

Unit-3

Sorting is a process of arranging elements in a group in a particular order, i.e., ascending order, descending order, alphabetic order, etc. Here we will discuss the following –

- Enumeration Sort
- Odd-Even Transposition Sort
- Parallel Merge Sort
- Hyper Quick Sort

Sorting a list of elements is a very common operation. A sequential sorting algorithm may not be efficient enough when we have to sort a huge volume of data. Therefore, parallel algorithms are used in sorting.

Enumeration Sort

Enumeration sort is a method of arranging all the elements in a list by finding the final position of each element in a sorted list. It is done by comparing each element with all other elements and finding the number of elements having smaller value.

Therefore, for any two elements, a_i and a_j , any one of the following cases must be true –

- $a_i < a_j$
- $a_i > a_j$
- $a_i = a_j$

Algorithm

```
procedure ENUM_SORTING (n)
```

```
begin
```

```
  for each process  $P_{i,j}$  do
```

```
     $C[j] := 0$ ;
```

```
  for each process  $P_{i,j}$  do
```

```
    if (A[i] < A[j]) or A[i] = A[j] and i < j) then
        C[j] := 1;
    else
        C[j] := 0;

    for each process P1, j do
        A[C[j]] := A[j];

end ENUM_SORTING
```

Odd-Even Transposition Sort

Odd-Even Transposition Sort is based on the Bubble Sort technique. It compares two adjacent numbers and switches them, if the first number is greater than the second number to get an ascending order list. The opposite case applies for a descending order series. Odd-Even transposition sort operates in two phases – odd phase and even phase. In both the phases, processes exchange numbers with their adjacent number in the right.

Unsorted								
9	7	3	8	5	6	4	1	Phase 1(Odd)
7	9	3	8	5	6	1	4	Phase 2(Even)
7	3	9	5	8	1	6	4	Phase 3(Odd)
3	7	5	9	1	8	4	6	Phase 4(Even)
3	5	7	1	9	4	8	6	Phase 5(Odd)
3	5	1	7	4	9	6	8	Phase 6(Even)
3	1	5	4	7	6	9	8	Phase 7(Odd)
1	3	4	5	6	7	8	9	
Sorted								

