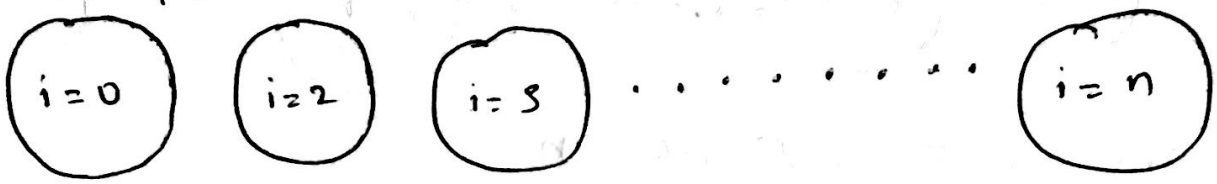
AKARSH GUPTA (B00969888)

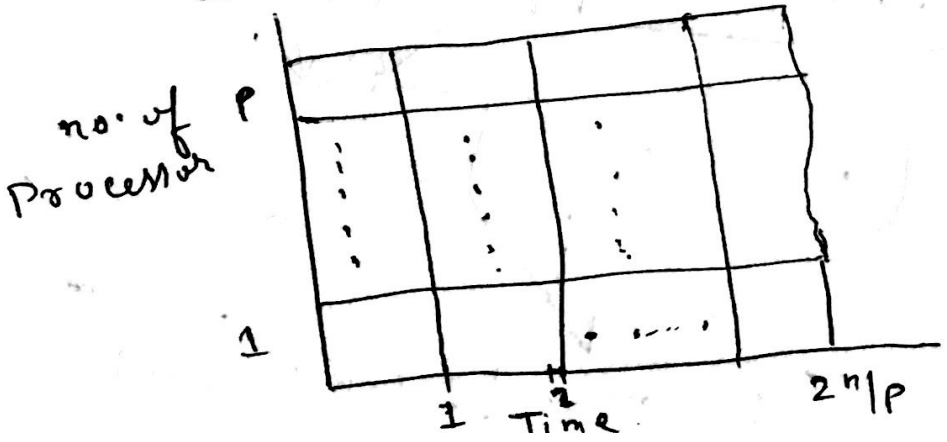
ASSIGNMENT-1, PARALLEL COMPUTING

Sol 1.a). In the given Transform method, there is an instruction inside the loop, this instruction is independent in each iteration, hence the loop iteration can be given to multiple processor at the same time. Therefore the dependency graph will look like below; There is a Read After write dependency (RAW)
So f(a[i]) → b[i]

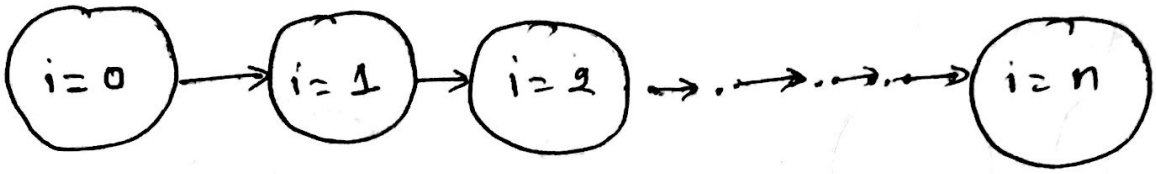


1b.) As from the above figure, we can see there is no dependency, hence all the tasks are independent and each task takes O(1). The critical path for this is C.P = 2n/p Total work can be calculated as the cost is 2n O(1) and runs of n, hence the total work is = 2n, since there are total two tasks in each iteration and both takes O(1) and width = n.

1c). Schedule for 'p' processors.



Sol 2 → In this code snippet, the method takes operator and an array and that operation is performed on each element and saved as result to be performed on the next element. Here In the first case as the operator is sum and array type is Int, the order of iteration does not matter but the tasks are dependent on the result from the previous task. So the dependency graph will look like,

