Assignment 1

Nitin Purohit (800956312)

1) a) As, each instance of loop is independent, each loop iteration can be given to different processors. There will not be any dependency between each loop iterations. But, to calculate b[i], we need to find f(a[i]). So, dependencies will look like,

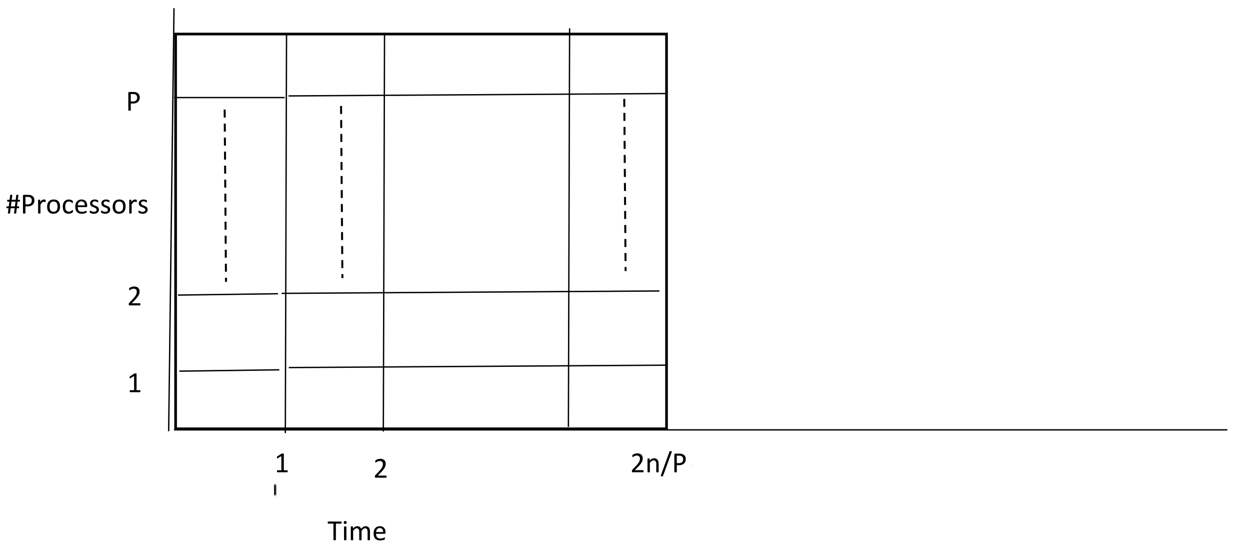
   F(a[i]) -> b[i], it is a RAW(Read after Write) type of dependency,

   b) The total work, for this will be 2n, as, there are two tasks, and each takes, O(1), So, **work** = 2n.

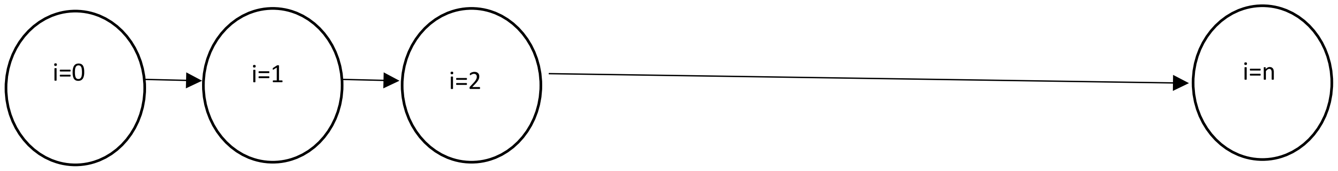
   Width = n, as each loop can be handled by a different processor. So, we can use n processors.

