COMP 322: Fundamentals of Parallel Programming

Lecture 11: Parallel Prefix Sum (contd), Parallel Quicksort

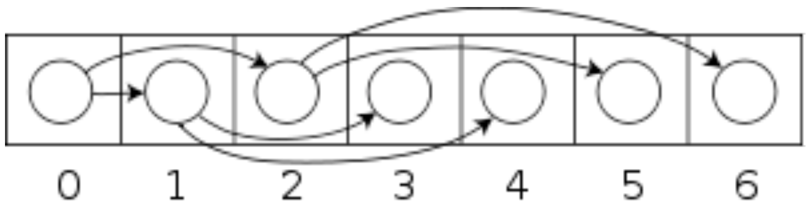
Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Goals for Today's Lecture

- <u>Parallel prefix sum algorithm (contd)</u>
- Parallel quicksort algorithm

Representing a complete binary tree as an array



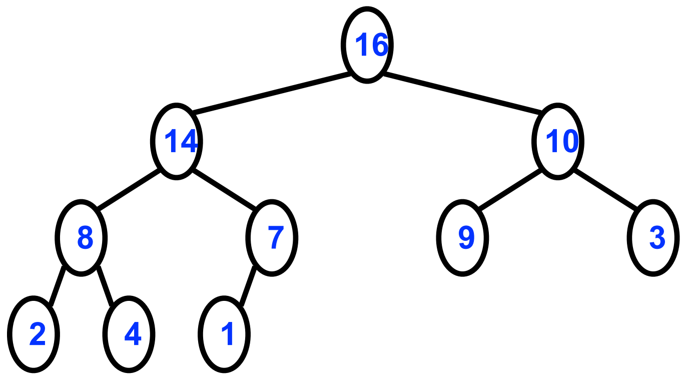
A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible

- To represent a complete binary tree as an array, arrange the nodes in breadth-first order:
  - The root node is A[0]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[(i-1)/2] (note: integer divide)
  - The left child of node *i* is A[2i+1]
  - The right child of node i is A[2i+2]

A =

16	14	10	8	7	9	3	2	4	1

=



Prefix Sum (Scan) Problem Statement

Given input array A, compute output array X as follows

$$X[i] = \sum_{0 \leq j \leq i} A[j]$$

- The above is an <u>inclusive</u> prefix sum since X[i] includes A[i]
- For an <u>exclusive</u> prefix sum, perform the summation for 0 <=j <i
- It is easy to see that prefix sums can be computed sequentially in O(n) time

```
// Copy input array A into output array X
X = new int[A.length]; System.arraycopy(A,0,X,0,A.length);
// Update array X with prefix sums
for (int i=1 ; i < X.length ; i++ ) X[i] += X[i-1];
```

Parallel Prefix Sum: General Idea

Observation: each prefix sum can be decomposed into reusable terms of power-of-2-size e.g.

$$\begin{aligned} X[6] &= A[0]+A[1]+A[2]+A[3]+A[4]+A[5]+A[6] \\ &= (A[0]+A[1]+A[2]+A[3])+(A[4]+A[5])+A[6] \end{aligned}$$

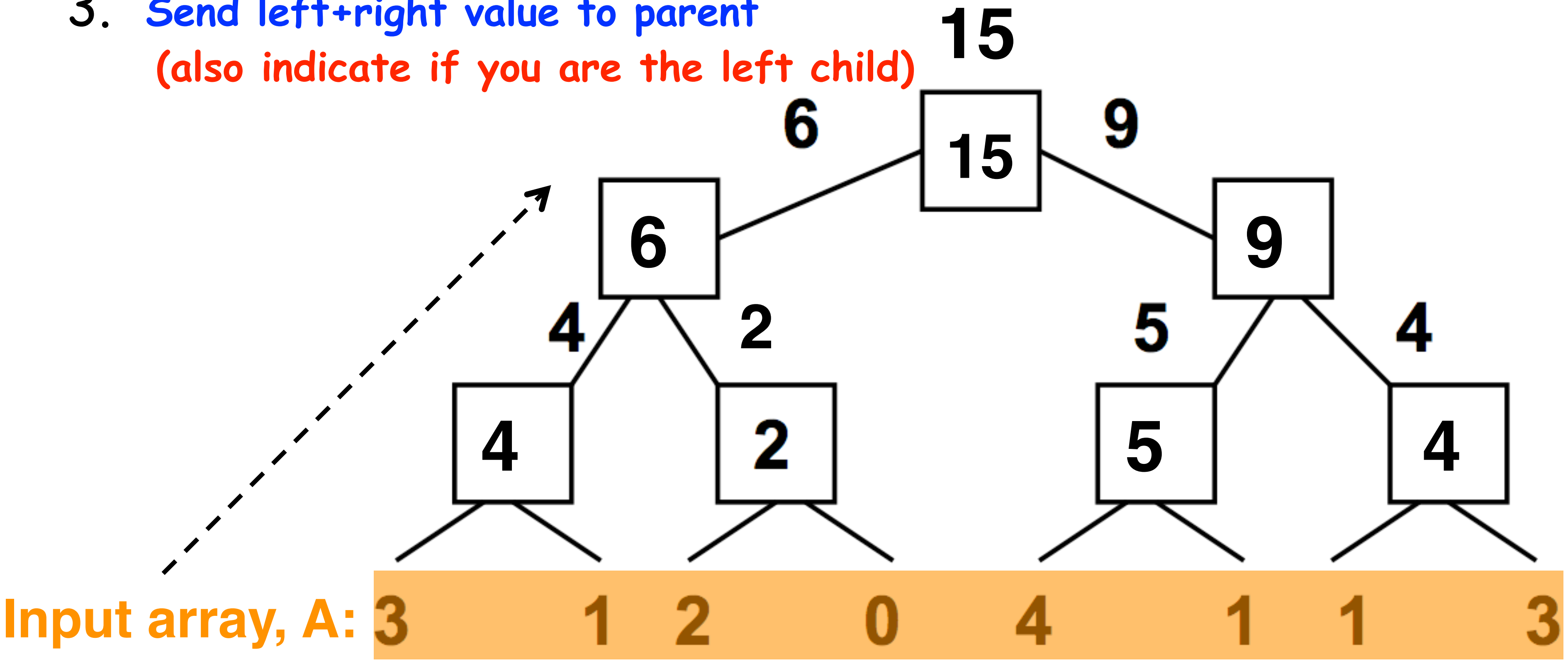
Approach:

- Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential Prefix Sum
- Use an “upward sweep” to perform parallel reduction, while storing partial sum terms in tree nodes
- Use a “downward sweep” to compute prefix sums while reusing partial sum terms stored in upward sweep

Parallel Prefix Sum: Upward Sweep (Alternate formulation to Lecture 10)

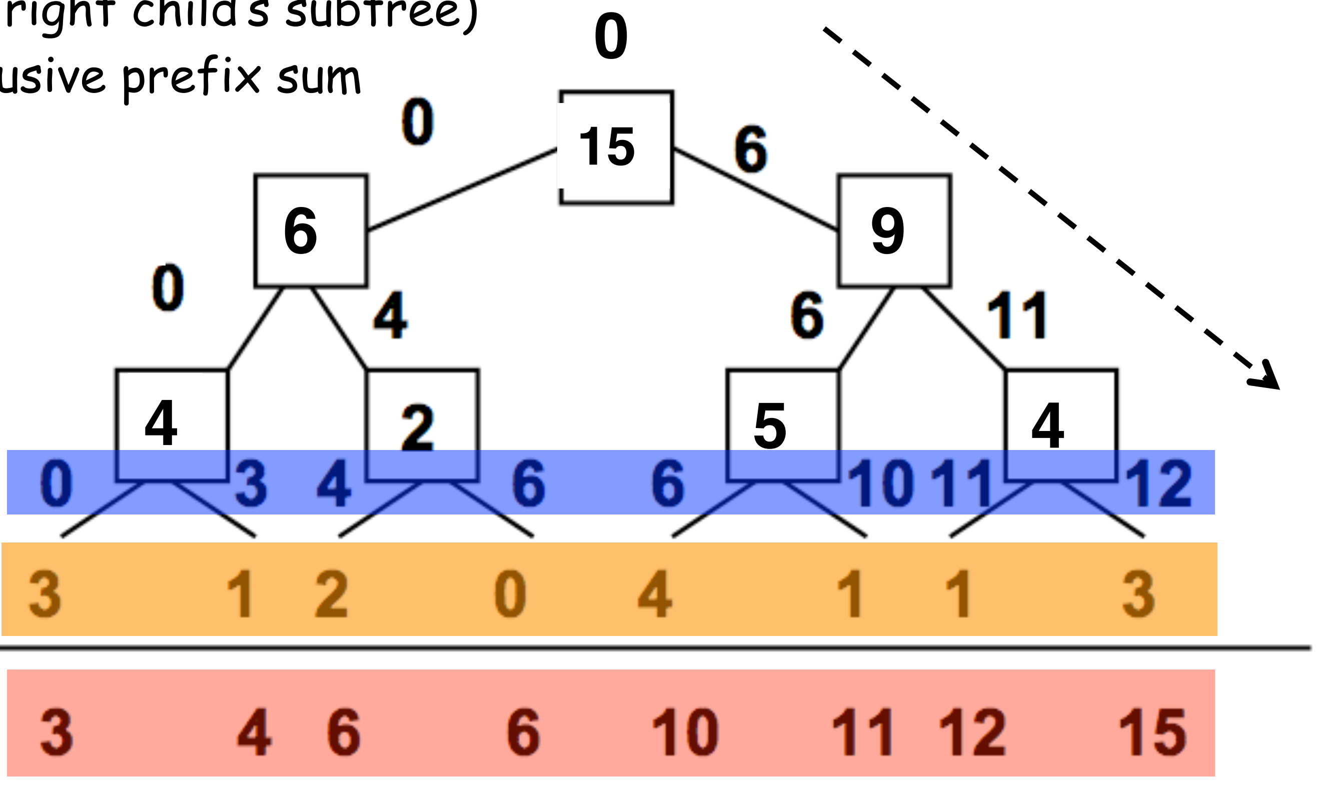
Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way

1. Receive values from left and right children
2. Compute left+right and store in box
3. Send left+right value to parent
   (also indicate if you are the left child)



Parallel Prefix Sum: Downward Sweep (Alternate formulation to Lecture 10)

1. Receive value from parent (root receives 0)
2. Send parent's value to left child (prefix sum for elements to left of left child's subtree)
3. Send parent's value+ left child's box value to right child (prefix sum for elements to left of right child's subtree)
4. Add A[i] to get inclusive prefix sum



Exclusive prefix sums	0	3	4	6	6	10	11	12
+ A[i]	3	1	2	0	4	1	1	3
Inclusive prefix sums	3	4	6	6	10	11	12	15

