PRAM algorithms continued

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 only distributed to the students taking this class with A. Gerbessiotis in Fall 2004; distribution outside this group of students is NOT allowed.

Integer Sorting

We introduce a special case of sorting n keys that are integers whose values are in the range of $[0..\lg n-1]$, where $\lg n$ we assume it is also an integer. This is the well-known problem of count-sort. Sorting n keys in the range $0..k-1$ requires sequential time $O(n+k)$. We show below that sorting n integer keys in the specified range with $P = n/\lg n$ processors can be done in time $T = O(\lg n)$ using $W = W_2 = \Theta(n)$. Note that count-sort does not sort in place i.e. we are going to use different arrays for input and output. Let $M[1..n-1]$ be the input and $N[1...n-1]$ the output arrays.

The idea is to assign $\lg n$ keys per processor; for example if keys are in $M[1..n-1]$, processor $i = 0, \ldots, p-1$ deals with keys $i \lg n + j + 1$, where $j = 0, \ldots, \lg n - 1$. The P processors collectively create an $P \times \lg n$ array C initialized to 0. We assign to each procesor a single row of the array. Thus initialization take $O(\lg n)$ steps to zero the entries of row i assigned to processor i. Entry (i, j) of the table would indicate how many keys with value j processor i is assigned to. This information can be collected easily: processor i scans its keys and for each key it updates the counters of row i of C. Total time is $O(\lg n)$. Then a parallel prefix operation is formed. It consists of the first column, the second column and so on the last column. The purpose is to count all the keys with values 0 before the ones with value 1, before those with value 2 and so on. Note that the prefix sequence is of length n. If we have n processors we can work it out in $O(\lg n)$ time. Now that we have only P processors we can invoke Brent's principle to do it in $O(\lg n)$ time as well but with $O(n)$ work. Note that during the prefix operation C is not overwritten; a new array D will hold the results. After the prefix operation if the entry that corresponded initially to the (i, j) element of C has value t, this means that processor i will store the keys assigned to it with value j to consecutive positions ending with memory location t of the output. Thus if for example we have that $C(i, j) = 3$, then N[t-2] N[t-1] and N[t] will hold the three keys with value j of processor i. Processor i, after D becomes available scans its keys and writes them into N as appropriately.

Thm. Sorting n keys in the range $[0..\lg n-1]$ with $P = n/\lg n$ processors can be done in time $T = O(\lg n)$ and work $W = W_2 = \Theta(n)$.

Applying this theorem t times (i.e. use t rounds of count-sort to obtain a radix-sort algorithm) the following is derived.

Thm. Sorting n keys in the range $[0..\lg^t n-1]$ with $P = n/\lg n$ processors can be done in time $T = O(t \lg n)$ and work $W = W_2 = \Theta(tn)$.

Parallel Count-Sort: Sequential algorithm

We show the corresponding sequential algorithm which also indicates the "sequential work" that each one of the P processors will perform on its own $\lg n$ assigned keys.

```
 Count-Sort(M[1..n],N[1..n],n,k)
   // Initialize Counter array C
1. for(i=0;i<k;i++)   C[i]=0;
3. for(j=1;j<=n;j++)  C[M[j]] ++;
4. D[0]=C[0]; //Note that the extra D is not required; we can reuse C
5. for(i=1;i<k;i++)
6.   D[i] = C[i]+D[i-1];
7. for(j=n;j>=1;j--) {
8.    N[D[M[j]]]=M[j];
9.    D[M[j]]--;
   }
```

Parallel Count-Sort: An example

```
Step 1:  Split keys and initialize C; Count keys and update C
M: 0 2 2 0   1 3 0 0   1 3 0 3  2 1 3 2          (n=16,lgn=4)
P:    0         1         2        3


C:
  p   0  1  2 3
   0  2  0  2 0 <<< Processor zero counted two 0's and two 2's for its elements
   1  2  1  0 1
   2  1  1  0 2
   3  0  1  2 1

Step 3:  For a prefix: take first column  second colum  ...  lgn-th column

Prefix: Input:     2 2 1 0   0 1 1 1   2  0  0 2   0  1 2  1    : C array info
       Output:     2 4 5 5   5 6 7 8  10 10 10 12 12 13 15 16   : D array info

The length of the prefix is n.  n processors can do it in O(lg{n}) time.
n/lgn processors in O(lg{n}) time and O(n) work.
                   2 4 5 5   5 6 7 8  10 10 10 12 12 13 15 16   : D array info
                   *         *         *          *
Processor 0 recovers four numbers 2,5,10,12 the last position occupied by
                              a   0,1, 2, 3 if it has them
It has 2 0's that will occupy positions 1 and 2  of the output
and    2 2's that will occupy positions 9 and 10 of the output
He copies them into N as dictated by array D (observe sequential algorithm)
```

