Cilk and Cilk++
Lecture 3

- Merge sort
- Prefix sums

Parallel Merge Sort


```
Merge-Sort ( A, p, r ) {      // sort the elements in A[ p ... r ]

    if p < r then
        q ← ⌊ ( p + r ) / 2 ⌋
        Merge-Sort ( A, p, q )
        Merge-Sort ( A, q + 1, r )
        Merge ( A, p, q, r )
}
```



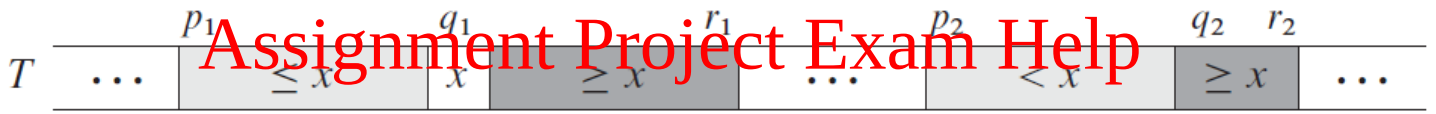
```
Par-Merge-Sort ( A, p, r ) {      // sort the elements in A[ p ... r ]

    if p < r then
        q ← ⌊ ( p + r ) / 2 ⌋
        cilk_spawn Par-Merge-Sort ( A, p, q )
                   Par-Merge-Sort ( A, q + 1, r )
        cilk_sync
        Merge ( A, p, q, r )      // time bottleneck
}
```

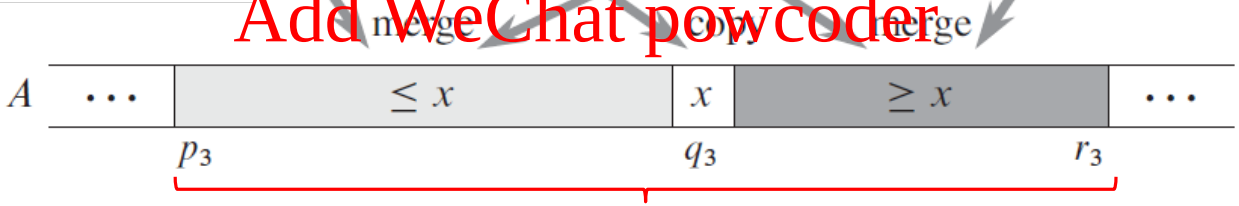
Parallel Merge

$n_1 = r_1 - p_1 + 1$ $\quad$ $n_2 = r_2 - p_2 + 1$

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



suppose: $n_1 \geq n_2$



merged output: $A[p_3..r_3]$

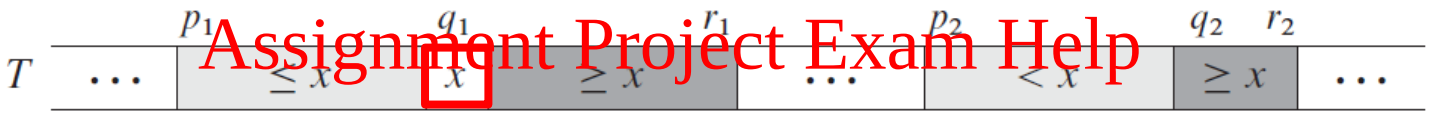
$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

