Data Structures & Algorithms for Problem Solving (CS1.304)

Sorting & Parallelism

Avinash Sharma

Center for Visual Information Technology (CVIT),

IIIT Hyderabad

Introduction

- Sorting is a fundamental concept in Computer Science.
  - several applications and a lot of literature.
  - We shall see two algorithms for sorting today
  - Try to introduce parallelism in sorting

QuickSort

- The quick sort algorithm designed by Tony Hoare is a simple yet highly efficient algorithm.
- It works as follows:
  - Start with the given array A of n elements.
  - Consider a pivot, say A[n].
  - Now, partition the elements of A into two arrays A_L and A_R such that:
    - the elements in A_L are less than A[n]
    - the elements in A_R are greater than A[n].
  - Sort A_L and A_R, recursively.

QuickSort

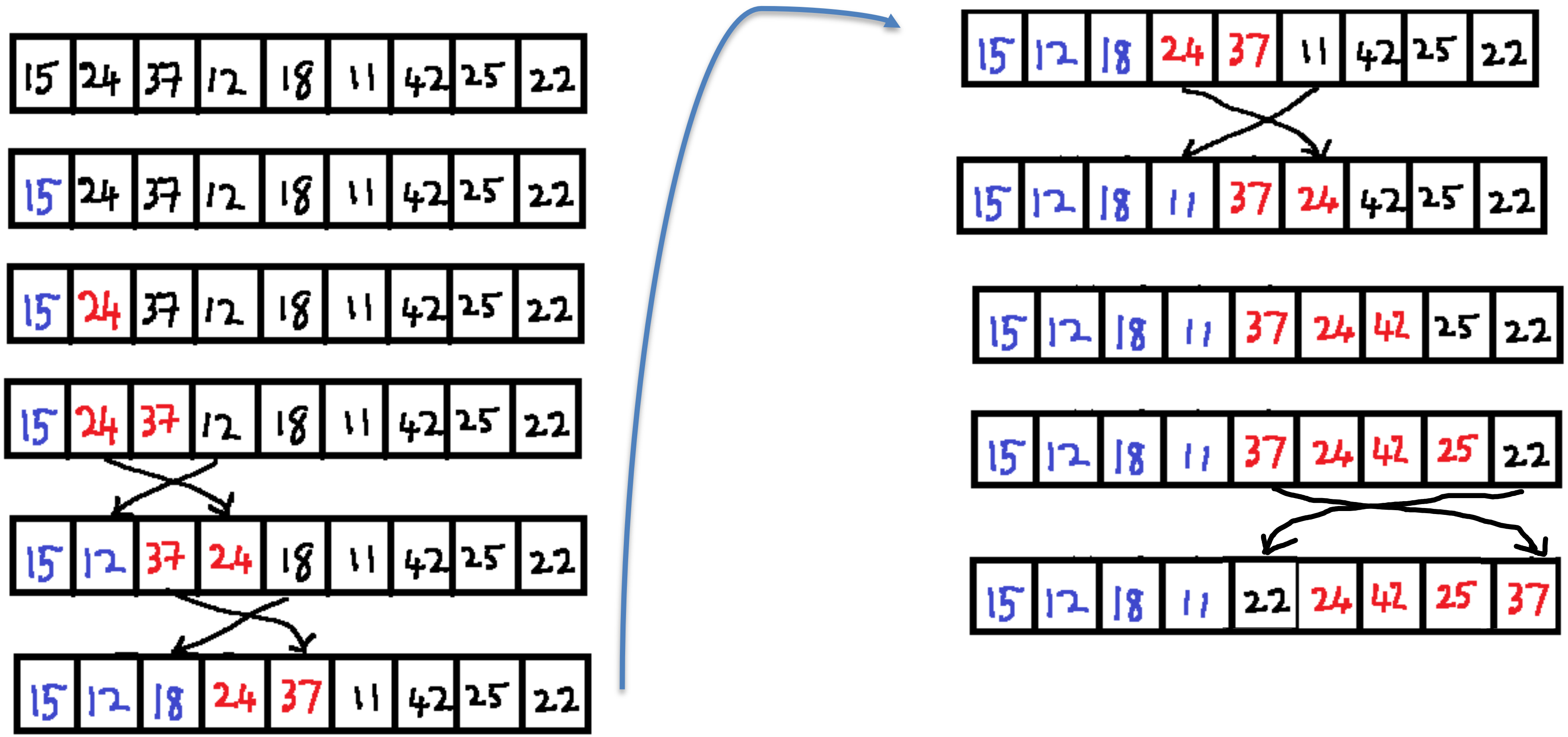
- How to partition ?
  - Suppose we take each element, compare it with A[n] and then move it to A_L or A_R accordingly.
  - Works in O(n) time.
  - Can write the program easily.
  - But, recall that space is also an resource. The above approach requires extra space for the arrays A_L and A_R
  - A better approach exists.

