings for G_1 and G_2 . Let W be the sum of the weights of these colorings. Then it is enough to prove that \mathcal{C} , the coloring that *COLOR* produces, has weight at most $2W - \lceil n/2 \rceil$. To prove this, we give each vertex a sequence number. The sequence numbers are chosen so that the sequence numbers assigned to each color class C of \mathcal{C} are $1, 2, \dots, p_C$. Let A_k be the set of vertices with sequence number k , and let l be the number of such vertices. Then, the total weight assigned to A_k is $l(l-1)/2$. Suppose the l_1 of these vertices came from G_1 , so $l-l_1$ came from G_2 . Then the total weight assigned to A_k by the colorings of G_1 and G_2 is $W_k = l_1(l_1-1)/2 + (l-l_1)(l-l_1-1)/2$. If we think of W_k as a function of l_1 , its minimum value occurs when $l_1 = l/2$. Thus, $W_k \geq 2(l/2)(l/2-1)/2 = l(l-2)/4 = l(l-1)/4 - l/4$. Therefore, the total weight assigned to A_k by \mathcal{C} is at most $2W_k - l/2$. Summing over the sequence numbers, we see that the weight of \mathcal{C} is at most $2W - n/2$. Since the weight of \mathcal{C} is an integer, it is also at most $2W - \lceil n/2 \rceil$. ■

4. Implementation Details. A graph is represented by an array of vertices and an array of edges. The *number* of a vertex is its position in the vertex array. Each vertex v has a pointer into the array of edges showing where the list of edges incident to v starts.

The most expensive part of the whole computation is finding spanning trees, the connected components, and blocks of graphs. These steps are done in $O(\log n)$ time on a CRCW PRAM with $O((m+n)\alpha(m,n)/\log n)$ processors by using the CTV-algorithm appropriately modified for our purposes. It is not hard to see how to do the other steps in $O(\log n)$ time on a PRAM with $O(n+m)$ processors, by the liberal use of a sorting routine. However, it is not obvious how the processor count can be reduced to $O((m+n)/\log n)$. The necessary techniques based on the parallel prefix computation are described here.

Often *IND* and *COLOR* need to compute $G[A]$ for various $A \subset V$. For example, after *DIVIDE* finds a Δ -free partition (A, A') it calculates $G[A]$ and $G[A']$. To do this, it is enough to renumber the vertices of the original graph G so that the vertices in A and the vertices in A' have consecutive numbers. With a parallel prefix computation, each vertex in A can compute the number of vertices in A with a lower original number; this will be the vertex's new number. Similarly, each vertex in A' can compute its new number. Since an n -element parallel prefix computation can be done in $O(\log n)$ time on an EREW PRAM with $O(n/\log n)$ processors [16], the computation of $G[A]$ and $G[A']$ is not an important contribution to the overall running time of *DIVIDE*.

Note that the technique described above works so that the algorithm runs in $O(\log n)$ time and uses $O(n/\log n)$ processors only when it divides the graph into a bounded number of pieces. Thus, when a routine divides a graph into its connected components, it needs to use a different technique.

Lemma 8. *Given a graph G and rooted spanning trees of each connected component of G , it is possible to renumber the vertices of G so that each connected component consists of consecutively numbered vertices, in $O(\log n)$ time on an EREW PRAM with $O((n+m)/\log n)$ processors.*

Proof. Consider the routine *RENUMBER* that does the renumbering described above. Using the Eulerian tour technique [20], *RENUMBER* computes the number of vertices in each connected component. Then it assigns to the root of each spanning tree the number of vertices in its connected component and to each other vertex the value zero. A parallel prefix computation determines the range of new numbers for each vertex. Actually assigning the new vertex numbers and rearranging the edges lists can be done with an application of the Eulerian tour technique and some parallel prefix computations. The parallel prefix and Eulerian tour computations can be done in $O(\log n)$ time on a EREW PRAM with $O(n/\log n)$ processors [20, 16, 6]. ■

Some procedures, including *DIVIDE*, need to be able to identify Δ and near Δ -

graphs. It is done by finding the blocks of the graph in question. Unfortunately, the straightforward use of the CTV-algorithm is not always helpful, since its output has to be in a different form. Effectively the CTV-algorithm returns a list of the edges in each block, while *DIVIDE* and the other procedures need a data structure based on the decomposition tree; we call it the *decomposition tree data structure*. In addition to the decomposition tree, this data structure includes the information showing, for every noncut vertex, the block it belongs to, and pointers in both directions between every cut vertex and corresponding node ⁵ in the decomposition tree. Furthermore, every block node contains a list of the noncut vertices that belong to that block.⁶