Next we will describe each step of this algorithm in more detail

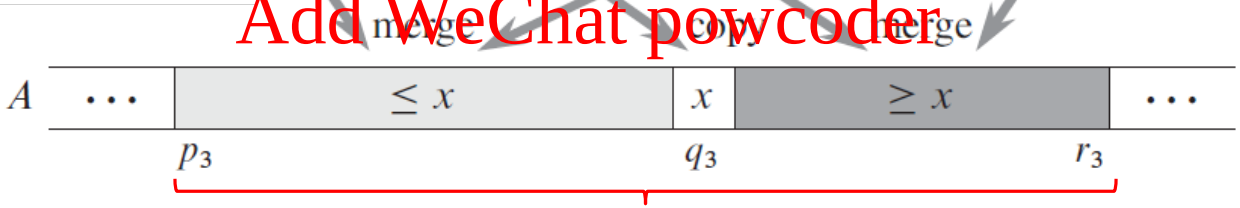
Parallel Merge

$$n_1 = r_1 - p_1 + 1 \qquad n_2 = r_2 - p_2 + 1$$

subarrays to merge: $T[p_1..r_1]$ $T[p_2..r_2]$



suppose: $n_1 \geq n_2$



merged output: $A[p_3..r_3]$

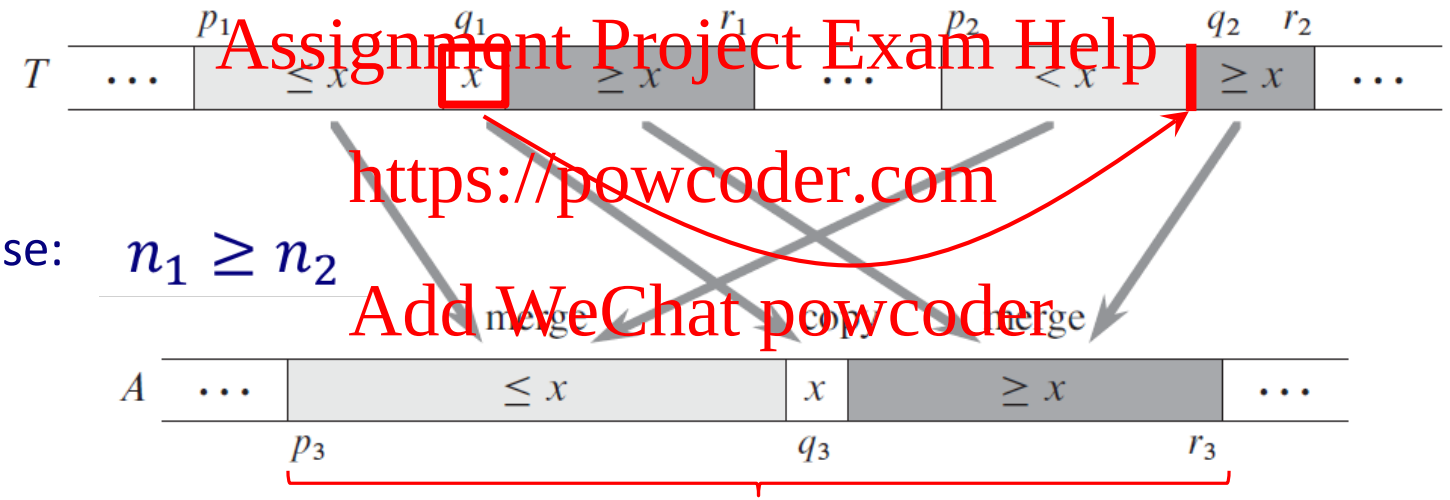
$$n_3 = r_3 - p_3 + 1 = n_1 + n_2$$

Step 1: Find $x = T[q_1]$, where q_1 is the midpoint of $T[p_1..r_1]$

Parallel Merge

$n_1 = r_1 - p_1 + 1$ $\quad$ $n_2 = r_2 - p_2 + 1$

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



suppose: $n_1 \geq n_2$

merged output: $A[p_3..r_3]$

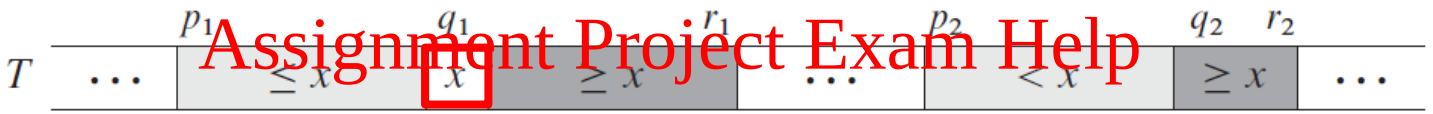
$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