QuickSort

```
Procedure Partition(A,n)
begin
  pivot = A[n];
  less = 0; more = 1;
  for more = 1 to n-1 do
     if A[more] < pivot then
        less++;
        swap(A[more], A[less]);
     end
   end
   swap (A[less+1], A[n]);
end
```

15	24	37	12	18	11	42	25	22

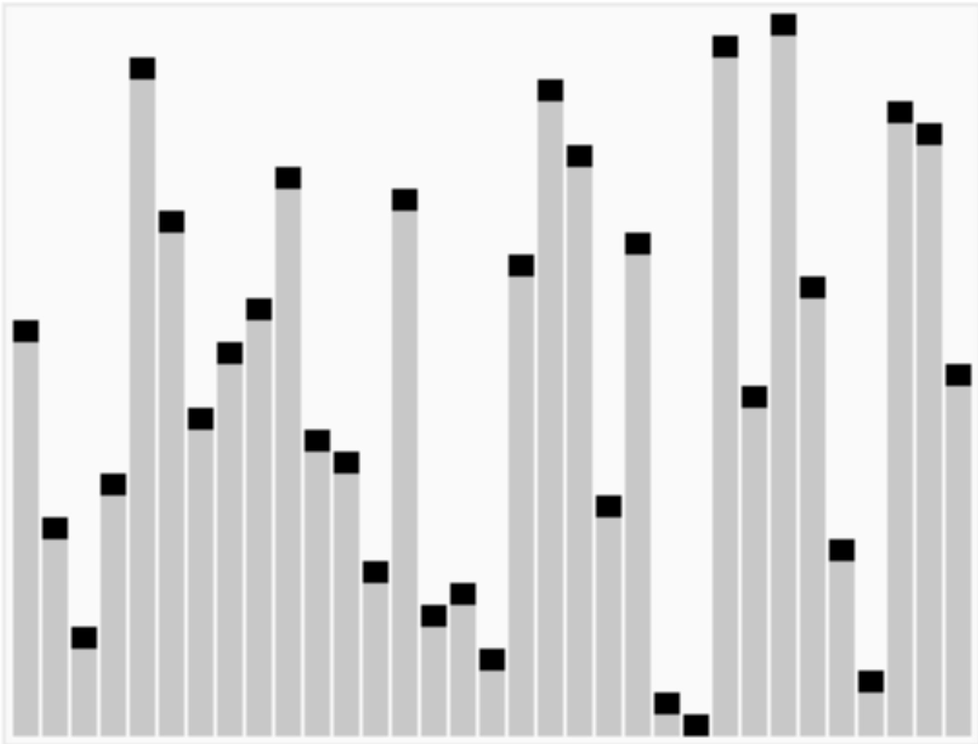
QuickSort



15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 24 37 12 18 11 42 25 22
15 12 37 24 18 11 42 25 22
15 12 18 24 37 11 42 25 22
15 12 18 24 37 11 42 25 22
15 12 18 11 37 24 42 25 22
15 12 18 11 37 24 42 25 22
15 12 18 11 37 24 42 25 22
15 12 18 11 22 24 42 25 37

QuickSort

- Graphical Visualization of recursive partitioning,



Graphics Courtsey: Wikipidea

Analyzing Quick Sort

- Suppose we run quick sort with A[n] as the pivot.
- Let A_L and A_R be the two subarrays obtained after partitioning.
- What is the time taken by quicksort?
- As a recurrence relation, $T(n) = T(|A_L|) + T(|A_R|) + O(n)$.
- To be able to solve this recurrence relation, need to know the sizes of arrays A_L and A_R.