Algorithm

```

procedure ODD-EVEN_PAR (n)
begin
  id := process's label

  for i := 1 to n do
    begin
      if i is odd and id is odd then
        compare-exchange_min(id + 1);
      else
        compare-exchange_max(id - 1);

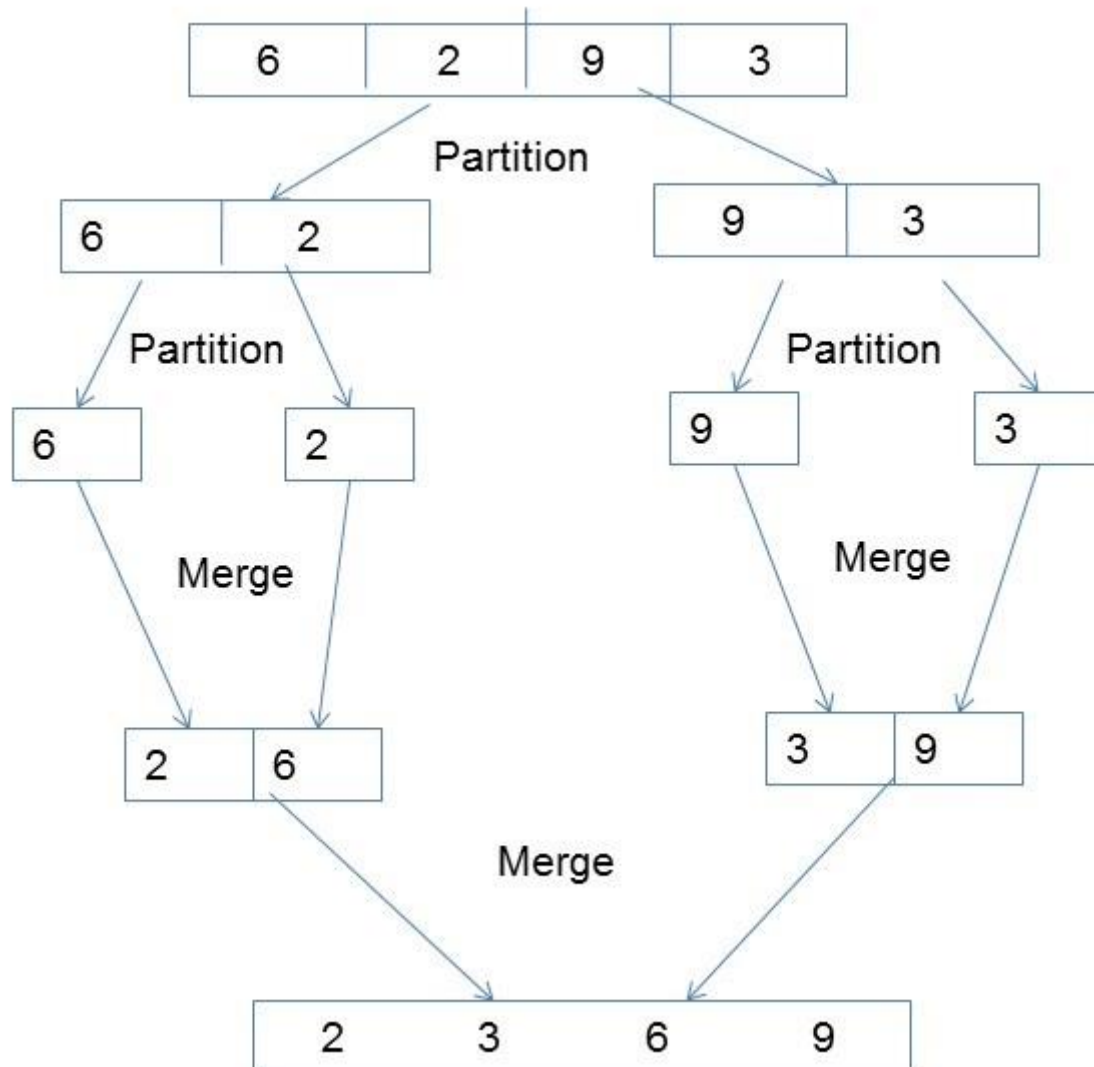
      if i is even and id is even then
        compare-exchange_min(id + 1);
      else
        compare-exchange_max(id - 1);
    end
  end

```

```
end for  
end ODD-EVEN_PAR
```

Parallel Merge Sort

Merge sort first divides the unsorted list into smallest possible sub-lists, compares it with the adjacent list, and merges it in a sorted order. It implements parallelism very nicely by following the divide and conquer algorithm.



Algorithm

```
procedureparallelmergesort(id, n, data, newdata)

begin
  data = sequentialmergesort(data)

  for dim = 1 to n
    data = parallelmerge(id, dim, data)
  endfor

  newdata = data
end
```

Hyper Quick Sort

Hyper quick sort is an implementation of quick sort on hypercube. Its steps are as follows –

- Divide the unsorted list among each node.
- Sort each node locally.
- From node 0, broadcast the median value.
- Split each list locally, then exchange the halves across the highest dimension.
- Repeat steps 3 and 4 in parallel until the dimension reaches 0.

Algorithm

```
procedure HYPERQUICKSORT (B, n)
begin

  id := process's label;

  for i := 1 to d do
    begin
      x := pivot;
```

```

partition B into B1 and B2 such that  $B1 \leq x < B2$ ;
if ith bit is 0 then

begin
    send B2 to the process along the ith communication link;
    C := subsequence received along the ith communication link;
    B := B1 U C;
endif

else
    send B1 to the process along the ith communication link;
    C := subsequence received along the ith communication link;
    B := B2 U C;
end else
end for

sort B using sequential quicksort;

end HYPERQUICKSORT

```

Searching is one of the fundamental operations in computer science. It is used in all applications where we need to find if an element is in the given list or not. In this chapter, we will discuss the following search algorithms –

- Divide and Conquer
- Depth-First Search
- Breadth-First Search
- Best-First Search

Divide and Conquer

In divide and conquer approach, the problem is divided into several small sub-problems. Then the sub-problems are solved recursively and combined to get the solution of the original problem.

