Parallel Computing

Assignment-2

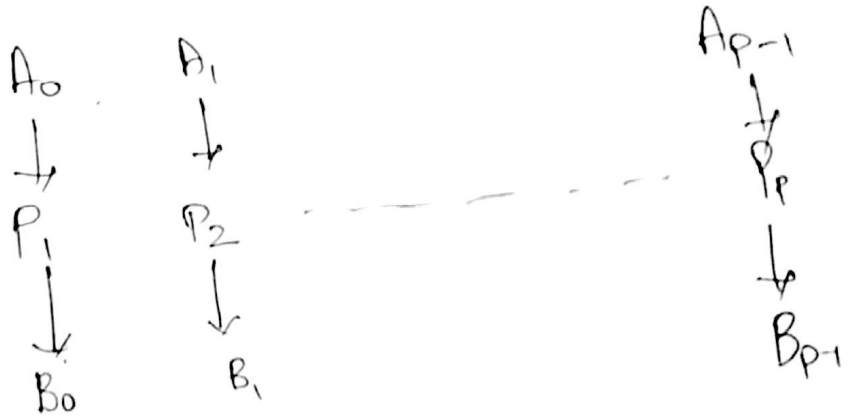
SHARAN GIRDHANI

1) a) Read-After-Write (RAW) Dependency.
As we are reading the data from A after it being returned as the output.
(writing)

b) Let the given Array be $A_0, A_1 \ldots A_{n-1}$.
and let the no. of processors be P.

So, At first we take first P elements of the array A and assign it to P processors.

Graph would look like:-



After this we repeat this process until it covers all the elements.

So, width = P

Now, Assuming that every operation here takes O(1) time.

$2 * O(1) \longrightarrow P$ elements.

To compute the entire array of n elements:-

$$C \cdot P = 2 \cdot O(1) \cdot \frac{n}{p} = \frac{2 \cdot n}{p}$$

Now, for writing each element of A into B, 2 processes are happening:-

1) $f \rightarrow O(1)$ time.

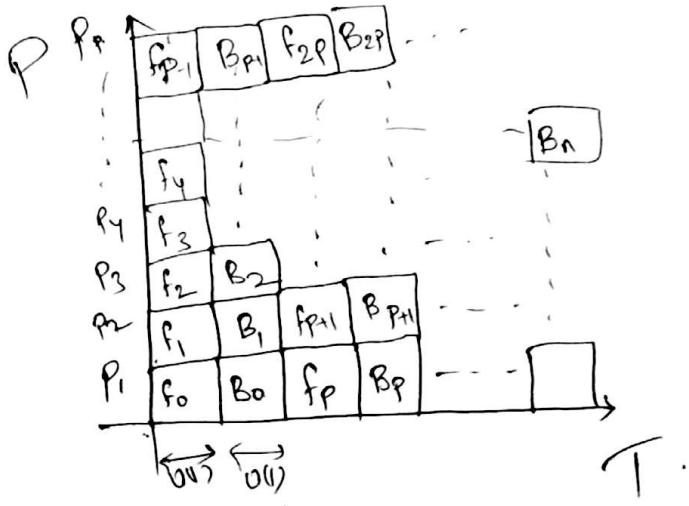
2) $B_j = f(A_j) \rightarrow O(1)$ time.

for n elements.

Total work = $2n$

e) Schedule for p processors:-

Since, we can say that All the individual processes are taking O(1) time, I am making the schedule based on List Scheduling.



2.1) a) (int, sum)

for each i,

result = sum(result, array[i])

Initially taking 1 processor for the sequential approach:-

Width = 1.

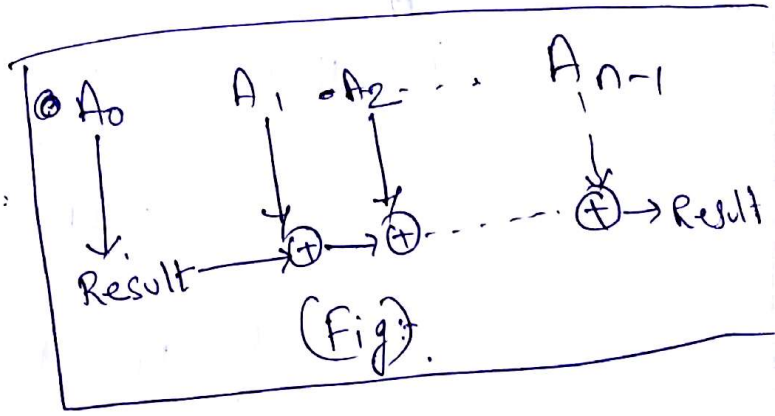
CP = n

Work = n.