Lemma 9. *Given a connected graph G , a spanning tree T of G and the name of the block that each edge belongs to, it is possible to build the decomposition tree data structure in $O(\log n)$ time on a EREW PRAM with $O((n + m)/\log n)$ processors.*

Proof. Consider the procedure *BUILD_TREE* that creates the decomposition tree. First *BUILD_TREE* consults the adjacency list of each vertex v to see if all the edges incident to v are in the same block. If they are, v is not a cut vertex, otherwise v is a cut vertex. The parent of the node corresponding to a cut vertex v is the block containing the edge from v to its parent in T . If v is the root of T , then the node corresponding to v is the root of the decomposition tree. To find the parents of the block nodes, *BUILD_TREE* forms a list of the tree edges in each block. For each block B , it then finds the endpoint v of these edges that is highest (closest to the root) in the spanning tree T . The cut node corresponding to v is the parent of B unless v is not a cut vertex and is the root of T . In this case, B is the root of the decomposition tree. Recall that every edge of T has a parent endpoint and a child endpoint. To find the children of a block node B , *BUILD_TREE* makes a list of the child endpoints of the tree edges in B . The child endpoints that are cut vertices correspond to the children of B in the decomposition tree. The child endpoints that are not cut vertices are the noncut vertices in B . All of these computations can be done in $O(\log n)$ time on a EREW PRAM with $O((n + m)/\log n)$ processors. ■

Now we show how to determine if a graph is a Δ -graph, or a near Δ -graph.

Lemma 10. *Given the decomposition tree data structure, an EREW PRAM with $O((m + n)/\log n)$ processors can determine if a connected graph G is a Δ -graph, a near Δ -graph or neither in $O(\log n)$ time.*

Proof. We describe the procedure *IS_DELTA* that determines if a connected graph is a Δ -graph ⁷

First, *IS_DELTA* determines the number of vertices in each block B by computing the number of noncut vertices in B and the number of cut nodes adjacent to B in the decomposition tree. If one of the blocks has an even number of vertices, then G is not a Δ -graph; otherwise, it might be one.

At this point, *IS_DELTA* needs to determine if every block is a clique. Let $|B|$ be the number of vertices in a block B . To determine if a block B is a clique, *IS_DELTA* first computes the degree of each noncut vertex in B . If one or more of these vertices has a degree that is not $|B| - 1$, then B is not a clique, and *IS_DELTA* returns “no”. Otherwise, *IS_DELTA* considers the edges incident to the cut vertices corresponding to the children of B in the decomposition tree and activates some of them. An edge (u, v) incident to a cut vertex u is activated if v is a noncut vertex in B or if v is a cut

⁵ We will use the term *node* to refer a vertex in the decomposition tree; the term *vertex* refers to a vertex in G or a subgraph of G .

⁶ Obviously, the list of children of a block node can contain duplicates. This is a misfeature of the data structure and complicates its use, but there does not seem to be any way to eliminate the duplicates without using too much resources.

⁷ The procedure for determining if a graph is a near Δ -graph is similar and is omitted.

vertex and the father of the corresponding node in the decomposition tree is B . The edges of the form (u, w) , where u corresponds to a child of B and w corresponds to the father of B are also activated. (If B is the root of the decomposition tree, there are no edges of this form.) If for all the vertices u corresponding to children of B , the number of activated edges is $|B| - 1$, then B is a clique; otherwise it is not. All of these computations can be done with the resources allowed. ■

The techniques described above are enough to show that step 1 can be implemented efficiently.

Lemma 11. *Step 1 can be done in $O(\log n)$ time on a CRCW PRAM with $O((n + m)\alpha(m, n)/\log n)$ processors.*

Proof. The running time and processor count of the first step, and of the whole procedure, is dominated by the resources required to find the spanning tree T of G . This can be done in $O(\log n)$ time on a CRCW PRAM with $O((m + n)\alpha(m, n)/\log n)$ processors [5]. The number of descendants of each vertex in the spanning tree T can be found in $O(\log n)$ time on an EREW PRAM with $O(n/\log n)$ processors [20]. Given this information, v (which must be unique) can be found easily. Finding the connected components of $G[D]$ requires another application of the spanning tree algorithm. If the largest connected component of $G[D]$ has fewer than $n/3$ vertices, then the connected components that *DIVIDE* assigns to A can be identified by a parallel prefix computation. This computation can be done in $O(\log n)$ time on an EREW PRAM with $O(n/\log n)$ processors [20, 16]. If the largest connected component of $G[D]$ has between $n/3$ and $2n/3$ vertices, the partition can be found in $O(1)$ steps. Finally, if there is a connected component of $G[D]$ with more than $2n/3$ vertices, *DIVIDE* finds the portion of this component that it puts in A with a parallel prefix computation and a connected components computation. This is done in $O(\log n)$ time on a CRCW PRAM with $O((m + n)\alpha(m, n)/\log n)$ processors. ■

The last difficult step is step 4.