Summary of Parallel Prefix Sum Algorithm

- Critical path length, CPL = O(log n)
- Total number of add operations, WORK = O(n)
- Optimal algorithm for P = O(n/log n) processors
  - Adding more processors does not help
- Like Array Sum Reduction, Parallel Prefix Sum has several applications that go beyond computing the sum of array elements

Example Applications of Parallel Prefix Algorithm

- Prefix Max with Index of First Occurrence: given an input array A, output an array X of objects such that X[i].max is the maximum of elements A[0…i] and X[i].index contains the index of the first occurrence of X[i].max in A[0…i]
- Filter and Packing of Strings: given an input array A identify elements that satisfy some desired property (e.g., uppercase), and pack them in a new output array. (First create a 0/1 array for elements that satisfy the property, and then compute prefix sums to identify locations of elements to be packed.)
  - Also useful for partitioning step in Parallel Quicksort algorithm

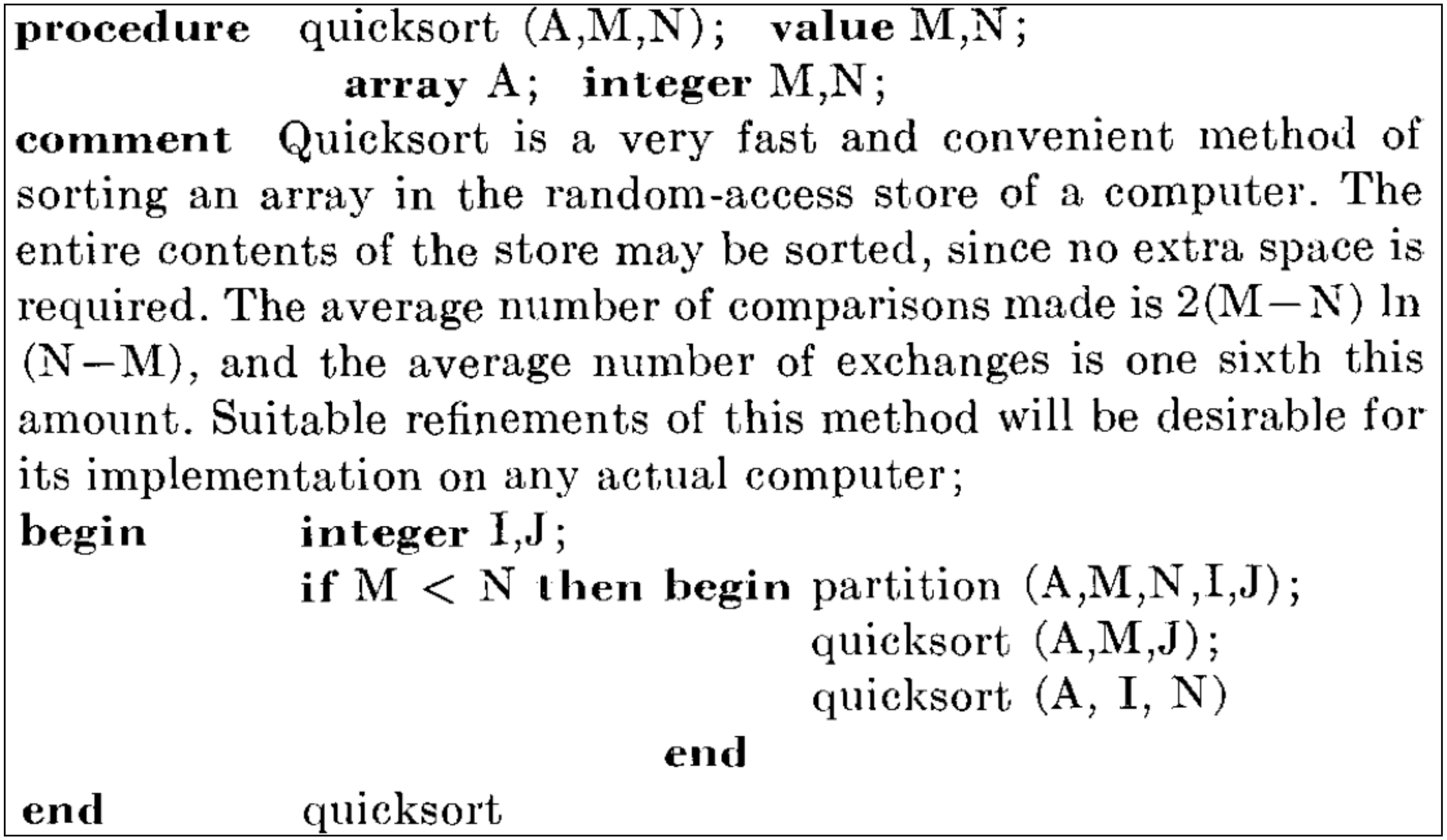
Goals for Today's Lecture

- Parallel prefix sum algorithm (contd)
- <u>Parallel quicksort algorithm</u>

Quicksort

- Classical sequential sorting algorithm introduced by C.A.R. Hoare in 1961 [3]
- Some reasons why Quicksort is still in use today:
  - Simple to implement
  - Worst case $O(n^2)$ execution time, but executes in O(n log n) time in practice (with high probability)
  - "In place'' sorting algorithm -- does not need allocation of a second copy of the array.
  - Exemplar of divide-and-conquer paradigm

Original description of Quicksort algorithm (CACM 1961)

