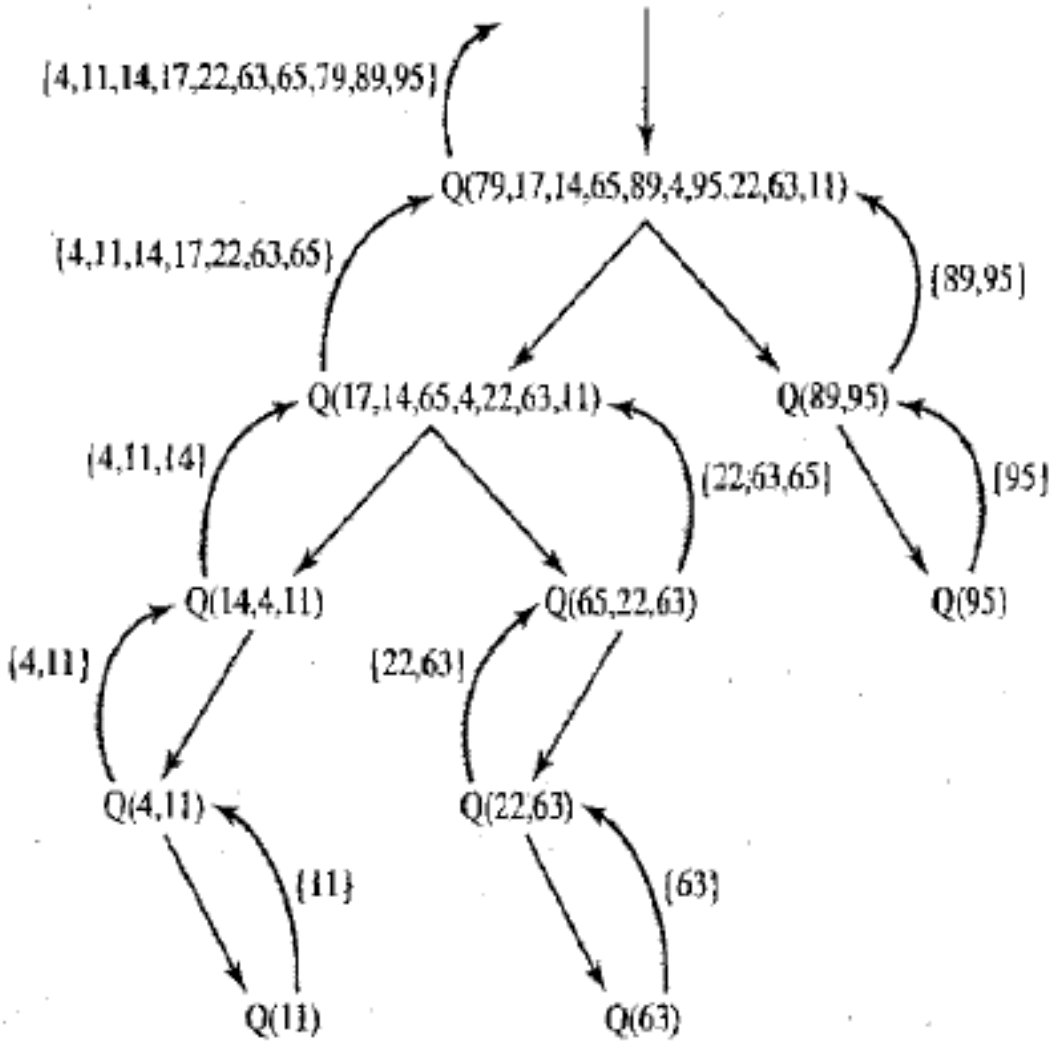
PARALLEL ALGORITHMS

Dr. Lavika Goel

Quicksort


{4,11,14,17,22,63,65,79,89,95}
Q(79,17,14,65,89,4,95,22,63,11)
{4,11,14,17,22,63,65}
{89,95}
Q(17,14,65,4,22,63,11)
Q(89,95)
{4,11,14}
{22,63,65}
{95}
Q(14,4,11)
Q(65,22,63)
Q(95)
{4,11}
{22,63}
Q(4,11)
Q(22,63)
{11}
{63}
Q(11)
Q(63)

Quicksort Pseudocode (Pivot as last element)

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

```
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes:  0   1   2   3   4   5   6

low = 0, high =  6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping
arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped
```

Quicksort

- Good starting point for a parallel sorting algorithm
- Fasting sorting algorithm based on comparison of keys
- It has natural concurrency as when quicksort calls itself, two calls may be independently executed
- As quicksort function calls itself twice, the number of leaves in the call graph is a power of 2
- The unsorted values are distributed among the memories of the processes
- We choose a pivot and broadcast it

Quicksort

- Each process divides its unsorted numbers into two lists, less than and greater than pivot
- Each process in the upper half of the process list sends its low list to a partner process in the lower half of the process list and receives a high list in return
- Now processes in the upper half of the process list have values greater than the pivot and the processes in the lower half of the process list have values less than or equal to the pivot