Analyzing Quick Sort

- We know that $|A_L| + |A_R| = n-1$.
- But, if the pivot is such that all elements are smaller (or larger) than the pivot, then $|A_L|$ (or $|A_R|$) $= n-1$.
- The recurrence relation in that case is

$$T(n) = T(n-1) + O(n).$$

- Suppose the same situation happens over every recursive call. So, the above recurrence relation holds during every recursive call.
- When will this happen ?

Analyzing Quick Sort

$$\underbrace{12 \quad 15 \quad 17 \quad 24 \quad 29 \quad 36}_{A_L} \mid 42$$

$$\underbrace{12 \quad 15 \quad 17 \quad 24 \quad 29}_{A_L} \mid 36$$

$$\underbrace{12 \quad 15 \quad 17 \quad 24}_{A_L} \mid 29$$

$$\underbrace{12 \quad 15 \quad 17}_{A_L} \mid 24$$

$$\underbrace{12 \quad 15}_{A_L} \mid 17$$

Analyzing Quick Sort

- Is it always that bad?
- What if the pivot is such that each recursive iteration, the sizes of $|A_L|$ and $|A_R|$ is exactly the same?
- The recurrence relation then stands as:

  $T(n) = 2T(n/2) + O(n).$

- Which element as the pivot ensures that the sizes of $|A_L|$ and $|A_R|$ are exactly the same?
- Can that happen in every run when you pick a pivot to be the last element? Or the first element? Or even uniformly at random?

Analyzing Quick Sort

- In general, if the sizes of $|A_L|$ and $|A_R|$ are such that they are a constant away from each other, then the recurrence relation is:

  $$T(n) = T(an) + T((1-a)n) + O(n)$$

  where a is a constant < 1.

- In practice, it turns out that most often the partitions are not too skewed.
- So, quick sort runs in $O(n \log n)$ time almost always.

Merge Sort

- Another sorting technique.
- Based on the divide and conquer principle.
- We will first explain the principle and then apply it to merge sort.

Divide and Conquer

- Divide the problem P into $k \geq 2$ sub-problems P_1, P_2, …, P_k.
- Solve the sub-problems P_1, P_2, …, P_k.
- Combine the solutions of the sub-problems to arrive at a solution to P.

Divide and Conquer

- A useful paradigm with several applications.
- Examples include merge sort, convex hull, median finding, matrix multiplication, and others.
- Typically, the sub-problems are solved recursively.
  - Recurrence relation

$$T(n) = \underbrace{D(n)}_{\text{Divide time}} + \underbrace{\sum_i T(n_i)}_{\text{Recursive cost}} + \underbrace{C(n)}_{\text{Combine time}}$$

Divide and Conquer

- Combination Procedure: Merge

$\longrightarrow$ 8 10 12 27 $\qquad\qquad$ $\longrightarrow$ 15 17 24 32

8

8 10

8 10 12

8 10 12 15

8 10 12 15 17

8 10 12 15 17 24

8 10 12 15 17 24 27

8 10 12 15 17 24 27 32

Algorithm Merge

```
Algorithm Merge(L, R)
// L and R are two sorted arrays of size n each.
// The output is written to an array A of size 2n.
int i=1, j=1;
L[n+1] = R[n+1] = MAXINT; // so that index does not
                          // fall over
for k = 1 to 2n do
        if L[i] < R[j] then
                A[k] = L[i]; i++;
        else
                A[k] = R[j]; j++;
end-for
```

Time complexity is O(n).

