

CSE 4351/5351 Notes 9: PRAM and Other Theoretical Models

Shared Memory Model

Traditional Sequential Algorithm Model

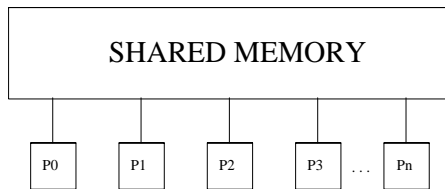
RAM (Random Access Machine)

Uniform access time to memory

Arithmetic operations performed in $O(1)$ time (simplification, really $O(\lg n)$)

Generalization of RAM to Parallel Computation

PRAM (Parallel Random Access Machine)



FAMILY of models - different concurrency assumptions imply DRASTIC differences in hardware/software support or may be impossible to physically realize.

MAJOR THEORETICAL ISSUE: What is the penalty for modifying an algorithm in a flexible model to satisfy a restrictive model?

MAJOR PRACTICAL ISSUES: Adequacy for mapping to popular topologies. Developing topologies/algorithms to support models with minimum penalty.

Other models:

Valiant's BSP (bulk-synchronous parallel), CACM 33 (8), August 1990

"Large" supersteps

Bulk communication/routing between supersteps

Synchronization to determine superstep completion

Karp's LogP, ACM Symp. on Parallel Algorithms and Architectures, 1993

Processors can be:

Operational: Computing, receiving, submitting a message

Stalling: Delay in message being accepted by communication medium

Parameters:

L: Maximum time for message to arrive

o: Overhead for preparing each message

g: Time units between consecutive communication handling steps

P: Maximum processor computation time

PRAM algorithms are usually described in a SIMD fashion

Highly synchronized by constructs in programming notation

Instructions are modified based on the processor id

Algorithms are simplified by letting processors do useless work

Lends to data parallel implementation - PROGRAMMER MUST REDUCE SYNCHRONIZATION OVERHEAD

Example PRAM Models

EREW (Exclusive Read, Exclusive Write) - Processors must access different memory cells. Most restrictive.

CREW (Concurrent Read, Exclusive Write) - Either multiple reads or a single writer may access a cell

ERCW (Exclusive Read, Concurrent Write) - Not popular

CRCW (Concurrent Read, Concurrent Write) - Most flexible.

What is a concurrent write?

Priority write by processor id

Equal write requirement

Combining: Sum of all written values, highest value, etc.

Sample Problem 1: Summing numbers

Input: N numbers in cells $A[0] \dots A[N-1]$

N processors, $0 \dots N-1$

Output: result in $A[N-1]$

CRCW with concurrent write by addition:

1. For all processors i : read $A[i]$ into local variable x

2. For all processors i : $A[N-1] := x$

RUNS IN $O(1)$ TIME!?!?

CREW: (Assume A may be overwritten)

```

for  $i:=0$  to  $\text{floor}(\lg(N-1))$  do
  for all processors  $j:=0$  to  $N-1$  do
    Read  $A[j]$  into local variable  $x$ 
    if  $j - 2^i \geq 0$  then
      Read  $A[j-2^i]$  into local variable  $y$ 
       $A[j] := x+y$ 

```

Runs in $O(\lg N)$ time

EREW: Same as CREW

Sample Problem 2: Sorting

Input: N numbers in cells $A[1] \dots A[N]$

N^2 processors: $P_{i,j}$, $1 \leq i \leq N$, $1 \leq j \leq N$

Output: Sorted numbers in array A

CRCW with concurrent write by addition:

Counting sort using array $c[]$ to maintain counts:

Assume input array A is

1	2	3	4
1	3	2	1

Processor $P_{i,j}$ compares $A[i]$ and $A[j]$ and "stores" in hypothetical array element $h[i][j]$:

	1	2	3	4
1	0	0	0	0
2	1	0	1	1
3	1	0	0	1
4	1	0	0	0

Now, add the elements of row i to get $c[i]$, which tell us where to place input $A[i]$:

	1	2	3	4
	0	3	2	1

```

for all processors  $P_{i,j}$  do
  Read  $A[i]$  into local variable  $x$ 
  Read  $A[j]$  into local variable  $y$ 
  if  $x > y$  or ( $x = y$  and  $i > j$ ) then
     $c[i] := 1$ 
  else
     $c[i] := 0$ 

```

```

for all processors  $P_{i,1}$  do
   $A[1+c[i]] := x$ 

```

Runs in $O(1)$ time, but not efficient.

CREW:

```

for all processors  $P_{i,j}$  do
  Read  $A[i]$  into local variable  $x$ 
  Read  $A[j]$  into local variable  $y$ 
  if  $x > y$  or ( $x = y$  and  $i > j$ ) then
     $c[i,j] := 1$ 
  else
     $c[i,j] := 0$ 

for  $k := 0$  to  $\text{floor}(\lg(N-1))$  do
  for all processors  $P_{i,j}$  do
    Read  $c[i,j]$  into local variable  $z$ 
    if  $j - 2^k \geq 1$  then
      Read  $c[i, j-2^k]$  into local variable  $y$ 
       $c[i,j] := z+y$ 

```

```

for all processors  $P_{i,N}$  do
   $A[1+c[i,N]] := x$ 

```

Runs in $O(\lg N)$ time

EREW:

Problem: Handling concurrent reads in first "for all" of CREW algorithm

Solution: Broadcasting. For example, replace "Read A[i] into local variable x" by

```

if j=1 then
    val[i,j] := A[i]
else
    val[i,j] := 0

for k:=0 to floor(lg (N-1)) do
    if j+2k ≤ N then
        Read val[i,j] into local variable v
        val[i,j+2k] := v
    Read val[i,j] into local variable x
  
```

For n=8:

1	2	3	4	5	6	7	8
X	A	B	C	D	E	F	G
X	X	A	B	C	D	E	F
X	X	X	X	A	B	C	D
X	X	X	X	X	X	X	X

THE TOUGHER NUT OF ARBITRARY CONCURRENT READS ("Multiple Broadcasting") ALSO INCURS $O(\lg N)$ PENALTY

