Scan operation and parallelization

Reduction - recall

- we have seen that a sum reduction takes O(n) work to run to completion.
- The parallel sum reduction takes the same order of work (operations) to complete as the serial version.
- Therefore, we say a sum reduction is work-efficient.

Scan (Prefix Sum)

- Given an array $A = [a_0, a_1, \ldots, a_{n-1}]$ and a binary associative operator $\oplus$ with identity I,

$$\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$$

Computing Cumulative Sum

- Prefix sum: if $\oplus$ is addition, then scan on the series

3	**1**	**7**	**0**	**4**	**1**	**6**	**3**

returns the series

????

Scan (Parallel Prefix Sum)

- Given an array $A = [a_0, a_1, \ldots, a_{\underline{n}-1}]$
  and a binary associative operator $\oplus$ with identity *I*,

$$\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$$

- Prefix sum: if $\oplus$ is addition, then scan on the series

3	**1**	**7**	**0**	**4**	**1**	**6**	**3**

returns the series

0	**3**	**4**	**11**	**11**	**15**	**16**	**22**

Scan is a simple & useful parallel building block for many parallel algorithms. Cutting example.
Difficult to parallelize !!! Each element depends on the previous.

Scan : types

Sum of all the elements that preceded **current element +**

Inclusive Scan

3	1	7	0	4	1	6	3

returns the series

3	4	11	11	15	16	22	25

Exclusive Scan

3	1	7	0	4	1	6	3

returns the series

0	3	4	11	11	15	16	22

Difference between reduction and scan?

Scan (prefix sum) : important concepts

- Input → array of elements

- Binary operator

- Identity element/ value → when operate with any element gives us back the same element.

- Associative and commutative operator.

- Running Sum of Inputs

Uses for Scan

- Sorting
- Lexical Analysis
- String Comparison
- Polynomial Evaluation
- Stream Compaction
- Building Histograms and Data Structures

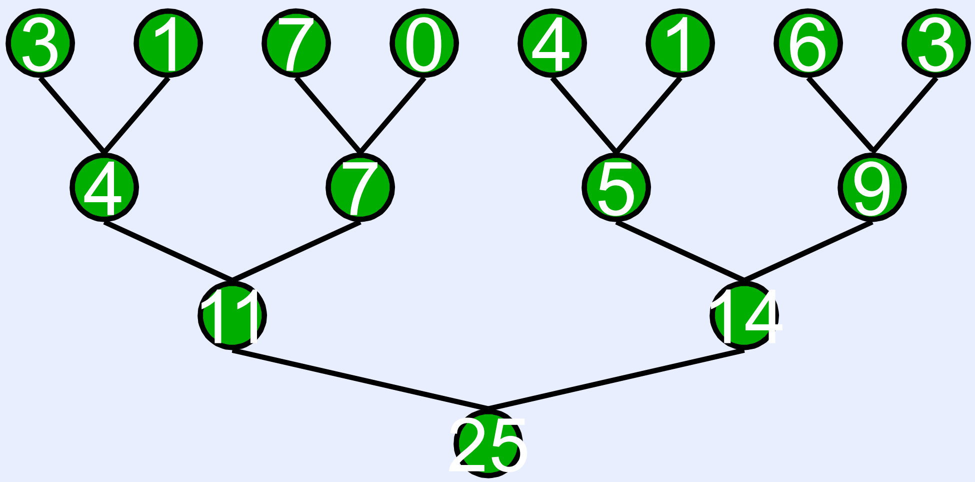
Serial Scan


```
int input[8] = {3, 1, 7, 0, 4, 1, 6, 3};
int result[8];
int running_sum = 0;
for(int i = 0; i < 8; ++i)
{
  result[i] = running_sum;
  running_sum += input[i];
}
```

```
prefixSum[0] = 0;
for(i = 1; i < N; i++)
  prefixSum[i]=prefixSum[i-1]+arr[j-1];

// result = {0, 3, 4, 11, 11, 15, 16, 22}
```

