Example: Prefix Sum

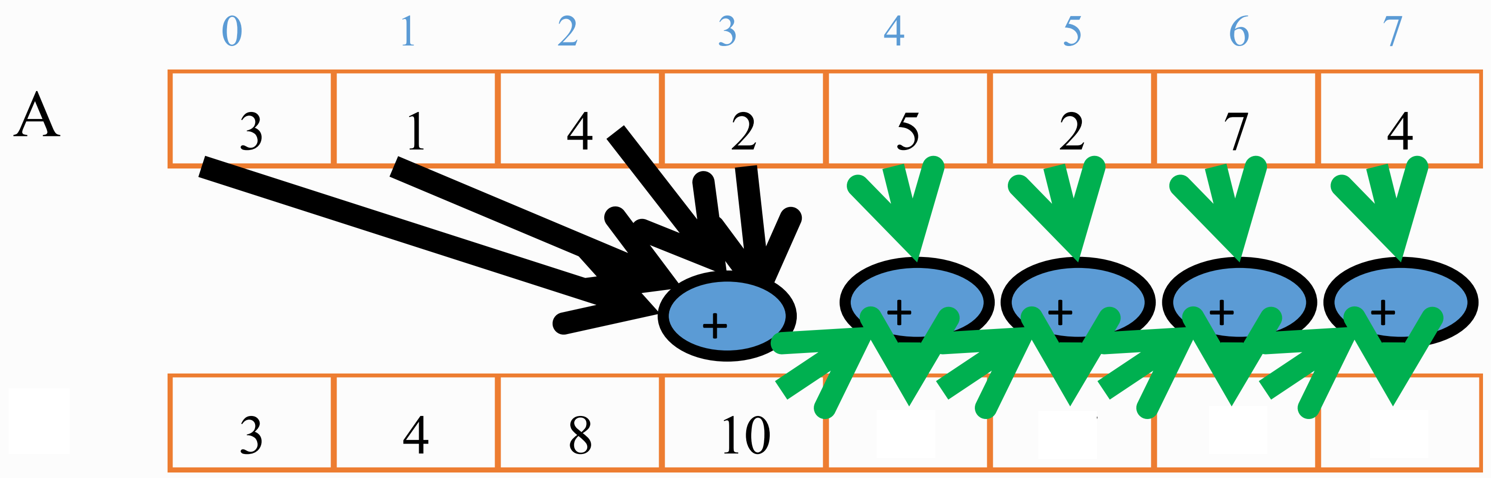
Recursive Doubling with Barriers

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

© 2018 L. V. Kale at the University of Illinois Urbana

Prefix Sum Problem

- Given array A[0..N-1], produce B[N], such that B[k] is the sum of all elements of A up to A[k]



B[3] is the sum of A[0], A[1], A[2], A[3]
But B[3] can also be calculated as B[2]+ A[3]