"Oblivious" Algorithm - read/write/communication pattern is unaffected by the particular data for the problem size

Oblivious \Rightarrow good static mapping to particular topologies or derivation of similar algorithm for topology

Not Oblivious \Rightarrow the read/write/communication pattern must be handled dynamically

Examples:

Summing - oblivious

Counting sort - not oblivious

Matrix multiplication - oblivious

Most sorts - not oblivious (but bitonic mergesort is)

Dynamic programming - oblivious in most simple cases

Huffman coding - not oblivious

Graph algorithms - common sequential algorithms are not oblivious, but matrix-based algorithms are oblivious

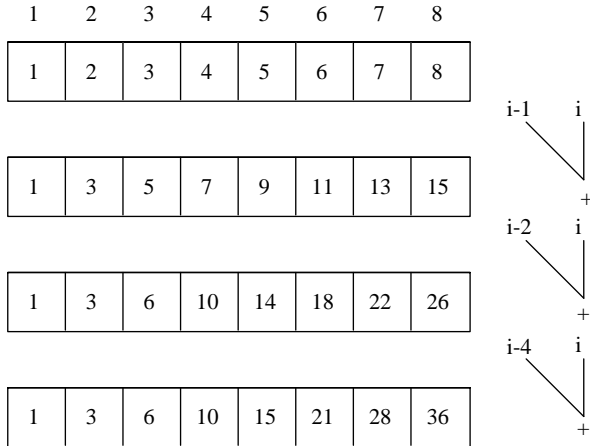
KMP string search - not oblivious

PARALLEL PREFIX (SUMS)

Suppose that $f(x,y)$ is an associative ($f(x,f(y,z)) = f(f(x,y),z)$) function. The parallel prefix computation of x_1, x_2, \dots, x_n under f is z_1, z_2, \dots, z_n , where $z_1 = x_1$ and $z_i = f(z_{i-1}, x_i)$ for $i \geq 2$. Most commonly, $f(x,y) = x + y$, i.e. compute array S where:

$S_i = S_{i-1} + A_i$, that is, $S_1 = A_1$ and $S_j = \sum_{i=1}^j A_i$. Assuming that an instance of $f(x,y)$ is evaluated in constant time, parallel prefix may be computed in $\theta(\lg n)$ time on n processors.

Example:



Analysis:

$$P = N$$

$$\Gamma = \theta(N)$$

$$T = \theta(\lg N)$$

$$S = \theta\left(\frac{N}{\lg N}\right)$$

$$W = \theta(N \lg N)$$

$$E = \theta\left(\frac{1}{\lg N}\right)$$

Even though the algorithm is asymptotically inefficient, an efficient blocked version may be derived.

Blocked Prefix Sum

$$\text{Let } P = \frac{N}{\lg N} \quad \left(\text{Assume } N \text{ is a power of } 2 \text{ for convenience} \right)$$

Algorithm:

for all processors i do
 processor i computes sum of N/P input elements
 processor i writes its sum in $B[i]$

Use original prefix sum algorithm to sum the P results in Array B

Processor i re-sums its elements, but also includes $B[i-1]$ (if $i > 1$).

$$\Gamma = \theta(N)$$

$$T = \theta(\lg N + \lg N - \lg \lg N) = \theta(\lg N)$$

$$S = \theta\left(\frac{N}{\lg N}\right)$$

$$W = \theta(N)$$

$$E = \theta(1)$$

Example: Evaluate linear recurrence $x_i = a_i x_{i-1} + d_i$ to find the value of x_i , $1 \leq i \leq n$ given the values of $x_0, a_1, a_2, \dots, a_n$ and d_1, d_2, \dots, d_n . Matrices are a convenient way to capture this problem as parallel prefix. If we view the recurrence as:

$$\begin{pmatrix} x_i \\ 1 \end{pmatrix} = \begin{pmatrix} a_i & d_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ 1 \end{pmatrix}$$

then the input to parallel prefix is the sequence of matrices:

$$\begin{pmatrix} x_0 \\ 1 \end{pmatrix}, \begin{pmatrix} a_1 & d_1 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} a_2 & d_2 \\ 0 & 1 \end{pmatrix}, \dots, \begin{pmatrix} a_n & d_n \\ 0 & 1 \end{pmatrix} \text{ and } f(x,y) = yx \text{ (i.e. matrix multiplication)}$$

Example: Evaluate linear recurrence $x_i = a_i x_{i-1} + b_i x_{i-2} + c_i$ to find the value of x_i , $2 \leq i \leq n$ given the values of $x_0, x_1, a_2, \dots, a_n, b_2, b_3, \dots, b_n$ and c_2, c_3, \dots, c_n . This recurrence may be viewed as

$$\begin{pmatrix} x_i \\ x_{i-1} \\ 1 \end{pmatrix} = \begin{pmatrix} a_i & b_i & c_i \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_{i-1} \\ x_{i-2} \\ 1 \end{pmatrix}$$

The input to the parallel prefix is the sequence of matrices:

$$\begin{pmatrix} x_1 \\ x_0 \\ 1 \end{pmatrix}, \begin{pmatrix} a_1 & b_1 & c_1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} a_2 & b_2 & c_2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \dots, \begin{pmatrix} a_n & b_n & c_n \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and $f(x,y) = yx$ (i.e. matrix multiplication).

Example: Binary addition in a bit-level parallel model

Ordinary serial addition takes $O(n)$ time, where n is the maximum number of bits in either of the two numbers.

If rippling of carries can be anticipated (carry-lookahead), then addition can be achieved in $O(\lg n)$ time.