[Assuming the sum function takes O(1) time]

Dependency :- RAW

Read-after-write.



(Fig).

b). Introducing Mutual Exclusion in this case does not help. as there is dependency between every iteration irrespective of the order. ~~Even if we do it in any order, the result is going to be the same.~~

c). Rewriting the code Assuming p processors.

```
int reduce(int *array, size_t n)
{
    // for Processor (Pi), i ∈ [1, n], where Pi = {0}.
        for (j = 2*(i-1); j<n; j=j+p) {
            Pi = Pi + (A[j] + A[j+1]);
        }
```

// Now, for these processors we repeat the process. $i \in (1, P/2)$ $j = j+2$

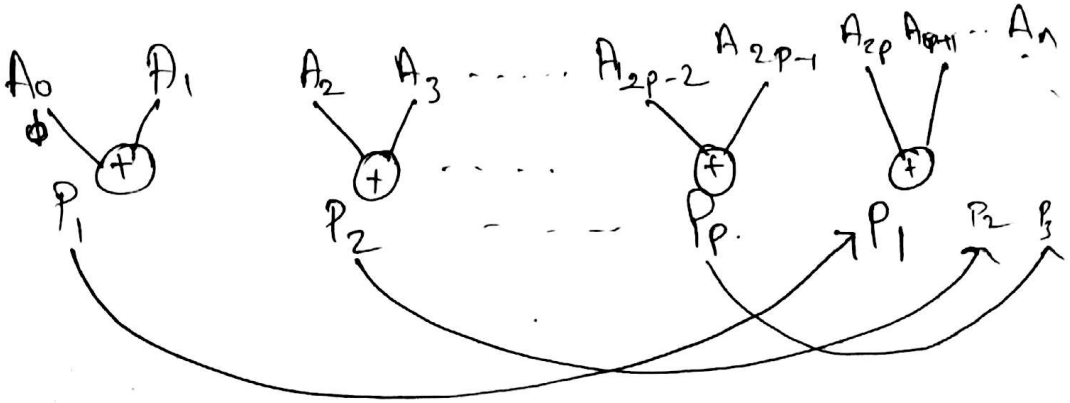
```
    for (j=0; j<= P; j++)
    {
        Pi = $sum(Pi, Pj+1)
    }
```

// and so on for $i \in (1, P/4)$, then $P/8$, $P/16$ ....

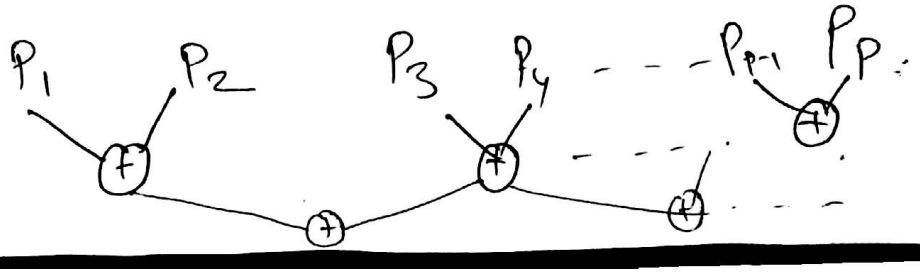
```
}
```

Dependency:

RAW!



