The Parallel Random Access Machine

Disclaimer: These notes DO NOT substitute the textbook for this class. The notes should be used in conjunction with the textbook and the material presented in class. If a statement in these notes seems to be incorrect, report it to the instructor so that it be fixed immediately. These notes are distributed to the students of CIS 668; distribution outside this group of students is prohibited.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

The Parallel Random Access Machine (PRAM) is one of the simplest ways to model a parallel computer. A PRAM consists of a collection of (sequential) processors that can *synchronously* access a global *shared* memory in unit time. Each processor can thus access its shared memory as fast (and efficiently) as it can access its own local memory. The main advantages of the PRAM is its simplicity in capturing parallelism and abstracting away communication and synchronization issues related to parallel computing. Processors are considered to be in abundance and unlimited in number. The resulting PRAM algorithms thus exhibit *unlimited parallelism* (number of processors used is a function of problem size). The abstraction thus offered by the PRAM is a fully synchronous collection of processors and a shared memory which makes it popular for parallel algorithm design. It is, however, this abstraction that also makes the PRAM unrealistic from a practical point of view. Full synchronization offered by the PRAM is too expensive and time demanding in parallel machines currently in use. Remote memory (i.e. shared memory) access is considerably more expensive in real machines than local memory access as well and UMA machines with unlimited parallelism are difficult to build.

Depending on how concurrent access to a single memory cell (of the shared memory) is resolved, there are various PRAM variants. ER (Exclusive Read) or EW (Exclusive Write) PRAMs do not allow concurrent access of the shared memory. It is allowed, however, for CR (Concurrent Read) or CW (Concurrent Write) PRAMs. Combining the rules for read and write access there are four PRAM variants: EREW, ERCW, CREW and CRCW PRAMs. Moreover, for CW PRAMs there are various rules that arbitrate how concurrent writes are handled.

Convention: In this subject we name processors arbitrarily either $0, 1, \ldots, p-1$ or $1, 2, \ldots, p$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

(1) in the *arbitrary* PRAM, if multiple processors write into a single shared memory cell, then an arbitrary processor succeeds in writing into this cell,

(2) in the *common* PRAM, processors must write the same value into the shared memory cell,

(3) in the *priority* PRAM the processor with the highest priority (smallest or largest indexed processor) succeeds in writing,

(4) in the *combining* PRAM if more than one processors write into the same memory cell, the result written into it depends on the combining operator. If it is the *sum* operator, the sum of the values is written, if it is the *maximum* operator the maximum is written.

The EREW PRAM is the weakest among the four basic variants. A CREW PRAM can simulate an EREW one. Both can be simulated by the more powerful CRCW PRAM. An algorithm designed for the common PRAM can be executed on a priority or arbitrary PRAM and exhibit similar complexity. The same holds for an arbitrary PRAM algorithm when run on a priority PRAM.

Assumptions

In this handout we examine parallel algorithms on the PRAM. In the course of the presentation of the various algorithms some common assumptions will be made. The input to a particular problem would reside in the cells of the shared memory. We assume, in order to simplify the exposition of our algorithms, that a cell is wide enough (in bits or bytes) to accommodate a single instance of the input (eg. a key or a floating point number). If the input is of size n, the first n cells numbered $0, \ldots, n-1$ store the input. In the discussion below, we assume that the number of processors of the PRAM is n or a polynomial function of the size n of the input. Processor indices are $0, 1, \ldots, n-1$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Parallel Sum

Problem. Compute $x_0 + x_1 + \ldots + x_{n-1}$.

A sequential algorithm that solves this problem requires $n - 1$ additions.

For a PRAM implementation, value x_i is initially stored in shared memory cell i. The sum $x_0 + x_1 + \ldots + x_{n-1}$ is to be computed in $T = \lg n$ parallel steps. Without loss of generality, let n be a power of two. If a combining CRCW PRAM with arbitration rule *sum* is used to solve this problem, the resulting algorithm is quite simple. In the first step processor i reads memory cell i storing x_i. In the following step processor i writes the read value into an agreed cell say 0. The time is $T = O(1)$, and processor utilization is $P = O(n)$.