Step 1: Label each bit position in $O(1)$ time based on single-bit addition. Carry bit is labeled as:

g = generates carry to the left by adding together two ones, carry received from right is irrelevant
 p = will propagate a carry received from the right to the left, since a 0 and a 1 were added
 s = stops carry received from right, since two zeroes were added

```

1 0 0 1 0 1 0 1 1 0 0 1
0 1 1 1 0 0 1 0 1 0 1 1
-----

```

1 1 1 0 0 1 1 1 0 0 1 0 \leftarrow one's bits

p p p g s p p p g s p g s \leftarrow the rightmost position corresponds to carry-in

Step 2: Carries are rippled using a right-to-left prefix sum in $O(\lg n)$ time using the following associative operator

<u>right</u>	left		
			s p g
	s		s s g
	p		s p g
	g		s g g

Confirm the following: if rightmost bit (carry-in) is s or g, then result will not have a p.

Phases of right-to-left prefix sum:

```

0:  p p p g s p p p g s p g s (initial)
1:  p p g g s p p g g s g g s (offset by 1)
2:  g g g g s g g g g s g g s (offset by 2)
3:  g g g g s g g g g s g g s (offset by 4)
4:  g g g g s g g g g s g g s (offset by 8)

```

Step 3: Take result from Step 2, replace g's by 1's and s's by 0's, then add (discarding carry) to one's bits from Step 1

```

      1 1 1 0 0 1 1 1 0 0 1 0  $\leftarrow$  one's bits
      1 1 1 1 0 1 1 1 1 0 1 1 0  $\leftarrow$  from phase 4 of Step 2
-----
      1 0 0 0 0 1 0 0 0 0 1 0 0  $\leftarrow$  Final result

```

PRAM matrix multiplication of two $n \times n$ matrices:

1. $n = 2^q$
2. $n^3 = 2^{3q}$ processors
3. Processor $P(i,j,k)$ computes $A[i,k] * B[k,j]$
4. Processors $P(i,j,*)$ use prefix sum to compute output value $C[i,j]$

Takes $\theta(\lg n)$ time, but has $\theta(1/\lg n)$ efficiency. Easy to make efficient by using $n^3/\lg n$ processors:

1. Processors indexed as $P(i,j,k)$. $0 \leq i < n$, $0 \leq j < n$, $0 \leq k < n/\lg n$
2. Processor $P(i,j,k)$ computes $\sum A[i,k'] * B[k',j]$ for $k * \lg n \leq k' < (k + 1) \lg n$,
i.e. $\lg n$ products per processor
3. Processors $P(i,j,*)$ perform prefix sum to output value $C[i,j]$ in $\theta(\lg n)$ time

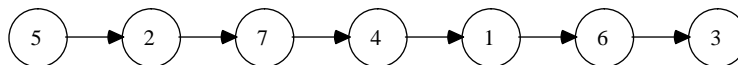
Overall: $\theta(1)$ efficiency

LIST RANKING

Another viewpoint: What if prefix sum is on a linked list instead of an ordered array?

Of course, if we have an arbitrary (e.g. malloc'd) linked list this is not possible. If the list is stored contiguously in an array, then a number of creative methods exist.

The most straightforward method is to convert the linked list into an array, which will be emphasized here. Suppose the following linked list:



This list may be stored as an array of successors. Notice that there is no node 0, but array location 0 points to the first node.

i	successor
0	5
1	6
2	7
3	0
4	1
5	2
6	3
7	4

By performing list ranking, we will be able to create a table of the node numbers in list traversal order:

i	node#
0	5
1	2
2	7
3	4
4	1
5	6
6	3
7	0

The most commonly cited method, pointer jumping, is horribly inefficient if applied blindly. A better method, randomization, has excellent expected performance:

0. Clear the marks. Each node will have one associated mark.
1. Randomly choose a subset S of the nodes and mark them. This subset should be larger than the set of processors and must include the first node in the list.
2. In parallel, using dynamic scheduling, traverse starting from each node in S . As unmarked nodes are encountered, mark them. When a marked node is encountered, we have reached the beginning of another sublist, so we save the indices of the starting node and the previously marked node that terminated this traversal along with the number of nodes in the sublist.
3. Determine the relative positions of the sublists in the entire list. This involves sorting the triples gathered in step 2 by the beginnings of the sublists and then using binary search to match up each terminator with the list that it begins.
4. In parallel, using dynamic scheduling based on the linked list of sublists constructed by step 3, construct the table listing node indices in traversal ordering.

The pthreads program listRankNotesPT.c compares sequential traversal, pointer jumping and the random technique:

```

Enter power of 2 for n
22
List size = 4194304
Enter seed
4444
Generate by Gray code (0) or 3**10 (1)?
1
List generated by jumping 3**10
Nodes traversed 4194304
Traversal CPU 1.270000
Enter # of threads for pointer jumping (0 to bypass)
2
  
```



```

Pointer jumping for thread 0 took 22.150000
#rounds 22
Pointer jumping for thread 1 took 22.120001
Parallel pointer jumping worked
Enter # of threads for pointer jumping (0 to bypass)
0
Enter # of threads & # of sublists for random (0 to bypass)
2 1000
Randomized list ranking took 3.260000
Randomized list ranking took 3.460000
Parallel randomized worked
Enter # of threads & # of sublists for random (0 to bypass)
4 1000
Randomized list ranking took 1.450000
Randomized list ranking took 1.430000
Randomized list ranking took 1.690000
Randomized list ranking took 1.720000
Parallel randomized worked
Enter # of threads & # of sublists for random (0 to bypass)
0 0

```

The MPI program `listRankNotesM.c` compares sequential traversal, pointer jumping and the random technique, but with the additional time needed for passing messages. The successor table is partitioned among all processors in a contiguous fashion. The "output" is to simply associate with each node its position from the beginning of the list. The characteristics of the five versions are:

`sequentialListTraversal` - Uses one token to traverse the list, so extreme amounts of blocking are used.

`pointerJumping1` - In each round of pointer jumping, each processor will request the successor from each of its elements whose pointer is not nil. Inefficiency results from the inherent number of rounds and the small messages that are used.

`pointerJumping2` - Similar to `pointerJumping1`, but longer messages are used in each round to group together several messages with the same destination.

`randomized1` - Similar in concept to the randomized pthreads program, but uses a token for collecting the nodes in each sublist. The computation (e.g. sorting, binary search) for ordering the sublists is duplicated on the processors. After ordering the sublists, each sublist is traversed again.