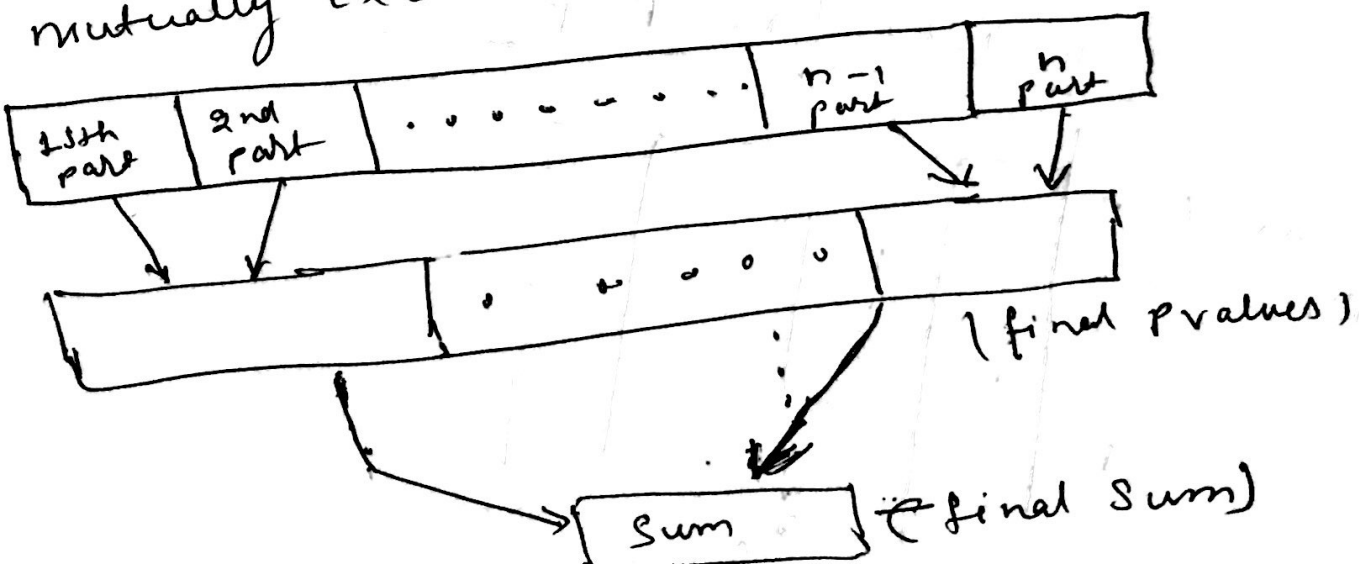


So, the work will be = n

$$C \cdot P = n$$

and width = 1

Since it is a sum operation and order of iteration does not matter or affect in this case, so we can divide the array size in 'p' parts, let P be the number of processors. and we divided the array in P parts to process them parallely, Each part of array will be mutually exclusive.



work = n

c.p = $(n/p + p)$ for p processors.

width = p.

```
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[i*n/p + j]

      }
     result = result sum;

   }
  return result;
}
```

Sol 3.1 algorithm for sequential find first.

```
find_first(arr, val)
    pos = -1
    for(i=0 to len(arr))
        If(arr[i] = val)
            pos = i
            return pos;
~~If(pos == -1)~~
return(~~pos~~ pos)
```

2.2 a) Yes, the parallel version will be correct for int and Max, as the order of iteration does not matter and we can repeat the above divide array approach.

b) NO, In concat & String order of iteration matter as it will change the output with different iteration hence we cannot remove dependencies in this case by dividing the array.

c) Yes, Changing data type of elements in array does not affect in iteration hence it won't affect parallelization.

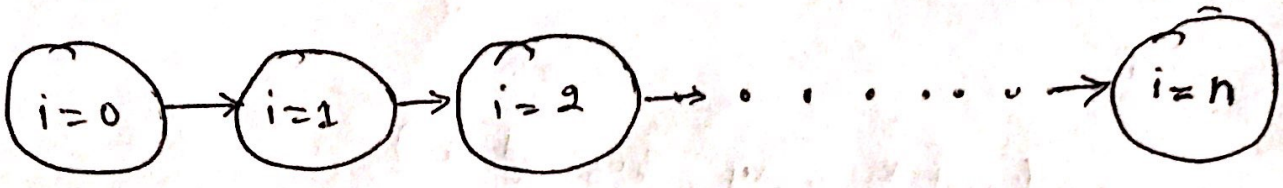
d) Yes, As above changing data type here does not make any changes in iteration hence dependency remains intact as earlier max, Int case and won't affect the parallelism.

3.1 Sequential algorithm for finding first element in an array of size 'n', returning the position (pos) in the array (arr) where arr[pos] = val else return 'n', The complexity of the algorithm will be

case 1: if (arr[pos] == val). $\Rightarrow O(pos)$

case 2: val not found in arr $\Rightarrow O(n)$

Sol 4. The dependency graph for this problem is below:-



Every iteration is dependent on the previous iteration. giving us the complexity of O(n).

prefix Sum can be parallelised using Binary Tree:-

In this Binary Tree approach, the array values are treated as the leaf node. we then start to add each subtree in a bottom-up approach, for each subtree we find the intermediate sums till the root node and we will get our last element. Now to find the intermediate prefix sum, we again iterate each subtree in top-down approach. In the bottom-up approach we found the last element, so we replace that with zero and we keep adding the root of intermediate subtree with left-child and keep the sum in right of root node. This is folding left to right so the right becomes sum of all values to left of it. and keep shifting zero to the root of left subtree. After log(n) iteration we get our final result.