Step 2: Use binary search to find the index q_2 in subarray $T[p_2..r_2]$ such that $T[q_2 - 1] < x \leq T[q_2]$

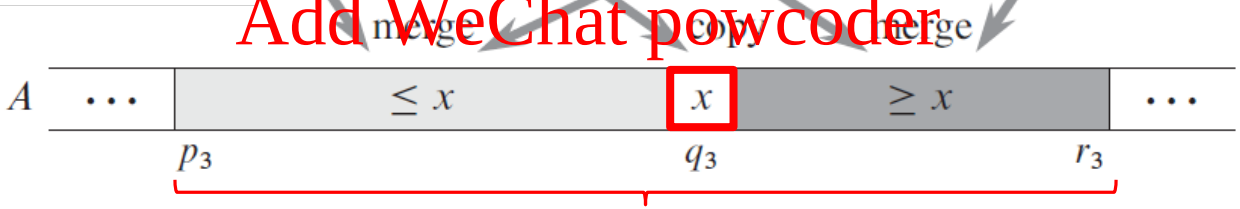
Parallel Merge

$$n_1 = r_1 - p_1 + 1 \qquad n_2 = r_2 - p_2 + 1$$

subarrays to merge: $T[p_1..r_1]$ $T[p_2..r_2]$



suppose: $n_1 \geq n_2$



merged output: $A[p_3..r_3]$

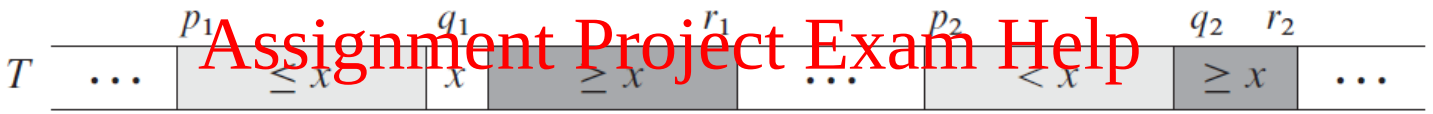
$$n_3 = r_3 - p_3 + 1 = n_1 + n_2$$

Step 3: Copy x to $A[q_3]$, where $q_3 = p_3 + (q_1 - p_1) + (q_2 - p_2)$

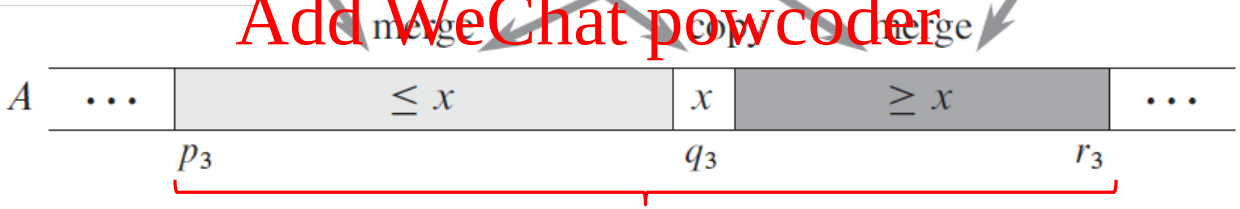
Parallel Merge

$n_1 = r_1 - p_1 + 1$ $\quad$ $n_2 = r_2 - p_2 + 1$

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$



suppose: $n_1 \geq n_2$



merged output: $A[p_3..r_3]$

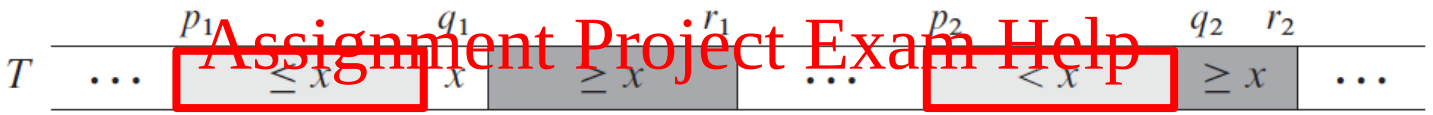
$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