`randomized2` - Avoids the second traversal of sublists by saving additional information during the first traversal. For each node, we record the sublist (by using the index of the beginning sublist element) and the position of the node within the sublist. After the list ordering computation, each list node has its position augmented by the number of nodes in the sublists that precede its sublist.

One Processor (ketchup)

```

Enter sublist size as power of 2
20
Enter seed
111
Enter # of sublists for randomized
200
1048576 elements, 1048576 per processor
Calling generateListGray
Calling sequentialListTraversal
Processor 0 sequentialListTraversal time 6
pointerJumping1 skipped
pointerJumping2 skipped
Calling randomized1
pid 0: 200 lists with sublistTotal 1048576 into sort
Randomized list ranking 1 complete!
Processor 0 randomized1 time 51
Calling randomized2
Randomized list ranking 2 complete!
Processor 0 randomized2 time 34

```

Two Processors (ketchup)

Enter sublist size as power of 2

20

Enter seed

111

Enter # of sublists for randomized

200

1048576 elements, 524288 per processor

Calling generateListGray

Calling sequentialListTraversal

Processor 0 sequentialListTraversal time 6

pointerJumping1 skipped

pointerJumping2 skipped

Calling randomized1

pid 0: 200 lists with sublistTotal 1048576 into sort

Randomized list ranking 1 complete!

Processor 0 randomized1 time 12

Calling randomized2

pid 1: 200 lists with sublistTotal 1048576 into sort

Randomized list ranking 2 complete!

Processor 0 randomized2 time 8

Two Processors (ketchup, mustard)

Enter sublist size as power of 2

20

Enter seed

111

Enter # of sublists for randomized

200

1048576 elements, 524288 per processor

Calling generateListGray

Calling sequentialListTraversal

Processor 0 sequentialListTraversal time 7

pointerJumping1 skipped

pointerJumping2 skipped

Calling randomized1

pid 0: 200 lists with sublistTotal 1048576 into sort

Randomized list ranking 1 complete!

Processor 0 randomized1 time 11

Calling randomized2

pid 1: 200 lists with sublistTotal 1048576 into sort

Randomized list ranking 2 complete!

Processor 0 randomized2 time 8

Four Processors (two on each)

Enter sublist size as power of 2

20

Enter seed

111

Enter # of sublists for randomized

200

1048576 elements, 262144 per processor

Calling generateListGray

Calling sequentialListTraversal

Processor 0 sequentialListTraversal time 6

pointerJumping1 skipped

pointerJumping2 skipped

Calling randomized1

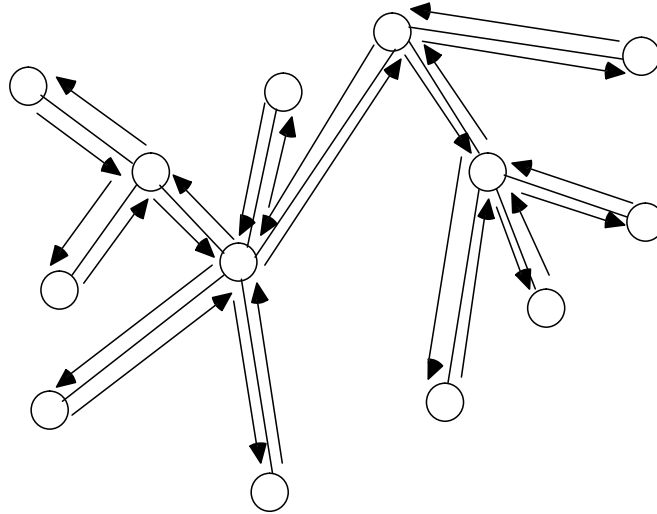
pid 0: 200 lists with sublistTotal 1048576 into sort

Randomized list ranking 1 complete!

Processor 0 randomized1 time 5
 Calling randomized2
 pid 2: 200 lists with sublistTotal 1048576 into sort
 pid 3: 200 lists with sublistTotal 1048576 into sort
 pid 1: 200 lists with sublistTotal 1048576 into sort
 Randomized list ranking 2 complete!
 Processor 0 randomized2 time 3

EULER TOURS FOR TREES

Given a free tree (i.e. a tree without a root, a connected acyclic undirected graph), determine a way to walk the tree such that each edge is traversed exactly once in each direction:



Of course, the tour is not unique. Finding a tour is easy if the graph is represented as an adjacency list structure with two table entries (one for each direction) for each edge:

Data Structure

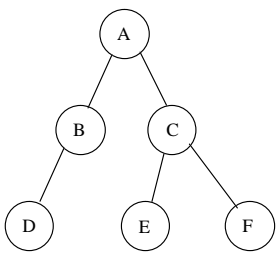
Each edge appears twice - once in each direction

Each edge has pointer to its reverse edge.

Each edge has pointer to the next edge with the same tail:

reverse next

A	B	C	D	E	F
1	2	6	4	8	10



1	AB	2	5
2	BA	1	3
3	BD	4	-
4	DB	3	-
5	AC	6	-
6	CA	5	7
7	CE	8	9
8	EC	7	-
9	CF	10	-
10	FC	9	-

Construction of linked Euler Tour for a Tree

Edges are handled in a highly concurrent fashion:

If reversal of edge (u,v) has a non-nil next pointer, then
 edge is followed by the next edge for its reversal
 else
 edge is followed by the first edge for v .