A more interesting algorithm is the one presented below for the EREW PRAM. The algorithm consists of $\lg n$ steps. In step i, processor $j < n/2^i$ reads shared-memory cells $2j$ and $2j + 1$ combines (sums) these values and stores the result into memory cell j. After $\lg n$ steps the sum resides in cell 0.

```
//   pid() returns the id of the processor issuing the call.
begin PARALLEL_SUM  (n)
1.    i = 1 ; j = pid();
2.    while (j < n/2^i)
3.        a = C[2j];
4.        b = C[2j + 1];
5.        C[j] = a + b;
6.        i = i + 1;
7.    end
end PARALLEL_SUM
```

Algorithm PARALLEL_SUM has $T = O(\lg n)$, $P = n$ and $W = O(n \lg n)$, $W_2 = O(n)$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Parallel Sum continued

Algorithm PARALLEL_SUM can be easily extended to include the case where n is not a power of two. PARALLEL_SUM is the first instance of a sequential problem that has a trivial sequential but more complex parallel solution. Instead of operator *Sum* other operators like *Multiply*, *Maximum*, *Minimum*, or in general, any associative operator could have been used. As associative operator $\otimes$ is one such that $(a \otimes b) \otimes c = a \otimes (b \otimes c)$.

Exercise 1 *Can you improve* PARALLEL_SUM *so that T remains the same, $P = O(n/\lg n)$, and $W = O(n)$? Explain.*

Exercise 2 *What if i have p processors where $p < n$? (You may assume that n is a multiple of p).*

Exercise 3 *Generalize algorithm to any associative operator.*

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Broadcasting

A message (say, a word) is stored in cell 0 of the shared memory. We would like this message to be read by all n processors of a PRAM. On a CREW PRAM this requires one parallel step (processor i concurrently reads cell 0). On an EREW PRAM broadcasting can be performed in $O(\lg n)$ steps. The structure of the algorithm is the reverse of the previous one. In $\lg n$ steps the message is broadcast as follows. In step i each processor with index j less than 2^i reads the contents of cell j and copies it into cell $j + 2^i$. After $\lceil \lg n \rceil$ steps each processor i reads the message by reading the contents of cell i.

```
begin BROADCAST (M)
1.    i = 0 ; j = pid(); C[0]=M;
2.    while (2^i < P)
3.        if (j < 2^i)
5.            C[j + 2^i] = C[j];
6.        i = i + 1;
6.    end
7.    Processor j reads M from C[j].
end BROADCAST
```

A CR?W PRAM algorithm that solves the broadcasting problem has performance $P = O(n)$, $T = O(1)$, and $W = O(n)$.

The EREW PRAM algorithm that solves the broadcasting problem has performance $P = O(n)$, $T = O(\lg n)$, and $W = O(n \lg n)$, $W_2 = O(n)$.

Exercise 4 *Broadcasting on a hypercube and a butterfly (Hint: Base you solution to the* BROADCAST *algorithm).*

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Given a set of n values $x_0, x_1, \ldots, x_{n-1}$ and an associative operator, say $+$, the *parallel prefix* problem is to compute the following n results/"sums".

0: x_0,

1: $x_0 + x_1$,

2: $x_0 + x_1 + x_2$,

...

$n-1$: $x_0 + x_1 + \ldots + x_{n-1}$.

Parallel prefix is also called *prefix sums* or *scan*. It has many uses in parallel computing such as in load-balancing the work assigned to processors and compacting data structures such as arrays. An algorithm for parallel prefix on an EREW PRAM would require $\lg n$ phases. In phase i, processor j reads the contents of cells j and $j - 2^i$ (if it exists) combines them and stores the result in cell j.

The EREW PRAM algorithm that solves the parallel prefix problem has performance $P = O(n)$, $T = O(\lg n)$, and $W = O(n \lg n)$, $W_2 = O(n)$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Parallel Prefix Example

For visualization purposes, the second step is written in two different lines. When we write $x_1 + \ldots + x_5$ we mean $x_1 + x_2 + x_3 + x_4 + x_5$.

```
   x1       x2         x3         x4          x5         x6       x7        x8


1.         x1+x2       x2+x3     x3+x4       x4+x5     x5+x6     x6+x7      x7+x8

2.                 x1+(x2+x3)           (x2+x3)+(x4+x5)    (x4+x5)+(x6+x7)
2.                           (x1+x2)+(x3+x4)     (x3+x4)+(x5+x6)    (x5+x6+x7+x8)

3.                                       x1+...+x5          x1+...+x7
3.                                                 x1+...+x6           x1+...+x8

Finally
F. x1   x1+x2    x1+...+x3  x1+...+x4  x1+...+x5 x1+...+x6 x1+...+x7 x1+...+x8
```

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Matrix Multiplication

A simple algorithm for multiplying two $n \times n$ matrices on a CREW PRAM with time complexity $T = O(\lg n)$ and $P = n^3$ follows. For convenience, processors are indexed as triples (i, j, k), where $i, j, k = 1, \ldots, n$. In the first step processor (i, j, k) concurrently reads a_{ij} and b_{jk} and performs the multiplication $a_{ij}b_{jk}$. In the following steps, for all i, k the results $(i, *, k)$ are combined, using the parallel sum algorithm to form $c_{ik} = \sum_j a_{ij}b_{jk}$. After $\lg n$ steps, the result c_{ik} is thus computed.