SKIP THE NEXT 6 slides

Suppose we are given a polynomial $f(x)$ such that

$$f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$$

where the indeterminate is x and the coefficients $a_i \in R$. Depending on the circumstances a_i could belong to the set of integer numbers (denoted by Z) or the set of rational numbers (denoted by Q). Suppose we are given $c \in R$. Polynomial evaluation deals with finding

$$f(c).$$

In order to compute $f(c)$ the naive method for evaluation first finds

$$1, c, c^2, \ldots, c^n \text{ for a total of } (n-1) \text{ multiplications}$$

then computes all products of c^i with a_i

$$a_0, a_1 c, a_2 c^2, \ldots, a_n c^n \text{ for a total of } n \text{ multiplications}$$

and finally adds up the partial products to derive $f(c)$

$$f(c) = a_0 + a_1 c + a_2 c^2 + \ldots + a_n c^n \text{ for a total of } n \text{ additions}$$

```
   Naive(a[0..n],n,x)
1. result = a[0]; xn=1;
2. for(j=1;j<=n;j++) {
3.  xn *= x;
4.  result = result + xn * a[j];
5. }
6. return(result);
```


Therefore the total cost of this method is : $(2n-1)$ M's (for multiplications) and n A's (for additions).

There is a faster way to perform polynomial evaluation however using a method that is known as Horner's method/rule or Horner-Newton rule. According to this method

$$f(x) = ((a_n x + a_{n-1})x + a_{n-2})x + \ldots + a_1)x + a_0.$$

i.e. $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = ((a_3x + a_2)x + a_1)x + a_0$. This method requires n M's and n A's, i.e. it is faster than the naive method.

```
   Horner(a[0..n],n,x)
1. result = a[n];
2. for(j=n-1;j>=0;j--) {
3.  result = result * x + a[j];
4. }
5. return(result);
```


Theorem PSum. Let the minimum number of operations for computing an expression E be $m \geq p$. Then evaluation of E in parallel using p processors requires time at least $T \geq (m+1)/p + \lg p - 1$.

Proof. We assume that each processor computes at every step a simple expression $z = x\ o\ y$, where $o \in \{+, -, /, *\}$.

Let $k = \lfloor \lg p \rfloor$. In the last step of the evaluation of E, the T-th step, a single operation like $E = z_1 o z_2$, is performed on a single processor.

Then, in step $T-1$, operations $z_1 = y_1 o y_2$ and $z_2 = y_3 o y_4$ may be performed, i.e. at most two operations are performed in parallel on two processors.

Then in the $T-i$-th step 2^i operations are performed in parallel on 2^i processors. In the last step $T-k$ we have $2^k = 2^{\lfloor \lg p \rfloor}$ operations are performed on the same number of processors. If $2^k = p$, then call $a_1, \ldots, a_p$ these operations. Operation $a_i = b_i o c_i$ and all suboperations required for b_i, c_i will then be performed sequentially in processor i for all $1 \leq i \leq p$.

Therefore the first $1 + 2 + 2^2 + \ldots + 2^k = 2^{k+1} - 1 = 2^{\lfloor \lg p \rfloor + 1} - 1$ operations are performed on a fraction of the processors (some processors remained idle during those computations) in $k + 1 = \lfloor \lg p \rfloor + 1$ parallel steps. This is at least $2^{\lceil \lg p \rceil} - 1$ operations.

In the remaining $T - k - 1$ steps, the remaining operations can be performed each processor performing one such operation per step. Therefore the total number of operations that can be performed in these steps is $p(T - k - 1) = p(T - \lfloor \lg p \rfloor - 1) \geq p(T - \lceil \lg p \rceil)$.