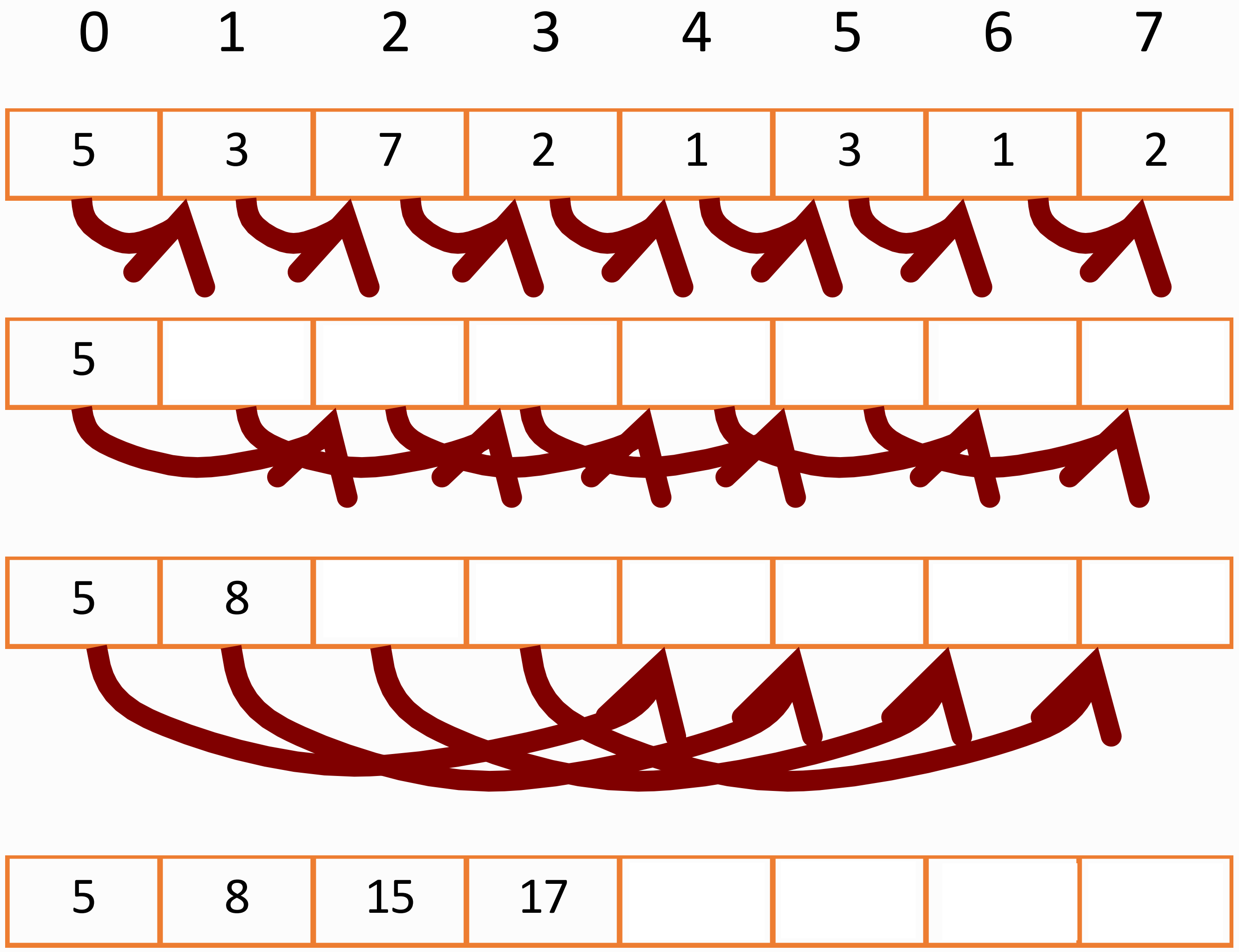
Prefix Sum: A good Sequential Algorithm

- Data dependency from iteration to iteration
    - How can this be parallelized at all?

```
B[0] = A[0];
for (i=1; i<N; i++)
  B[i] = B[i-1] + A[i];
```

- It looks like the problem is inherently sequential, but theoreticians came up with a beautiful algorithm called recursive doubling or just parallel prefix