The same algorithm also works on the EREW PRAM with the same time and processor complexity. The first step of the CREW algorithm need to be changed only. We avoid concurrency by broadcasting element a_{ij} to processors $(i, j, *)$ using the broadcasting algorithm of the EREW PRAM in $O(\lg n)$ steps. Similarly, b_{jk} is broadcast to processors $(*, j, k)$.

The above algorithm also shows how an n-processor EREW PRAM can simulate an n-processor CREW PRAM with an $O(\lg n)$ slowdown.

```
                                          CREW    EREW
1. aij to all            (i,j,*) procs    O(1)    O(lgn)
   bjk to all            (*,j,k) procs    O(1)    O(lgn)
2. aij*bjk at            (i,j,k) proc     O(1)    O(1)
3. parallel sum aij *bjk (i,*,k) procs    O(lgn)  O(lgn)   n procs participate
             j

4. cik = sum aij*bjk                      O(1)    O(1)
           j


               3        3                3
 T=O(lgn),P=O(n ) W=O( n  lgn)    W  = O(n )
                                   2
```

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Let $G = (V, E)$ be an undirected graph. V is the set of vertices and E is the set of edges. By convention $|V| = n$ and $|E| = m$.

(a) Two vertices u, v are connected by an edge if $(u, v) \in E$.

(b) The degree of node u is the number of edges incident on u, ie the number of v such that $(u, v) \in E$.

A directed graph G is like an undirected one but the edges are assigned directions. We represent G by $G = (N, A)$, where N is the set of nodes(vertices of a directed graph) and A is the set of arcs (directed edges). If u, v are connected by an arc from u to v, then $< u, v > \in A$. For simplicity, we may write $(u, v) \in A$ as well, using the same symbols for both a directed and undirected graph.

(a) The out-degree of a node u is the number of vertices v such that $< u, v > \in A$. The in-degree of v is the number of vertices w such that $< w, v > \in A$.

(b) For an undirected or directed graph G, a path is a sequence of vertices $v_1, v_2, \ldots, v_j$ such that $(v_1, v_2), (v_2, v_3), \ldots, (v_{j-1}v_j)$ are edges in the undirected case or $< v_1, v_2 >, < v_2, v_3 >, \ldots, < v_{j-1}v_j >$ are arcs in the directed case.

(c) The length of the path is the number of edges/arcs on the path ie $j - 1$ in the example above.

For undirected graphs discussed in this handout we assume that they are simple i.e. they have no self-loops (edges (v, v)) or multiple edges.

(i)An undirected graph G is connected if there is a path connecting every pair of vertices. (ii) If a simple connected undirected graph has $n - 1$ edges it is called a tree. (iii) A collection of trees forms a forest.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Pointer Jumping-Introduction

A rooted directed tree $T = (V, A)$ is a directed graph with a special node r called the root such that (a) $\forall v \in V - \{r\}$ node v has out-degree 1, and r has out-degree 0, and (b) $\forall v \in V - \{r\}$ there exists a unique directed path from v to r. In other words, T is rooted if the undirected graph resulting from T is a tree. The level of a vertex/node in a tree is the number of edges on the path to the root.

Let F be a forest consisting of a set of rooted directed trees. Forest F is represented by an array P (P stands for "parent") of length n such that $P(i) = j$ if $< i, j >$ is an arc of F. For a root i, it is $P(i) = i$.

We examine a technique called **pointer jumping** that finds many applications in designing algorithms for linked list and graph theory problems.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Problem Given forest F and array P construct array S where $S(j)$ is the root of the tree containing node j.
Proof. We use pointer jumping, that is we iteratively make the successor of any node i to become the successor of its successor. This way the distance of a node from its root is halved after a single pointer jumping step. After k iterations (pointer jumping steps) the distance, in the original graph, between i and its current successor $S(i)$ is 2^k (in terms of number of edges in the original graph) unless $S(i)$ is the root (in the original forest represented by P). In the latter case the procedure is successfully completed. The PRAM algorithm FIND_ROOT implements pointer jumping.

```
begin FIND_ROOT  (P,S)
1.    in parallel: S(i) = P(i) ;
2.    while (S(i) ≠ S(S(i)))
3.          S(i) = S(S(i)).
end FIND_ROOT
```

Let h be the maximum height of any tree in forest F. The running time of this algorithm on an CREW PRAM is is $T = O(\lg h)$, $P = O(n)$ and $W = W_2 = O(n \lg h)$.