The total number of operations performed is thus at least $2^{\lceil \lg p \rceil} - 1 + p(T - \lceil \lg p \rceil)$. Since the expressions requires m operations to be evaluated for $2^{\lceil \lg p \rceil} - 1 + p(T - \lceil \lg p \rceil) > m$, T must be at least

$$T \geq \lceil \frac{m - 2^{\lceil \lg p \rceil} + 1}{p} \rceil + \lceil \lg p \rceil$$

This simplifies as $T = (m - p + 1)/p + \lg p = (m+1)/p + \lg p - 1$.

Munro-Paterson method ***WILL NOT BE COVERED***

Theorem M-P. Let $n = 2m + 1$ be odd. Let $f(x) = a_n x^n + \ldots + a_1 x + a_0$, and let $p \lg n \leq n$. Then $f(x)$ can be evaluated in $T = 2n/p + \lg p + c$ multiplications and additions, where c is a constant less than 4. The constant c absorbs the contributions of $\lceil \lg p \rceil$ factors that are simplified as $\lg p$.

Proof. Phase 1. We keep processors busy by performing a Horner rule computation and a parallel prefix computation $f(x) = ((a_n x + a_{n-1})x + \ldots$ by working as follows.

(a) We first compute in parallel $b_0 = a_0 + a_1 x,\ b_1 = a_2 + a_3 x,\ \ldots, b_m = a_{n-1} + a_n x$. The number of terms we compute in parallel is $(n + 1)/2 = m + 1$, and we thus need to perform a total of $m + 1$ A's and $m + 1$'s M's to complete this subtask of Phase 1. The total parallel running time is $2(m + 1)/p = (n + 1)/p$.

(b) We then compute $t = x^2$ and then by performing a parallel prefix operation we compute t, t^2, t^3, t^p. This requires a total of p M's. Since these can be performed in parallel no processor performs more than $\lceil \lg p \rceil = \lg p + 1$ M's (and the 1 is absorbed in c hereafter).

The two steps can be interleaved. Between the two steps, the computation of step (a) takes more time. Therefore the total time for Phase 1 is $(n + 1)/p$.

Munro-Paterson method (continued) ***WILL NOT BE COVERED***

Phase 2. Let $b_0 = a_0 + a_1x$, $b_1 = a_2 + a_3x \ldots, b_m = a_{n-1} + a_nx$.
Then, for $t = x^2$, $f(x)$ can be rewritten as follows.

$$f(x) = b_0 + b_1x^2 + b_2x^4 + \ldots + b_mx^{2m} = b_0 + b_1t + b_2t^2 + \ldots + b_mt^m.$$

We note that the m-th term $b_mt^m = (a_{n-1} + a_nx)t^m = (a_{n-1} + a_nx)x^{2m} = (a_{n-1}x^{n-1} + a_nx^n)$ contributes the two highest degree terms of $f(x)$. Then, we can rewrite $f(x)$ as follows.

$$\begin{aligned} f(x) &= t^0(b_0 + b_pt^p + b_{2p}t^{2p} + \ldots + b_{k_0p}t^{k_0p}) + \\ &\quad t^1(b_1 + b_{p+1}t^p + b_{2p+1}t^{2p} + \ldots + b_{k_1p+1}t^{k_1p}) + \\ &\ldots \\ &\quad t^{p-1}(b_{p-1} + b_{2p-1}t^p + b_{3p-1}t^{2p} + \ldots + b_{k_{p-1}p+1}t^{k_{p-1}p}) \\ &= t^0g_0(t^p) + t^1g_1(t^p) + \ldots + t^ig_i(t^p) + \ldots + t^{p-1}g_{p-1}(t^p) \end{aligned}$$

where $g_i(t^p)t^i$ is the sum of all b_jt^j such that $j \equiv i \bmod p$. Each $g_i(t^p)$ is a polynomial of degree $k_i \leq m/p$ in t^p.

Phase 2 then consists of the following three steps (a), (b) and (c).

Munro-Paterson method (continued) ***WILL NOT BE COVERED***

(a) Assign processor i to computing $g_i(t^p)$ for all $0 \leq i \leq p-1$. t^p was computed in step (b) of Phase 1. Processor i then evaluates polynomial $g_i(t^p)$ given by

$$g_i(t^p) = b_i + b_{p+i}(t^p)^1 + b_{2p+i}(t^p)^2 + \ldots + b_{k_ip+i}(t^p)^{k_i}$$

The polynomial is of degree an most m/p in t^p, and thus its evaluation requires m/p A's and m/p M's by using Horner's rule for a total of $2m/p = (n-1)/p$.