Parallel Prefix: Recursive Doubling



N Data Items

P Processors

N=P

Log P Phases

P additions in each phase

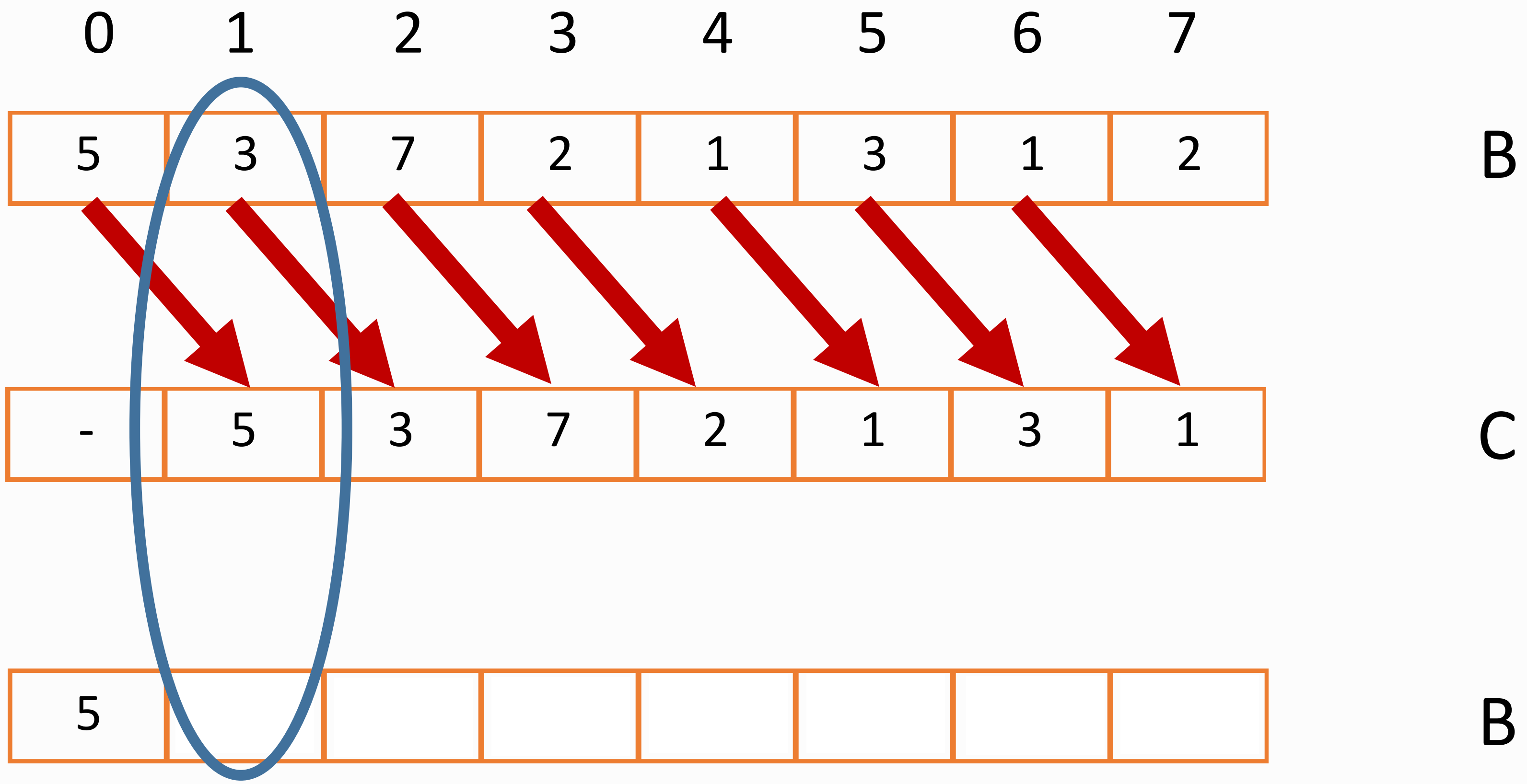
P log P operations

Completes in O(logP) time

OpenMP Formulation for Parallel Prefix

- We don't have n processors
  - I.e., the number of threads will be much smaller than the size of the data array
- So, we will simulate n processors using p threads
- Notice that each phase of the computation must finish before the next phase of the computation starts
  - We will use OpenMP's **`barrier`** directive for that
- Basic description of actions in each phase with distance d
  - Every "processor" i adds its value to the value held by a processor distance d away
  - Simulation: `B[i+d] += B[i]`, but you have to be careful to avoid dependencies
    - I.e., copy `B[i]` into a temporary variable at `i+d`, say `C[i+d]` and then add `C[i]` to `B[i]` for every i
  - Note d doubles in every phase