Next we can do steps 4(a) and 4(b) in parallel

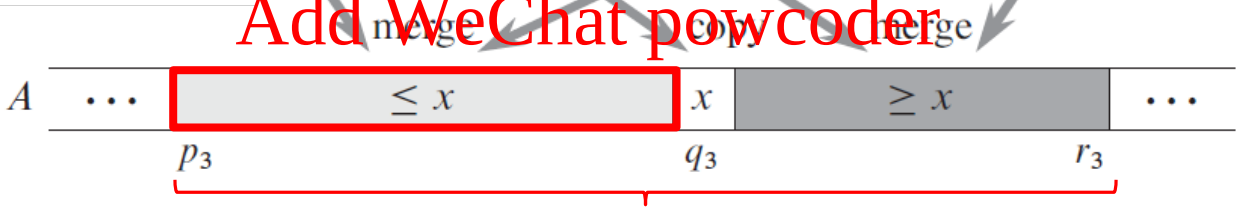
Parallel Merge

$n_1 = r_1 - p_1 + 1$ $n_2 = r_2 - p_2 + 1$

subarrays to merge: $T[p_1..r_1]$ $T[p_2..r_2]$



suppose: $n_1 \geq n_2$



merged output: $A[p_3..r_3]$

$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

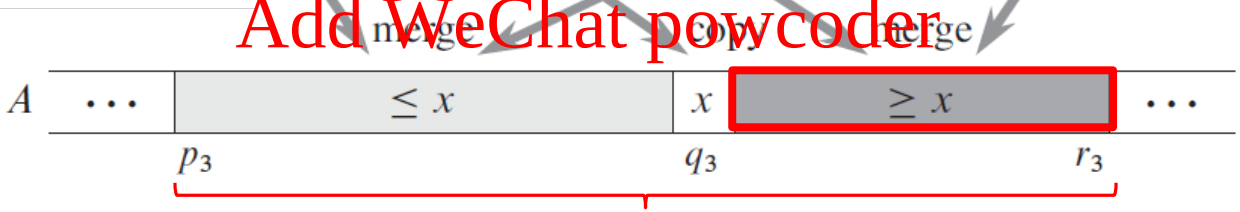
Step 4(a): Recursively merge $T[p_1..q_1 - 1]$ with $T[p_2..q_2 - 1]$, and place the result into $A[p_3..q_3 - 1]$

Parallel Merge

$n_1 = r_1 - p_1 + 1$ $\quad$ $n_2 = r_2 - p_2 + 1$

subarrays to merge: $T[p_1..r_1]$ $\quad$ $T[p_2..r_2]$

suppose: $n_1 \geq n_2$



merged output: $A[p_3..r_3]$

$n_3 = r_3 - p_3 + 1 = n_1 + n_2$

Step 4(b): Recursively merge $T[q_1 + 1..r_1]$ with $T[q_2..r_2]$, and place the result into $A[q_3 + 1..r_3]$

Parallel Merge


```
Par-Merge ( T, p1, r1, p2, r2, A, p3 ) {
    n1 ← r1 - p1 + 1,   n2 ← r2 - p2 + 1
    if n1 < n2 then
        p1 ↔ p2,  r1 ↔ r2,  n1 ↔ n2
    if n1 = 0 then return
    else
        q1 ← ⌊ (p1 + r1) / 2 ⌋
        q2 ← Binary-Search ( T[ q1 ],  T,  p2, r2 )
        q3 ← p3 + ( q1 - p1 ) + ( q2 - p2 )
        A[ q3 ] ← T[ q1 ]
        cilk_spawn Par-Merge ( T, p1, q1 - 1, p2, q2 - 1, A, p3 )
                   Par-Merge ( T, q1 + 1, r1, q2, r2, A, q3 + 1 )
        cilk_sync
}
```