(b) Compute on processor i, $g_i(t^p)t^i$, in parallel, by performing one multiplication per processor noting that all the t^i's have become available since the parallel prefix of Phase 1.

(c) The p results $g_i(t^p)t^i$ of step (b) are summed up in $\lg p$ steps to evaluate f at x. Using a parallel sum no processor performs more than $\lceil \lg p \rceil$ A's for a total of $\lg p$ parallel A's (and one more unit contribution to c).

Total running time of all steps is $(n+1)/p + (n-1)/p + \lg p + 1 + d = 2n/p + \lg p + c$. Note that d is no more than 2, and thus c is no more than 3.

Let $G = (V, E)$ be an undirected graph. 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 we ought to write $< u, v > \in A$. For simplicity, we will 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 a 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.

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** or also called **pointer doubling** that finds many applications in designing algorithms for linked list and graph theory problems.

Problem Given forest F and array P construct array R where $R(j)$ 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 $R(i)$ is 2^k (in terms of number of edges in the original graph) unless $R(i)$ 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,R)
Input:  Forest on n vertices represented by the parent array P[...].
Output: An array R[...] giving the root of the tree containing each vertex
1.      ∀ i dopar
2.          R(i) = P(i) ;
3.      enddo
4.      ∀ i dopar
5.          while(R(i) ≠ R(R(i)))
6.                R(i) = R(R(i)).
7.      enddo
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, in the remainder, we primarily use the alternative definition of work W, i.e. 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.

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 FIND_ROOT (P,R)
Input:  Forest on n vertices represented by the parent array P[...].
Output: An array R[...] giving the root of the tree containing each vertex
1.      ∀ i dopar
2.          R(i) = P(i) ; W(i)=V(i);          ⟸ : New line
3.      enddo
4.      ∀ i dopar
5.          while(R(i) ≠ R(R(i)))
5a              W(i) = W(i) + W(R(i)).   ⟸ : New line
6.              R(i) = R(R(i)).
7.      enddo
end FIND_ROOT
```


Cycle Coloring - Symmetry Breaking

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 $c : V \rightarrow \{0, \ldots, k-1\}$ 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 property $P(S(i)) = i$. For a number i let $i = i_{n-1} \ldots i_1 i_0$ be its binary representation. Then i_k is the k-th lsb **(least significant bit)** of i. A wrapper function (initialization function for all coloring algorithms) is Color0 below.

```
0   3 Array S                   Cycle
1   0                           2->1->0->3->4->5->7->6-|
2   1                           ^                      |
3   4                           |______________________|
4   5
5   7
6   2
7   6                   Figure 1.
```

begin Color0 (P, S, c, n)
Input: A circular linked-list (ring) of n keys and $S(.)$
Output: Coloring c of n colors
1. $\forall 0 \leq i < n$ **dopar**
2. C(i)=i;
3. **if** $S(i) \neq$NULL $P(S(i)) = i$;
3. Color1 (P, S, c);

begin COLOR1 (P,S,c)
Input: Coloring c with r colors
Output: Coloring c with $2 \lg r$ colors
1. $\forall \quad 0 \leq i \quad < n$ **dopar**
2. Let k be the lsb position that $c(i)$ and $c(S(i))$ differ;
3. Set $nc(i) = 2k +$ (k-th lsb of $c(i)$); i.e. the pair (k, k-th lsb of $c(i)$);
4. Set $c(i) = nc(i)$.
end COLOR1

Claim 1. 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 $nc(i) = nc(j)$ for $(i, j) \in E$. These colors were obtained after an application of step 3, i.e. $nc(i) = 2k + c(i)|_k$ and $nc(j) = 2l + c(j)|_l$. Since $nc(i) = nc(j)$ we must have that $k = l$ and moreover $c(i)|_k = c(j)|_k$, 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. □

A question might arise in how to do the computation of step 3 of the algorithm. This would require a few computational operations. Consider $A = i - S(i)$, $B = A - 1$, and $A \oplus B$ ($\oplus$ = exclusive OR) to computer k.

A has zeroes in lsb positions with 1 in position k (where i and $S(i)$ differ). $B = A - 1$ has 0 in position k and 1's in lesser lsb positions. $A \oplus B$ can provide $k + 1$ in unary representation.

What Claim 1 tells us is that starting with an n coloring of a cycle we can get in one parallel step an $2(\lg n - 1) + 2$ coloring. This is because n colors can increase $nc(i)$ in line 3 into $2(\lceil \lg n \rceil - 1) + 1$ and thus the number of colors taking values $[0..2\lceil \lg n \rceil - 1]$ is $2\lceil \lg n \rceil$.

Let $\lg^{(1)} n = \lg n$, and let $\lg^{(i)} n = \lg \lg^{(i-1)} n$. Then, $\lg^* n$ is the minimum i such that $\lg^{(i)} n \leq 2$.

If we repeat COLOR 1 about $O(\lg^* n)$ times starting from an n coloring, we first get a $O(\lg n)$ coloring, then an $O(\lg \lg n)$ coloring and so on. After $O(\lg^* n)$ time steps and iteration of the basic coloring algorithm we end up say with a 3-bit coloring i.e. an 8-coloring. Can we get less? One more iteration turns this 8-coloring into a $2 * (3 - 1) + 1 + 1 = 6$ coloring. This is COLOR2. Having had a 6-coloring can we go further down? COLOR3 achieves a 3-coloring by perturbing a 6-coloring.