Now, where

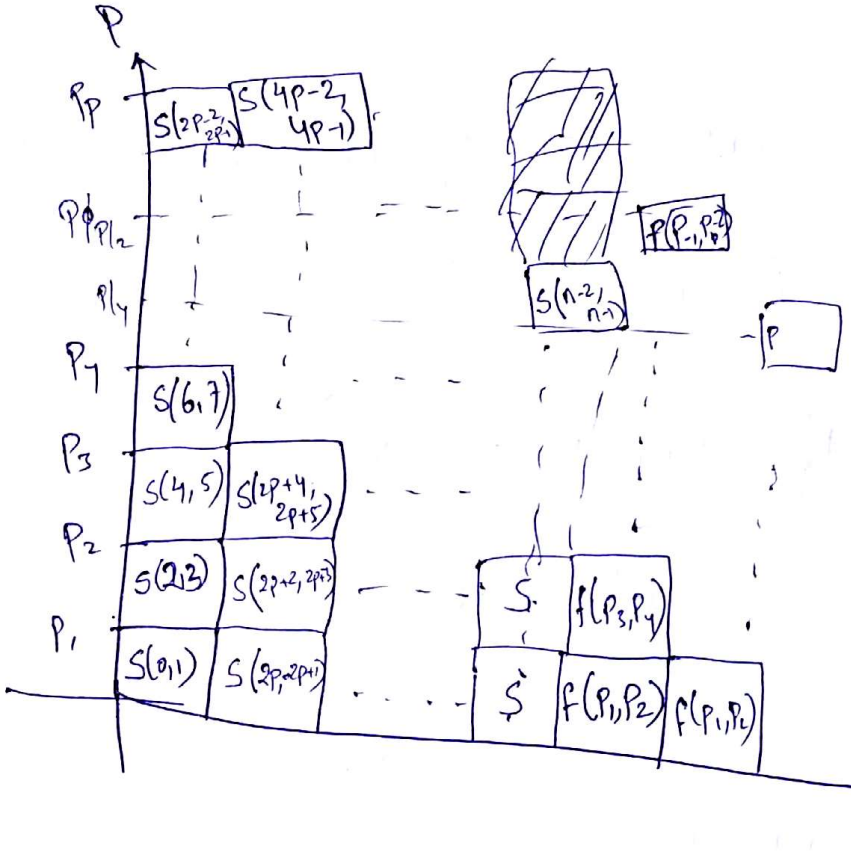


Width = P

C·P = $\log_2 n$ → ~~As we do~~ After the first step, we are adding 2 elements in a Processors.

Work = n-1

d)



(S(i,j) denotes the sum of i^{th} & j^{th} index of Given Array)

&

f(Pi, Pj) denotes the sum of local variables of Processor i & j.

(2.2) a) For int, max, it's the same case as (int, sum) as the order doesn't matter. Hence, Yes, it is correct for both Mutual Exclusions and local variable

b) For (String, Concat) the order of Concatenation matters. Hence, there will be no help with Mutual Exclusion. but parallel version with local variable is correct if we keep the order same.

c) float, sum is exact case as int, sum.
Hence correct for both.

d) float, max is exact same case as int, max.
Hence, correct for both

3) Find first.

3.1 a).

Algo:- for an array 'arr' of n elements, and 'val' be the search element

```
for (i=0; i<n; i++)
{
    if (arr[i] == val)
    {
        return i;
    }
}
return n;
```

(Here 'i' is 'pos')

b) The above Algorithm has a Complexity of O(pos). irrespective of n.

But if the element is not found in the array, then

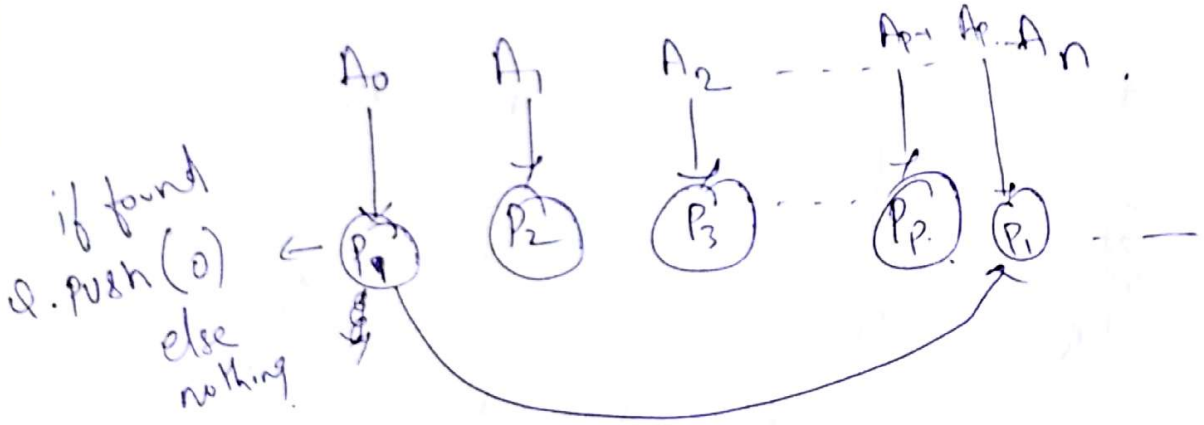
Complexity :- O(n).