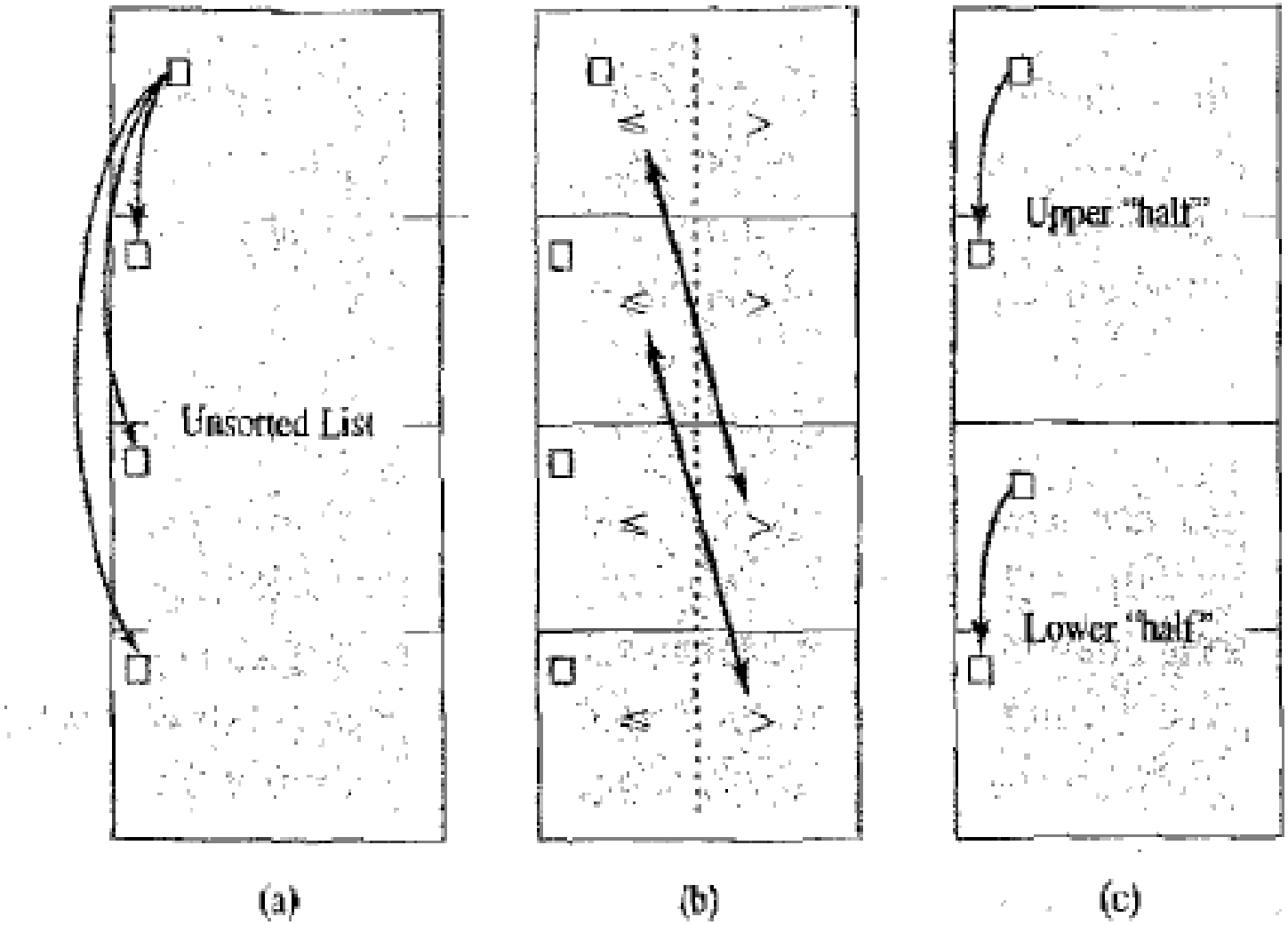
Quicksort

- At this point the processes divide themselves into two groups and the algorithm recurses
- In each process group the pivot value is broadcast
- Processes divide their lists and swap values with partner processes
- After log p recursions every process has an unsorted list of values completely disjoint from the values held by the other processes
- The largest value held by process ‘i’ is smaller than the smallest value held by the process ‘i+1’