Example:

		reverse	next	ET successor
1	AB	2	5	3
2	BA	1	3	5
3	BD	4	-	4
4	DB	3	-	2
5	AC	6	-	7
6	CA	5	7	1
7	CE	8	9	8
8	EC	7	-	9
9	CF	10	-	10
10	FC	9	-	6

Euler tour may now be extracted into an array by:

1. Having one processor (P_n) read root of tree (which is arbitrary) and make $\text{etourLink}[e] := e$ for some edge into the root.
2. Apply list ranking.
3. For each edge i , its $\text{posn}[i] = 2|E| - \text{rank}(i)$, i.e. the position of the edge in the tour

Example:

Let B be the root and perform list ranking:

		reverse	next	ET link	rank
1	AB	2	5	1	0
2	BA	1	3	5	1
3	BD	4	-	4	1
4	DB	3	-	2	1
5	AC	6	-	7	1
6	CA	5	7	1	1
7	CE	8	9	8	1
8	EC	7	-	9	1
9	CF	10	-	10	1
10	FC	9	-	6	1

		reverse	next	ET link	rank
1	AB	2	5	1	0
2	BA	1	3	5	7
3	BD	4	-	4	9
4	DB	3	-	2	8
5	AC	6	-	7	6
6	CA	5	7	1	1
7	CE	8	9	8	5
8	EC	7	-	9	4
9	CF	10	-	10	3
10	FC	9	-	6	2

posn[i]	1	2	3	4	5	6	7	8	9	10
i	3	4	2	5	7	8	9	10	6	1
i	1	2	3	4	5	6	7	8	9	10
posn[i]	10	3	1	2	4	9	5	6	7	8

SOLVING TREE PROBLEMS

Can be solved quickly given that the Euler tour has been found and extracted for a particular root.

Forward Edge - an edge traversed before its reverse (in example: 3, 2, 5, 7, 9)

Retreat Edge - an edge traversed after its reverse (in example: 4, 1, 6, 8, 10)

Finding Parents - If (x,y) is forward, then x is the parent of y.

Preorder Numbering -

Suffix sum using posn array. Forward edges have a 1, retreat edges a 0.

preorder number for (x,y), a forward edge, is |E| - suffixSum for (x,y) + 1, root is 0

posn[i]	1	2	3	4	5	6	7	8	9	10	
i	3	4	2	5	7	8	9	10	6	1	
Tour	B	D	B	A	C	E	C	F	C	A	B
Sum-initial	1	0	1	1	1	0	1	0	0	0	
Suffix Sum	5	4	4	3	2	1	1	0	0	0	
Position of head in preorder	1		2	3	4		5				

(Example ordering is: B D A C E F, where B is the root which is not the head of any forward edge)

Postorder - count the number of retreat edges in Euler tour before a vertex (via prefix sums)

Number of Descendants -

For the tree's root, this is simply the number of vertices |V|

For all other vertices w, there must be a single forward edge (v,w).

The number of descendants for w is the difference in the suffix sum (as computed for preorder) for (v,w) and (w,v)

posn[i]	1	2	3	4	5	6	7	8	9	10	
i	3	4	2	5	7	8	9	10	6	1	
Tour	B	D	B	A	C	E	C	F	C	A	B
Sum-initial	1	0	1	1	1	0	1	0	0	0	
Suffix Sum	5	4	4	3	2	1	1	0	0	0	
# of descendants	1		4	3	1		1				
w	D		A	C	E		F				

Levels of Vertices -

Assign forward edges -1 and retreat edges +1.

Compute suffix sums.

head of forward edge has level = suffixSum+1

posn[i]	1	2	3	4	5	6	7	8	9	10	
i	3	4	2	5	7	8	9	10	6	1	
Tour	B	D	B	A	C	E	C	F	C	A	B
Sum-initial	-1	+1	-1	-1	-1	+1	-1	+1	+1	+1	
Suffix Sum	0	1	0	1	2	3	2	3	2	1	
level	1		1	2	3		3				
w	D		A	C	E		F				

root (B) has level 0

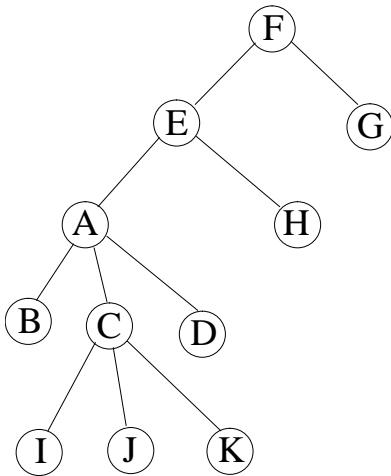
Notes 4 Problems:

1. Give an EREW algorithm to “cyclically” replicate the values in array locations a[0], a[1], . . . , a[k-1] throughout the array a[0], a[1], . . . , a[n-1]. The value of array element a[i] after executing the algorithm will be the value of array element a[i mod k] before executing the algorithm.
2. Use prefix sum concepts to evaluate the linear recurrence $x_i = a_i x_{i-1} + d_i$ to find the value of x_i , $1 \leq i \leq n$ given the following:

$x_0 = 10$

i	a_i	d_i
1	2	3
2	4	5
3	-6	7
4	2	3
5	4	5
6	2	3
7	4	5

3. Use prefix sum concepts to add 11001101 and 11110101.
4. Give the data structure, Euler tour, preorder traversal, postorder traversal, number of descendants, and levels of vertices when vertex A is used as the root of the tree. Order each linked list in ascending order by the head vertices.



1. Give an EREW algorithm to “cyclically” . . .

```

#include <stdio.h>
#include <stdlib.h>

main()
{
  int k,n,i,j,rounds,offset;
  int *table,*tabcpy;

  printf("Enter k & n: ");
  fflush(stdout);
  scanf("%d %d",&k,&n);

  table=(int*) malloc(n*sizeof(int));
  tabcpy=(int*) malloc(n*sizeof(int));

  for (i=0;i<n;i++)
    table[i]=i;

  i=k;
  for (rounds=0;i<n;rounds++)
    i*=2;

  offset=k;
  for (i=0;i<rounds;i++)
  {
    for (j=0;j<n;j++)
      tabcpy[j]=table[j];
    for (j=offset;j<n;j++)
      table[j]=tabcpy[j-offset];
    printf("-----\n");
    for (j=0;j<n;j++)
      printf("%2d %2d\n",j,table[j]);
    offset*=2;
  }
}