The divide and conquer approach involves the following steps at each level –

- Divide – The original problem is divided into sub-problems.

- Conquer – The sub-problems are solved recursively.
- Combine – The solutions of the sub-problems are combined to get the solution of the original problem.

Binary search is an example of divide and conquer algorithm.

Pseudocode

```

Binarysearch(a, b, low, high)

if low < high then
    return NOT FOUND
else
    mid ← (low+high) / 2
    if b = key(mid) then
        return key(mid)
    else if b < key(mid) then
        return BinarySearch(a, b, low, mid-1)
    else

        return BinarySearch(a, b, mid+1, high)

```

Depth-First Search

Depth-First Search (or DFS) is an algorithm for searching a tree or an undirected graph data structure. Here, the concept is to start from the starting node known as the root and traverse as far as possible in the same branch. If we get a node with no successor node, we return and continue with the vertex, which is yet to be visited.

Steps of Depth-First Search

- Consider a node (root) that is not visited previously and mark it visited.
- Visit the first adjacent successor node and mark it visited.
- If all the successors nodes of the considered node are already visited or it doesn't have any more successor node, return to its parent node.

Pseudocode

Let v be the vertex where the search starts in Graph G .

DFS(G,v)

Stack S := {};

for each vertex u, set visited[u] := false;

push S, v;

while (S is not empty) do

 u := pop S;

 if (not visited[u]) then

 visited[u] := true;

 for each unvisited neighbour w of u

 push S, w;

 end if

 end while

END DFS()

Breadth-First Search

Breadth-First Search (or BFS) is an algorithm for searching a tree or an undirected graph data structure. Here, we start with a node and then visit all the adjacent nodes in the same level and then move to the adjacent successor node in the next level. This is also known as level-by-level search.

Steps of Breadth-First Search

- Start with the root node, mark it visited.
- As the root node has no node in the same level, go to the next level.
- Visit all adjacent nodes and mark them visited.
- Go to the next level and visit all the unvisited adjacent nodes.
- Continue this process until all the nodes are visited.

Pseudocode

Let v be the vertex where the search starts in Graph G.

BFS(G,v)

```

Queue Q := {};

for each vertex u, set visited[u] := false;
insert Q, v;
while (Q is not empty) do
    u := delete Q;

    if (not visited[u]) then
        visited[u] := true;
        for each unvisited neighbor w of u
            insert Q, w;
        end if
    end while
end while

END BFS()

```

Best-First Search

Best-First Search is an algorithm that traverses a graph to reach a target in the shortest possible path. Unlike BFS and DFS, Best-First Search follows an evaluation function to determine which node is the most appropriate to traverse next.

Steps of Best-First Search

- Start with the root node, mark it visited.
- Find the next appropriate node and mark it visited.
- Go to the next level and find the appropriate node and mark it visited.
- Continue this process until the target is reached.

Pseudocode

```

BFS( m )

Insert( m.StartNode )
Until PriorityQueue is empty
    c ← PriorityQueue.DeleteMin
    If c is the goal

```

```

Exit
Else

  Foreach neighbor n of c
    If n "Unvisited"
      Mark n "Visited"
      Insert( n )
    Mark c "Examined"

End procedure

```

Our first parallel algorithm operates on lists. We can store a list in a PRAM much as we store lists in an ordinary RAM. To operate on list objects in parallel, however, it is convenient to assign a "responsible" processor to each object. We shall assume that there are as many processors as list objects, and that the i th processor is responsible for the i th object. Figure 30.2(a), for example, shows a linked list consisting of the sequence of objects $\langle 3, 4, 6, 1, 0, 5 \rangle$. Since there is one processor per list object, every object in the list can be operated on by its responsible processor in $O(1)$ time.

Suppose that we are given a singly linked list L with n objects and wish to compute, for each object in L , its distance from the end of the list. More formally, if $next$ is the pointer field, we wish to compute a value $d[i]$ for each object i in the list such that

$$d[i] = \begin{cases} 0 & \text{if } next[i] = \text{NIL} , \\ d[next[i]] + 1 & \text{if } next[i] \neq \text{NIL} . \end{cases}$$

We call the problem of computing the d values the *list-ranking problem*.