Lemma 12. *Step 4 can be done in $O(\log n)$ time on a CRCW PRAM with $O((n + m)\alpha(m, n)/\log n)$ processors.*

Proof. Sub-step 4.1 can be done with the resources stated by the CTV-algorithm. The only other difficult sub-steps are 4.7, 4.9 and 4.11.

To do 4.7, *DIVIDE* first calculates the blocks of $G[A \cap B]$. Then, for each block B_i , it finds L_i , the list of vertices in A' adjacent to B_i . Note that a vertex $v \in A'$ can appear in several of the L_i and even several times in the same L_i . However, the total length of the L_i is at most m .

The procedure processes each L_i in parallel; it looks for vertices x and y in L_i that are not part of the same block. First, it determines if any of the vertices in the list are not cut vertices of $G[A']$. If there is a vertex $x \in A' \cap B$ that is not a cut vertex, *DIVIDE* finds the block C'' of $G[A']$ containing x . Next, it looks for a vertex $y \in B \cap A'$ that is not in C'' . If such a vertex exists, *DIVIDE* returns x and y ; otherwise the desired vertices do not exist.

Alternatively, all vertices in $B \cap A'$ are cut vertices. Given a vertex $v \in B \cap A'$, let $P(v)$ be the parent of the node in the decomposition tree corresponding to v ; if v is the root of the decomposition tree, let $P(v)$ be null. *DIVIDE* calculates $P(v)$ for each vertex in $B \cap A'$. If all these parent blocks are the same, the desired vertices do not exist. Otherwise, *DIVIDE* finds two vertices x and y with different parent blocks. If the node corresponding to y is an ancestor of the node corresponding to x , *DIVIDE* switches x and y . If the node corresponding to x is not the parent of $P(y)$, the parent block of y , then x and y are the desired vertices. Otherwise, *DIVIDE* looks for a vertex z that is not in $P(y)$. (If z doesn't exist, the desired vertices do not exist.) If the parent node of $P(z)$ corresponds to y , then x and z are the desired vertices; otherwise y and z are.

To do step 4.9, *DIVIDE* creates a list of the vertices in C' that are adjacent to some vertex in $B \cap A$. If this list is not all of C' , *DIVIDE* chooses x and y to be the first two vertices on this list, and it chooses z to be some vertex not on the list. Alternatively, if all the vertices in C' are adjacent to some vertex in $B \cap A$, *DIVIDE* chooses z to be some vertex in C' not adjacent to all the vertices in $B \cap A$, and x and y to be two other vertices in C' .

Finally, we come to step 4.11. To do step 4.11, *DIVIDE* first identifies x by computing the number of vertices in $A \cap B$ adjacent to each vertex in $A' \cap B$ and choosing x to be one of the ones adjacent to fewer than all of them. Once *DIVIDE* has chosen x , it can choose C_0 to be a block of $G[A \cap B]$ containing a vertex not adjacent to x . If there is a vertex in C_0 that is adjacent to two or more vertices in C' , *DIVIDE* chooses that vertex to be u , it chooses v to be some other vertex in C_0 adjacent to some vertex z in C' , and it chooses $y \neq z$ to be a vertex in C' adjacent to u . Alternatively, if every vertex in C_0 is adjacent to at most one vertex in C' , *DIVIDE* chooses u and v so that the vertices that they are adjacent to in C' are different.

Therefore, *DIVIDE* executes step 4 in $O(\log n)$ time on a CRCW PRAM with only $O((n+m)\alpha(n,m)/\log n)$ processors. ■

Putting this all together, we get

Theorem 5. *The procedure DIVIDE requires $O(\log n)$ time on a CRCW PRAM with $O((n+m)\alpha(m,n)/\log n)$ processors.*

5. Conclusions. One way to cope with hard problems is to find a solution that is not necessarily optimal but has a guaranteed quality. We present an efficient parallel algorithm that finds an independent set of a guaranteed size. It uses, as a subroutine, a procedure that finds a partition that cuts a guaranteed number of edges. The partitioning procedure can also be used to obtain an efficient parallel algorithm to find a light coloring. The problem of finding a fast parallel algorithm that finds a minimal coloring remains open, however.