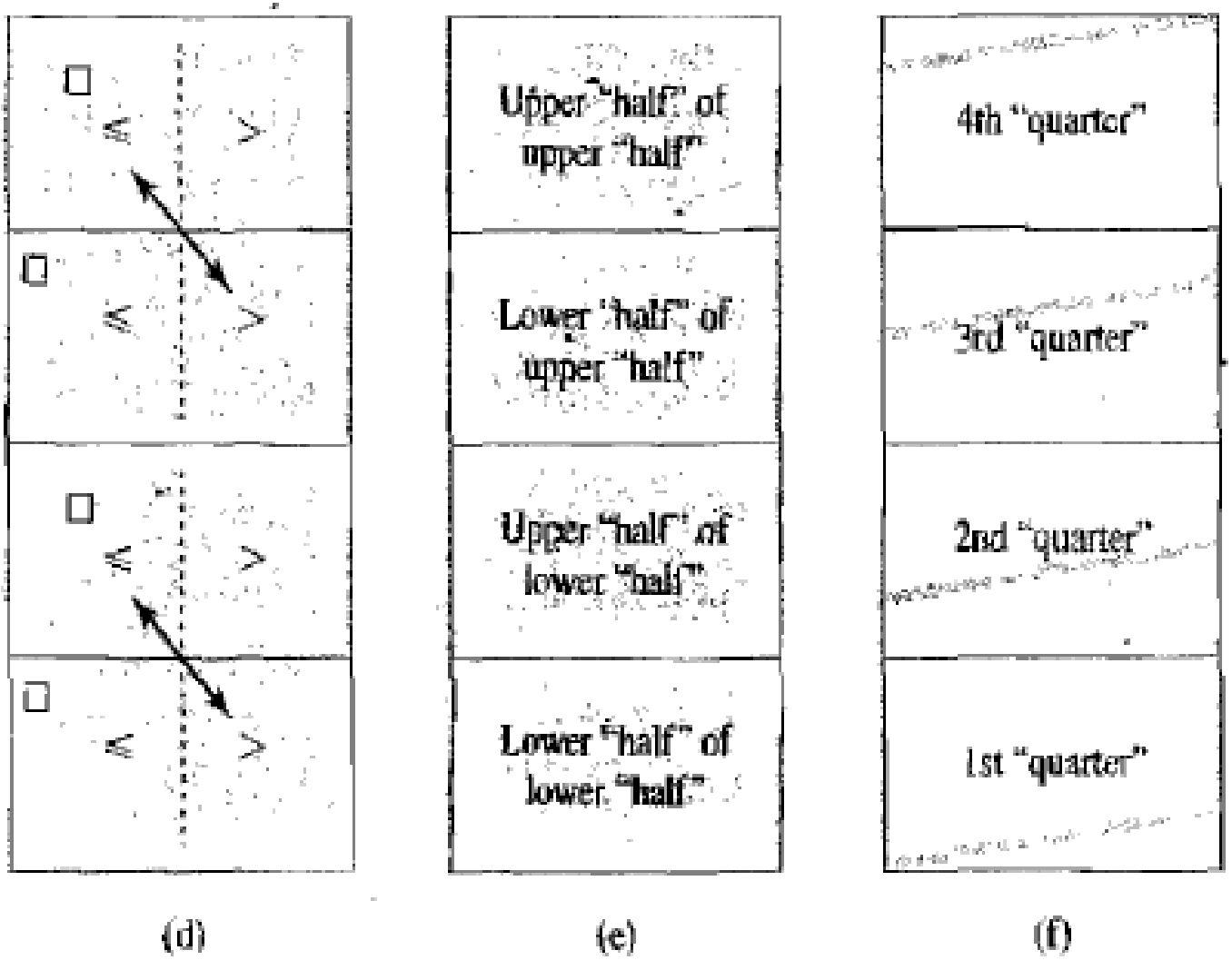
Quicksort

Figure 14.2 High-level view of a parallel quicksort algorithm. (a) Initially the unsorted values are distributed among the memories of all the processes. A single value is chosen as the pivot. The pivot is broadcast to the other processes. (b) Processes use the pivot to divide their numbers into those in the "lower half" and those in the "upper half." Each process in the upper half swaps values with a partner in the lower half. (c) The algorithm recurses. A single value from each "half" is chosen as the pivot for that "half" and broadcast to the other process responsible for that "half." (d) As in step (b), processes use the pivot to divide their numbers. Upper processes swap with lower processes, swapping smaller values for larger values. (e) At this point the largest value held by process i is less than the smallest value held by process $i + 1$. (f) Each process uses quicksort to sort the elements it controls. The list is now sorted.