Parallel Algorithm with $O(n)$ work :- It can be parallelized by dividing the array and giving these parts of array to different processors, As we know the algorithm takes $O(n)$ when the value is not found. So even ~~array~~ it can be processing parallely as the order does not matter. Let's take 'p' processor so the

$c \cdot p \geq n/p$ and width $= p$.

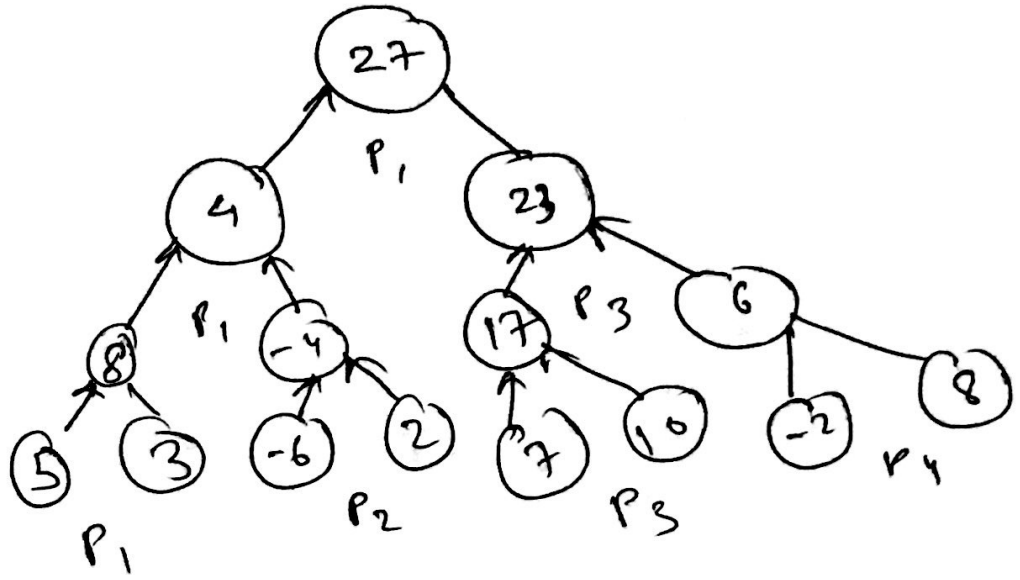
Parallel Algorithm with $O(pos)$:- It cannot be parallelised because the algorithm is to find first match and hence as above we cannot divide the array to be processed in multiple processors, Now if the val occurs multiple times and we make it parallel ~~we width~~ we may get wrong result. hence It cannot be parallelized.

Sol 3.2. As we know, linked List is a sequential data ~~structure~~ structure and accessing elements happens in $O(n)$, so we cannot seperate the linked list like arrays as every node in the linked has the link to the next node. hence we cannot make it parallel.

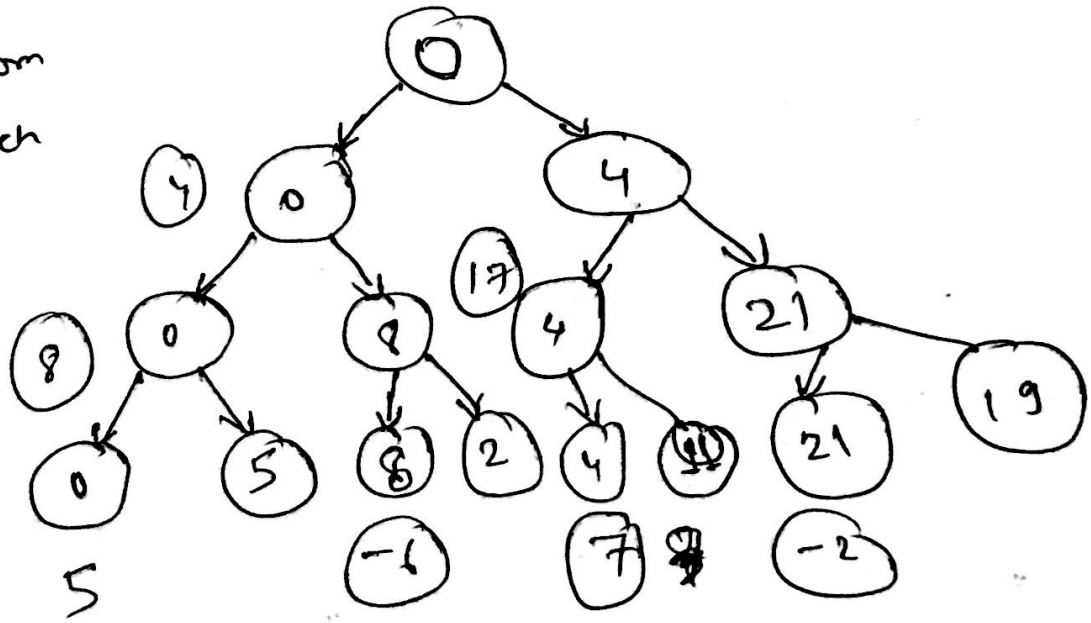
figure for prefix sum.