Approaching other combinatorial problems similarly should lead to interesting results. In parallel, the matching problem may be hard; no deterministic \mathcal{NC} -algorithm is known. Thus, we are led to the question “how big a matching can we be sure of finding deterministically in \mathcal{NC} ?” If the edges have weights, we can ask a similar question about the weight of the matching. This question is particularly interesting since matching is the bottleneck in Anderson and Aggarwal’s algorithm that finds a depth-first search tree of an undirected graph [1]. Therefore, studying algorithms that find big matchings may lead to a deterministic \mathcal{NC} -depth first search algorithm.

Acknowledgment. The authors would like to thank H. Kierstead for finding an error in an earlier version of *DIVIDE*, and the reviewers for several useful comments.

REFERENCES

- [1] A. Aggarwal and R. Anderson, *A Random NC-algorithm for Depth First Search*, in Proc. 19th Annual ACM Symposium on Theory of Computing, (1987), pp. 325–334.
- [2] N. Alon, L. Babai, and A. Itai, *A fast and simple randomized parallel algorithm for the maximal independent set problem*, J. Algorithms, **7** (1986), pp. 567–583.
- [3] J. A. Bondy, U. S. R. Murty, *Graph Theory with Applications*, North Holland, 1979.
- [4] R. P. Brent, *The parallel evaluation of general arithmetic expressions*, J. ACM, **2** (1974), pp. 201–208.
- [5] R. Cole, U. Vishkin, *Approximate and exact parallel scheduling with applications to list, tree, and graph problems*, in Proc. 27th Annual Symposium on Foundations of Computer Science, (1986), pp. 478–491.
- [6] R. Cole and U. Vishkin, *Approximate parallel scheduling. I. The basic technique with applications to optimal parallel list ranking in logarithmic time*, SIAM J. Comput. (1988), pp. 128–142.

- [7] P. Erdős, *On even subgraphs of graphs* (in Hungarian) *Mat. Lapok.*, **18** (1964), pp. 283–288.
- [8] M. Goldberg, *Parallel algorithms for three graph problems*, *Congressus Numerantium*, **54** (1986), pp. 111–121.
- [9] M. Goldberg, S. Lath, and J. Roberts, *Heuristics for the graph bisection problem*, Tech. Report, 86-8, Rensselaer Polytechnic Institute, Troy, NY, 1986.
- [10] M. Goldberg, T. Spencer, *A new parallel algorithm for the maximal independent set problem*, *SIAM J. on Comp.* **18** (1989), pp. 419–427.
- [11] M. Goldberg, T. Spencer, *Constructing a maximal independent set in parallel*, *SIAM J. on Discr. Math.*, **2** (1989), pp. 322–328.
- [12] Y. Han and R. A. Wagner *An efficient and fast parallel connected component algorithm* *JACM*, (1990), pp. 626–642.
- [13] D. Johnson, *Approximate algorithms for combinatorial problems*, *J. of Computer and System Sciences* **9**, (1974), pp. 256–278.
- [14] R. M. Karp, V. Ramachandran, *Parallel algorithms for shared memory machines*, in *Handbook of Theoretical Computer Science*, MIT press, Cambridge, Ma., pp. 871–942. 1990.
- [15] R. M. Karp, A. Wigderson, *A fast parallel algorithm for the maximal independent set problem*, in *Proc. 16th ACM Symp. on Theory of Computing*, 1984, pp. 266–272.
- [16] R. E. Ladner, M. J. Fischer, *Parallel prefix computation*, *J. ACM* **27** (1980), pp. 831–838.
- [17] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, *SIAM J. Comp.*, **15** (1986), pp. 1036–1053.
- [18] M. Luby, *Removing randomness in parallel computation without a processor penalty*, in *Proc. 29th Annual Symp. on Foundation of Computer Science*, (1988), pp. 162–173.
- [19] Y. Shiloach, U. Vishkin, *An $O(\log n)$ parallel connectivity algorithm*, *J. Algorithms*, **3** (1982), pp. 57–63.
- [20] R. Tarjan, U. Vishkin *An Efficient parallel biconnectivity algorithm*, *SIAM J. Comp.*, **14**, (1985), pp. 862–874.
- [21] P. Turán, *On the theory of graphs*, *Colloq. Math.* **3** (1954), pp. 19–30.