Parallel Merge Sort with Parallel Merge


```
Par-Merge-Sort ( A, p, r ) {    // sort the elements in A[ p ... r ]
  if p < r then
    q ← ⌊ ( p + r ) / 2 ⌋
    cilk_spawn Par-Merge-Sort ( A, p, q )
               Par-Merge-Sort ( A, q + 1, r )
    cilk_sync
    Par-Merge ( A, p, q, r )      // efficient now
}
```

parmergesort.cpp


```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cilk/cilk.h>
using namespace std;

int BinarySearch (int x, int T[ ], int low, int high) {
        if (low>high) return low;
        int mid=(low+high)/2;
        if (T[mid]==x) return mid;
        if (T[mid]<x) return BinarySearch (x, T, mid+1, high);
        return BinarySearch (x, T, low, mid-1);
}
```

parmergesort.cpp (continued)


```
void ParMerge (int T[ ], int p1, int r1, int p2, int r2, int A[ ], int p3) {
        int n1=r1-p1+1, n2=r2-p2+1;
        if (n1<n2) {
                int z=p1; p1=p2; p2=z;
                z=r1; r1=r2; r2=z;
                z=n1; n1=n2; n2=z;
        }
        if (n1==0) return;
        int q1=(p1+r1)/2;
        int q2=BinarySearch (T[q1], T, p2, r2);
        int q3=p3+(q1-p1)+(q2-p2);
        A[q3]=T[q1];
        cilk_spawn  ParMerge (T, p1, q1-1, p2, q2-1, A, p3);
                    ParMerge (T, q1+1, r1, q2, r2, A, q3+1);
        cilk_sync;
}
```

parmergesort.cpp (continued)


```
void ParMergeSort (int A[ ], int p, int r, int B[ ], int s) {
        int n=r-p+1;
        if (n==1) { B[s]=A[p]; return; }
        int *T = new int[n];
        int q=(p+r)/2;
        int q2=q-p+1;
        cilk_spawn ParMergeSort (A, p, q, T, 0);
                           ParMergeSort (A, q+1, r, T, q2);
        cilk_sync;
        ParMerge (T, 0, q2-1, q2, n-1, B, s);
        delete[ ] T;
}
```

parmergesort.cpp (continued)


```
int main (int argc, char *argv[ ]) {
        int size=atoi (argv[1]);
        int *A = new int[size];
        int *B = new int[size];
        time_t t;
        srand ((unsigned) time(&t));
        for (int k=0; k<size; k++)
                A[k]=rand( );
```

parmergesort.cpp (continued)


```
    clock_t start=clock( );
    ParMergeSort (A, 0, size-1, B, 0);
    clock_t finish=clock( );
    double duration=(double)(finish-start)/CLOCKS_PER_SEC;
    cout << "ParMergeSort took " << duration << " seconds" << endl;
    cilk_for (int k=1; k<size; k++)
        if (B[k-1]>B[k])
            cout << "Error in sort" << endl;
    return 0;
}
```

Compile and run


```
> cilk++  parmergesort.cpp  –o  parmergesort
> ./parmergesort  100000
ParMergeSort took 0.80716 seconds

> ./parmergesort  1000000
ParMergeSort took 3.32211 seconds

> ./parmergesort  10000000
ParMergeSort took 32.1453 seconds
        (about 1.5 seconds in real time)

> ./mergesort  10000000
MergeSort took 9.32856 seconds
```

Prefix sums

Recall in Haskell:

foldl (+) 0 [5,3,7,1,3,6,2,4] ⇒ 31

scanl (+) 0 [5,3,7,1,3,6,2,4] ⇒ [0,5,8,15,16,19,25,27,31]

foldl1 (+) [5,3,7,1,3,6,2,4] ⇒ 31

scanl1 (+) [5,3,7,1,3,6,2,4] ⇒ [5,8,15,16,19,25,27,31]

Prefix sums is just another name for scanl1

Parallel Prefix Sums

Input: A sequence of n elements $\{x_1, x_2, \ldots, x_n\}$ drawn from a set S with a binary associative operation, denoted by $\oplus$.