c) for O(n) work, with ~~we take~~ p processors,

we take first p elements and then search ~~in to~~ 1 element assigned to each processor, and record the value (found, not found).

if found, we stored the index of the element into a Queue.

Finally we pop out the queue, first element would be our required index.



Work :- O(n).

C.P :- (n/p) +1 , if n%p != 0, else (n/p).

width :- p.

d) We follow the same procedure as 'c' part, but when we find the first element that is found, we ~~return~~ exit the search process then and there and return the index of the element.

Work = $\theta(pos)$

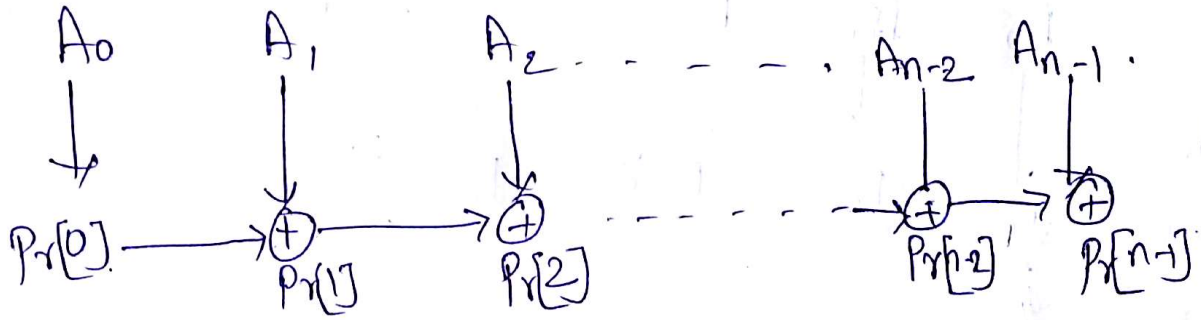
C.P = $\left((pos)/p + 1\right)$ if pos%p != 0, else $(pos)/p$.

Width :- p.

3.2. Since Linked list is a sequential data structure where you cannot jump from 1 index to another, without traversing. Hence, No parallelism is possible in this case.

4)

a)



→ Read After Write (RAW) Dependency.

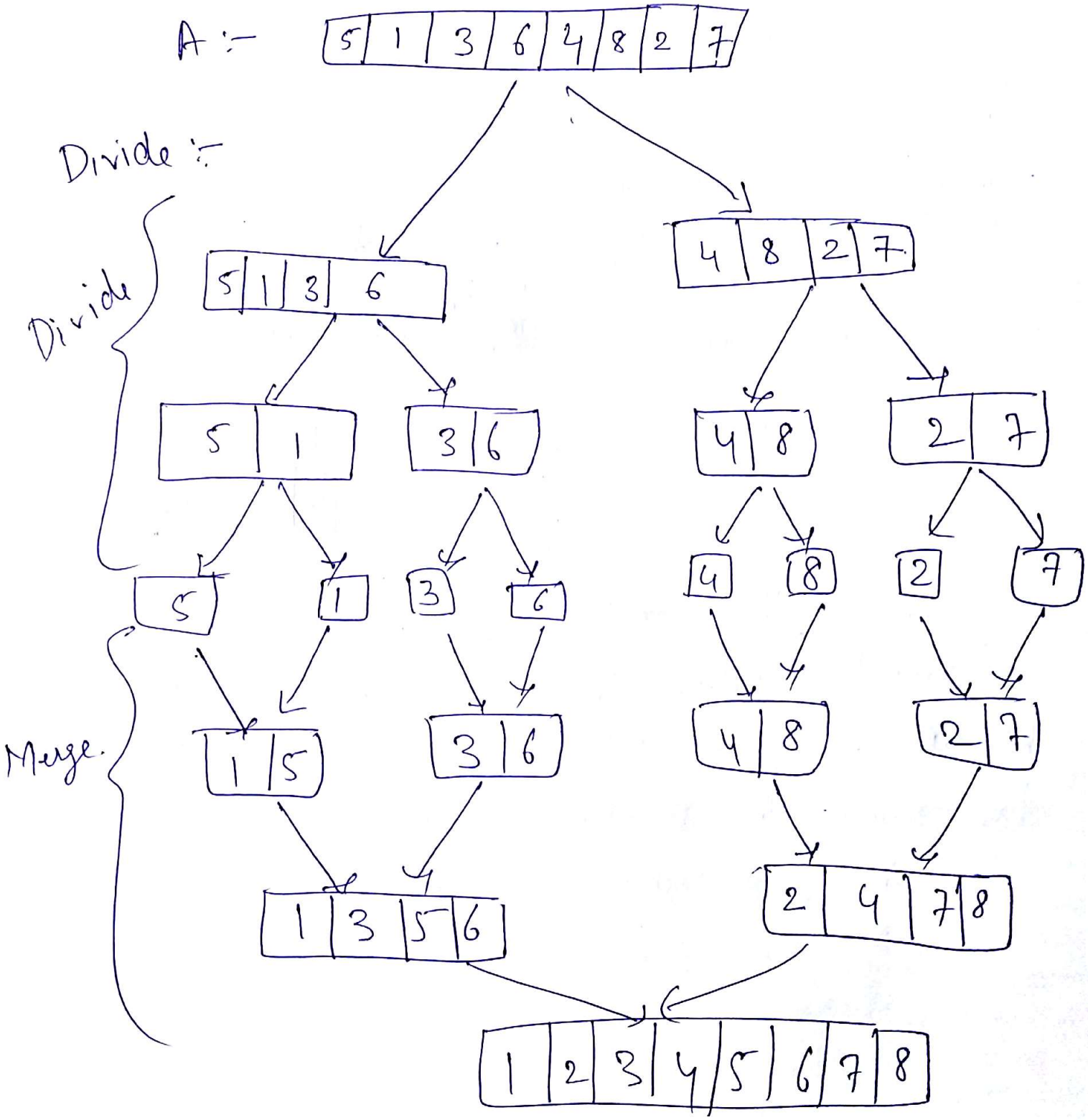
** NOTE :- 4(b) written at the end.

5)

a) Merge Sort is divided into 2 parts :-

Divide → & then Merge.

Lets explain Merge Sort through an example



Algorithm wise:- (Recursive).

n → total no. of elements
A → Given Array.
P → first element
r → Last element.

```
Merge_sort (A, p, r)
{
    if (p < r)
    {
        q = (p+r)/2
        Merge_sort (A, p, q)
        Merge_sort (A, q+1, r)
        Merge (A, p, q, r)
    }
}
```

divide part

Merge → (Each stage).

Initially:- p = 0, r = n-1

b).

→ Dependency here can be on the levels of the time of Divide & Merge. We cannot ~~complete~~ compute of level until the previous level is finished. So, find of RAW dependency

→ C.P is the longest dependency route.

n numbers → divides into $\log_2 n$ levels and then ~~builds~~ Meges in another $\log_2 n$ levels.

So, $C.P = 2 . \log_2 n$

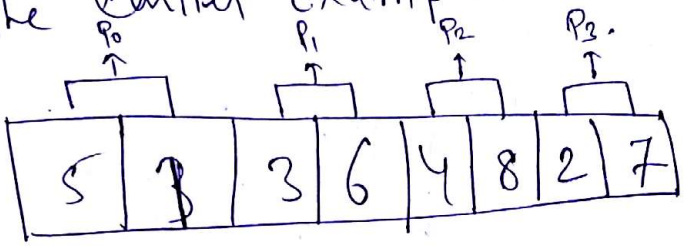
→ Work = n-1

width = P (upto $P = n/2$, then it should be constant $n/2$).

c) for a parallel Algorithm with p=4.

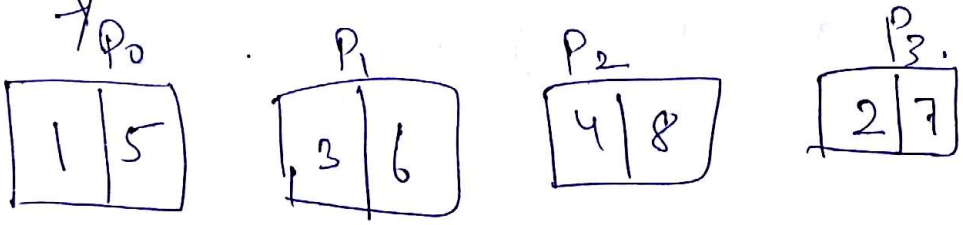
first we divide the array into 4 parts. for each processor.

Taking the earlier Example:-



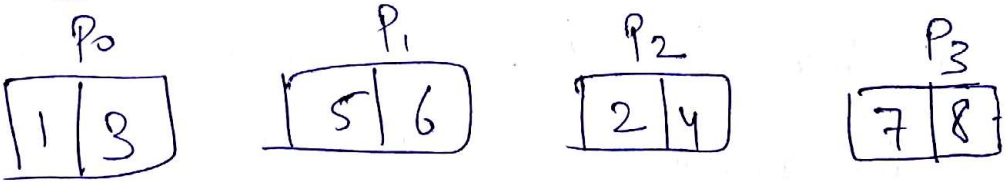
→ First we sort the individual parts locally using sequential Merge sort.

Step - 1



Sort ~~Merge~~ (P_0, P_1) and (P_2, P_3).

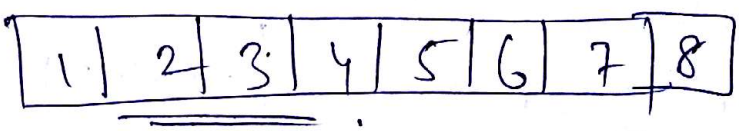
Step - 2 →



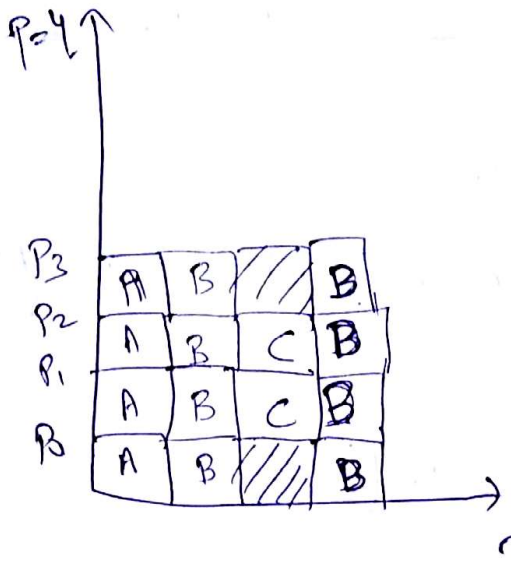
Sort (P_1, P_2)

Step - 3.

Repeat Step 2. & Merge :-



Schedule :-



Array

A :- divides into 4 parts.

B :- Sorting (P_0, P_1) & (P_2, P_3).

C :- Sorting (P_1, P_2)

~~And the~~

(Rough Diagram).

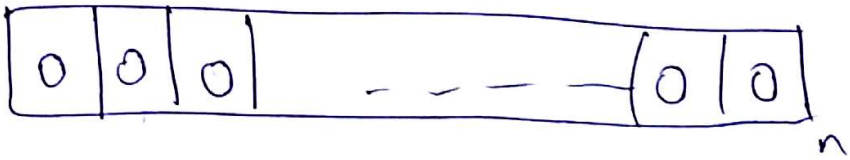
d) following the same process as 'c' as we increase, the time reduces.

~~This~~ Since, all the steps parallelly are in the linear form, The time Complexity should be of the order $O(n)$. ~~for sufficiently large~~

4(b). Let the Array be like

To explain the parallel algorithm, ~~let us take~~ ~~Processors~~ we take an empty array B of size n and initialize it to 0.

B



Now, we take n processors.

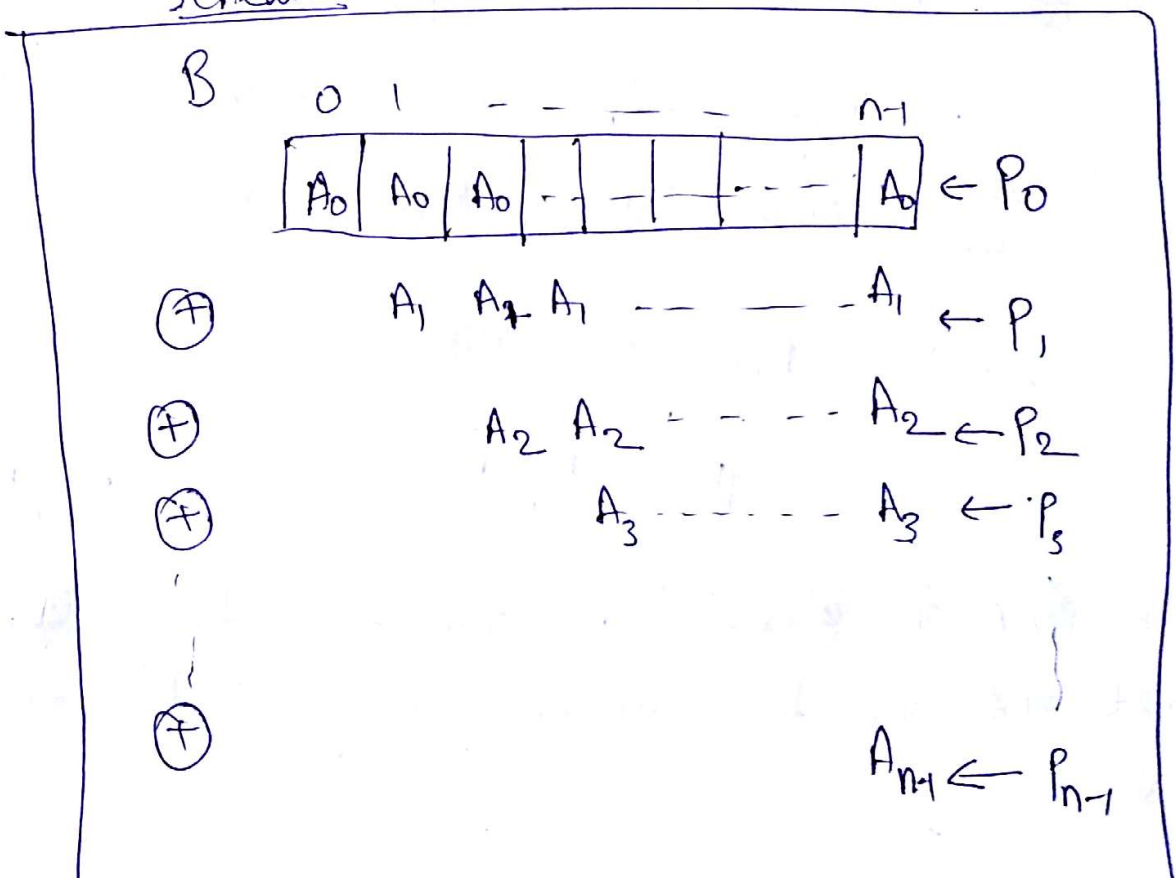
and Each of the processor will undergo the following algorithm:-

for any ~~Process~~ P_i $[i \in [0, n-1]]$

```
for(j = n-1; j >= i; j--)
{
  B[j] = B[j] + A[i];
}
```

which will look like:-

Schedule:-



Note:- We can also take P processors for this process and repeat the process until it reaches to 'n'.