One solution to the list-ranking problem is simply to propagate distances back from the end of the list. This method takes $\Theta(n)$ time,

since the k th object from the end must wait for the $k-1$ objects following it to determine their distances from the end before it can determine its own. This solution is essentially a serial algorithm.

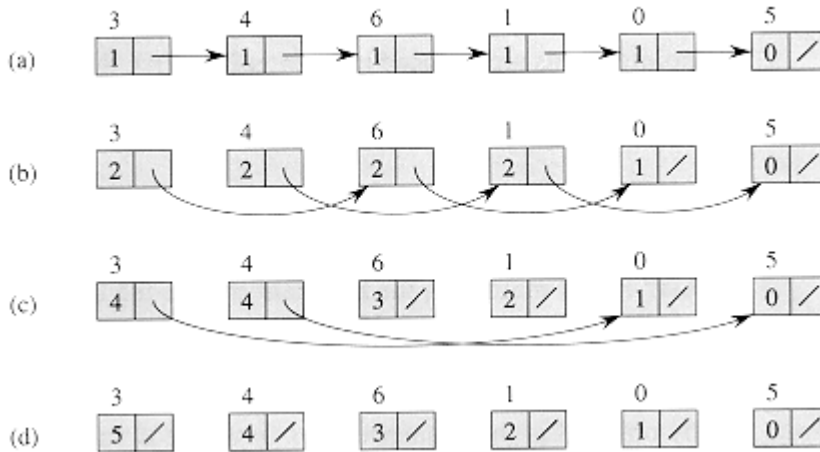


*Figure 30.2 Finding the distance from each object in an n -object list to the end of the list in $O(\lg n)$ time using pointer jumping. (a) A linked list represented in a PRAM with d values initialized. At the end of the algorithm, each d value holds the distance of its object from the end of the list. Each object's responsible processor appears above the object. (b)-(d) The pointers and d values after each iteration of the while loop in the algorithm *LIST-RANK*.*

An efficient parallel solution, requiring only $O(\lg n)$ time, is given by the following parallel pseudocode.

```

LIST-RANK( $L$ )
1  for each processor  $i$ , in parallel
2      do if  $next[i] = NIL$ 
3          then  $d[i] \leftarrow 0$ 
4          else  $d[i] \leftarrow 1$ 
5  while there exists an object  $i$  such that  $next[i] \neq NIL$ 
6      do for each processor  $i$ , in parallel
7          do if  $next[i] \neq NIL$ 
8              then  $d[i] \leftarrow d[i] + d[next[i]]$ 
9               $next[i] \leftarrow next[next[i]]$ 

```

Figure 30.2 shows how the algorithm computes the distances. Each part of the figure shows the state of the list before an iteration of the while loop of lines 5-9. Part (a) shows the list just after initialization. In the first iteration, the first 5 list objects have non-nil pointers, so that lines 8-9 are executed by their responsible processors. The result appears in part (b) of the figure. In the second iteration, only the first 4 objects have non-NIL pointers; the result of this iteration is shown in part (c). In the third iteration, only the first 2 objects are operated on, and the final result, in which all objects have NIL pointers, appears in part (d).

The idea implemented by line 9, in which we set $next[i] \leftarrow next[next[i]]$ for all non-nil pointers $next[i]$, is called *pointer jumping*. Note that the pointer fields are changed by pointer jumping, thus destroying the structure of the list. If the list structure must be preserved, then we make copies of the *next* pointers and use the copies to compute the distances.

Types of Traversals:

There are a number of orders in which a processor can traverse, or visit tree nodes, exactly once, in a specific order, in a binary tree. Some of the most common ones are listed and described in the table below.