From this point on, we primarily use the alternative definition of work W, ie W_2 to indicate the actual number of operations performed by all the processors (which is not necessarily $P \cdot T$); in most of the cases to be examined it will not make a difference which definition is used.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Pointer Jumping continued

Problem Assume that associated with each node i of forest F is a value $V(i)$. Compute $W(i)$, for all i, where $W(i)$ is the sum of the $V(j)$ over all nodes j in the path from i to its root (a parallel prefix-like operation in a list/tree).
Proof. The PRAM algorithm works as follows.

```
begin PJ  (P,S,V)
1.    in parallel: S(i) = P(i), W(i) = V(i) ;
2.    while (S(i) ≠ S(S(i)))
3.        W(i) = W(i) + W(S(i)).
4.        S(i) = S(S(i)).
end PJ
```

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Logical AND operation

Problem. Let $X_1 \ldots, X_n$ be binary/boolean values. Find $X = X_1 \wedge X_2 \wedge \ldots \wedge X_n$.

The sequential problem accepts a $P = 1, T = O(n), W = O(n)$ direct solution.

An EREW PRAM algorithm solution for this problem works the same way as the PARALLEL SUM algorithm and its performance is $P = O(n), T = O(\lg n), W = O(n \lg n)$ along with the improvements in P and W mentioned for the PARALLEL SUM algorithm.

In the remainder we will investigate a CRCW PRAM algorithm. Let binary value X_i reside in the shared memory location i. We can find $X = X_1 \wedge X_2 \wedge \ldots \wedge X_n$ in constant time on a CRCW PRAM. Processor 1 first writes an 1 in shared memory cell 0. If $X_i = 0$, processor i writes a 0 in memory cell 0. The result X is then stored in this memory cell.

```
begin LOGICAL_AND  (X_1 ... X_n)
1.    Proc 1    writes in cell 0.
2.    if  X_i = 0 processor i writes 0 into cell 0.
end LOGICAL_AND
```

The result stored in cell 0 is 1 (TRUE) unless a processor writes a 0 in cell 0; then one of the X_i is 0 (FALSE) and the result X should be FALSE, as it is.

Exercise 5 *Give an $O(1)$ CRCW algorithm for LOGICAL OR.*

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Problem. Let $X_1 \ldots, X_N$ be n keys. Find $X = \max\{X_1, X_2, \ldots, X_N\}$.

The sequential problem accepts a $P = 1, T = O(N), W = O(N)$ direct solution.

An EREW PRAM algorithm solution for this problem works the same way as the PARALLEL SUM algorithm and its performance is $P = O(N), T = O(\lg N), W = O(N \lg N), W_2 = O(N)$ along with the improvements in P and W mentioned for the PARALLEL SUM algorithm.

In the remainder we will investigate a CRCW PRAM algorithm. Let binary value X_i reside in the local memory of processor i.

The CRCW PRAM algorithm MAX1 to be presented has performance $T = O(1)$, $P = O(N^2)$, and work $W_2 = W = O(N^2)$.

The second algorithm to be presented in the following pages utilizes what is called a doubly-logarithmic depth tree and achieves $T = O(\text{lglg}N)$, $P = O(N)$ and $W = W_2 = O(N\,\text{lglg}N)$.

The third algorithm is a combination of the EREW PRAM algorithm and the CRCW doubly-logarithmic depth tree-based algorithm and requires $T = O(\text{lglg}N)$, $P = O(N)$ and $W_2 = O(N)$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

begin MAX1 $(X_1 \ldots X_N)$
1. **in proc** (i,j) **if** $X_i \geq X_j$ **then** $x_{ij} = 1$;
2. **else** $x_{ij} = 0$;
3. $Y_i = x_{i1} \wedge \ldots \wedge x_{in}$;
4. Processor i reads Y_i ;
5. **if** $Y_i = 1$ processor i writes i into cell 0.

end MAX1

In the algorithm, we rename processors so that pair (i,j) could refer to processor $j \times n + i$. Variable Y_i is equal to 1 if and only if X_i is the maximum.

The CRCW PRAM algorithm MAX1 has performance $T = O(1)$, $P = O(N^2)$, and work $W_2 = W = O(N^2)$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Doubly Logarithmic-Depth Tree

In preparation of algorithm MAX2 we introduce a **doubly logarithmic-depth tree.**
Let $N = 2^{2^n}$, for some integer n.
A *doubly logarithmic-depth* tree with N leaves is defined as follows.

(1) The root of the tree at level 0 has $2^{2^{n-1}} = N^{1/2}$ children in level 1.