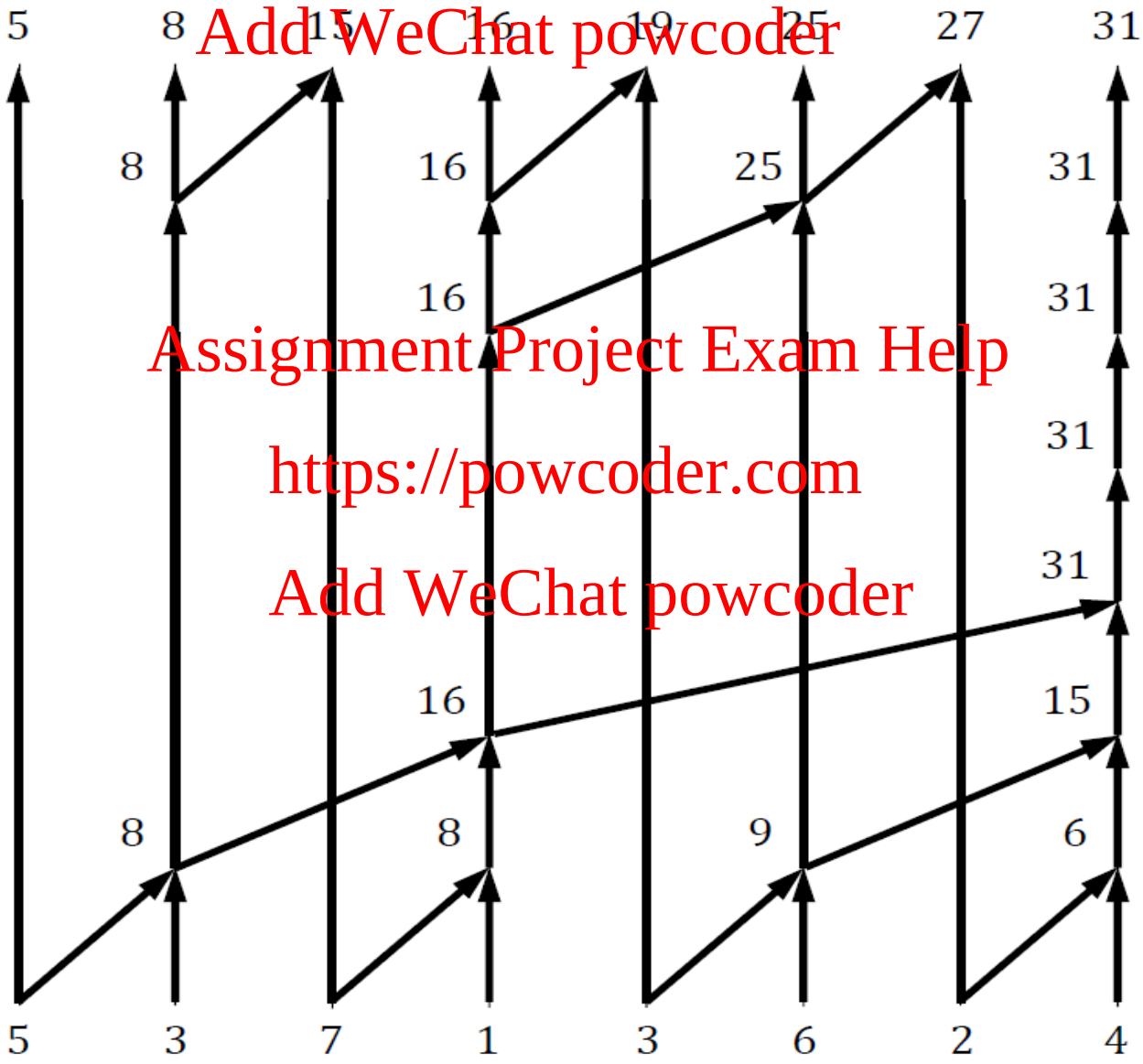
Output: A sequence of n partial sums $\{s_1, s_2, \ldots, s_n\}$, where $s_i = x_1 \oplus x_2 \oplus \ldots \oplus x_i$ for $1 \leq i \leq n$.