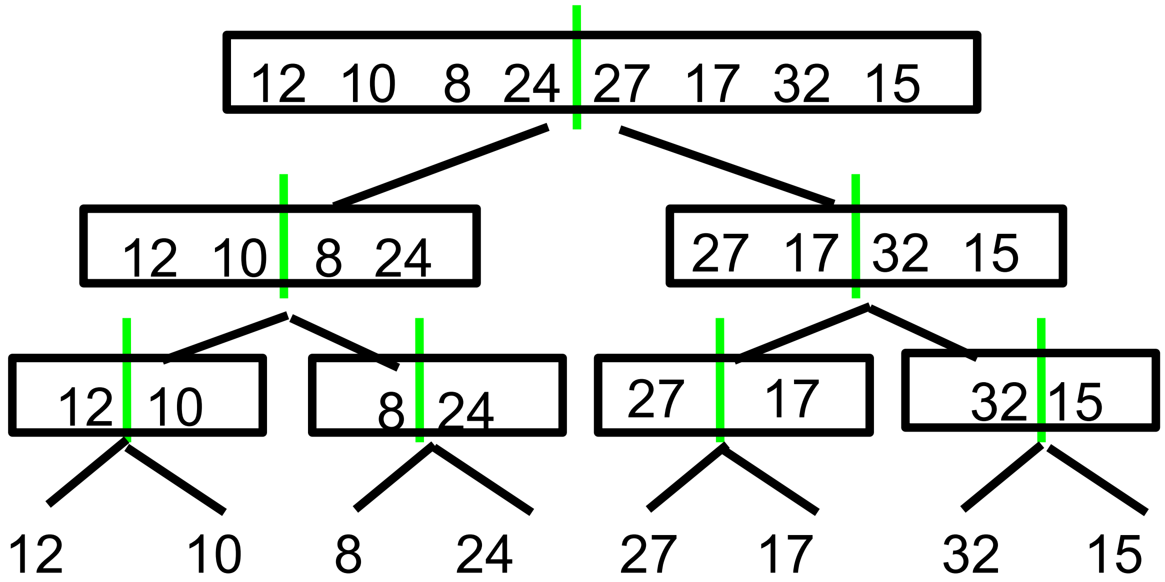
From Merging to Sorting

- How to use merging to finally sort?
- Using the divide and conquer principle
  - Divide the input array into two halves.
  - Sort each of them.
  - Merge the two sub-arrays. This is indeed procedure Merge.
- The algorithm can now be given as follows.

Algorithm Merge Sort

```
Algorithm MergeSort(A)
begin
        mid = n/2; //divide step
        L = MergeSort(A[1..mid]);
        R = MergeSort(A[mid+1..n]);
        Merge(L, R); //combine step
end-Algorithm
```

Divide & Conquer Merge Sort



- Example via merge sort: 1) Divide is split into two parts

  2)Recursively solve each subproblem

Analyzing Merge Sort

- Recurrence relation for merge sort as:

$$T(n) = 2T(n/2)+O(n).$$

  - This can be explained by the O(n) time for merge and
  - The two subproblems obtained during the divide step each take T(n/2) time.
  - Now use the general format for divide and conquer based algorithms.
- Solving this recurrence relation is done using say the substitution method giving us T(n) = O(n log n).
  - Look at previous examples.

Introducing Parallelism in Computing

- My laptop has 8 cores.
- What does it mean?
  - In principle, each core can run some instructions on its own independently.
  - So, while one core is possibly running the browser, the one can run an editor, the third can run a PDF reader, and the fourth can run a mail client, etc.
  - Is that helpful in all situations?

Introducing Parallelism in Computing

- There are two reasons why the previous model is not enough.
- There are applications that are time critical and can use ALL the cores for themselves.
  - Examples include weather forecasting, cyber-security, and the like.
- Secondly, while the number of cores is increasing, per core performance is dropping.

Introducing Parallelism in Computing

- So, how can MY program use ALL the cores simultaneously?
- Can the architecture help?
- Can the OS help?
- Can the compiler help?
- Can I help?
  - Very much. That is what parallel computing is all about.
- How can I help?
  - Algorithm Designer
  - Programmer
  - Computer Science student,
  - All of the above.