(2) Each node at level 1 has $2^{2^{n-2}} = N^{1/2^2}$ children in level 2.

(3) Each node of level i has $2^{2^{n-(i+1)}} = N^{1/2^{i+1}}$ children in level $i+1$.

(4) Each node of level $n-1$ (the level before the last) has $2^{2^{n-n}} = N^{1/2^n} = 2$ children in level $n = \lg\lg N$.

(5) The nodes of level n are the **leaves** of the tree.

Some properties of a *doubly logarithmic-depth* tree are listed below.

(1) The **height** of the tree is $n = \lg\lg N$.

(2) A node of level i has $2^{2^{n-(i+1)}}$ children in level $i+1$.

(3) The TOTAL number of level i nodes is $2^{2^{n-1}} 2^{2^{n-2}} \ldots 2^{2^{n-i}} = 2^{2^n - 2^{n-i}}$.

(4) The Product $(2^{2^{n-i-1}})^2 \times 2^{2^n - 2^{n-i}}$ is $O(2^{2^n}) = O(N)$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Algorithm MAX2

Algorithm MAX2 below achieves better work performance than MAX1 $T = O(\lg\lg N)$, $P = O(N)$, and $W = W_2 = O(N \lg\lg N)$.

Algorithm MAX2 works as follows.

(1) Initialyy, items X_i are on the N leaves of the tree.

(2) The root will hold the result at the completion of the algorithm.

(3) Processors are assigned to the nodes of the tree in some predetermined fashion.

(4) All nodes of the tree other than the leaves hold an UNDEFINED value in the beginning of the execution.

(5) If a node u at level i holds an UNDEFINED value and its M children hold some intermediate results M^2 processors are assigned to u to find the maximum of M numbers (the partial results of the children of u) using MAX1 in constant time. Node u then holds the computed maximum (and ceases to hold an UNDEFINED value).

```
begin MAX2  (X_1 ... X_N)
0.    The i-th cell contains X_i; N = 2^{2^n}.
1.    for  (i = n − 1; i ≥ 0; i − −) do
2         begin
3.            Assign to each node u of level i, (2^{2^{n−i−1}})^2 processors (i.e. the square of its children in level i + 1.) ;
4.            Use algorithm MAX1 and these processors to find the maximum
              of the values stored at the children of u and store the result at u;
5.        Node u ceases to hold an UNDEFINED.
6.        end
end MAX2
```

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Algorithm MAX2 has $W = W_2 = O(N \lg\lg N)$. Algorithm MAX3 below has $W_2 = O(N)$. It uses MAX2 and the EREW PRAM algorithm as subroutines.

The EREW algorithm finds the SUM or the MAXIMUM of N numbers by working from the leaves to the root of a binary tree, ie $\lg N$ levels. If we stop the computation after i levels, we have $N/2^i$ partial results, each result being the MAXIMUM of 2^i numbers initially stored in the leavs of the subtree rooted at the partial result.

Max3 first runs the EREW PRAM algorithm for $i = \lg\lg\lg N$ levels so that a total of $N/\lg\lg N$ partial MAXIMA are computed.

Then it applies MAX2 where N in MAX2 is equal to the number of partial results ie $N/\lg\lg N$.

begin MAX3 $(X_1 \ldots X_N)$
0. The i-th cell contains X_i;
1 **begin**
2. Use the EREW PRAM algorithm on a complete binary tree on N leaves to reduce the original problem to computing the maximum of $N/\lg\lg N$ values (ie proceed from the leaves up to the nodes of level $\lg N - \lg\lg\lg N$);
3. Use MAX2 to find the maximum of the $N/\lg\lg N$ values of Step2;
4. **end**

end MAX3

Step 2 of algorithm MAX3 requires $T = O(\lg\lg\lg N)$, $P = O(N)$ and $O(N)$ work (number of comparisons is at most the number of edges of the tree).

Step 3 requires $P = O(N)$, $T = O(\lg\lg N)$ and total work $W = W_2 = O(N)$ by the analysis of MAX2.

Question. Is there a $p \leq n$ processor CRCW PRAM algorithm that finds the maximum of N keys faster than MAX2 or MAX3?

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

A matching lower bound

Definition. On an undirected graph $G = (V, E)$, an **independent set** is a set of vertices such that no two vertices are connected by an edge.

Theorem Let $G = (V, E)$ be an undirected graph, where $|V| = n$ and $|E| = m$. Graph G has an independent set of size at least $n^2/(2m + n)$.

The following theorem can then be proved.

Theorem 1 Computing the maximum of n keys requires at least $\lg\lg n$ parallel steps with $p \leq n$ processors.