x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8
5	3	7	1	3	6	2	4

$\oplus$ = **binary addition**

5	8	15	16	19	25	27	31
s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8

Parallel Prefix Sums

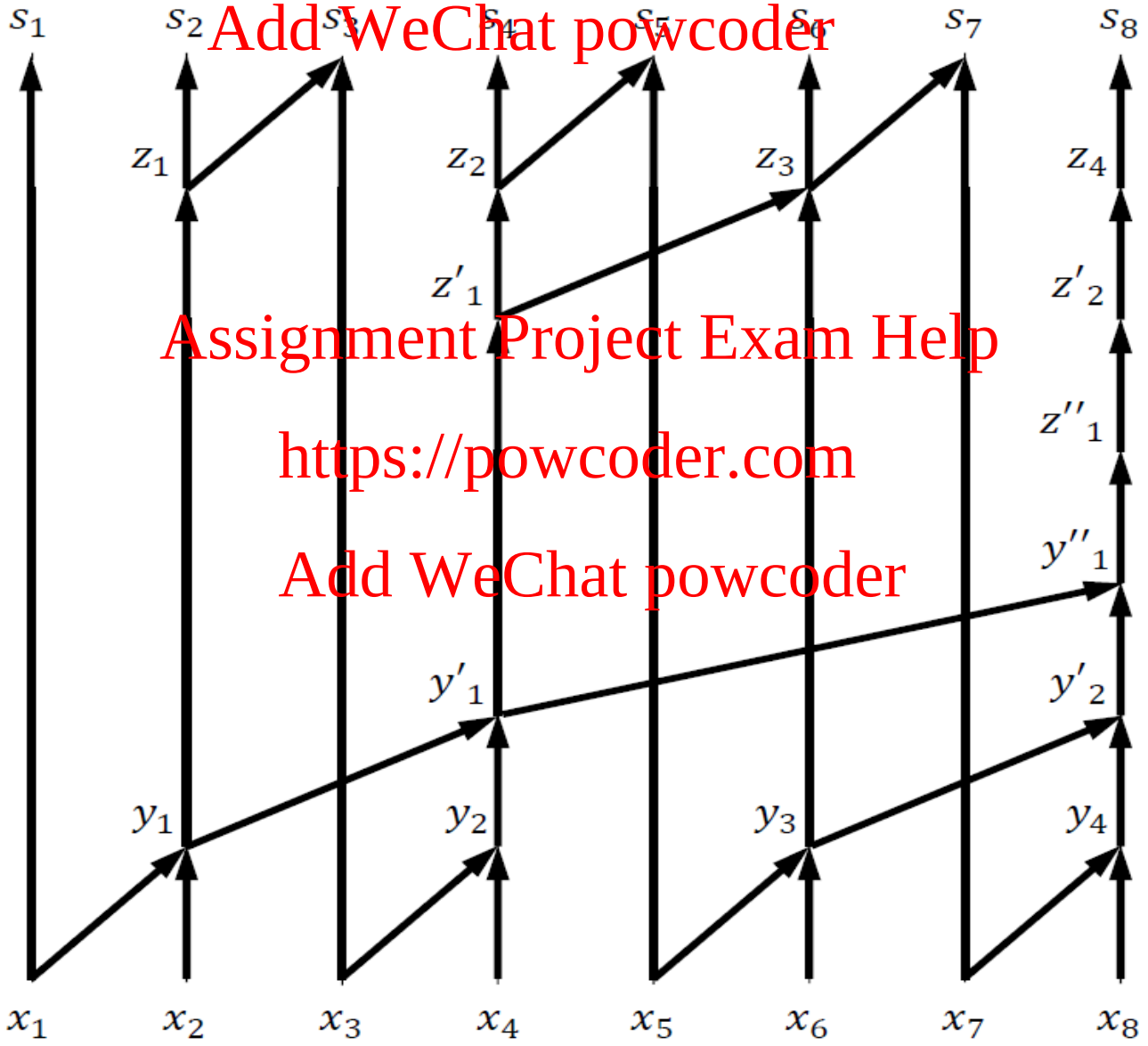


Parallel Prefix Sums


```
Prefix-Sum ( ⟨x_1, x_2, ..., x_n⟩, ⊕ )   { n = 2^k for some k ≥ 0.
                                           Return prefix sums
                                           ⟨s_1, s_2, ..., s_n⟩ }
1.  if n = 1 then
2.      s_1 ← x_1
3.  else
4.      parallel for i ← 1 to n/2 do
5.          y_i ← x_{2i-1} ⊕ x_{2i}
6.      ⟨z_1, z_2, ..., z_{n/2}⟩ ← Prefix-Sum( ⟨y_1, y_2, ..., y_{n/2}⟩, ⊕ )
7.      parallel for i ← 1 to n do
8.          if i = 1 then s_1 ← x_1
9.          else if i = even then s_i ← z_{i/2}
10.             else s_i ← z_{(i-1)/2} ⊕ x_i        { z_0 is identity }
11. return ⟨s_1, s_2, ..., s_n⟩
```


parallel for $\Rightarrow$ cilk_for
(pseudo-code)

Parallel Prefix Sums



prefixsum.c


```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cilk/cilk.h>

int add (int j, int k) { return j+k; }
int min (int j, int k) { return j<k ? j : k; }
int max (int j, int k) { return j>k ? j : k; }
int xor (int j, int k) { return j^k; }
```

prefixsum.c (continued)


```
void ParPrefixSum (int (*f)(int,int), int *X, int n) {
    if (n==1) return;
    int m=n/2;
    int *Y = malloc (m * sizeof(int));
    cilk_for (i=0; i<m; i++)
        Y[i] = f (X[2*i], X[2*i+1]);
    ParPrefixSum (f, Y, m);
    cilk_for (i=0; i<m; i++)
        X[2*i+1] = Y[i];                    // odd case