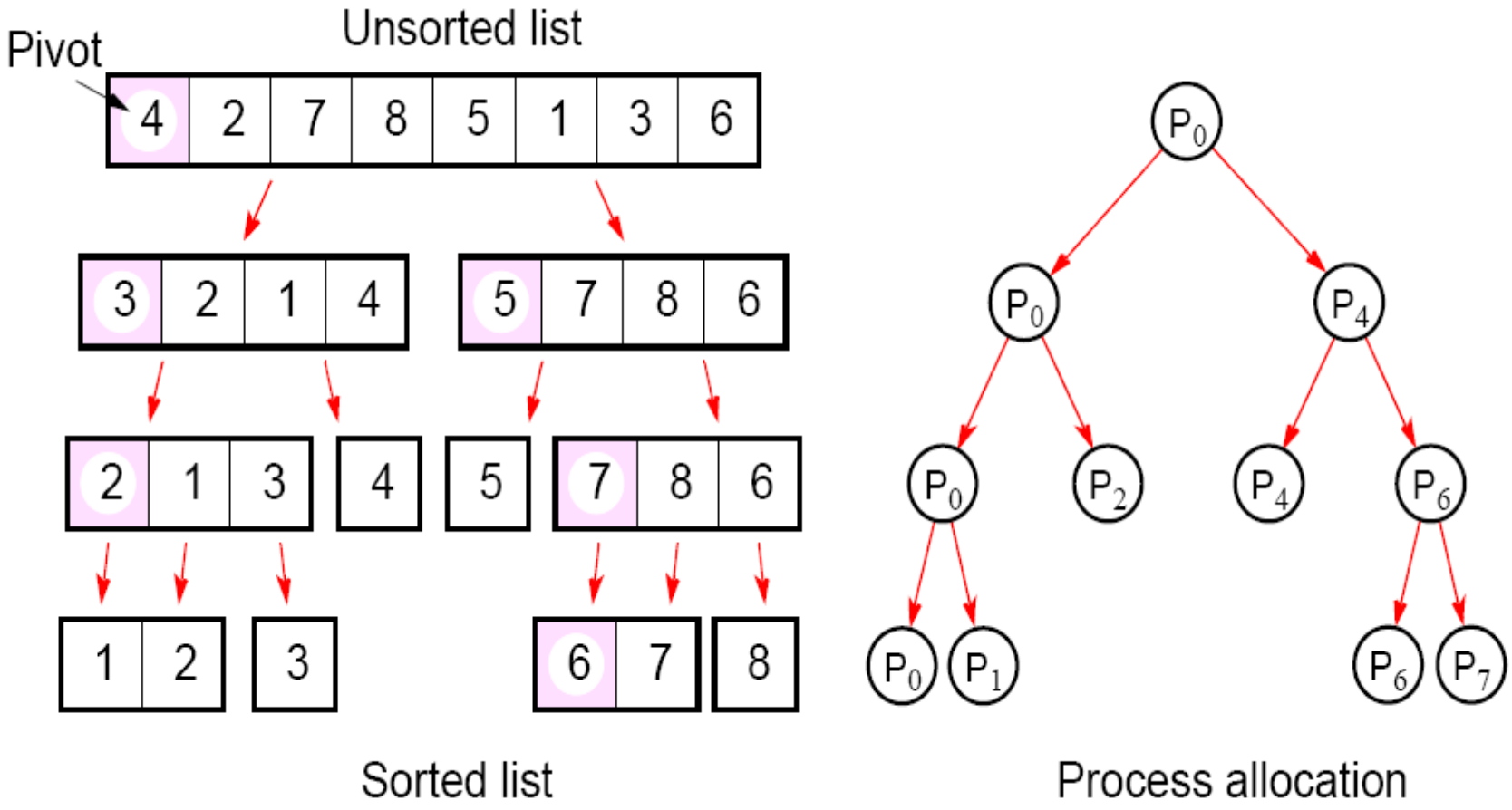
Quicksort



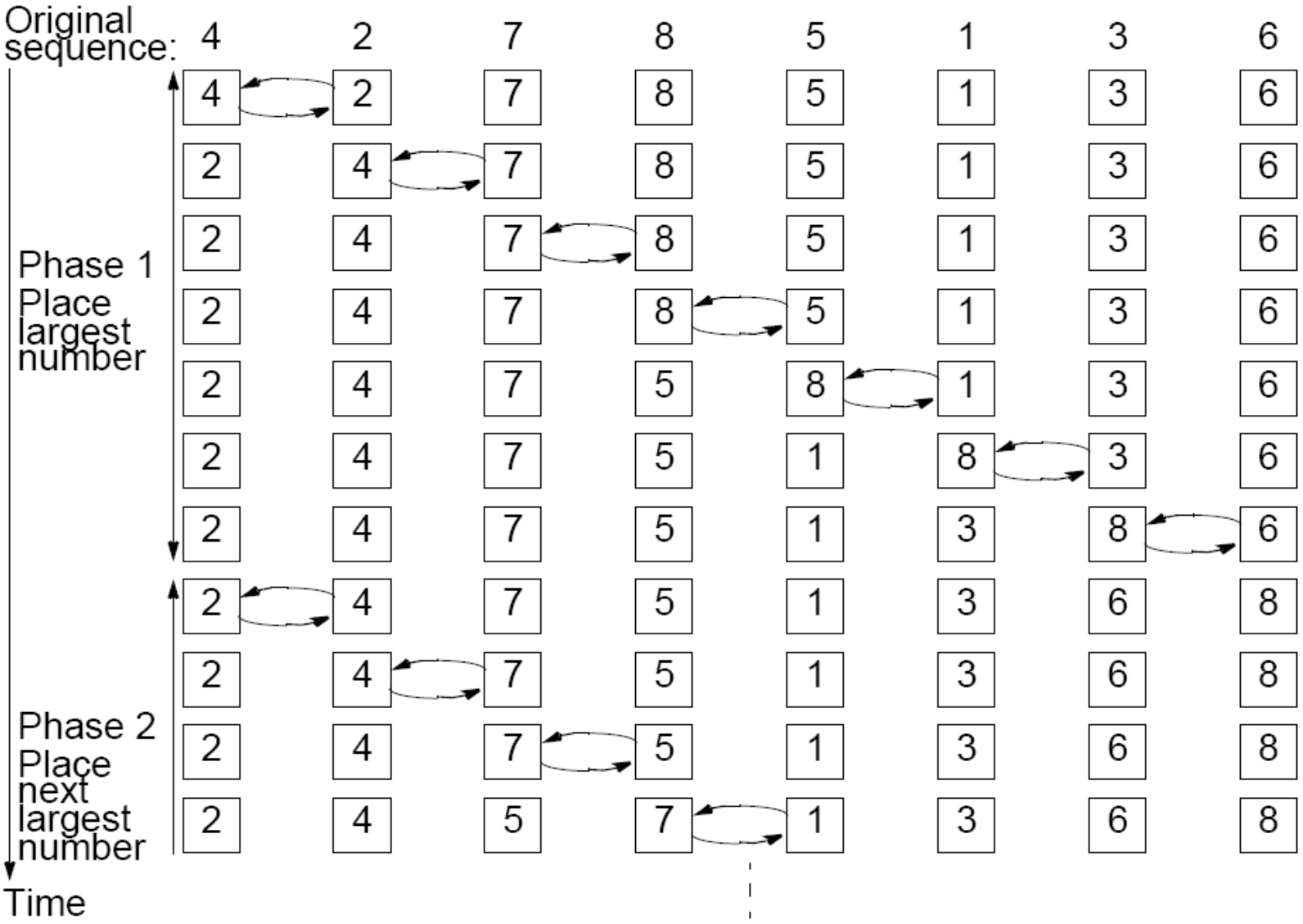
Quicksort


Upper "half" of upper "half"
Lower "half" of upper "half"
Upper "half" of lower "half"
Lower "half" of lower "half"
4th "quarter"
3rd "quarter"
2nd "quarter"
1st "quarter"
(d)
(e)
(f)