Proof. (by induction) It is proved by what we call **an adversary argument** through induction. An **adversary** for this problem is allowed to choose the input keys by modifying their values in such a way so as to force the algorithm to run for at least $\lg\lg n$ steps.

These modifications should not invalidate, however, the operations of the algorithm already performed. Since the proof is inductive, at the end of step $i + 1$ an adversary can specify an input such that the maximum will lie in a set S_{i+1} of size s_{i+1} with the following properties:

(1) no two keys of S_{i+1} have been compared before,

(2) $s_{i+1} \geq s_i^2/(s_i + 2p)$.

For $i = 0$, $s_0 = n$. Each one of the keys is in set S_0. Any parallel algorithm for finding the maximum could perform at most p comparisons in the first parallel step (at most one per processor). For each key in S_0 a graph G is formed with an edge joining two vertices if the corresponding keys were compared in that step. G has n vertices and at most p edges and an independent set S_1 of size at least $s_1 = t \geq n^2/(2p + n)$.

Let the vertices of this set be $v_{j_1}, \ldots, v_{j_t}$. An adversary can then choose the keys x_i's in such a way that x_{j_k} is larger than all its neighbors. This is possible as v_{j_k} is not connected to any v_{j_l} (by the independent set property). Set S_1 satisfies the inductive hypothesis.

This way we prove that $s_{i+1} \geq s_i^2/(s_i + 2p) \geq s_i^2/3n$, $s_0 = n$. Therefore, $s_i \geq n/3^{2^i-1}$.

The algorithm terminates when $s_i \leq 1$. The number of parallel steps is then $\Omega(\lg\lg n)$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Definition 1 *A* **directed cycle** *is a directed graph $G = (V, E)$ such that the in-degree and out-degree of every node is one. Then, for every $u, v \in V$ there is a directed path from u to v (and from v to u as well). A k-coloring of G is a mapping $c : V \to \{0, \ldots, k-1\}$ such that $c(i) \neq c(j)$, $\forall i \neq j$ and $< i, j > \in E$.*

We are interested in 3-colorings of directed cycles. In the sequential case, this problem is easy to solve. Color vertices of the cycle alternately with two colors 0 and 1 and at the end, a third color may be required for the last node of the cycle, if the first and the node before the last are colored differently. In a parallel setting this problem looks difficult to parallelize because it looks so symmetric! All vertices look alike. In order to solve this problem in parallel we represent the graph by defining $V = \{0, \ldots, n-1\}$ and an array S, the successor array, so that $S(i) = j$ if $< i, j > \in E$. A predecessor array can be easily derived from the identity $P(S(i)) = i$. For a number i let $i = i_n \ldots i_2 i_1$ be its binary representation. Then i_k is the k-th lsb **(least significant bit)** of i.

begin COLOR1 (P,S,c)
1. **in parallel** $\forall 0 \leq i < n$
2. Let k be the lsb position that $c(i)$ and $c(S(i))$ differ;
3. Set $c(i) = 2(k-1) + (k$-th lsb of $c(i)$;

end COLOR1

Claim After a single call to COLOR1 a (valid) coloring is derived from a previously (valid) coloring.
Proof. Before the call to COLOR1 adjacent vertices are colored differently by a coloring say coloring C_1. Let us assume for the sake of contradiction that an application of COLOR1 results in a coloring C_2 that fails to color properly two vertices i, j, i.e $c(i) = c(j)$ for $< i, j > \in E$. These colors were obtained after an application of step 3, i.e. $c(i) = 2(k-1) + c(i)|_k$ and $c(j) = 2(l-1) + c(j)|_l$. Since $c(i) = c(j)$ (because of the C_2 coloring) we must have that $k = l$ and $c(i)|_k = c(j)|_k$, i.e the previous colors of i and j (in C_1) agreed in the k-th lsb. This contradicts the assumption that k is the first lsb position where $c(i)$ and $c(j)$ differ under C_1.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Repeated application of COLOR1 results as it is proved below in a 6-coloring of a directed cycle as described in algorithm COLOR2A. Initially a trivial coloring of the n vertices with n colors is found, i.e. a mapping c such that $c(i) = i$. After a call to COLOR1, an initial n-coloring of a directed cycle gives rise to a $(2\lg n + 1)$-coloring. If at some point a coloring uses 3 bits (up to 8 colors) then a new coloring after another application of COLOR1 would require at most six colors ($c(i) = 2(k-1) + c(i)|_k \leq 5$, for $k \leq 3$).