    cilk_for (i=1; i<(n+1)/2; i++)
        X[2*i] = f (X[2*i], Y[i-1]);        // even case
    free(Y);
}
```

prefixsum.c (continued)


```
int main (int argc, char *argv[ ]) {
        int size=atoi (argv[1]);
        int *A = malloc (size * sizeof(int));
        int *B = malloc (size * sizeof(int));
        int *C = malloc (size * sizeof(int));
        int *D = malloc (size * sizeof(int));
        int *E = malloc (size * sizeof(int));
        time_t t;
        srand ((unsigned) time(&t));
        for (int k=0; k<size; k++)
                A[k] = B[k] = C[k] = D[k] = E[k] =
                                rand( ) % min(size,1000);
```

prefixsum.c (continued)


```
    clock_t start=clock( );
    ParPrefixSum (add, B, size);
    ParPrefixSum (min, C, size);
    ParPrefixSum (max, D, size);
    ParPrefixSum (xor, E, size);
    clock_t finish=clock( );
    double duration=(double)(finish-start)/CLOCKS_PER_SEC;
    printf("ParPrefixSums took %lf seconds\n\n", duration);
    printf ("A\tB\tC\tD\tE\n");
    for (int k=0; k<min(size,10); k++)
        printf ("%d\t%d\t%d\t%d\t%d\n", A[k], B[k], C[k], D[k], E[k]);
    return 0;
}
```

Compile and run


```
> cilk  prefixsum.c  –o  prefixsum
> ./prefixsum  10000000

ParPrefixSums took 2.695916 seconds
```


A	B	C	D	E
650	650	650	650	650
444	1094	444	650	822
177	1271	177	650	903
946	2217	177	946	53
428	2645	177	946	409
824	3469	177	946	673
55	3524	55	946	662
758	4282	55	946	96
772	5054	55	946	868
955	6009	55	955	223

Just prints the first 10 out of 10000000 values in each array

B: add

C: min

D: max

E: xor