   The critical path if we use P processors will be, **C.P.** = 2n/P, since, we can divide the entire loop iterations into P parts, where P are the number of processors.

   c) **Schedule for 'P' processors**,



2) 2.1 – a) As, we need to find the sum of array in this case, the order of each loop iteration does not matter, but, the tasks are dependent on each other, as each will require the sum calculated by previous tasks. So, the dependency graph will look as,



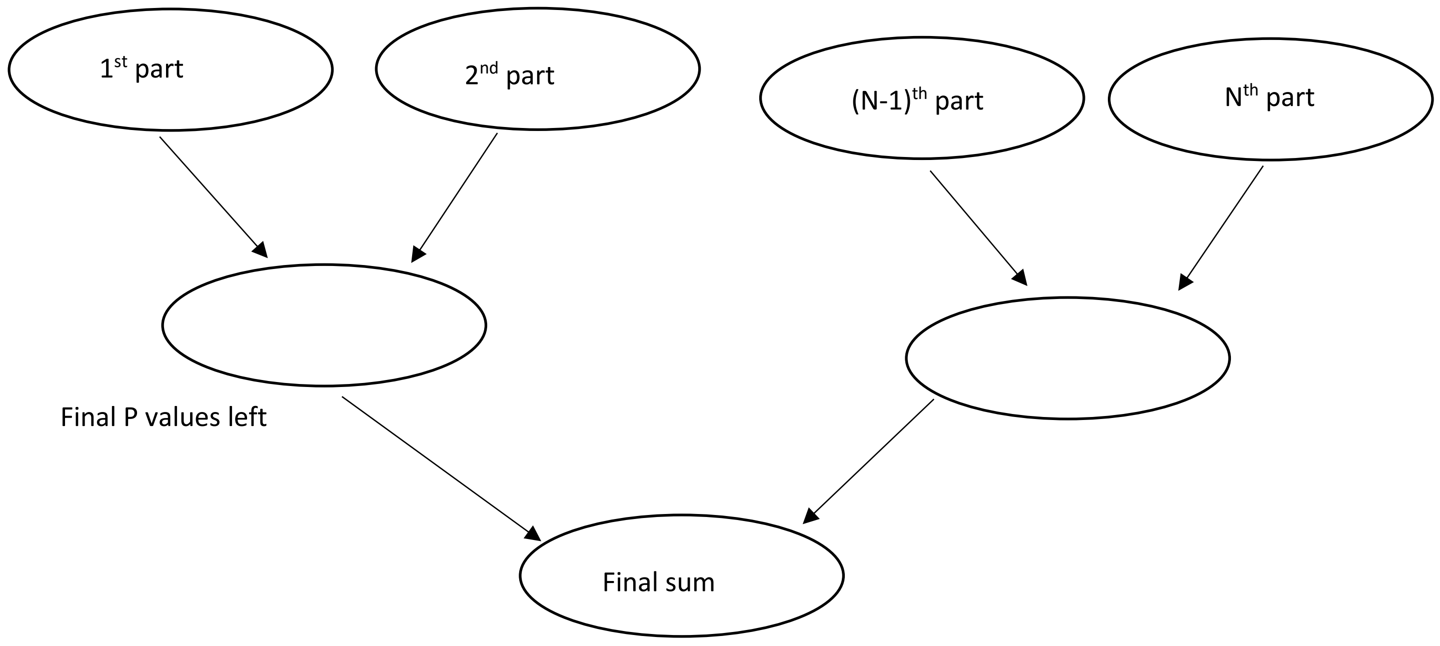
Work: n

Critical Path: n

Width : 1

b) Yes, introducing a mutual exclusion does help in this case. As iterations can be in any order, we can divide the array into the ‘P’ parts, and then each part of array can be processed by different processor. Each part of array will be mutually exclusive.

c) So, if we divide the array into ‘P’ parts, and use ‘P’ processors. The dependencies will look as below now,

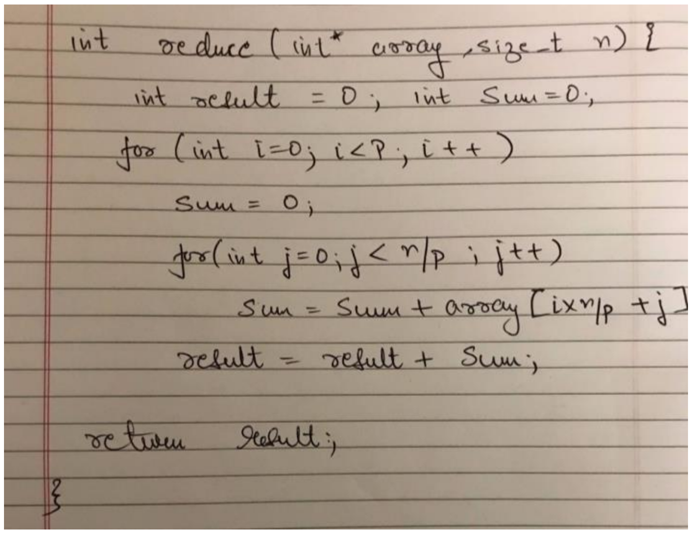


Work: n

Critical path: (n/P + P), for ‘P’ processors, where n/P is for finding sum of each part, and then P is to add final sum coming from each array chunk.