```

2. Use prefix sum concepts . . .

$$x_0 = 10$$

i	a_i	d_i
1	2	3
2	4	5
3	-6	7
4	2	3
5	4	5
6	2	3
7	4	5

```

[weems@va NOTES]$ cat linearRecurrence.c
#include <stdio.h>

main()
{
int i,j,k,p,offset;
int n=7;
int x[8];
int a[8]={-999,2,4,-6,2,4,2,4};
int d[8]={-999,3,5,7,3,5,3,5};

int rowDim[8],colDim[8];
int mat[8][2][2];
int matCpy[8][2][2];
int rowDimCpy[8],colDimCpy[8];

x[0]=10;

for (i=1;i<=n;i++)
    x[i]=a[i]*x[i-1]+d[i];

printf("Sequential using recurrence:\n");
for (i=0;i<=n;i++)
    printf("x[%d]=%d\n",i,x[i]);

printf("Parallel using prefix sum:\n");
rowDim[0]=2;
colDim[0]=1;
mat[0][0][0]=x[0];
mat[0][1][0]=1;

for (i=1;i<=n;i++)
{
    rowDim[i]=colDim[i]=2;
    mat[i][0][0]=a[i];
    mat[i][0][1]=d[i];
    mat[i][1][0]=0;
    mat[i][1][1]=1;
}

printf("Initial matrices:\n");
for (i=0;i<=n;i++)
{
    printf("matrix %d:\n",i);
    for (j=0;j<rowDim[i];j++)
    {
        for (k=0;k<colDim[i];k++)
            printf("%d ",mat[i][j][k]);
        printf("\n");
    }
}

for (offset=1;
    offset<n;
    offset*=2)
{
    for (i=0;i<=n;i++)
    {
        rowDimCpy[i]=rowDim[i];
        colDimCpy[i]=colDim[i];
        for (j=0;j<rowDim[i];j++)
            for (k=0;k<colDim[i];k++)
                matCpy[i][j][k]=mat[i][j][k];
    }

    for (i=offset;i<=n;i++)
    {
        /* Note reversing of matrix order */
        rowDim[i]=rowDimCpy[i];
        colDim[i]=colDimCpy[i-offset];
        for (j=0;j<rowDim[i];j++)
            for (k=0;k<colDim[i];k++)
                {

```



```

        mat[i][j][k]=0;
        for (p=0;p<colDimCpy[i];p++)
            mat[i][j][k]+=matCpy[i][j][p]*matCpy[i-offset][p][k];
    }
}
printf("matrices after offset %d:\n",offset);
for (i=0;i<=n;i++)
{
    printf("matrix %d:\n",i);
    for (j=0;j<rowDim[i];j++)
    {
        for (k=0;k<colDim[i];k++)
            printf("%d ",mat[i][j][k]);
        printf("\n");
    }
}

```

```

for (i=0;i<=n;i++)
    printf("x[%d]=%d\n",i,mat[i][0][0]);
}

```

```

[weems@va NOTES]$ a.out
Sequential using recurrence:

```

```

x[0]=10
x[1]=23
x[2]=97
x[3]=-575
x[4]=-1147
x[5]=-4583
x[6]=-9163
x[7]=-36647

```

```

Parallel using prefix sum:

```

```

Initial matrices:

```

```

matrix 0:

```

```

10
1

```

```

matrix 1:

```

```

2 3
0 1

```

```

matrix 2:

```

```

4 5
0 1

```

```

matrix 3:

```

```

-6 7
0 1

```

```

matrix 4:

```

```

2 3
0 1

```

```

matrix 5:

```

```

4 5
0 1

```

```

matrix 6:

```

```

2 3
0 1

```

```

matrix 7:

```

```

4 5
0 1

```

```

matrices after offset 1:

```

```

matrix 0:

```

```

10
1

```

```

matrix 1:

```

```

23
1

```

```

matrix 2:

```

```

8 17
0 1

```

```

matrix 3:

```

```

-24 -23
0 1

```

```

matrix 4:

```

```

-12 17
0 1

```

```

matrix 5:

```

```

8 17
0 1
matrix 6:
8 13
0 1
matrix 7:
8 17
0 1
matrices after offset 2:
matrix 0:
10
1
matrix 1:
23
1
matrix 2:
97
1
matrix 3:
-575
1
matrix 4:
-96 -187
0 1
matrix 5:
-192 -167
0 1
matrix 6:
-96 149
0 1
matrix 7:
64 153
0 1
matrices after offset 4:
matrix 0:
10
1
matrix 1:
23
1
matrix 2:
97
1
matrix 3:
-575
1
matrix 4:
-1147
1
matrix 5:
-4583
1
matrix 6:
-9163
1
matrix 7:
-36647
1
x[0]=10
x[1]=23
x[2]=97
x[3]=-575
x[4]=-1147
x[5]=-4583
x[6]=-9163
x[7]=-36647

```

3. Use prefix sum concepts to add 11001101 and 11110101.

```

11001101 number1
11110101 number2
ggppppsgs propagation
000111000 onesbits

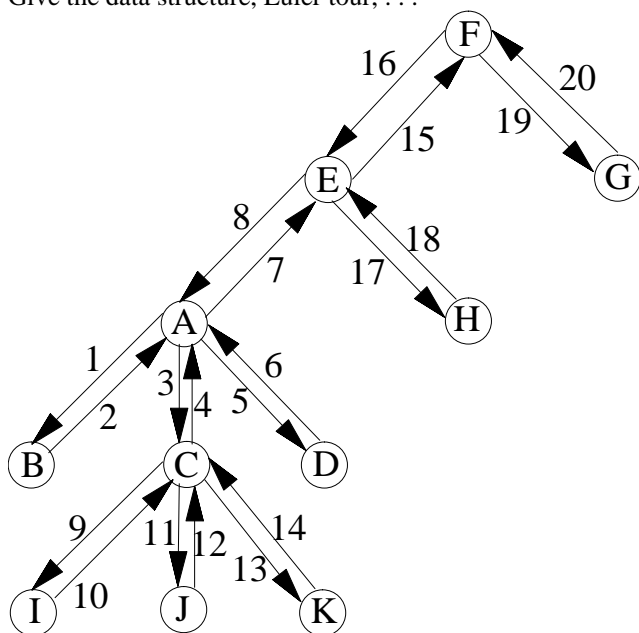
```

```

ggppppgs propagation from offset 1
gggggggs propagation from offset 2
gggggggs propagation from offset 4
gggggggs propagation from offset 8
11100010 output