Parallelizing Quicksort

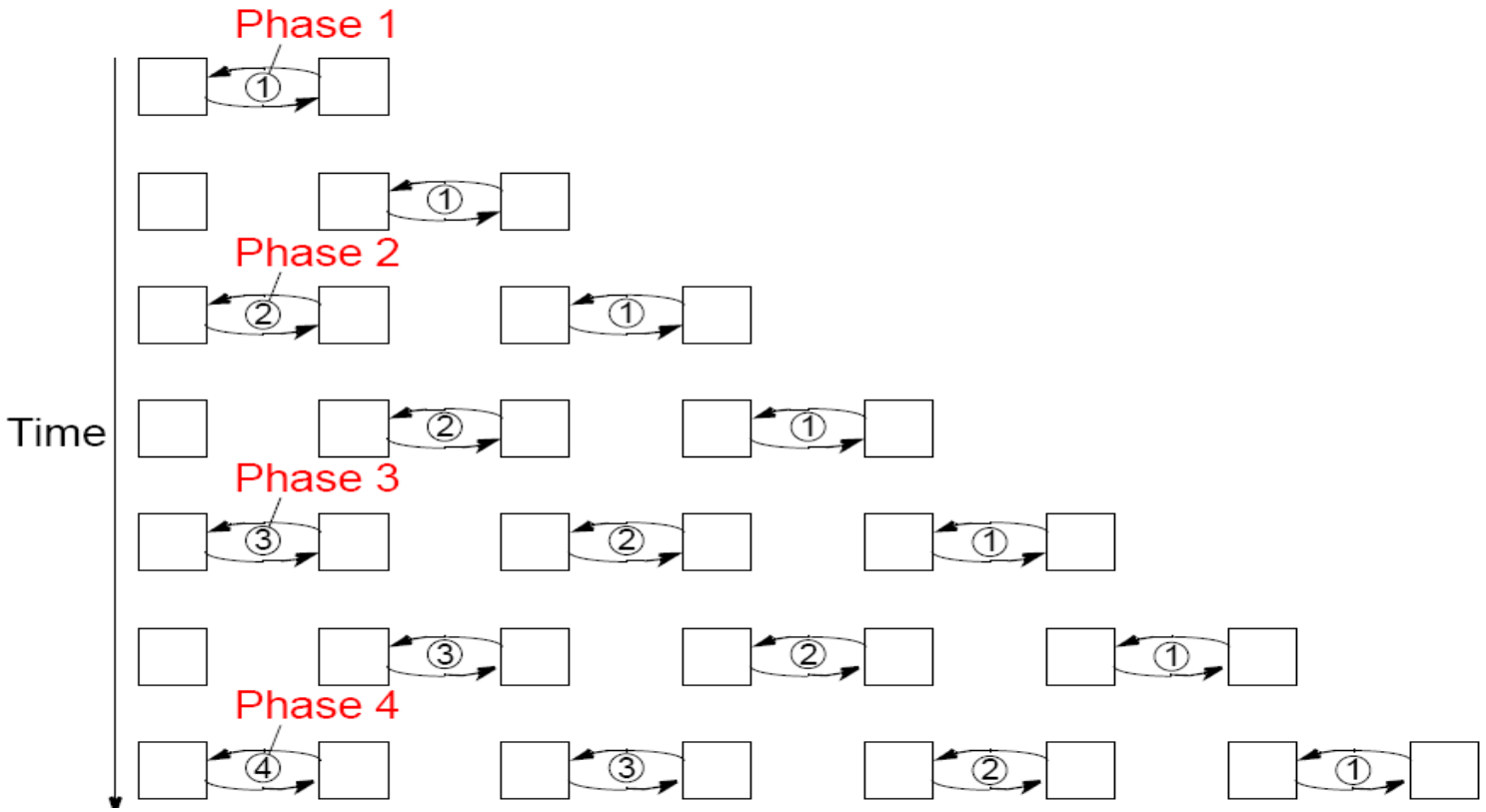


Bubble Sort


Original sequence:
4 2 7 8 5 1 3 6
Phase 1
Place largest number
Phase 2
Place next largest number
Time

Parallel Bubble Sort

Iteration could start before previous iteration finished if does not overtake previous bubbling action:



Odd-Even Transposition Sort

- Variation of bubble sort.
- Operates in two alternating phases, *even* phase and *odd* phase.
- **Even phase**

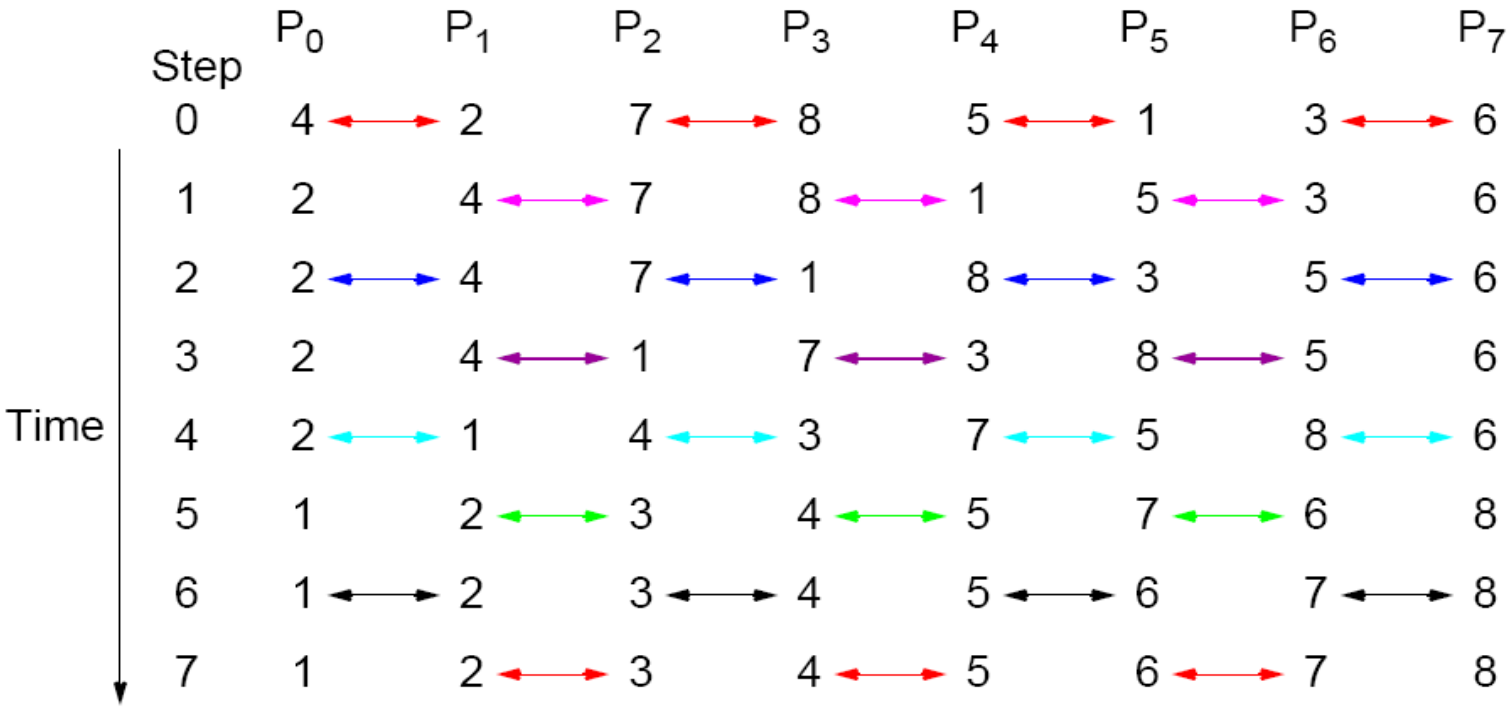
  Even-numbered processes exchange numbers with their right neighbor.