Claim Algorithm COLOR2A 6-colors a directed cycle.

```
begin COLOR2A  (P,S,c)
1.    in parallel ∀0 ≤ i < n set c(i) = i;
2.    repeat
3.        call Algorithm COLOR1 ;
4.    until at most 6 colors are used.
end COLOR2A
```

Proof. Algorithm COLOR2A initially colors the vertices with n colors using c bits, i.e $2^{c-1} \leq n < 2^c$. After the first iteration, $\lceil \lg c \rceil + 1$ bits are only used (colors $0, \ldots, 2c-1$ are used). Let us define $\lg^{(1)}(x) = \lg x$, $\lg^{(2)}(x) = \lg\lg x$ and in general $\lg^{(i)}(x) = \lg(\lg^{(i-1)}(x))$. We then define $\lg^*(x) = \min\{i : \lg^{(i)}(x) \leq 1\}$. After the first iteration of Loop 2-4 an $O(\lg n)$-coloring is derived. After the second iteration an $O(\lg\lg n)$-coloring is derived. After $O(\lg^*(n))$ iterations a 6-coloring is derived. The complexity of the algorithm is thus $T = O(\lg^*(n))$ and $W = O(n \lg^*(n))$.

A question arises whether a 3-coloring is possible. As soon as a 6-coloring is obtained, a 3-coloring can be derived by perturbing the 6-coloring as in step 3 of COLOR2 below that colors the vertices of a directed cycles with 3 colors.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

```
begin COLOR2  (P,S)
1.   Call COLOR2A;
2.   do for each 3 ≤ i ≤ 5 ;
3.       if a vertex is colored i recolor it with the smallest
         possible color from {0, 1, 2};
end COLOR2
```

Step 3 is realized in $O(1)$ parallel steps, loop 2 is repeated 3 times (once for each of the 3,4,5 colors) and COLOR2 has the same asymptotic time complexity as COLOR2A. Algorithm COLOR3 is more work-efficient (if array initialization costs are ignored) and is presented below. It utilizes parallel integer sorting (a combination of radix and count sort). The following result will be used and is the parallel equivalent of sequential count (sometimes referred to as bucket sort).

```
begin COLOR3  (P,S)
1.   in parallel ∀0 ≤ i < n set c(i) = i;
2.   Call COLOR1;
3.   Sort vertices by their respective color;
4.   for (i = 3; i ≤ ⌈lg n⌉; i++)
5        begin
6.           For all vertices of same color i do in parallel ;
7.               Color v with the smallest color in {0, 1, 2} that is
                 different from the colors of its two neighbors
end COLOR3
```

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Theorem Parallel sorting n integer keys with values in the range $[0 \ldots \lg n - 1]$ takes time proportional to the time required for parallel prefix plus an extra $O(\lg n)$ term on a p processor PRAM, where $n/\lg n \leq p \leq n$.

Proof. The idea behind this parallel-sort algorithm is sequential count-sort. We describe the algorithm for the case $p = n$. Each processor is assigned the i-th integer of the input. Each processor creates an array $\lg n \times 1$ of length $\lg n$ in **contiguous locations of the shared memory** and initializes its i-th entry to the number of keys it is assigned to with value i.

The set of n, $\lg n \times 1$ arrays spanned over the n processors can be viewed as a two-dimensional $\lg n \times n$ array. This virtual two-dimensional array that spreads over the n processors can also be viewed as an $n \times \lg n$ one-dimensional array by viewing the elements of the two-dimensional array in row major fashion (ie 1st element of processor 0, 1st element of processor 1, ..., 1st element of processor $n - 1$, 2nd element of processor 0 and so on).

As there are $n \lg n$ elements in the one-dimensional array and n processors we assign $\lg n$ elements to each processor. Each processor, starting from processor 0, gets $\lg n$ consecutive elements of the one-dimensional array. Such consecutive elements are not, however consecutive in the shared memory as they may belong to different $\lg n \times 1$ arrays (originally stored as a contiguous block in shared memory).

A reordering of the elements of the n original arrays is thus required to get this one-dimensional array assigned to the n processors so that the block assigned to each processor is in contiguous memory locations.

A parallel prefix on this one dimensional array of $n \lg n$ values is then initiated that takes $T = O(\lg n)$ time. Each entry in the resultant array indicates the rank of the corresponding key in the sorted output sequence.