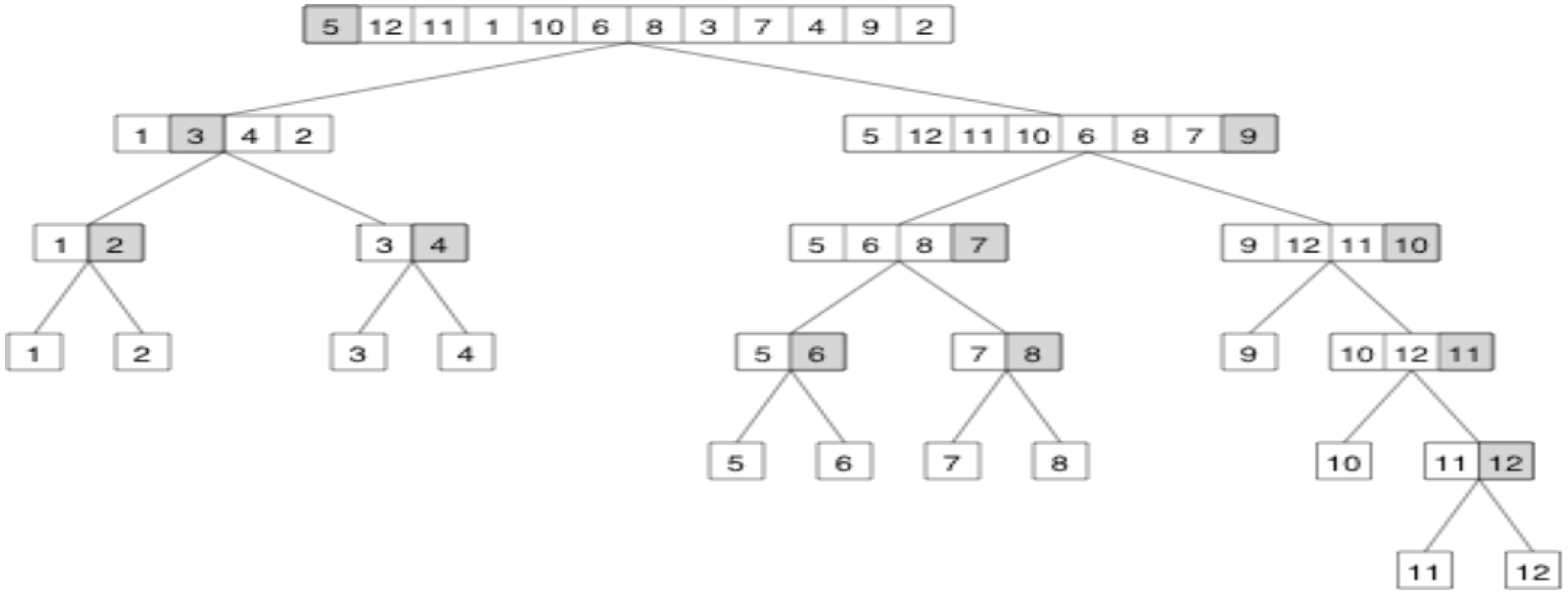


```
procedure   quicksort (A,M,N);   value M,N;
                array A;   integer M,N;
comment  Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin        integer I,J;
             if M < N then begin partition (A,M,N,I,J);
                                 quicksort (A,M,J);
                                 quicksort (A, I, N)
                           end
end          quicksort
```