```
begin COLOR2  (P,S,c)
1.   ∀  0 ≤ i < n dopar    c(i) = i;
2.   repeat
3.       COLOR1  (P,S,c) ;
4.   until at most 6 colors are used in c.
end COLOR2
```

Claim 2. Algorithm COLOR2 6-colors a directed cycle.

Proof. Algorithm COLOR2 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 $O(\lg^*(n))$ 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 COLOR3 below that colors the vertices of a directed cycles with 3 colors.

```
begin COLOR3 (P,S,c)
1.   COLOR2 (P,S,c);
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 COLOR3
```

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 COLOR3 has the same asymptotic time complexity as COLOR2.

Symmetry Breaking: A fourth approximation

Algorithms COLOR3 or COLOR2 are not work-efficient because of the repeated calls to COLOR1. COLOR4 is work-efficient; it uses the integer sorting algorithm we introduced earlier that is work efficient and requires only a single call to COLOR1. By grouping nodes (as a byproduct of integer sorting in step 3) according to their color, in steps 4-8 only nodes of a certain color are active at a time and the processors assigned to these nodes. Therefore the work performed by COLOR4 can be reduced to $W_2 = \Theta(n)$ even though $T(n) = \Theta(\lg n)$ and $W = O(n \lg n)$.

begin COLOR4 (P,S,c)
1. $\forall \;\; 0 \leq i < n$ **dopar** $\quad c(i) = i$;
2. COLOR1 (P,S,c) ;
3. Sort vertices with respect to $c(i)$;
4. **for** $(j = 3; j \leq \lceil \lg n \rceil; j++)$
5. **begin**
6. $\forall\ 0 \leq i < n$ **with** $c(i) = j$ **dopar**
8. Color i with the smallest color in $\{0, 1, 2\}$ that is different from the colors of its two neighbors

end COLOR4

List Ranking: Introduction

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, $S(t) = NULL$. The problem of *list-ranking* is to determine the distance/rank $V(i)$ of each node i from the tail of the list. If $S(i) == NULL$, then $V(i) = 0$, otherwise $V(i) = V(S(i))+1$.

The sequential complexity of list ranking is linear in n and the sequential problem is a prefix-like problem that can be solved for example as follows.

- Find for every element i, its predecessor $P(i)$, in the list. One can find the predecessor of every i as follows: `if S(i) != NULL` then `P(S(i))=i`.
- Trace the list backwards from tail to the head of the list and compute $V(i)$ for every i.

The sequential complexity with $P = 1$ is $T = W = W_2 = O(n)$.

Parallel list ranking has many applications in parallel graph theory in particular.

Algorithm List1a uses pointer jumping/doubling. One can improve List1a by List1

```
begin List1a (S)
Input:    S(.) matrix
Output:   V(i) is rank or distance from tail of node i;
1.        ∀0 ≤ i < n do in parallel
2.            if  (S(i) ≠ NULL)  V(i) = 1;
3.                          else  V(i) = 0;
4.        ∀0 ≤ i < n do in parallel
5             B(i) = S(i);
6.        // left empty
7.            while (B(i) ≠ NULL ∧ B(B(i)) ≠ NULL) do
8.                V(i) = V(i) + V(B(i));
9.                B(i) = B(B(i));
end List1a
```

The time complexity of List1a on an EREW PRAM is $T = O(\lg n)$ and $W = O(n \lg n)$. The while loop can be removed as follows.