Width: P

Algorithm (for P processors-each processor calculates a chunk of array):

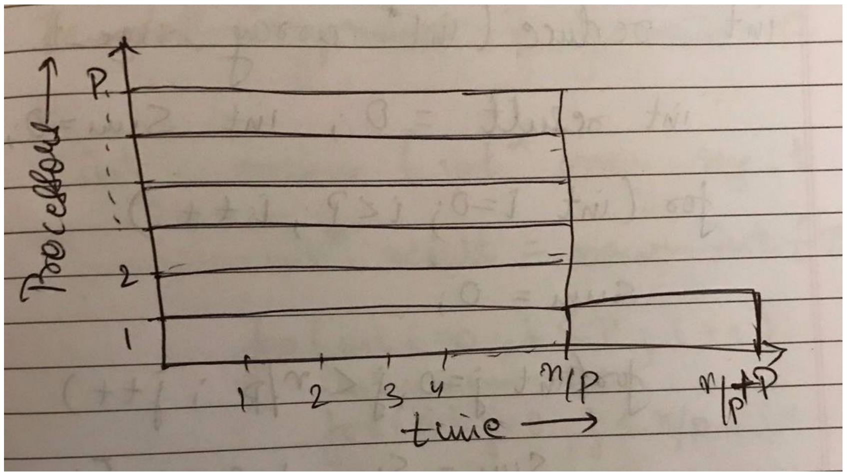


```
int reduce ( int* array ,size_t n) {
    int result = 0 ;  int sum=0;
   for (int i=0; i<P ; i++ )
       sum = 0;
       for(int j=0;j< n/p ; j++)
           sum = sum + array [ixn/p +j]
       result = result + sum;

  return result;
}
```

Each loop iteration of inner loop can be in the to a separate processor and then the output from each processor is added together to get the final sum.

d) **Schedule**:



2.2 –

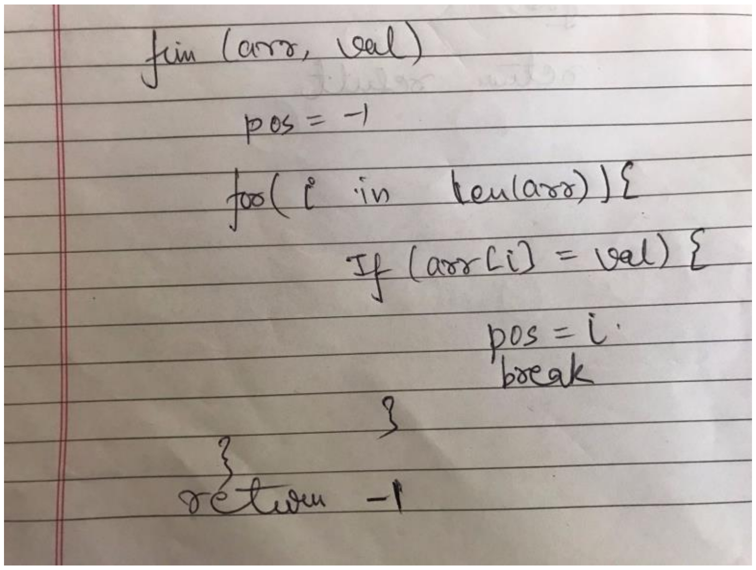
a) Yes, because, in max , we can still divide the array, as the order still does not matter.

b) No, in case of concat, the order matters, so we can’t divide the calculations. We need to maintain the same order, so, we can’t remove the dependencies.

c) Yes, changing the data type from int to float won’t make a difference in parallelization.

d) Yes, the same explanation as for sum, applies to max also, as changing data type from int to float won’t make a difference.