Sequential HJ implementation of Quicksort

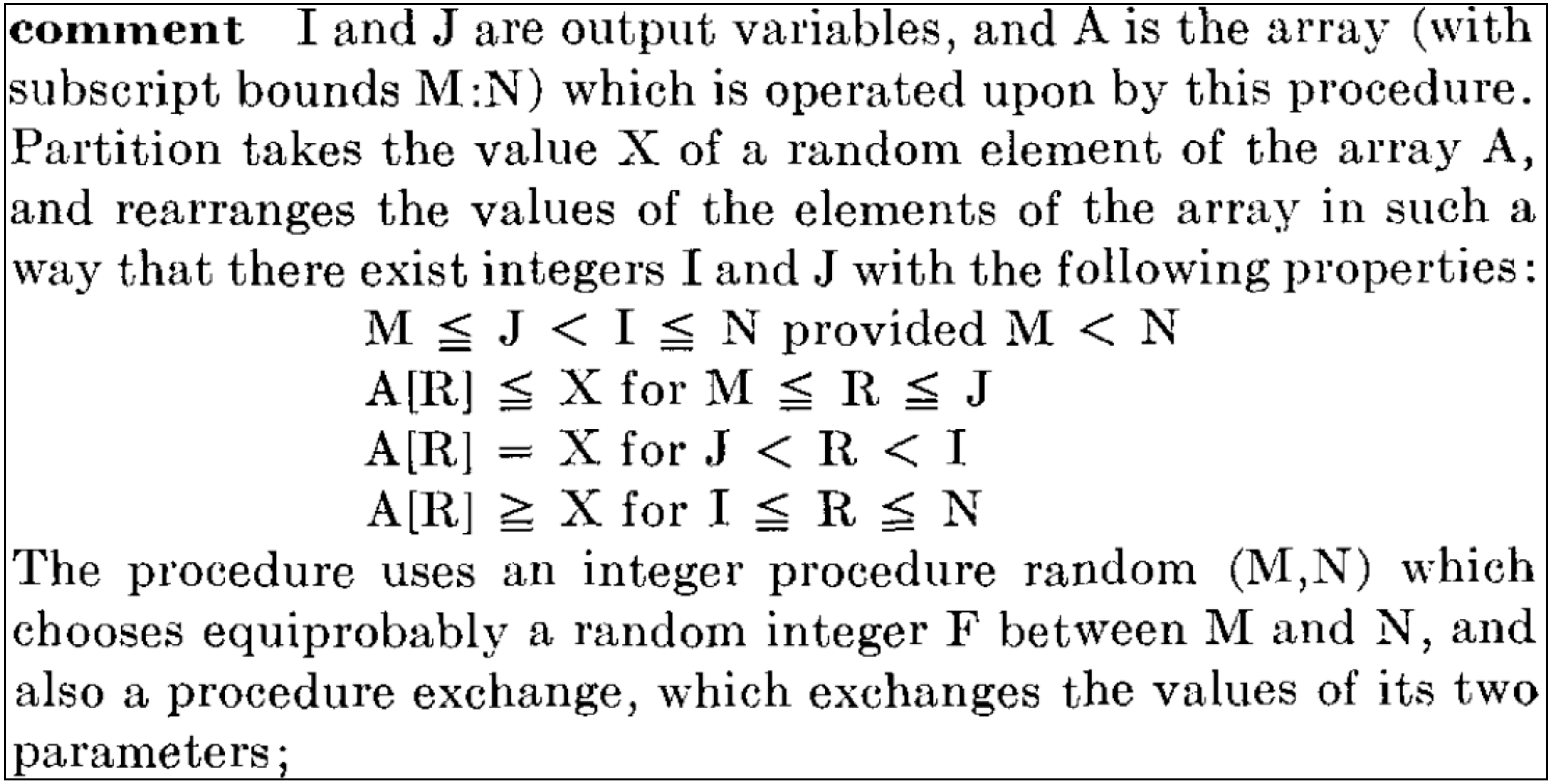
```
static void quicksort(int[] A, int M, int N) {
  if (M < N) {
    // partition() selects a pivot element in A[M…N]
    // to partition A[M…N] into A[M…J] and A[I…N]
    point p = partition(A, M, N); // p = [I,J]
    int I=p.get(0); int J=p.get(1);
    quicksort(A, M, J);
    quicksort(A, I, N);
  }
} //quicksort
. . .
quicksort(A, 0, A.length-1);
```

Example Execution of Quicksort algorithm



▨ Pivot element (can be selected randomly, or as median of three fixed elements, or by any other approach)

Original description of partition()