5	3	-6	2	7	10	-2	8

Bottom up



Top-Bottom Approach



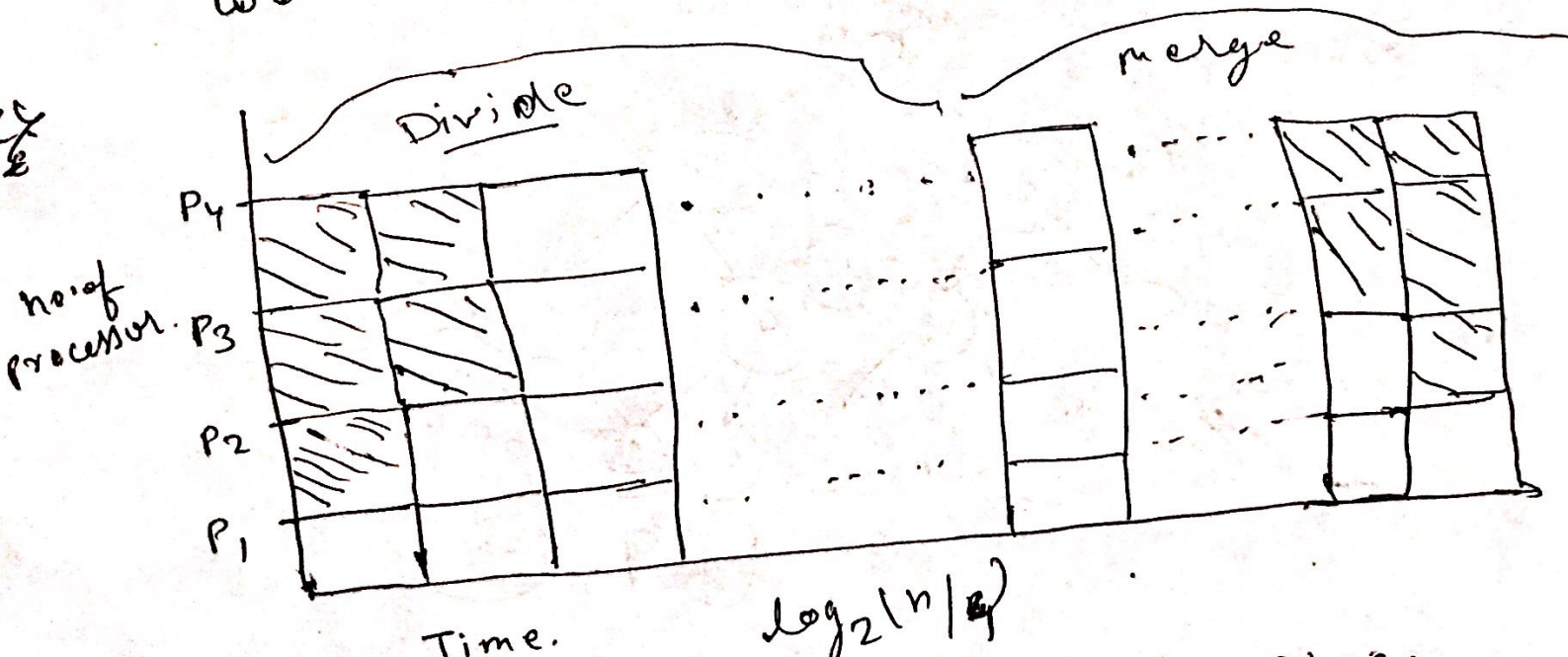
Sol 5) Mergesort :- Merge sort is a Divide and conquer algorithm, where the array is divided first recursively into smallest parts then, it is recursively merged and sorting is done on merge.

Algorithm for merge sort.