Maxima of n Numbers

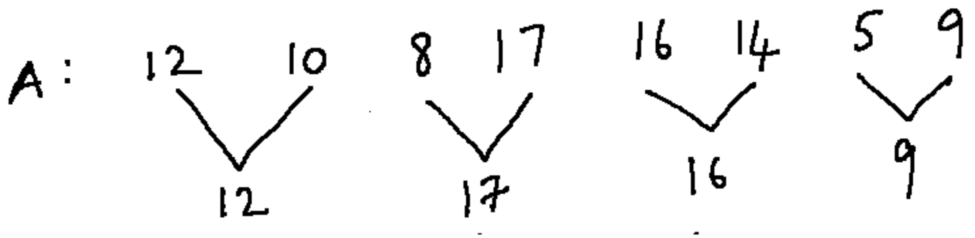
- A sequential program resembles the code below.

```
Program Maxima(A)
Begin
    int max = A[1];
    for i = 1 to n do
        if max < A[i]
            max = A[i]
        End-if
    End-for
End
```

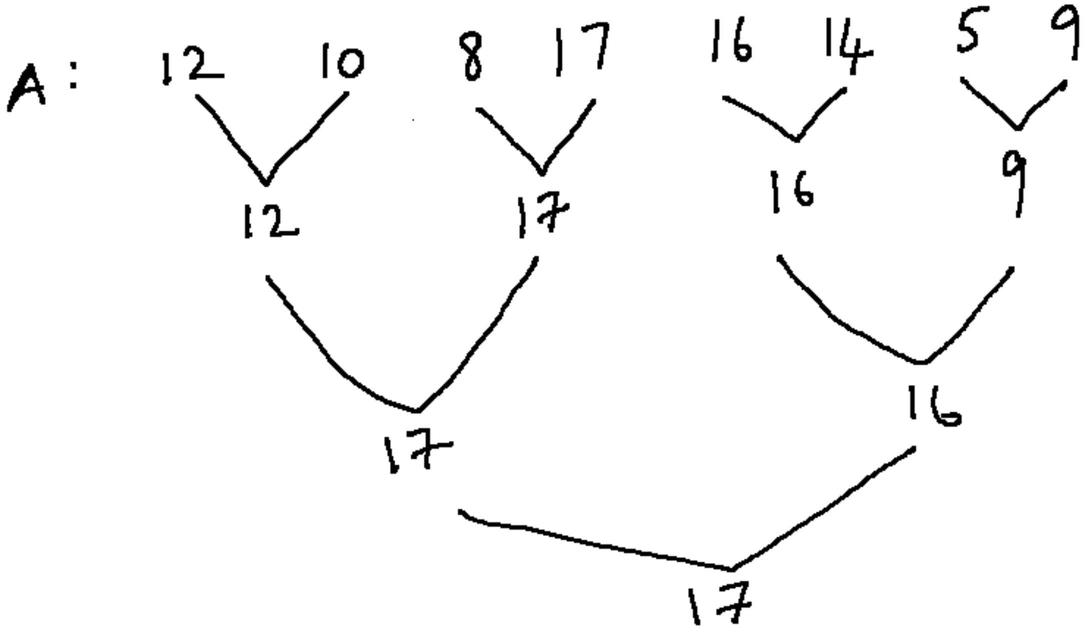
- How do we run this program in parallel?

Maxima of n Numbers

- Suppose we compare pairs of numbers and record the maximum.
- The overall maximum is one of these local maxima.
- So, we could compare pairs of local maxima, to get more local maxima.
- Continue until only one pair remains with the property that the maxima of the input is the maximum of the pair that remains.



Maxima of n Numbers



Maxima of n Numbers

- Can you write the above idea as a parallel algorithm?
- Some hints. Let n be the size of the array.
  - How many pairs are compared in the first iteration?
  - In the second iteration?
  - In the ith iteration?
  - How many iterations have to be executed to get the final maxima?
  - Which parts of the algorithm are parallel?
    - Within iterations or across iterations?