Odd phase

Odd-numbered processes exchange numbers with their right neighbor.

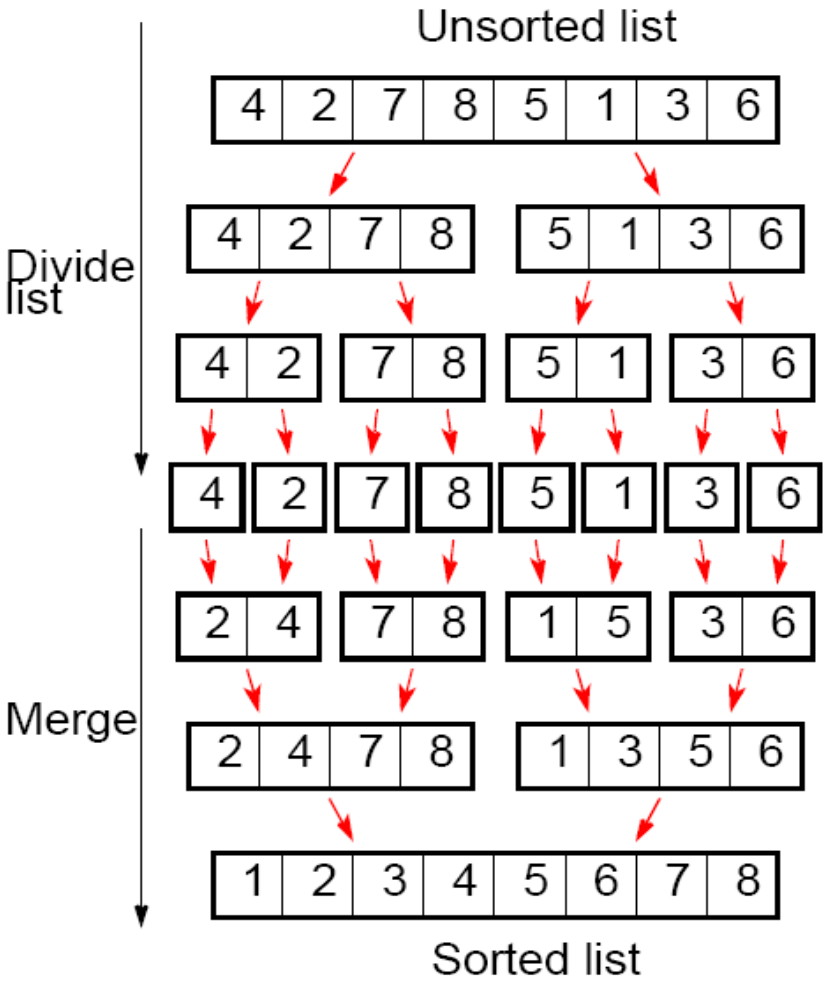
Odd-Even Transposition Sort

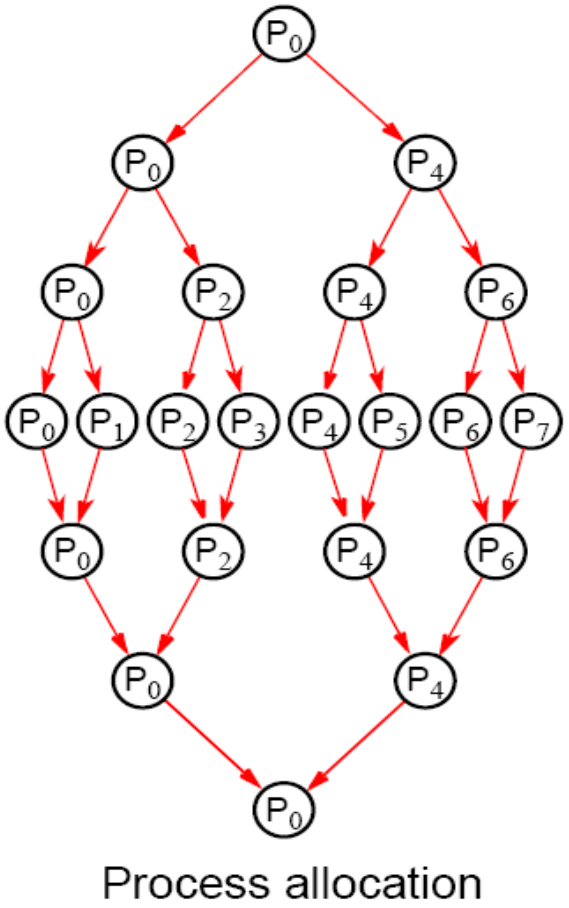
Sorting eight numbers



Parallelizing Mergesort

Using tree allocation of processes





Rank Sort

- The number of numbers that are smaller than each selected number is counted.
- This count provides the position of selected number in sorted list - its “rank.”
- First a[0] is read and compared with each of the other numbers, a[1] … a[n-1], recording the number of numbers less than a[0]
- Suppose this number is *x*. This is the index of the location in the final sorted list.