Traversal Type	Description	Real-World Use
Pre-order (depth-first)	Examine the root first, then the left child, then the right child	Creating (writing) a new tree
In-order (depth-first)	Examine the left child, then the root, then the right child	Searching a sorted binary tree
Post-order (depth-first)	Examine the left child, then the right child, then the root	Deleting an existing tree
Breadth-first	Examine each level of the binary tree before	Finding the shortest path between two

	moving to the next	nodes
--	--------------------	-------

Pre--Order Traversal:

A pre-order traversal of a binary tree is a recursive algorithm where the program first prints the label of the root of the tree. It then sends the same command to the left and right subtrees. An implementation of this algorithm in C is as follows:

```
void preordertraversal(struct node * root) {
    if(root != NULL) {
        printf("%d", root->label);
        preordertraversal(root->left);
        preordertraversal(root->right);
    }
    return;
}
```

Post--Order Traversal:

A post-order traversal of a binary tree is a recursive algorithm where the program first invokes itself on the left and right subtrees, traversing them completely, and then prints the label of the root. When the traversal of the left or right subtree reaches a node with no children, it prints the label of that node. An implementation of this algorithm in C is as follows:

```
void postordertraversal(struct node * root) {
    if(root != NULL) {
        postordertraversal(root->left);
        postordertraversal(root->right);
        printf("%d", root->label);
    }
    return;
}
```

In--Order Traversal:

An in-order traversal of a binary tree is a recursive algorithm where the program first invokes itself on its left subtree, then when that finishes prints the root, and finally invokes itself on its right subtree. An implementation of this algorithm in C is as follows:

```
void inordertraversal(struct node * root) {  
    if(root != NULL) {  
        inordertraversal(root->left);  
        printf("%d", root->label);  
        inordertraversal(root->right);  
    }  
    return;  
}
```

Breadth--First Traversal:

A breadth-first traversal works differently from a depth-first traversal. The program loops downward through each level of the tree, pushing all of the nodes at each row in the order in which they appear in the row. It does this by using two stacks: one to hold the nodes to be printed, and one to hold their children. An implementation of this algorithm in C is as follows:

```

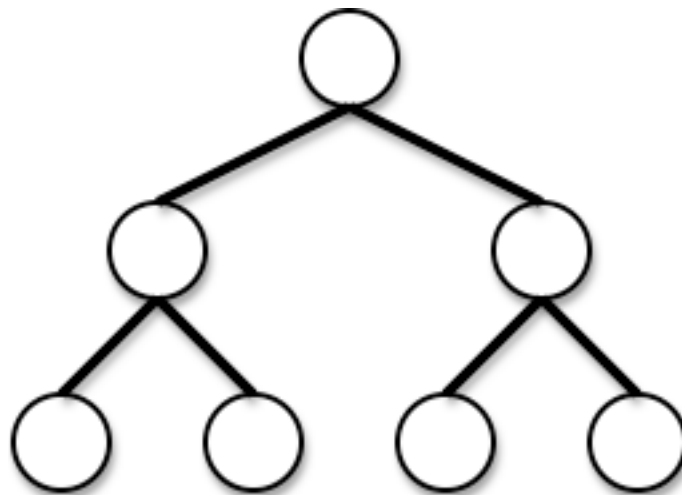
void breadthfirsttraversal(node * root) { stack
    activenodes;
    stack nextnodes;
    activenodes.push(root);
    while(activenodes.peek() != NULL) {
        while(activenodes.peek() != NULL) { node
            * current = activenodes.pop();
            printf("%d", current->label);
            nextnodes.push(current->left);
            nextnodes.push(current->right);
        }
        while(nextnodes.peek() != NULL) {
            activenodes.push(nextnodes.pop());
        }
    }
    return;
}

```

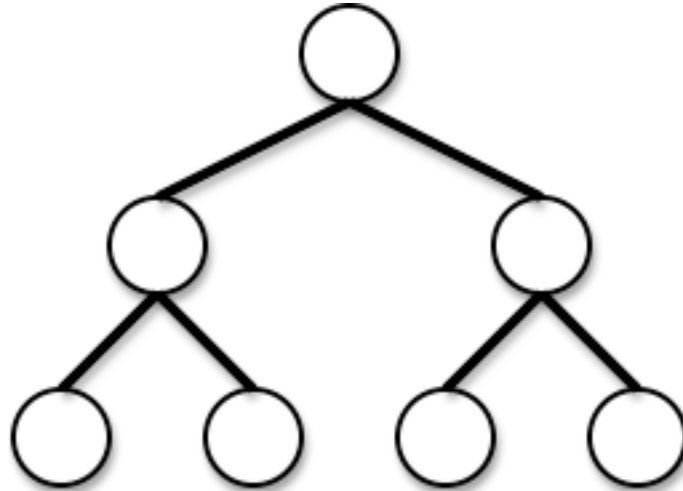
Exercise 1

Complete questions 1 through 4 below.