```
begin List1 (S)
Input:    S(.) matrix
Output:   V(i) is rank or distance from tail of node i;
1.        ∀0 ≤ i < n do in parallel
2.              if (S(i) ≠ NULL)      V(i) = 1;
3.                          else      V(i) = 0;
4.        ∀0 ≤ i < n do in parallel
5               B(i) = S(i);
6.        ∀0 ≤ i < n do in parallel
7.        for   k = 1  to   lg n  do
8.                        V(i) = V(i) + V(B(i));
9.                        B(i) = B(B(i));
end List1
```

Claim. Upon completion of List1, $V(i)$ has the rank of vertex i.

Proof. After iteration k, the following inveriants apply: (a) distance, i.e. number of edges between i and $S(i)$, is $V(i)$ (ignoring as edge the NULL pointer), and (b) as long as $S(i) \neq NULL$, $V(i) = 2^k$. Therefore after $\lceil \lg n \rceil$ iterations (to be more accurate) List1 computes the correct distance information. $\square$.

List Ranking: Improving upon LIST1

In the remainder we will show how we can reduce the work performed in list-ranking to linear; the corresponding running time will be $\omega(\lg n)$. We can reduce running time to $\Theta(\lg n)$ by a more elaborate algorithms and still maintain the linear work bound.

The idea behind the approach we will follow is easy: (a) do something elaborate but for few iterations. Few iterations means, that we will reduced the list ranking problems of ranking n elements into one of ranking $n/\lg n$ of them. (b) Use the non-optimal LIST1 algorithm to rank the $n/\lg n$ elements, and then (c) in $O(n)$ work time reduce the rank of each one of the original n elements from the ranks of the $n/\lg n$ elements.

Step (a) will use COLOR4. Step (b) uses LIST1 with $P = n/\lg n$, $T = O(\lg n)$, and thus $W = W_2 = O(n)$. The reconstruction of step3 will require, $O(\lg n)$ time and $O(n)$ work as a total.

We provide more details below. For the sake of an example, consider the very simple case where $S(i) = i + 1$, and $S(n-1) = NULL$. It is obvious now how to separate the odd-indexed from the even-indexed elements of the list in $O(1)$ time. If we manage to somehow rank the odd-indexed elements which are $n/2$ in number, then ranking the even-indexed elements is an $O(1)$ operation: for an even k, $V(k) = V(S(k)) + 1$, and $V(S(k))$ is the rank of an odd-indexed element whose rank was somehow computed. In practice however we cannot have such a nice even split.

List Ranking: Preliminaries for LIST2

The key to the list ranking algorithm LIST2 (L) that we will present are the following operations described in the previous page

- to find a large enough subset T of the nodes which contains at least a constant fraction of the elements of the list, but no two successive elements of input linked list L.
- Every element of T is then removed from L; if t is such an element we can do that by performing an $S(P(t)) = S(t)$ in parallel for every $t \in T$.
- the remainder $L - T$ is compacted with array S storing the $S(.)$ pointers residing into a smaller array M. The compaction operations uses parallel prefix (a homework problem).
- Recursively list-rank M.
- Use the M ranking to rank T then, thus ranking all elements of L. This is possible since elements of T are not successive in L, thus for $t \in T$ we know that $S(t) \notin T$.

We give below an outline of the work-optimal list-ranking algorithm in the form of the summarized LIST3.

begin LIST3 (L,S,P,V)
Input: Linked list L, with $S(.), P(.)$ successor/predecessor arrays
Output: $V(i)$ is rank or distance from tail of node i;
1. Run Algorithm LIST2 outlined above until problem size reduced to $n/\lg n$. // LIST2 code on page after next
2. Run LIST1 on the output;
end LIST3

The running time of LIST3 can be summarized by the following recurrence

$$T(n) = T(dn) + O(\lg n)$$

where b is a constant less than one. The solution to the recurrence is $O(\lg^2 n)$ if base case is $T(1) = 1$. dn is the size of the linked list remaining after the compaction. If however, we stop the algorithm, as we do, at $T(n/\lg n)$, then $T(n) = O(\lg n \lg\lg n)$. For work the corresponding recurrence is

$$W_2(n) = W_2(dn) + \Theta(n)$$

which gives $W_2(n) = \Theta(n)$.

We provide a helpful definition before we detail the steps of LIST2 that will complete the description of LIST3.

Definition. A set T which is a subset of a linked-list L is called a b-ruling set of L, $0 < b \leq 1/2$, if and only if the following two conditions apply

1. If an element t is not in T, then at least one of its b successors is in T; for example for a 2-ruling set $S(t)$ or $S(S(t))$ must be in T, if t is not in T.

2. If element t is in T, then $S(t)$ is not in T.

Thus finding T is equivalent to finding a ruling set. Finding T will be equivalent in filling array T such that $T(i) = 1$ indicates i is in T and $T(i) = 0$ indicates that i is not in T. Property 1 is equivalent to saying that any $b + 1$ successive elements of L contain at least one element in T.

We now make a connection between the ring-coloring algorithms and 2-ruling sets. If the nodes of a ring are 3-colored, then a 2-ruling set can be established by the nodes of a single color. A 2-ruling set exists of size at least $n/3$.

begin LIST2 (L,S,P,V,n)
Input: Linked list L, with $S(.), P(.)$ successor/predecessor arrays
Output: $V(i)$ is rank or distance from tail of node i;
1. **if** $n == 1$ **return**;
2. T=RULING_SET (b, n, L, S, P, V);
3. $\forall$ $0 \leq i < n$ **and** $i \in T$ **dopar**
4. V(P(i))=V(P(i))+V(i);
5. S(P(i))=S(i);
6. **enddo** ;
7. M=Compact $(L - T, T, L, P, S, V, n)$; //Compact L into M
8. LIST2(M, S, P, V, m); // $m =\mid L - T \mid$
9. $\forall$ $0 \leq i < n$ **and** $i \in T$ **dopar**
10. $V(i) = V(i) + V(S(i))$;
end LIST2

An example explains the steps