As soon as the rank of each key is known, it is written in the ranked location of the output (sorted) sequence and the sorting operation is completed. □

If the range of the input keys is $[0 \ldots \lg^2 n - 1]$, then two rounds of the previous algorithm are required (parallel radix-sort). With reference to Color3, Step 2 reduces the number of colors to $O(\lg n)$ and vertices are colored properly. Thus, when step 7 is performed, no two adjacent vertices have the same color. The total parallel running time of Color3 is thus $O(\lg n)$. All three coloring algorithms work on an EREW PRAM. Algorithm Color3 is to be used in the solution of the List-Ranking problem to follow.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Consider a linked list L of n nodes whose order is specified by the use of a successor array ($S(i)$ is the successor of i in the linked list, $0 \leq i < n$). If t is the tail of the list, $S(t) = 0$. The problem of *list-ranking* is to determine the distance of each node from the tail of the list. The sequential complexity of list ranking is linear in n and the sequential problem is a prefix-like problem. Parallel List-ranking has many applications in parallel graph algorithms. We first present a non-optimal parallel algorithm for list-ranking (LIST1).

```
begin LIST1 (P,S)
0.   Input S(.) matrix, Output V(i) is distance from tail of node i;
1.   ∀0 ≤ i < n do in parallel
2.       if (S(i) ≠ 0) V(i) = 1;
3.              else   V(i) = 0;
4.   ∀0 ≤ i < n do in parallel
5        B(i) = S(i);
6.       while (B(i) ≠ 0 ∧ B(B(i)) ≠ 0) do
7.           V(i) = V(i) + V(B(i));
8.           B(i) = B(B(i));
end LIST1
```

The time complexity of LIST1 on an EREW PRAM is $T = O(\lg n)$ and $W = O(n \lg n)$. We next develop a work-optimal algorithm (LIST2). We outline the strategy for the latter work-optimal algorithm: (a) Shrink L until $O(n/\lg n)$ nodes remain, (b) Apply LIST1 on the resulting list, and (c) Restore the original list and rank all nodes removed in step (a).

As step (c) is the reverse of step(a) the time-processor requirements are the same (asymptotically). In step (a) a set of nodes is removed and the V values of the remaining nodes are updated. The set of nodes to be removed is chosen so that the nodes of the set form an independent set (i.e. set I is an independent set if whenever $i \in I$ then $S(i) \notin I$.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Procedure LIST2A below removes nodes by adjusting the successor and predecessor nodes of a removed node. Its input consists of L represented by arrays P and S, array V, and independent set I whose nodes will be removed from L. The output is list L with nodes in I removed and the V values of the remaining nodes readjusted.

begin LIST2A (P,S,V,I)
1. Assign consecutive numbers $F(i)$ to elements of I such that $1 \leq F(i) \leq |I|$.
2. $\forall i \in I$ **do in parallel**
3. $\quad N(F(i)) =< i, S(i), P(i), V(i) >$;
4. $\quad V(P(i)) = V(P(i)) + V(i)$;
5 $\quad S(P(i)) = S(i)$;
6. $\quad P(S(i)) = P(i)$;

end LIST2A

Finding an independent set is realized by coloring the nodes of L; same color vertices form an independent set.

Definition 2 *A vertex u is local minimum (maximum) if its color is less (greater) than those of its two neighbors*

Lemma 1 *Given a k-coloring of L, set I of local minima (or maxima) is an independent set of size $\Omega(n/k)$. Set I can be determined in $T = O(1)$ time and $W = O(n)$.*

Proof. Let u, v be two successive local minima. Clearly, they are not adjacent and the number of vertices between them is at most $(k-2)+1+(k-2) = 2k-3$. Local minima thus form an independent set of size at least $n/(2k-3) = \Omega(n/k)$ as claimed.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.

Procedure LIST2A below removes nodes by adjusting the successor and predecessor nodes of a removed node.

begin LIST2 (P,S,V,I)
1. $n_0 = n$, $k = 0$;
2. **while** $n_0 \geq n/\lg n$
3. $k = k + 1$;
4. Color list with 3 colors and identify local minima (set I);
5 Remove I from list and store info of deleted nodes as in LIST2A;
6. Let n_k be size of remaining list. Compact it into consecutive memory locations;
8. Apply pointer jumping algorithm PJ on the result (of size at most $n/\lg n$);
9. Restore original list using info from step 5.

end LIST2

Claim LIST2 is a work-optimal list-ranking algorithm.
Proof. Because of step 4 and Lemma 1, cn_k nodes, for some constant c, are removed every time step 5 is executed and after $O(\lg\lg n)$ steps at most $n/\lg n$ nodes remain ($n_{k+1} \leq (1-c)n_k$ and thus $n_k \leq (1-c)^k n$). The k-th iteration requires $T = O(\lg n)$ and $W = O(n_k)$. The total running time is thus $T = O(\lg n \lg\lg n)$ and $W = O(n)$ on an EREW PRAM.

The parallel time can be further reduced to $O(\lg n)$ by a more complicated algorithm.

(c) Copyright Alexandros Gerbessiotis, Fall 2000. Not for distribution outside the New Jersey Institute of Technology.