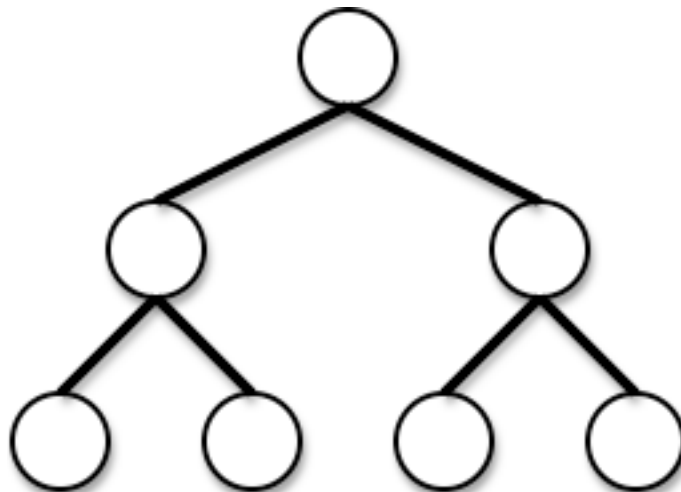
1. Label the nodes of the tree below 1 through 7 in the order they are processed by a pre--order traversal.



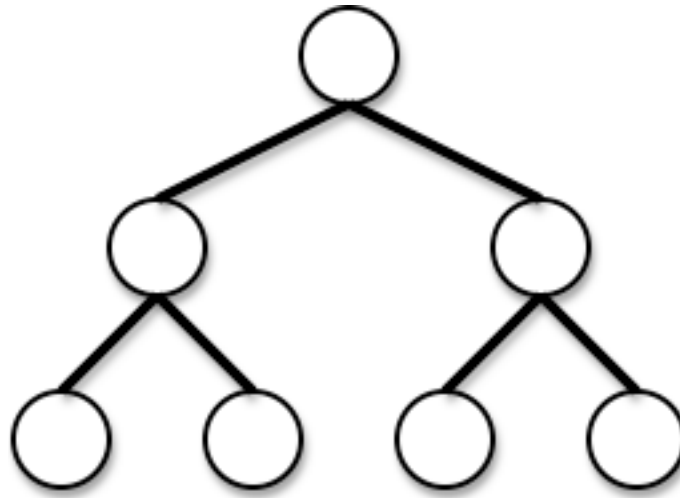
2. Label the nodes of the tree below 1 through 7 in the order they are processed by a post--order traversal.



3. Label the nodes of the tree below 1 through 7 in the order they are processed by an in--order traversal.



4. Label the nodes of the tree below 1 through 7 in the order they are processed by a breadth--first traversal.



Exercise 2

Follow the steps below on a computer, from the command line.

Example commands are shown below each instruction.

1. Change directories to the “code” directory (an attachment to this module).

```
cd BinaryTreeTraversal/code
```

2. Compile the code.

```
make traverse
```

3. Run the pre-order program and compare it to your result from Exercise 1 Question 1.

```
./traverse --order pre
```

4. Run the in-order program and compare it to your result from Exercise 1 Question 3.

```
./traverse --order in
```

5. Run the bread-first program and compare it to your result from Exercise 1 Question 4.

```
./traverse --order breadth
```

Parallelism and Scaling

The current setup works well enough on small amounts of data, but at some point data sets can grow sufficiently large that a single computer cannot hold all of the data at once, or the tree becomes so large that it takes an unreasonable amount of time to complete a traversal. To overcome these issues, parallel computing, or parallelism, can be employed. In parallel computing, a problem is divided up and each of multiple processors performs a part of the problem. The processors work at the same time, in parallel.

There are three primary reasons to parallelize a problem. The first is to achieve speedup, or to solve the problem in less time. As one adds more processors to solve the problem, one may find that the work is finished faster¹. This is known as strong scaling; the problem size stays constant, but the number of processors increases.