Parallelism in Computing

- Think of the sequential computer as a machine that executes jobs or instructions.
- With more than one processor, can execute more than one job (instruction) at the same time.
  - Cannot however execute instructions that are dependent on each other.
- This opens up a new world where computations have to specified in parallel.
- Sometimes have to rethink on known sequential approaches.

Parallelism in Computing

- How many independent operations can be done at a time?
  - Depends on the number of processors available.
- Assume that as many as n, or n/2, processors are available.
- Most often, our analysis suggests that the computation takes only O(log n) time, but we need n processors for this.
- Cannot ensure that the number of processors also grow with the input size.
- In practice, the number of processors on a machine does not change!

Parallelism in Computing

- The idea of the parallel algorithm is to show the extent of parallelism available in the computation.
- Plus, if there are fewer processors than what is required, can always simulate more processors.
- For instance, if there are p processors and n processors are required, then each of the p processors simulates the actions of n/p processors.

Parallel Merge Sort

- An algorithm is a sequence of tasks T1, T2, ...
- These tasks may have inter-dependecies,
  - Such as task Ti should be completed before task Tj for some i,j.
- However, it is often the case that there are several algorithms where many tasks are independent of each other.
  - In some cases, the algorithm or the computation has to be expressed in that independent-task fashion.

Parallel Merge Sort

- In such a setting, one can imagine that tasks that are independent of each other can be done simultaneously, or in parallel.
- Let us think of arriving at a parallel merge sort algorithm.

Parallel Merge Sort

- What are the independent tasks in merge sort?
  - Sorting the two parts of the array.
  - This further breaks down to sorting four parts of the array, etc.
  - Eventually, every element of the array is a sorted sub-array.
  - So the main work is in merge itself.

Parallel Merge Sort

- So, we just have to figure out a way to merge in parallel.

- Recall the merge algorithm as we developed it earlier.
  - Too many dependent tasks.
  - Not feasible in a parallel model.

```
for k = 1 to 2n do
        if L[i] < R[j] then
                A[k] = L[i]; i++;
        else A[k] = R[j]; j++;
end-for
```

Parallel Merge Sort

- Need to rethink on a parallel merge algorithm
- Start from the beginning.
  - We have two sorted arrays L and R.
  - Need to merge them into a single sorted array A.

Parallel Merge Sort

- Need to rethink on a parallel merge algorithm
- Start from the beginning.
  - We have two sorted arrays L and R.
  - Need to merge them into a single sorted array A.
- Define the rank of an element x in a sorted array A as the number of elements of A that are smaller than x.
- To merge L and R, need to know the rank of every element from L and R in the merged array L U R.

Parallel Merge Sort

- Importantly, for any x in L or R,

  Rank(x, L U R) = Rank(x, L) + Rank(x, R).

- So, merging is equivalent to finding the two ranks on the right hand side.

Parallel Merge Sort

- Now, consider an element x in L at index k.
- How many elements of L are smaller than x?
  - k-1.
- How many elements of R are smaller than x?
  - No easy answer, but
  - can do binary search for x in R and get the answer.
  - Say k' elements in R are smaller than x.

Parallel Merge Sort

- How many elements in LUR are smaller than x?
  - Precisely k + k‘ - 1.
- So, in the merged output, what index should x be placed in?
  - precisely at k+k'.
- Can this be done for every x in L?
  - Yes, it is an independent operation.
- Can this be done for every x in R also?
  - Yes, replace the roles of L and R.
- All these operations are independent.

Parallel Merge Sort

L = [8 10 12 27]

R = [15 17 24 32]

Element	8	10	12	27	15	17	24	32
Rank in L	0	1	2	3	3	3	3	4
Rank in R	0	0	0	3	0	1	2	3
Rank in L U R	0	1	2	6	3	4	5	7

L U R = [8 10 12 15 17 24 27 32]

Parallel Merge Sort

```
Algorithm ParallelMergeSort(A)
Begin
        mid = n/2; //divide step
        L = MergeSort(A[1..mid]);
        R = MergeSort(A[mid+1..n]);
        ParallelMerge(L, R); //combine step
  end-Algorithm
```

Thank You