```

4. Give the data structure, Euler tour, ...



```

A 1
B 2
C 4
D 6
E 8
F 16
G 20
H 18
I 10
J 12
K 14

```

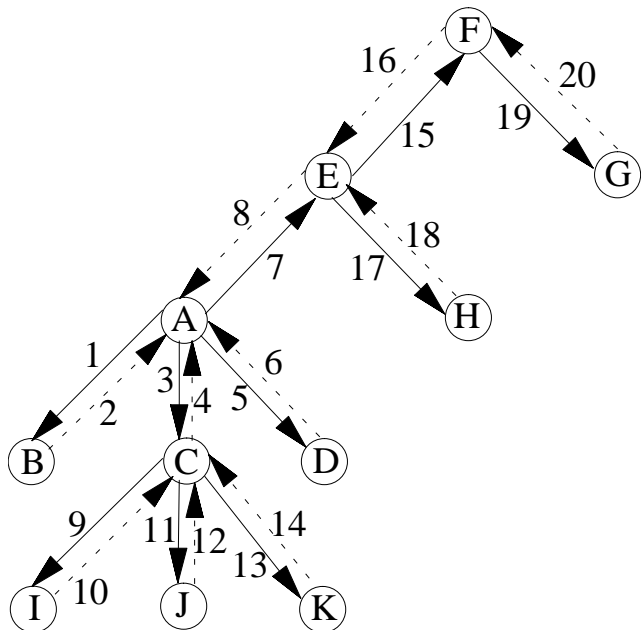
Edges:

	edge	reverse	next	ET successor	initial rank	final rank
1	AB	2	3	2	1	19
2	BA	1	-	3	1	18
3	AC	4	5	9	1	17
4	CA	3	9	5	1	10
5	AD	6	7	6	1	9
6	DA	5	-	7	1	8
7	AE	8	-	15	1	7
8	EA	7	15	(1) 8	0	0
9	CI	10	11	10	1	16
10	IC	9	-	11	1	15
11	CJ	12	13	12	1	14
12	JC	11	-	13	1	13
13	CK	14	-	14	1	12
14	KC	13	-	4	1	11
15	EF	16	17	19	1	6
16	FE	15	19	17	1	3
17	EH	18	-	18	1	2
18	HE	17	-	8	1	1

19	FG	20	-	20	1	5
20	GF	19	-	16	1	4

posn[edge]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
edge	<u>1</u>	2	<u>3</u>	<u>9</u>	10	<u>11</u>	12	<u>13</u>	14	4	<u>5</u>	6	<u>7</u>	<u>15</u>	<u>19</u>	20	16	<u>17</u>	18	8	
tour	A	B	A	C	I	C	J	C	K	C	A	D	A	E	F	G	F	E	H	E	A

Forward edges are underlined. In the following diagram forward edges are solid and retreat edges are dotted.

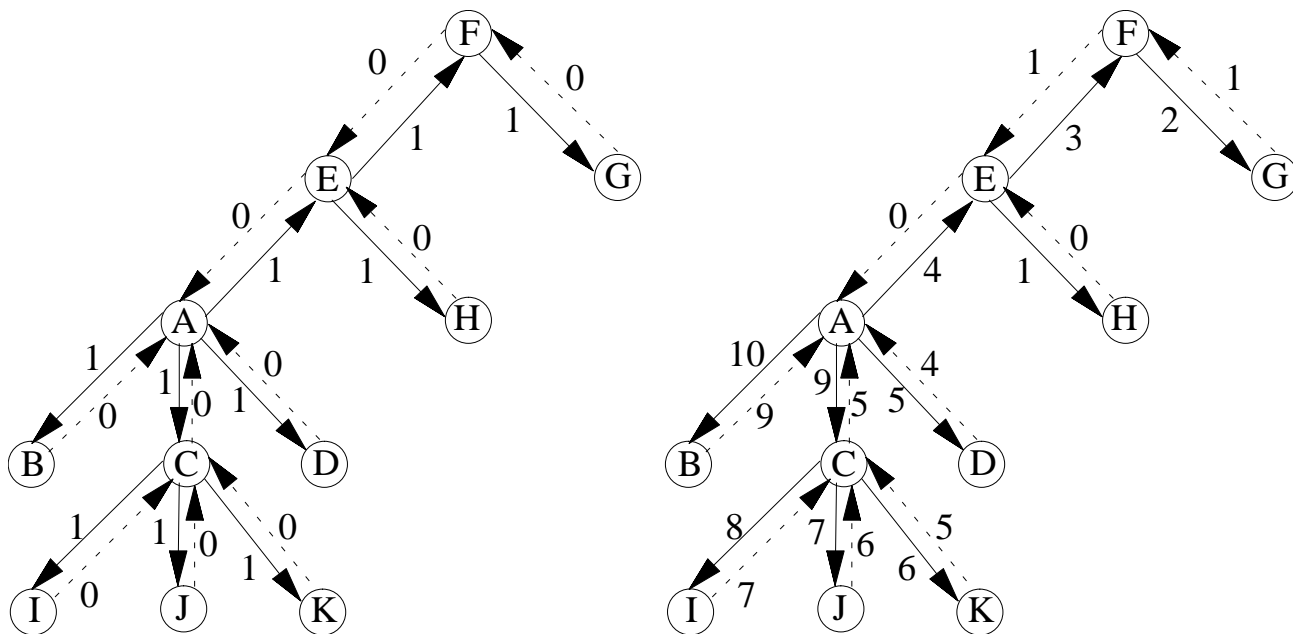


Preorder numbering - assign forward edges a 1, retreat edges a 0, then perform suffix sum:

posn[edge]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
edge	<u>1</u>	2	<u>3</u>	<u>9</u>	10	<u>11</u>	12	<u>13</u>	14	4	<u>5</u>	6	<u>7</u>	<u>15</u>	<u>19</u>	20	16	<u>17</u>	18	8	
tour	A	B	A	C	I	C	J	C	K	C	A	D	A	E	F	G	F	E	H	E	A
initial	1	0	1	1	0	1	0	1	0	0	1	0	1	1	1	0	0	1	0	0	
sum		10	9	9	8	7	7	6	6	5	5	5	4	4	3	2	1	1	1	0	0
E - sum + 1	1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
(E =10)																					