Parallel Prefix: Recursive Doubling



```
…
#pragma omp parallel for
     for(i=0;i<n;i++){B[i]=A[i];}
     int d=1;
     while(d<n)  // this loop will run for lg n steps
     {
          int i;
          #pragma omp parallel for
          for(i=d;i<n;i++)C[i]=B[i-d];

          #pragma omp parallel for
          for(i=d;i<n;i++)B[i]+=C[i];


          d*=2;
      }
…
```

Initialize B with values from A

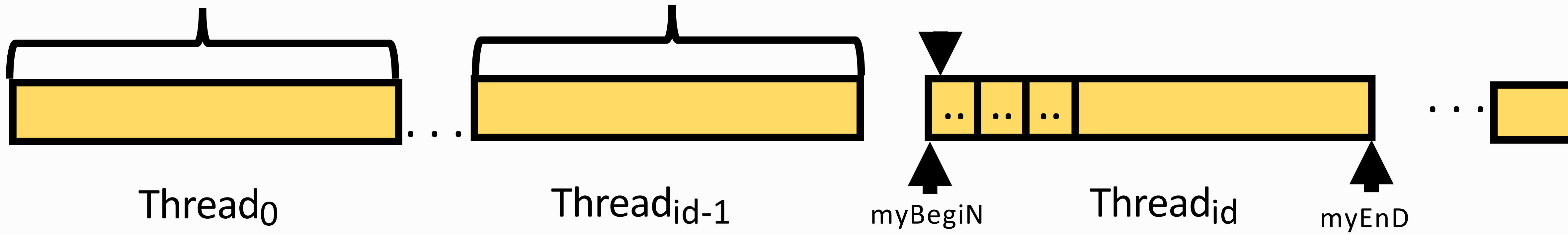
C[k] temporarily stores the value that we want to add to B[k]

Critique of Prefix Algorithm 1

- The sequential algorithm had n additions
- But the parallel algorithm is doing a total of n*(log n) additions
  - Although they are parallelized by p threads
  - This is an example of an algorithm that is not “work efficient”
- It uses log n barriers, which are expensive operations
- Maybe a thread oriented approach will avoid the log n factors

Prefix Sum Algorithm 2: A Thread Oriented Approach

- What if we let each thread calculate prefix sum over its own range of array?
  - I.e., thread id is responsible for range B$\left[\frac{\mathrm{n*id}}{\mathrm{p}} : \frac{n*(\mathrm{id}+1)}{\mathrm{p}} - 1\right]$
  - Id : my thread's serial number; p : total number of threads
  - Assuming n is a multiple of p
- But then each thread needs the sum of all numbers to its left