Rank Sort

- The number a[0] is copied into the final sorted list b[0] … b[n-1], at location b[x].
- Actions repeated with the other numbers.
- Overall sequential sorting time complexity of $O(n^2)$
- Not exactly a good sequential sorting algorithm!

Sequential Code

```
for (i   = 0; i < n;  i++)
{ /*  for each number */ x = 0;
   for (j = 0; j < n; j++)
            /*count number less than it */
      if(a[i] > a[j])
             x++;
      b[x] = a[i];
         /* copy number into correct place */
}
```

This code will fail if duplicates exist in the sequence of numbers.

Parallel Code

Using *n* Processors

One processor allocated to each number.

Finds final index in O(*n*) steps.

With all processors operating in parallel, parallel time complexity O(*n*).

Parallel time complexity, O(*n*), better than any sequential sorting algorithm.

Can do even better if we have more processors.

Parallel Code

Using *n* Processors

In **forall** notation, the code would look like

```
forall (i = 0;   i < n;   i++)
{
        x = 0;
      for (j = 0; j < n;  j++)
         (a[i] > a[j])
            x++;
      b[x] = a[i];
}
```

Sorting Conclusions

Computational time complexity using n processors

- Ranksort O(n)
- Odd-even transposition sort- O(n)
- Parallel mergesort - O(n) but unbalanced processor load and communication
- Parallel quicksort - O(n) but unbalanced processor load, and communication can generate to $O(n^2)$

Prefix Sums

Given a set of n values a1, a2,…, an and an associative operation @, the Prefix Sums problem is to compute the n quantities a1, a1@a2, a1@a2@a3, ... , a1@...@an

Example:

{2, 7, 9, 4} → {2, 9, 18, 22}

Doubling

- A processing technique in which accesses or actions are governed by increasing powers of 2
- That is, processing proceeds by 1, 2, 4, 8, 16, etc., doubling on each iteration

Prefix Sums by Doubling Example

4	9	5	2	10	6	12	8
4	13	14	7	12	16	18	20
4	13	18	20	26	23	30	36
4	13	18	20	30	36	48	56

T1 = O(n) Tp = O(log n)

Prefix Sums by Doubling Example

0	1	2	3	4	5	6	7
0	0,1	1,2	2,3	3,4	4,5	5,6	6,7
0	0,1	0,1,2	0,1,2,3	1,2,3,4	2,3,4,5	3,4,5,6	4,5,6,7
0	0,1	0,1,2	0,1,2,3	0,1,2,3,4	0,1,2,3,4,5	0,1,2 3,4,5,6	0,1,2,3,4,5,6,7

T1 = O(n) Tp = O(log n)

An example of prefix computation

- For example, if $\oplus$ is + and the input is the ordered set

  {5, 3, -6, 2, 7, 10, -2, 8}

  then the output is

  {5, 8, 2, 4, 11, 21, 19, 27}

- Prefix sum can be computed in $O(n)$ time sequentially.

Serial Algo

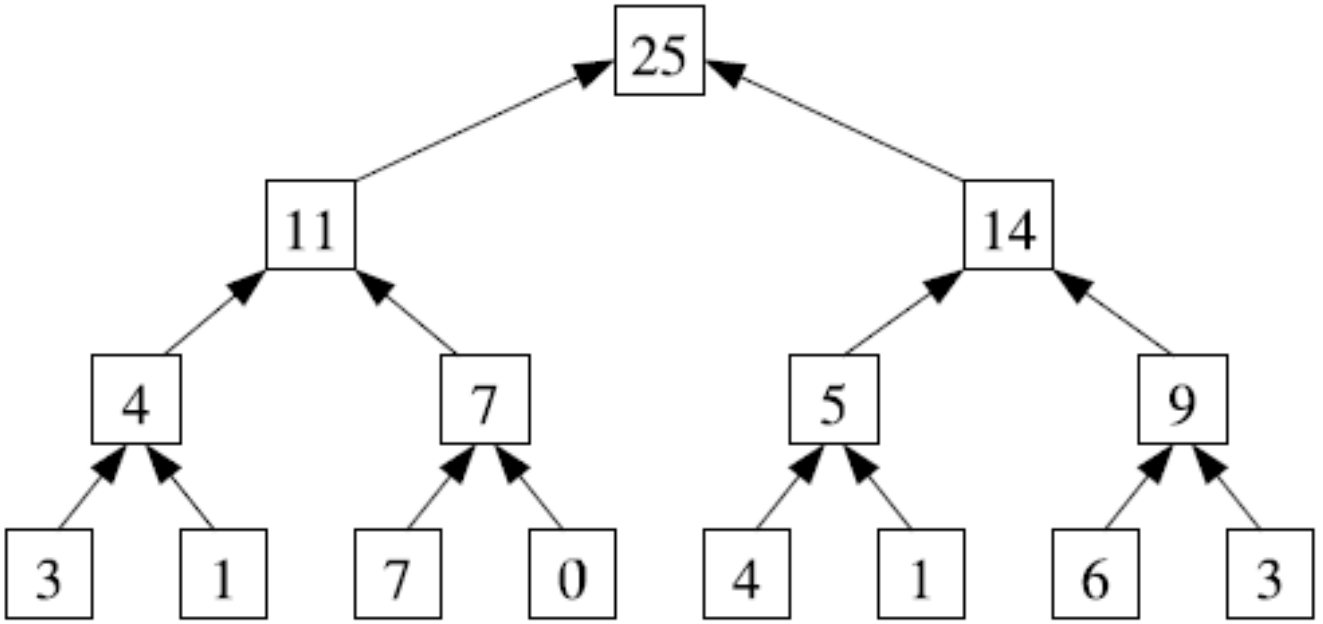
```
proc all-prefix-sums(Out, In)
    i ← 0
    sum ← In[0]
    Out[0] ← sum
    while (i < length)
        i ← i + 1
        sum ← sum + In[i]
        Out[i] ← sum
```