3) 3.1 – a) sequential algorithm:



```
fun (arr, val)
    pos = -1
    for( i in len(arr)){
        If (arr[i] = val) {
            pos = i.
            break
        }
    }
return -1
```

b) Complexity of algorithm:

If val is found in array=> theta(pos)

Else => theta(n)

c) Parallel algorithm with theta(n) work: We can make it parallel by dividing the array and giving a part of array to each processor to search. As, we know algorithm takes theta(n) only when the val is not found in array, so, the order won't matter. If we use 'P' processors,

Critical Path: n/P

Width: P

d) Parallel algorithm with theta(pos) work: We can't make it parallel, because if we divide the array, and val occurs more than once in the array, we will not be sure, which processor will find the val first. Therefore, it is not certain which location of val will we get as output.

3.2 –

With linked list, we can just perform the algorithm in sequential algorithm, as we can't separate the linked lists like arrays. We must traverse from starting to go to a specific location. So, parallel algorithm is not possible.

4) a) The dependency graph, without any parallelization will look as below,

$$\begin{tikzcd}
i=0 \arrow[r] & i=1 \arrow[r] & i=2 \arrow[r] & i=n
\end{tikzcd}$$

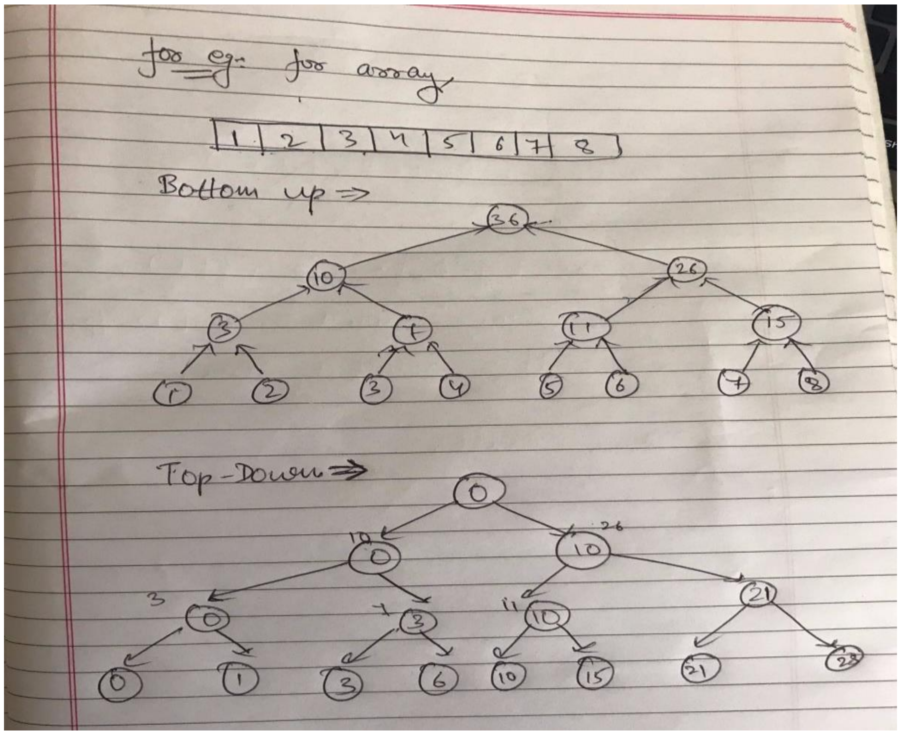
Each loop iteration is dependent on previous iteration. This will have complexity of O(n).

b) Prefix sum in parallel using Binary tree:

In this algorithm, we treat each value in array as a leaf node of tree. We then sum up each subtree in bottom-up fashion. Note that, for each subtree we find the intermediate sums, and we stop when we reach the root node. This will give us our last element.

Now, to find the intermediate prefix sums for each subtree correctly, we iterate the tree again in top-down fashion. As, we already have the last element correct, we replace that by 0. We will keep adding the root of intermediate subtree with left child, and put this sum into root of right subtree. We do this because, we are folding left to right here, and the right side will be the sum of all the values to the left of it. We keep shifting the 0 to the root of left subtree. After doing log(n), iterations, we reach all the leaves again.

This will give us our final answer, which is prefix sum of the initial array.



5) a) MergeSort: Merge sort works on divide and conquer terminology. We first divide the array recursively into smaller parts, and then we recursively merge the elements back.