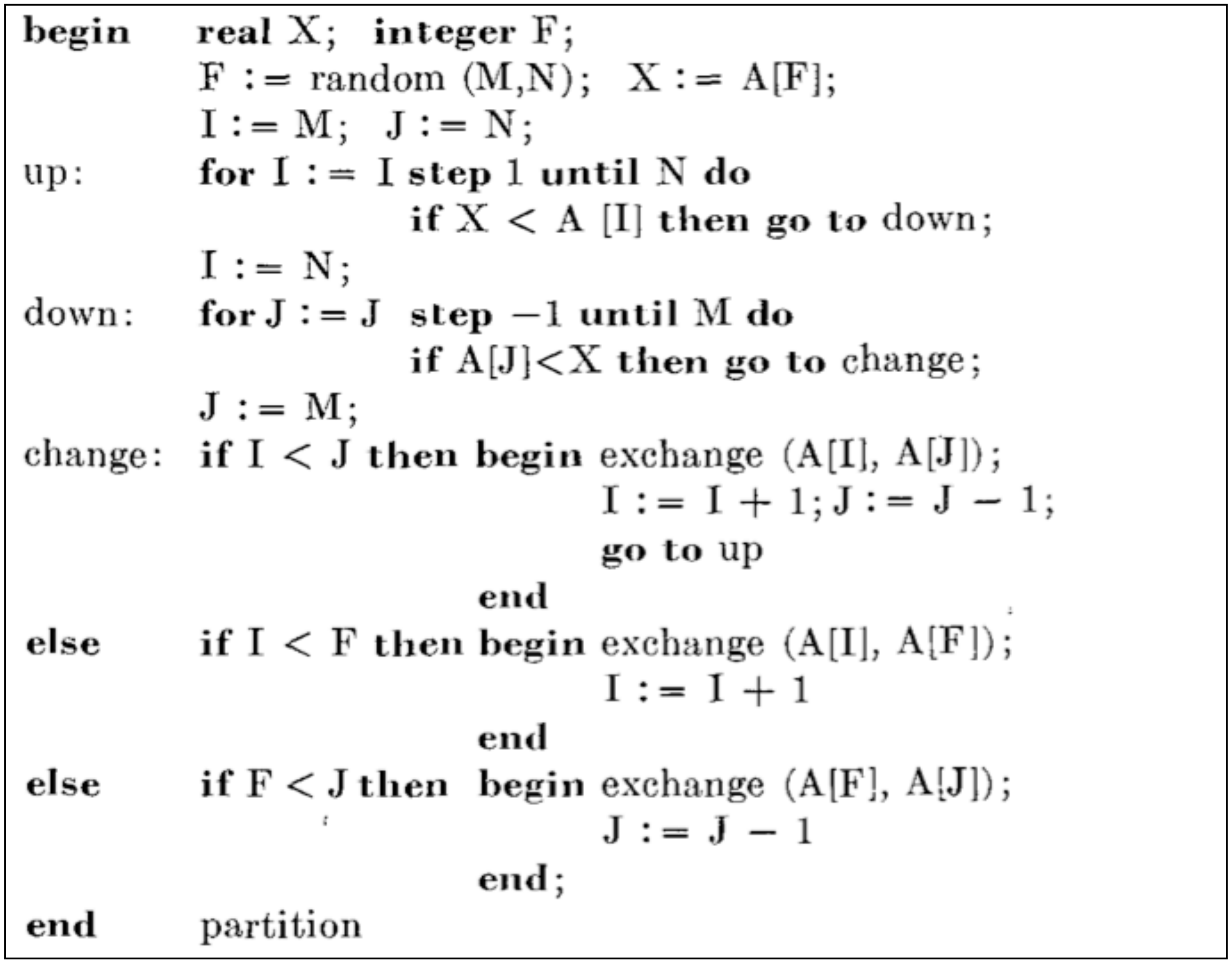


comment I and J are output variables, and A is the array (with subscript bounds M:N) which is operated upon by this procedure. Partition takes the value X of a random element of the array A, and rearranges the values of the elements of the array in such a way that there exist integers I and J with the following properties:

$$M \leqq J < I \leqq N \text{ provided } M < N$$
$$A[R] \leqq X \text{ for } M \leqq R \leqq J$$
$$A[R] = X \text{ for } J < R < I$$
$$A[R] \geqq X \text{ for } I \leqq R \leqq N$$

The procedure uses an integer procedure random (M,N) which chooses equiprobably a random integer F between M and N, and also a procedure exchange, which exchanges the values of its two parameters;

Original code for partition()



```
begin    real X;  integer F;
         F := random (M,N);  X := A[F];
         I := M;  J := N;
up:      for I := I step 1 until N do
                  if X < A [I] then go to down;
         I := N;
down:    for J := J  step −1 until M do
                  if A[J]<X then go to change;
         J := M;
change:  if I < J then begin exchange (A[I], A[J]);
                             I := I + 1; J := J − 1;
                             go to up
                       end
else     if I < F then begin exchange (A[I], A[F]);
                             I := I + 1
                       end
else     if F < J then begin exchange (A[F], A[J]);
                             J := J − 1
                       end;
end      partition
```

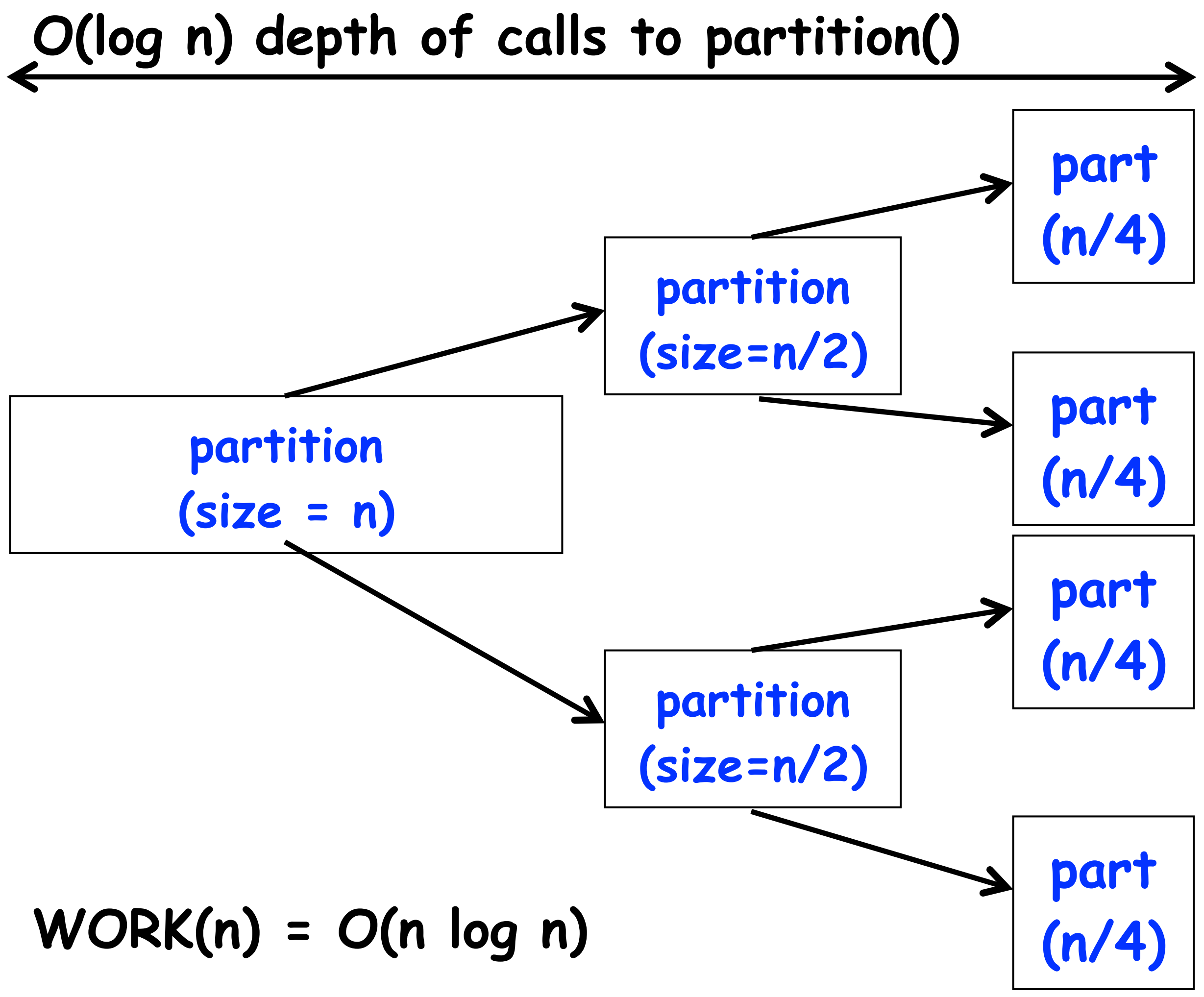
Sequential HJ implementation of partition()

```
static point partition(int[] A, int M, int N) {
  int I, J;
  Random rand = new Random();
  // Assign pivot to random integer in M..N
  int pivot = M + rand.nextInt(N-M+1);
  I = M; J = N;
  while (true) {
    /* up */ while (I<= N && A[I] <= A[pivot]) I++; if (I > N) I = N;
    /* down */ while (J>= M && A[J] >= A[pivot]) J--; if (J < M) J = M;
    /* change */
    if (I < J) { exchange(A, I, J); I++; J--; continue; }
    else if (I < pivot) { exchange(A, I, pivot); I++; break; }
    else if (pivot < J) { exchange(A, pivot, J); J--; break; }
    else break;
  } // while
  return [I, J];
} // partition
```