Example

$$[3 \quad 1 \quad 7 \quad 0 \quad 4 \quad 1 \quad 6 \quad 3],$$

$$[3 \quad 4 \quad 11 \quad 11 \quad 14 \quad 16 \quad 22 \quad 25].$$

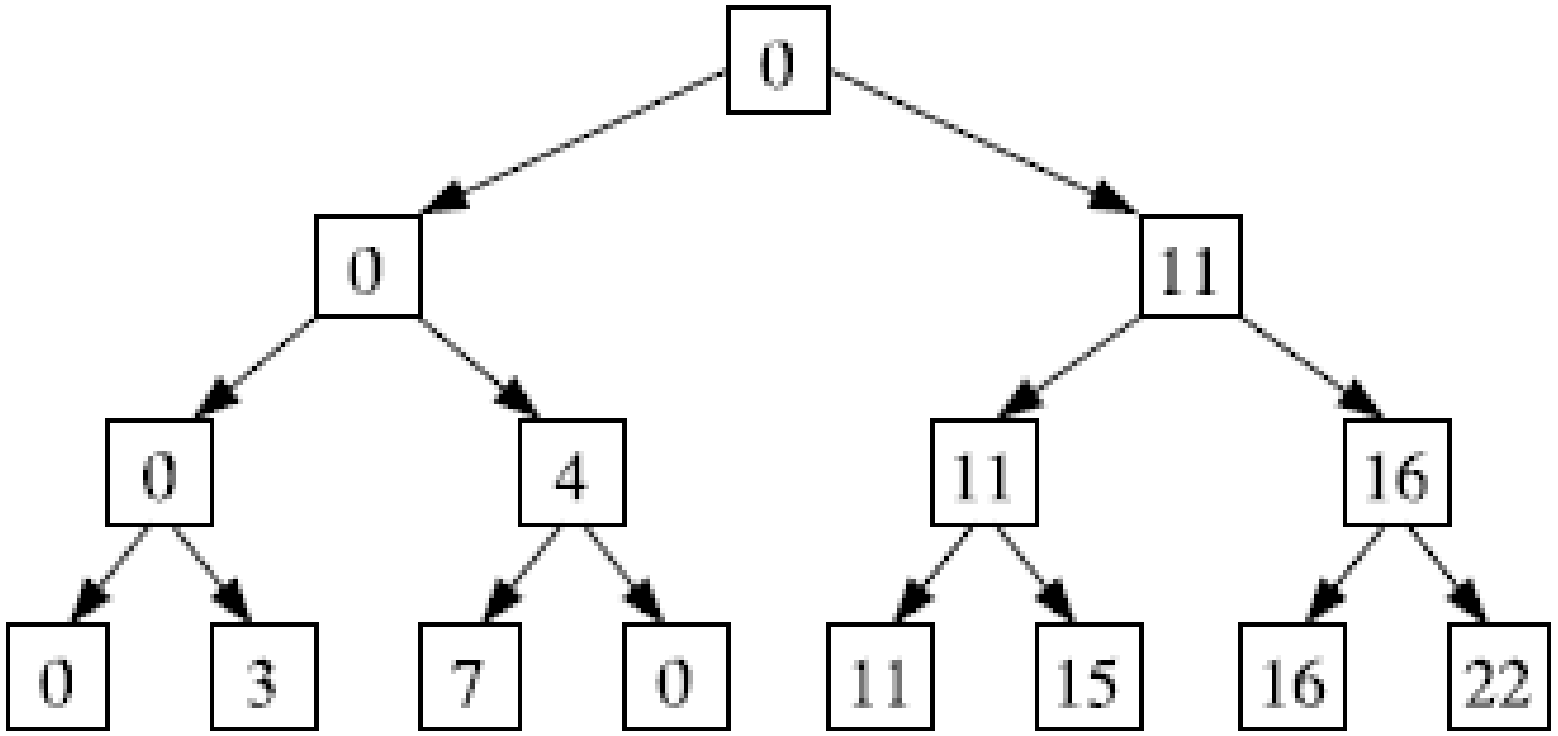
Up Tree



Pass 1

Step	Vector in Memory							
0	[**3**	**1**	**7**	**0**	**4**	**1**	**6**	**3**]
1	[3	**4**	7	**7**	4	**5**	6	**9**]
2	[3	4	7	**11**	4	5	6	**14**]
3	[3	4	7	11	4	5	6	**25**]

Down Tree



Pass 2

	Step	Vector in Memory							
	0	[**3**	**1**	**7**	**0**	**4**	**1**	**6**	**3**]
up	1	[3	**4**	7	**7**	4	**5**	6	**9**]
	2	[3	4	7	**11**	4	5	6	**14**]
	3	[3	4	7	11	4	5	6	**25**]
clear	4	[3	4	7	11	4	5	6	**0**]
down	5	[3	4	7	**0**	4	5	6	**11**]
	6	[3	**0**	7	**4**	4	**11**	6	**16**]
	7	[**0**	**3**	**4**	**11**	**11**	**15**	**16**	**22**]

First Attempt

- Steps 2 and 3 of the algorithm are shared between threads
- Each thread working on a different subset of the array
- To prevent race conditions,
  - we use an auxiliary array which stores writes from a thread
  - while other threads are reading from the other array.

Second Attempt

- Divide the original array into sub-arrays
- Number of which will be equal to the number of threads
- Each thread will, linearly, calculate the prefix-sum for its assigned sub-array.
- These prefix-sums will be less than the actual sums, since elements before the start of a particular sub-array are ignored.

Second Attempt

- The last elements of each of these sub-arrays is stored in another array.
- For this array, the prefix-sum array is calculated, and the corresponding values in original array are updated.
- This is done by adding to each element the requisite amount that was missing earlier.