```
          1   1   1   1   1   1   1   1   1   1   0
L         o-->o-->o-->o-->o-->o-->o-->o-->o-->o-->o
              *       *           *       *  : to become T

          2       2       1   2       2       1   0
L-T       o------>o------>o-->o------>o------>o-->o
           recur
          10      8       6   5       3       1   0
L-T       o------>o------>o-->o------>o------>o-->o
          retrieve L
          10  9   8   7   6   5   4   3   2   1   0
L         o-->o-->o-->o-->o-->o-->o-->o-->o-->o-->o
```


List Ranking: RULING_SET

Task: Starting with a linked list L and its successor array S find a 2-ruling set T.

```
begin RULING_SET (b,n,L,S,P,V)
Input:  Linked list L in the form of successor S(.) matrix
Output: T() is a 2-ruling set indicator array of L; if T(i) = 1, then i ∈ T.
1.      ∀0 ≤ i < n do in parallel
2.          P(S(i)) = i     if    S(i) ≠ NULL.
3.          c(i) = i;
4.      COLOR1 (P,S,c); // One iteration of COLOR1; c[i] ∈ [0, 2 lg n − 1].
5.      ∀0 ≤ i < n do in parallel
6.          T(i) =1;
7.      for  k = 0 to  2 lg n − 1      do
8.          ∀0 ≤ i < n with c(i) = k  do in parallel
9.              if    T(P(i)) == T(S(i)) == 1
10.             then T(i) = 0;
11.     endfor
end RULIN_SET
```

Claim. Set T is a 2-ruling set.

Proof. An element i with $c(i)$ is selected to be in T at iteration $c(i)$. $S(i)$ and $P(i)$ are of different colors from i and thus not looked at this iteration. If i is selected in T, then $S(i)$ will not be selected because of step 9; so won't be $P(i)$. If element i was not selected in T, then $P(i)$ or $S(i)$ must be; otherwise if both $P(i)$ and $S(i)$ are not selected, RULING_SET should select i. □. Steps 1-4 take $T = O(1)$ and $W = O(n)$. Steps 5-6 have the same bounds. Number of iterations is $O(\lg n)$ and so is $T = O(\lg n)$. $W_2 = O(n)$. □.

As a conclusion,

Thm. LIST3 has $T = O(\lg n \lg\lg n)$, $P = n/\lg n$, and $W = \Theta(n)$.