- 0 A
- 1 B
- 2 C
- 3 I
- 4 J
- 5 K
- 6 D
- 7 E
- 8 F
- 9 G
- 10 H

Figure for preorder numbering:

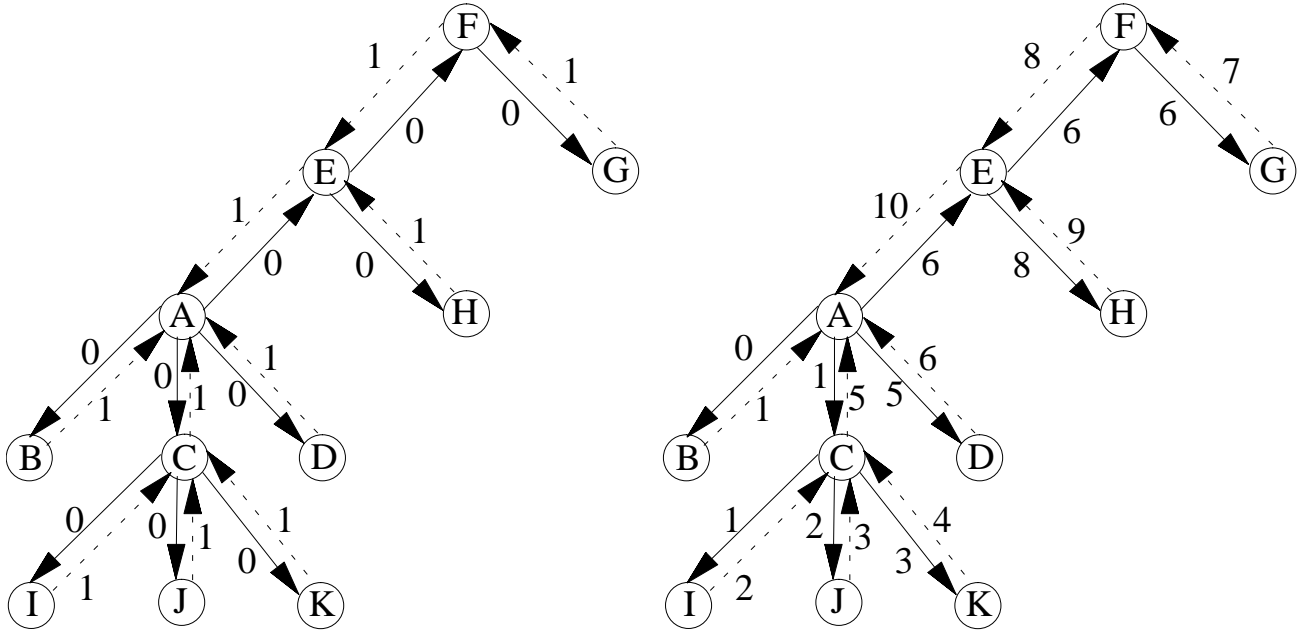


Postorder numbering - assign forward edges a 0, retreat edges a 1, then perform prefix sum. Assign number to tail of retreat edge:

posn[edge]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
edge	<u>1</u>	<u>2</u>	<u>3</u>	<u>9</u>	<u>10</u>	<u>11</u>	<u>12</u>	<u>13</u>	<u>14</u>	<u>4</u>	<u>5</u>	<u>6</u>	<u>7</u>	<u>15</u>	<u>19</u>	<u>20</u>	<u>16</u>	<u>17</u>	<u>18</u>	<u>8</u>	
tour	A	B	A	C	I	C	J	C	K	C	A	D	A	E	F	G	F	E	H	E	A
initial	0	1	0	0	1	0	1	0	1	1	0	1	0	0	0	1	1	0	1	1	
sum	0	1	1	1	2	2	3	3	4	5	5	6	6	6	6	7	8	8	9	10	

- 0 B
- 1 I
- 2 J
- 3 K
- 4 C
- 5 D
- 6 G
- 7 F
- 8 H
- 9 E
- 10 A (root)

Figure for postorder numbering:



Number of descendants - for root this is the number of vertices. For another vertex y, this is the difference between the forward edge sum for (x,y) and the retreat edge sum (y,x) from preorder numbering:

- A 11 = |V|
- B 1 = 10 - 9
- C 4 = 9 - 5
- D 1 = 5 - 4
- E 4 = 4 - 0
- F 2 = 3 - 1
- G 1 = 2 - 1
- H 1 = 1 - 0
- I 1 = 8 - 7
- J 1 = 7 - 6
- K 1 = 6 - 5

Levels of vertices - for root this is 0. Assign forward edges -1 and retreat edges +1. Perform suffix sum. Head of forward edge has level = suffix sum + 1

posn[edge]	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
edge	<u>1</u>	2	<u>3</u>	<u>9</u>	10	<u>11</u>	12	<u>13</u>	14	4	<u>5</u>	6	<u>7</u>	<u>15</u>	<u>19</u>	20	16	<u>17</u>	18	8	
tour	A	B	A	C	I	C	J	C	K	C	A	D	A	E	F	G	F	E	H	E	A
initial	-1	+1	-1	-1	+1	-1	+1	-1	+1	+1	-1	+1	-1	-1	-1	+1	+1	-1	+1	+1	
sum	0	1	0	1	2	1	2	1	2	1	0	1	0	1	2	3	2	1	2	1	

- A 0
- B 1
- C 1
- D 1
- E 1
- F 2
- G 3
- H 2
- I 2
- J 2
- K 2

Diagram for levels:

