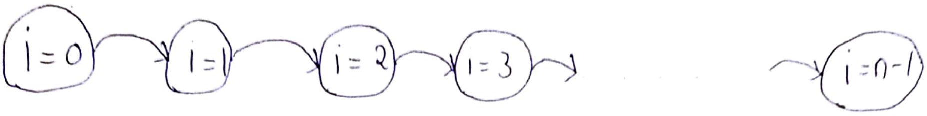
Parallel Computing
Assignment - 2 (main)

Prasanth Reddy Pallerla
ppallerl@uncc.edu
801033619

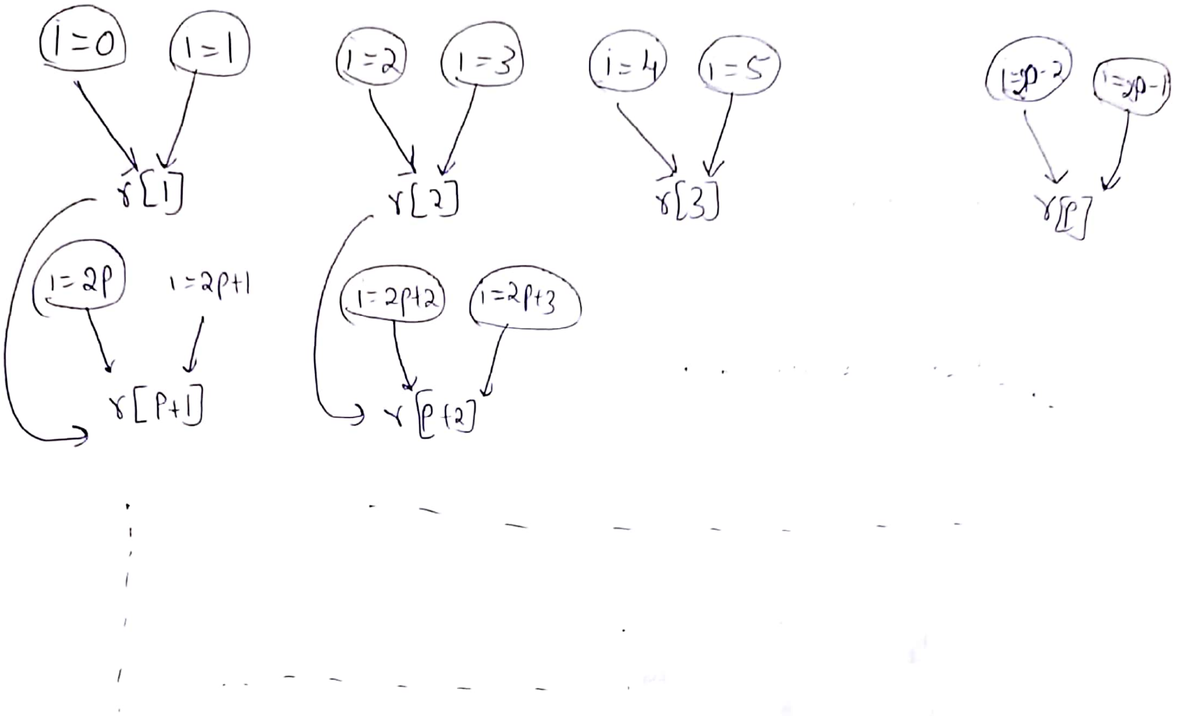
3.1 a)



Work = $\Theta(n)$
width = 1
critical path = $\Theta(n)$

b) ~~template <typename T, typename op>~~

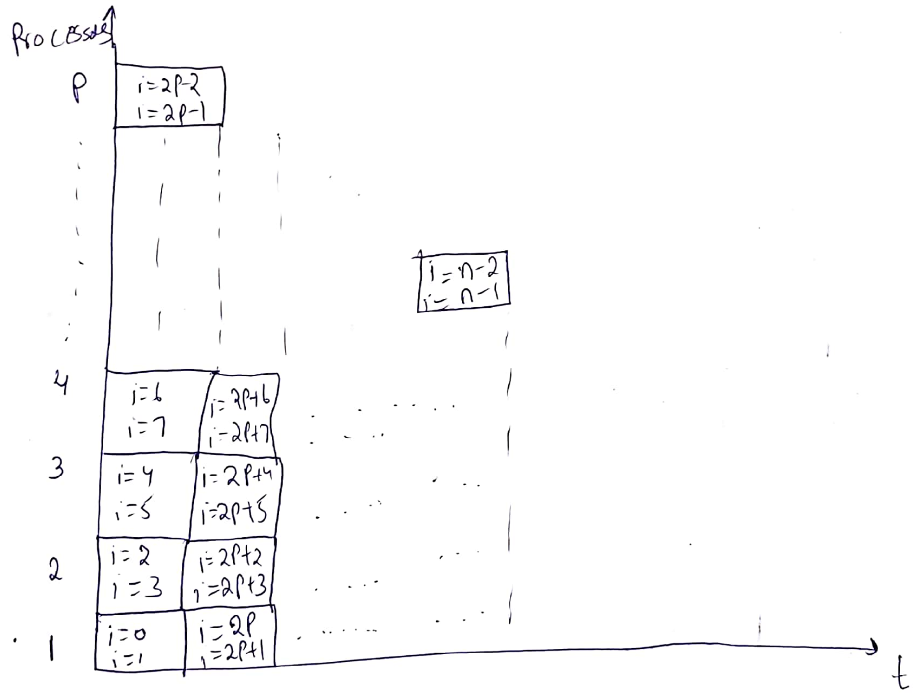
```
template < int T, sum op>
 T reduce (T* array , size_t n) {
  T r[p], result;

 if (p<n)
 {  for (i=1 ; i<=p; i++)
     r[i] = op (array[2i-2], array[2i-1]);

     int count = 1;

    for (i=p+1 ; i<=n-1 ;i++)
    { r[i - count*p] = op (r[i-count*p], array[2i-2], array[2i-1]);
       count ++;
    }

   result = ~~r[0]~~ r[1]+r[2] + .... r[p] ;
 }

 if (p>=n)
 {
    for(i=1 ; i<=n-1; i++)
      r[i] = op (array [2i-2], array [2i-1]);
     result = r[1] + r[2] + .... r[p];
  }
```



Work $= \Theta(n)$ width $= p$ critical path $= \Theta(1+1+2(n/p)+p-1)$

$= \Theta(2(n/p)+p+1)$

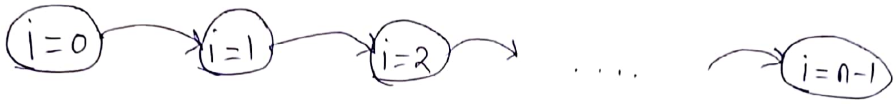
c)



3.2 Variants

The parallel version should work for all of the given variants max, concat, sum and any data type int, string, float. As the operation op takes two inputs every time which are independent pair and produces the output.

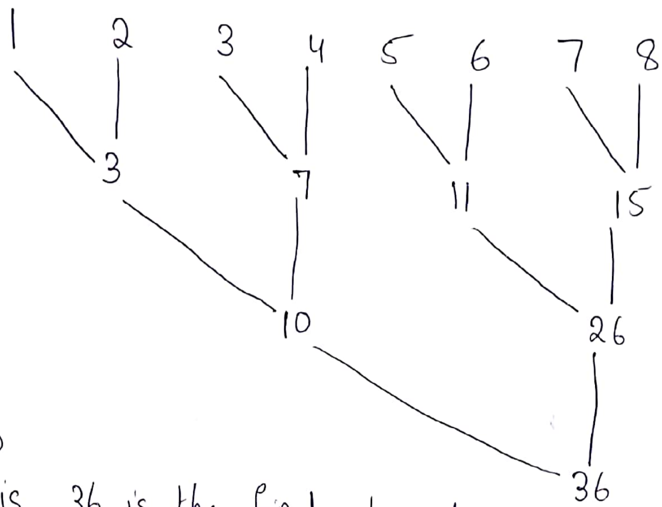
4 a) Prefix Sum



Work = $\Theta(n)$ width = 1 critical path = $O(n)$

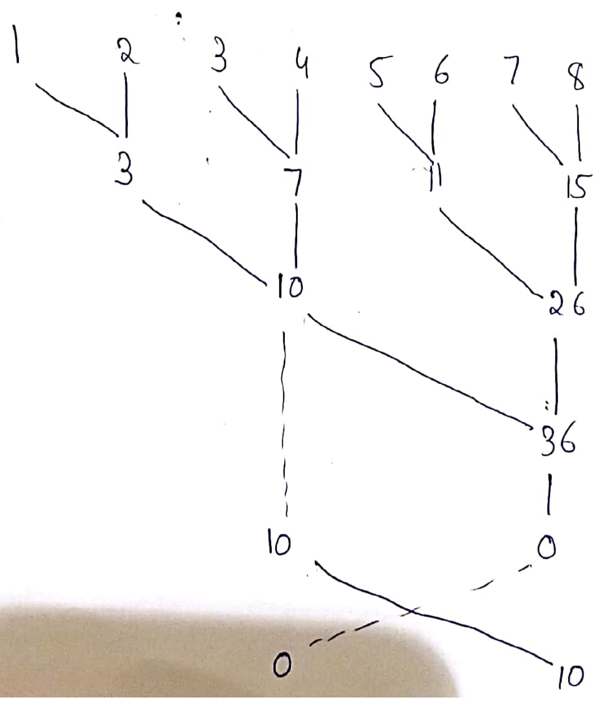
b) The parallel algorithm for prefix sum consists of two stages i) Reduction and ii) Down-Sweep

Reduction

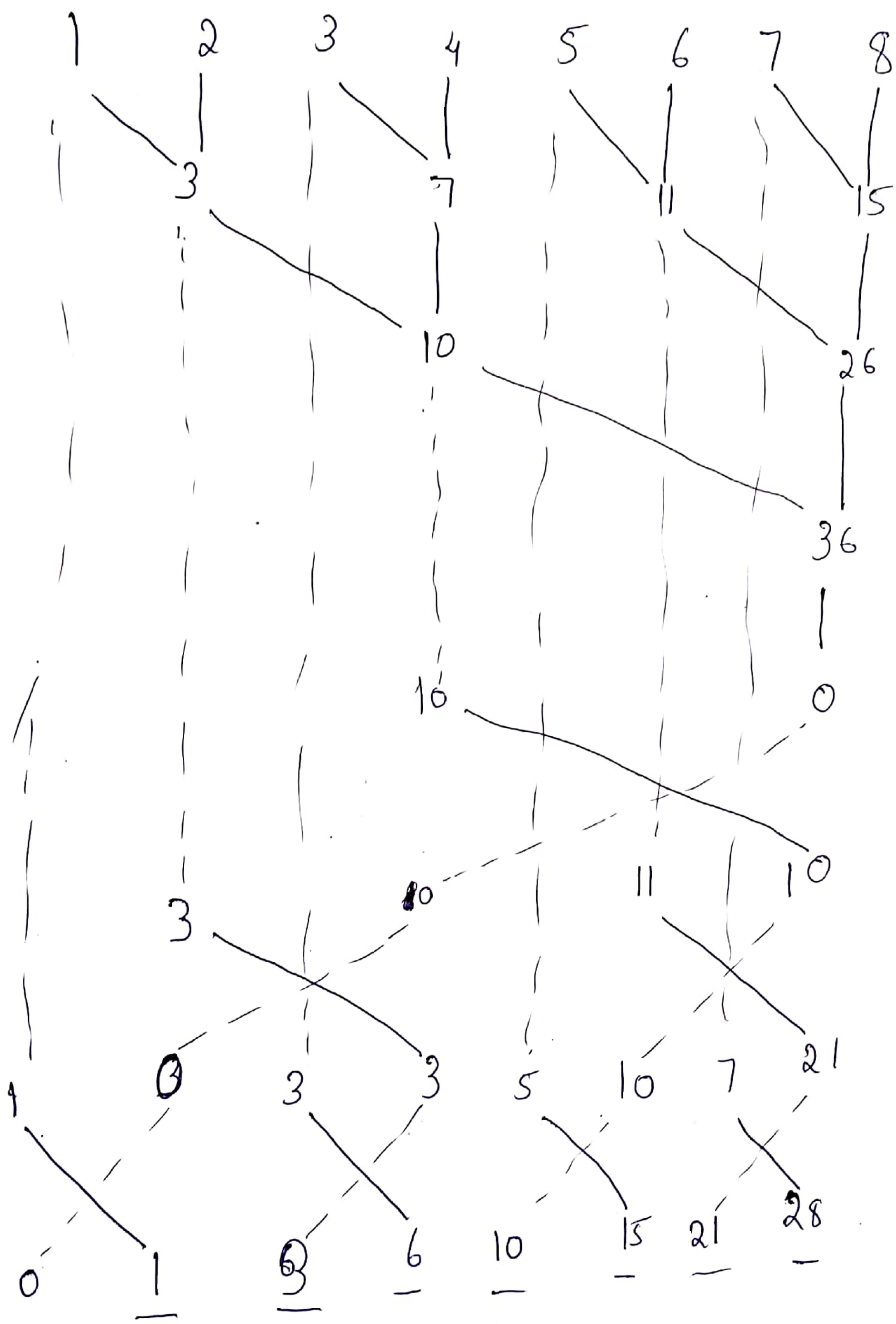


Down-sweep

Now this 36 is the final element of the prefix sum array. It is replaced with 0 to find the intermediate results (remaining elements of the array.

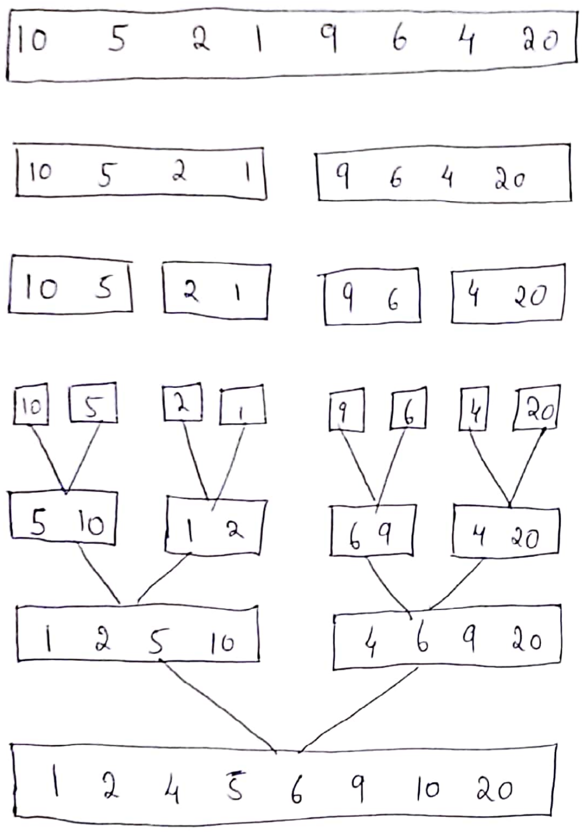


We will drag down each intermediate result and calculate the sum and push the sum to right side and the push the previous right element to leftside. By continuing the process we will obtain the final result.