The second reason to parallelize is to solve a bigger problem. More processors may be able to solve a bigger problem in the same amount of time that fewer processors are able to solve a smaller problem. If the problem size increases as the number of processors increases, we call it weak scaling.

The third reason to parallelize is that it allows a problem that is too big to fit in the memory of one processor to be broken up such that it is able to fit in the memories of multiple processors.

In many ways, a tree is the perfect candidate for parallelism. In a tree, each node/subtree is independent. As a result, we can split up a large tree into 2, 4, 8, or more subtrees and hold subtree on each processor. Then, the only duplicated data that must be kept on all processors is the tiny tip of the tree that is the parent of all of the individual subtrees. Mathematically speaking, for a tree divided among n processors (where n is a power of two), the processors only need to hold $n - 1$ nodes in common – no matter how big the tree itself is. A picture of this is shown below in Figure 2. In this picture and throughout the module, a processor's rank is an identifier to keep it distinct from the other processors.

¹ An observation known as Amdahl's Law says that there is a theoretical limit to the amount of speedup that can be achieved from

strong scaling. At some point, the number of processors exceeds the amount of work available to do in parallel, and adding processors results in no additional speedup. Worse, the time spent communicating between processors overwhelms the total time spent solving the problem, and it actually becomes slower to solve a problem with more processors than with fewer.

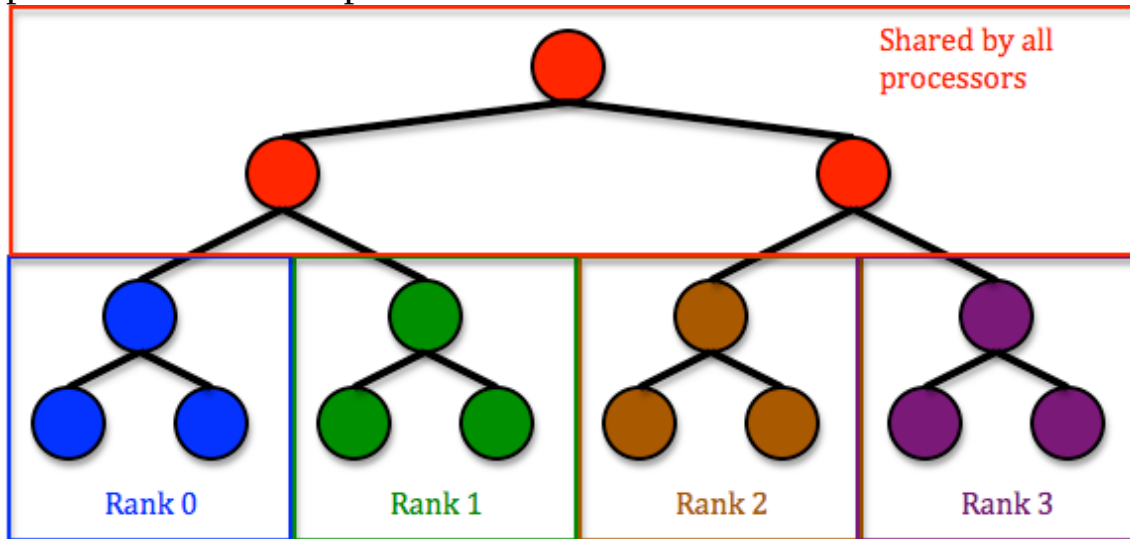


Figure 2: A tree of height 4 split among 4 processors

Parallel Traversals:

The fact that trees are comprised of independent sub-trees makes parallelizing them very easy. Properly done, the portion of these traversals that is parallelizable grows at 2^n for an n -generation tree, while the processors only need to synchronize once, at the end, so it approaches 100% for large trees (but keep in mind Amdahl's Law, footnote 1). The basic steps for parallelizing these traversals are as follows:

1. Perform the traversal on the parent part of the tree.
2. Whenever you get to a node that is only present on one processor, ask that processor to execute the appropriate C algorithm detailed above.
3. The processor will return its result that can be used exactly as if it was a serial processor.