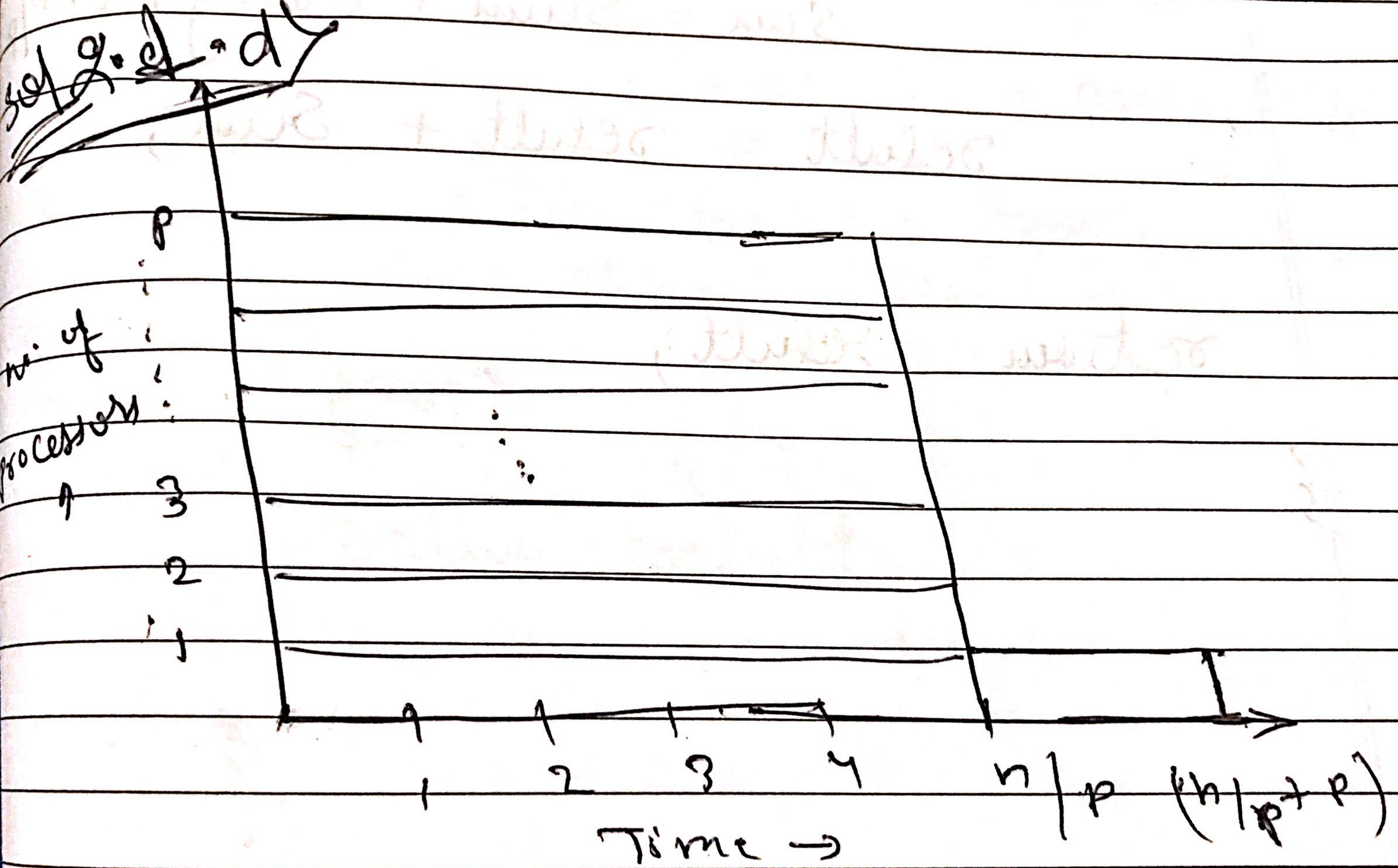
```
Mergesort (A, P, r)
{
  if (p<r)
  {  q = (p+r)/2
     mergesort (A, P, q)
     mergesort (A, q+1, r)
     merge (A, P, q, r)
  }
}
```

b) Here, we recursively divide the array and the next division need this previous divided array, so dependency is :-

work = $n \log n$, $c.p = n$, width = n



d) In the above parallel algorithm merge phase takes $O(n)$ to merge and dominates the whole parallelism. we can extract more parallelism by parallelising this merge phase also by finding the median and handling smaller and larger array seperately.



no. of processors ↑
p
3
2
1
1
2
3
4
n/p
(n/p + p)
Time →