So the prefix sum array is {1, 3, 6, 10, 15, 21, 28, 36}

5) a)

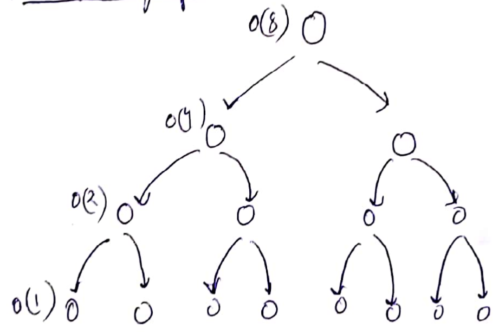


$mid = \frac{0+7}{2} = 3.5 \simeq 3$

```
void mergesort(int a[], int l, int r)
{  if(l<r)
   { mid = (l+r)/2;
     mergesort(a, l, mid);
     mergesort(a, mid+1, r);
     merge(a, l, mid, r);
   }
}
```

b) As the various calls to mergesort function as different sizes of the array, the processing time of each call is different. Specifically it decreases with the each call.

Dependency graph



As the last call to mergesort function takes $O(1)$, the previous one takes $O(2)$, $O(4)$ .... $O(n)$

work = $O(n \log n)$

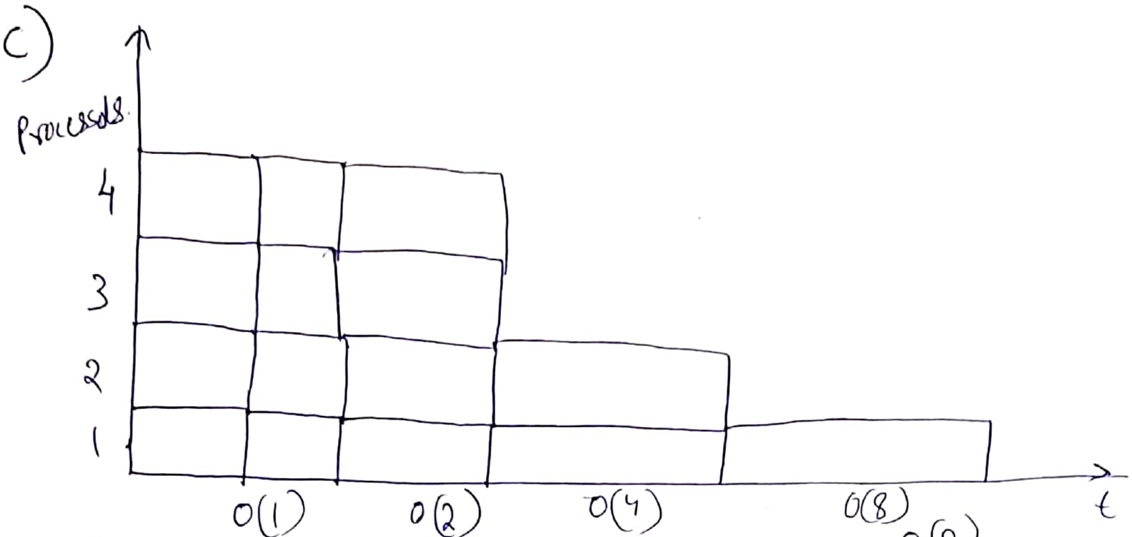
width = n

critical path = $O(1) + O(2) + O(4) + \dots + O(n)$

= ~~[illegible]~~

$= O([illegible]n) = O(n)$

c)



The ~~processor~~ schedule of the algorithm on 4 processors is shown above by considering the example in 5 a)

d) In the above sequential algorithm the first call to above merge sort takes O(n) time., which is the highest time taken call to the function. we will try to do this in parallel. This function call divides the array into two arrays of size $N/2$. each. we can sort them by picking one element of first array and doing binary search with second array for finding its position. This process is done repeatedly with all the elements of first array.

Each search takes O(logn) time. And each search is independent with another search. so we can take n processors to search the n elements independently. Hence the total time take is O(nlogn). Hence the time is increased from O(n)