Two Opportunities in Parallelizing Quicksort

```
procedure Quicksort(S) {
   if S contains at most one element then return S
   else {
       choose an element a randomly from S;
       // Opportunity: Parallelize partitioning
       let S1, S2 and S3 be the sequences of elements in S less
       than, equal to, and greater than a, respectively;
       // Opportunity: Parallelize recursive calls
       return (Quicksort(S1) followed by S2 followed by
               Quicksort(S3))
    } // else
} // procedure
```

Approach 1: sequential partition, parallel calls



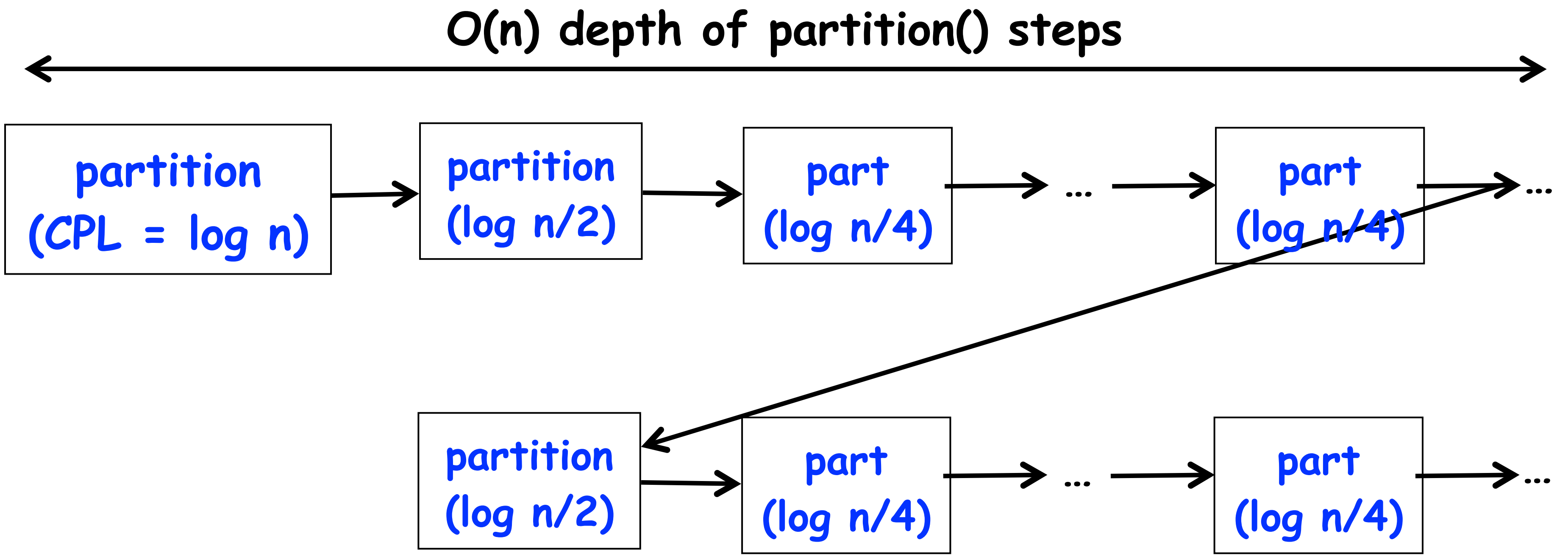
WORK(n) = O(n log n)

CPL(n) = O(n) + O(n/2) + O(n/4) + … = O(n)

Parallel HJ implementation of Quicksort for Approach 1

```
static void quicksort(int[] A, int M, int N) {
  if (M < N) {
    // partition() selects a pivot element in A[M…N]
    // to partition A[M…N] into A[M…J] and A[I…N]
    point p = partition(A, M, N);
    int I=p.get(0); int J=p.get(1);
    async quicksort(A, M, J);
    async quicksort(A, I, N);
  }
} //quicksort
 . . .
finish quicksort(A, 0, A.length-1);
```

Approach 2: Parallel partition, sequential calls



WORK(n) = O(n log n)

CPL(n) = log(n) + 2 log(n/2) + 4 log(n/4) + … = O(n)