A Breadth-First traversal is somewhat more complicated to

implement as a parallel system because at each level, it must access nodes from all of the parallel processors. Theoretically, a Breadth-First traversal can achieve the same 100% speedup of Pre-, In-, and Post-Order traversals. However, the amount of processor-processor data transmission adds in a greater potential for delays, thus slowing down the algorithm. Nevertheless, as the size of the tree increases the size of the generations grows at the rate of 2^n while the number of synchronizations grows at a rate of n for an n -generation tree, so the parallelizable portion of these traversals also approaches 100%. The basic steps for parallelizing this traversal are as follows:

1. Perform the traversal on the parent part of the tree.
2. Whenever you get to a node that is only present on one processor, ask that processor to execute the Breadth-First C algorithm detailed above, but wait after it finishes one generation.
3. Combine all the one-generation results from the different processors in the correct order.
4. Allow each processor to execute the next generation of the Breadth-First C algorithm detailed above, and then wait again.
5. Repeat Steps 3 and 4 until there are no nodes remaining.

PREFIX SUM

The technique of pointer jumping extends well beyond the application of list ranking. In the context of arithmetic circuits, a "prefix" computation can be used to perform binary addition quickly. We now investigate how pointer jumping can be used to perform prefix computations. Our EREW algorithm for the prefix problem runs in $O(\lg n)$ time on n -object lists.

A *prefix computation* is defined in terms of a binary, associative operator $\otimes$. The computation takes as input a sequence $\langle x_1, x_2, \dots, x_n \rangle$ and produces as output a sequence $\langle y_1, y_2, \dots, y_n \rangle$ such that $y_1 = x_1$ and

$$\begin{aligned} y_k &= y_{k-1} \otimes x_k \\ &= x_1 \otimes x_2 \otimes \dots \otimes x_k \end{aligned}$$

for $k = 2, 3, \dots, n$. In other words, each y_k is obtained by "multiplying" together the first k elements of the sequence of x_k --hence, the term

"prefix." (The definition in Chapter 29 indexes the sequences from 0, whereas this definition indexes from 1--an inessential difference.)

As an example of a prefix computation, suppose that every element of an n -object list contains the value 1, and let $\otimes$ be ordinary addition. Since the k th element of the list contains the value $x_k = 1$ for $k = 1, 2, \dots, n$, a prefix computation produces $y_k = k$, the index of the k th element. Thus, another way to perform list ranking is to reverse the list (which can be done in $O(1)$ time), perform this prefix computation, and subtract 1 from each value computed.

We now show how an EREW algorithm can compute parallel prefixes in $O(\lg n)$ time on n -object lists. For convenience, we define the notation

$$[i, j] = x_i \otimes x_{i+1} \otimes \dots \otimes x_j$$

for integers i and j in the range $1 \leq i \leq j \leq n$. Then, $[k, k] = x_k$ for

$$k = 1, 2, \dots, n, \text{ and}$$

$$[i, k] = [i, j] \otimes [j+1, k]$$

for $0 \leq i \leq j < k \leq n$. In terms of this notation, the goal of a prefix computation is to compute $y_k = [1, k]$ for $k = 1, 2, \dots, n$.

When we perform a prefix computation on a list, we wish the order of the input sequence $\langle x_1, x_2, \dots, x_n \rangle$ to be determined by how the objects are linked together in the list, and not by the index of the object in the array of memory that stores objects. (Exercise 30.1-2 asks for a prefix algorithm for arrays.) The following EREW algorithm starts with a value $x[i]$ in each object i in a list L . If object i is the k th object from the beginning of the list, then $x[i] = x_k$ is the k th element of the input sequence. Thus, the parallel prefix computation produces $y[i] = y_k = [1, k]$.

LIST-PREFIX(L)

```

1 for each processor  $i$ , in parallel
2   do  $y[i] \leftarrow x[i]$ 
3 while there exists an object  $i$  such that  $next[i] \neq \text{NIL}$ 
4   do for each processor  $i$ , in parallel
5     do if  $next[i] \neq \text{NIL}$ 
6       then  $y[next[i]] \leftarrow y[i] \otimes y[next[i]]$ 
7          $next[i] \leftarrow next[next[i]]$ 
```