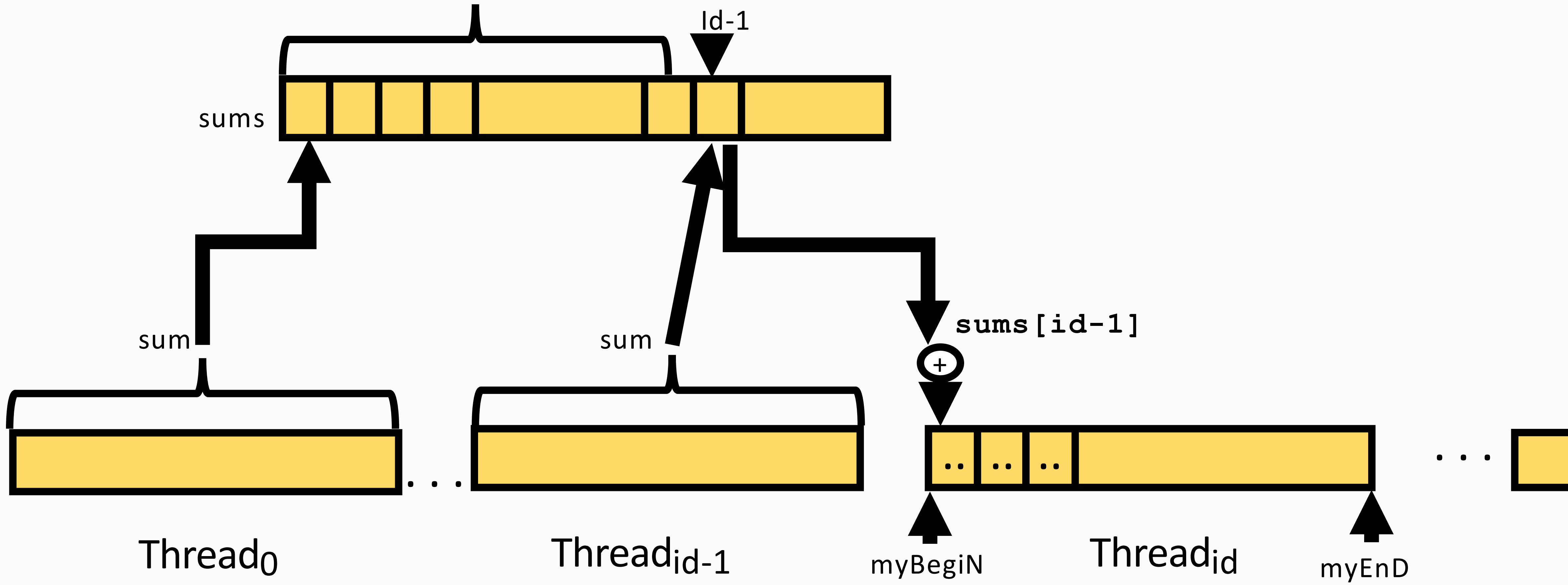

Thread$_0$
Thread$_{id-1}$
myBegiN
Thread$_{id}$
myEnD

Prefix Sum Algorithm 2: A Thread Oriented Approach

- What if we let each thread calculate prefix sum over its own range of array?
  - I.e., thread i is responsible for range B$[\frac{n*id}{p} : \frac{n*(id+1)}{p} - 1]$

  *id – my thread's serial number; *p – total number of threads

  Assuming n is a multiple of p
- But then each thread needs the sum of all numbers to its left
- If we are willing to double the amount of work, we can obtain this sum with a much smaller prefix sum problem of size p
  1. First loop: every thread calculates sum s over its sub-range and copies s into a shared array called sums at **`sums[id]`**
  2. Calculate prefix sum of the sums array
     - **`sums[id-1]`** has the sum of all values to the left of thread numbered id
  3. Second loop: every thread with serial number id calculates the prefix sum in array B using **`sums[id-1]`** and the values in A


Id-1
sums
sum
sum
sums[id-1]
+
..
..
..
. . .
. . .
Thread0
Threadid-1
myBegiN
Threadid
myEnD

```
…
omp_set_num_threads(p);

#pragma omp parallel
{
    int id=omp_get_thread_num();
    int myBegiN = (n*id)/p;
    int myEnD = min( (n*id+1)/p, n);


    int sum=0;
    for(int i=myBegiN;i<myEnD;i++)
        sum+=B[i];
    sums[id]=sum;
    #pragma omp barrier
    #pragma omp single
    {
      for(int i=1;i<p;i++)
          sums[i]+=sums[i-1];
    }
    if(id>0)B[myBegiN]+= sums[id-1]
    for(int i=myBegiN+1;i<myEnD/p;i++)
       B[i]+=B[i-1];
}
```

Form Local sum

- Calculate Prefix sum of size p
- Sums [id] now contains the sum of values of all previous threads' ranges
- This can be done in parallel but it's not worth it

Complete the Prefix sum