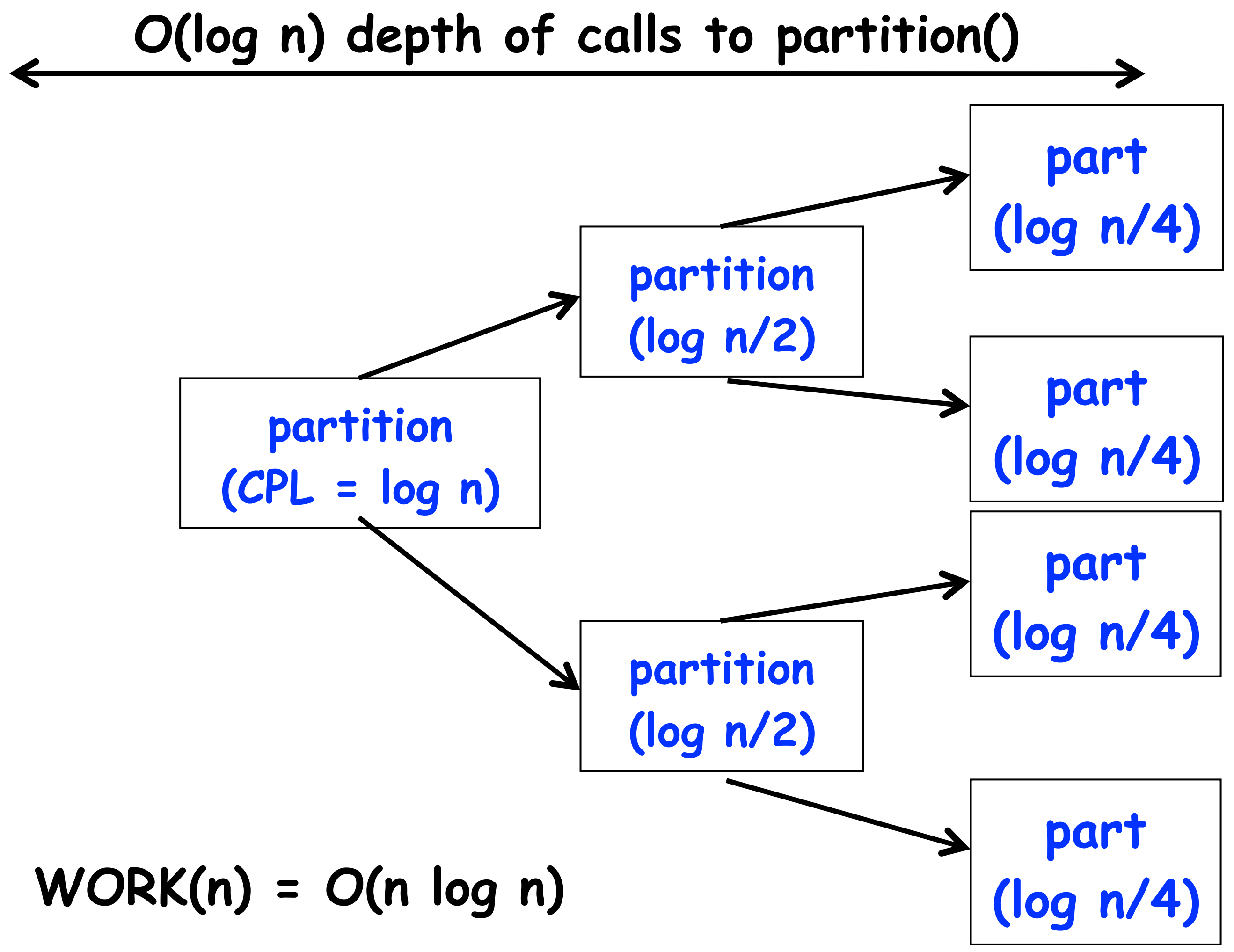
Parallel HJ implementation of partition() for Approach 2

```
static point partition(int[] A, int M, int N) {
  int I, J;
  int pivot = M + new java.util.Random().nextInt(N-M+1);
  int[] buffer = new int[N-M+1];
  int[] lt = new int[N-M+1];
  int[] gt = new int[N-M+1];
  int[] eq = new int[N-M+1];
  finish forasync(point [k] : [0:N-M]) {
    lt[k] = (A[M+k] < A[pivot] ? 1 : 0);
    eq[k] = (A[M+k] == A[pivot] ? 1 : 0);
    gt[k] = (A[M+k] > A[pivot] ? 1 : 0);
    buffer[k] = A[M+k];
  }
```

Parallel HJ implementation of partition() for Approach 2 (contd)

```
  int ltPsum = computePrefixSums(lt);   // Inclusive prefix sum
  int eqStart = M+ltPsum[ltPsum.length-1];
  int eqPsum = computePrefixSums(eq);   // Inclusive prefix sum
  int gtStart = eqStart + eqPsum[eqPsum.length-1];
  int gtCount = computePrefixSums(gt);  // Inclusive prefix sum
  finish forasync(point [k] : [0:N-M]) {
   if(ltCount[k]==1) A[M+lt[k]-1] = buffer[k];
   else if(eqCount[k]==1) A[eqStart+eq[k]-1] = buffer[k];
   else A[gtStart+gt[k]-1] = buffer[k];
  }
  if(eqStart == M) return [gtstart, eqStart];
  else if(eqStart == N) return [eqStart, eqStart-1];
  else return [gtStart, eqStart-1];
} // partition
```

Approach 3: parallel partition, parallel calls



WORK(n) = O(n log n)

CPL(n) = O(log n) + O(log n/2) + O(log n/4) + … = $O(\log^2 n)$