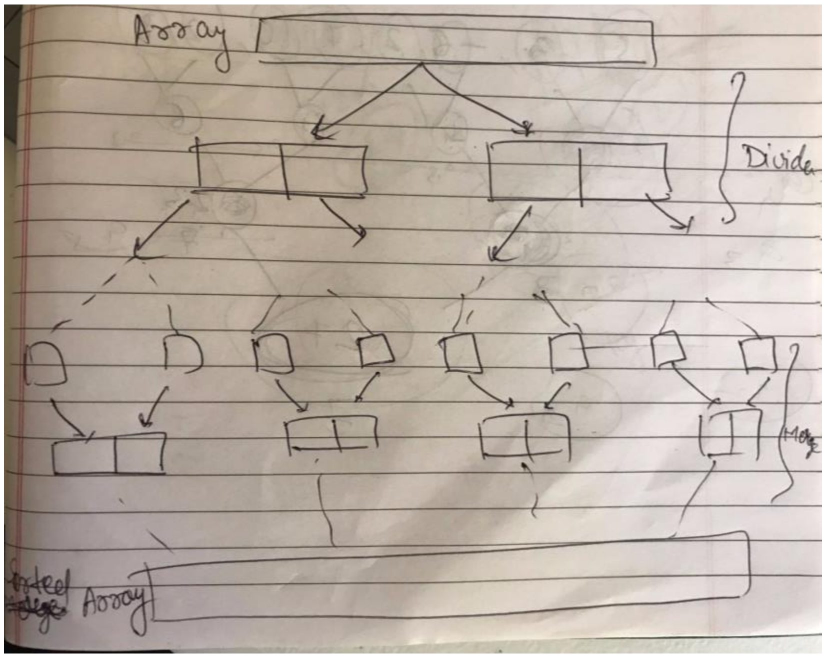
Algorithm:

```
merge_sort(A,p,r){

     If(p<r)
     {
         q = (p+r)/2
         merge_sort(A,p,q)          //dividing
         merge_sort(A,q+1,r)      //dividing
         merge(A,p,q,r)              //merge
    }
}
```

In merge, we combine the arrays back accordingly.

b) As we know, we divide the array recursively in merge sort, and these divided array chunks can be handled separately. But, as we recursively divide the array, the next division will need previous divide computation. So, dependency can be given as,

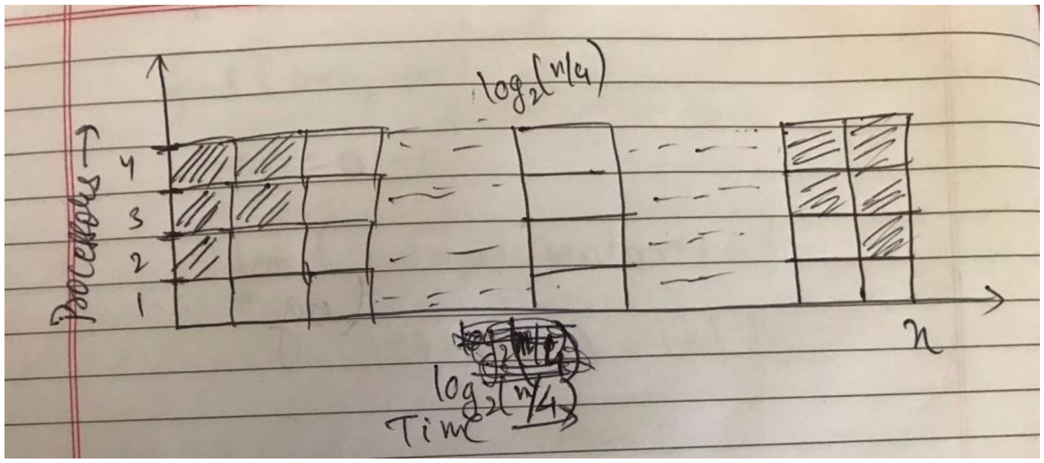


Work = nlogn

C.P. = n

Width = n

c) **Schedule for 4 processors**:



d) Now, as we can notice, the stand-out with our previous parallelism is only log(n). This is because of the merge phase, which takes O(n). This is dominating the overall performance of the merge sort algorithm. So, one way to extract more parallelism can be to if we parallelize the merge phase also. If we find the median element of the two arrays to be merged, then, handle the elements smaller than median and elements greater than median separately.