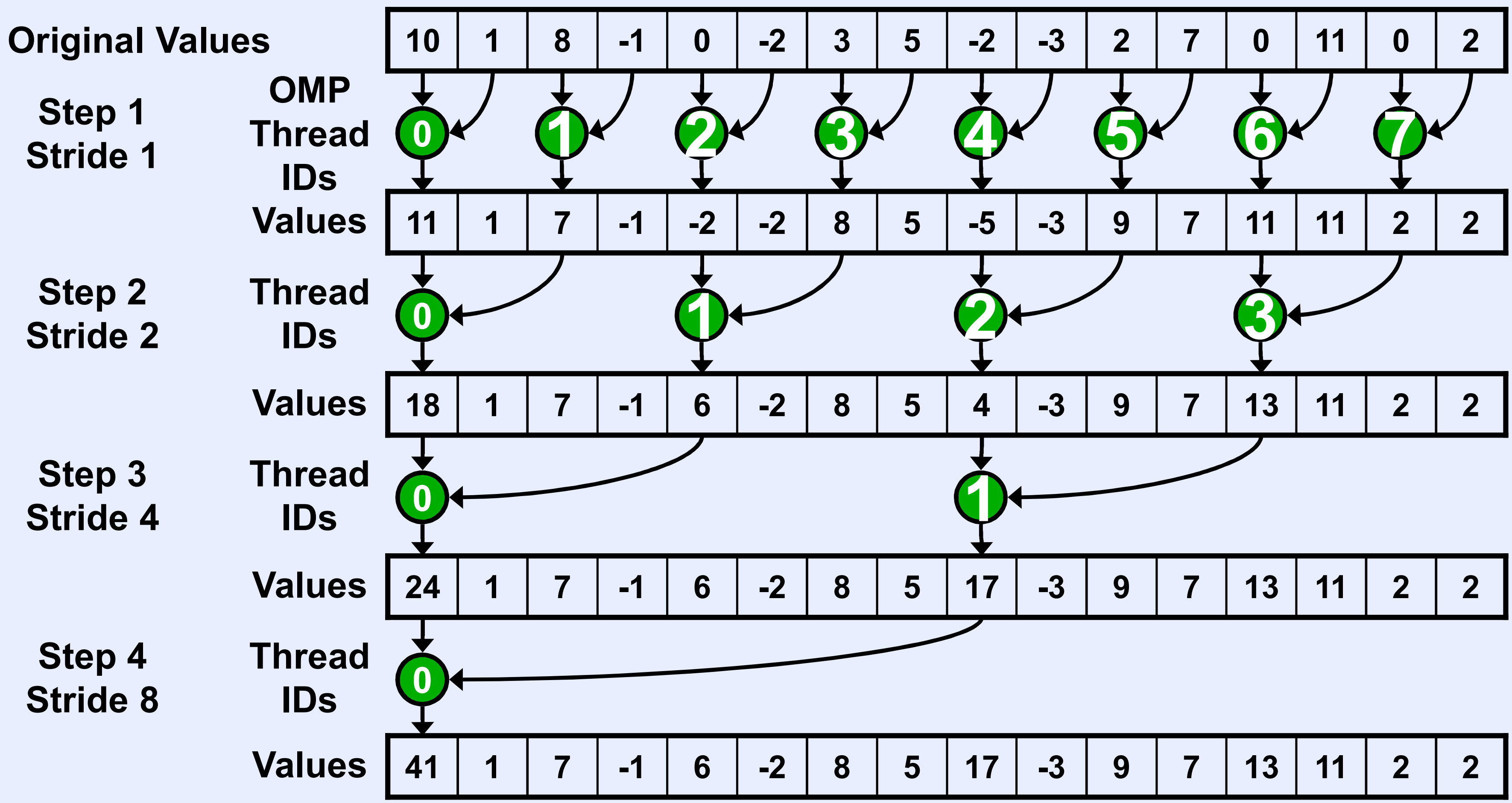
Reduction



Work Complexity → work efficient

Step Complexity → Step efficient

Parallel Reduction



Parallel Reduction Complexity

- Log(N) parallel steps, each step S does $N/2^S$ independent ops
  - *Step Complexity is O(log N)*
- For $N=2^D$, performs $\sum_{S\in[1..D]} 2^{D-S} = N-1$ operations
  - *Work Complexity is O(N) – It is work-efficient*
  - *i.e. does not perform more operations than a sequential algorithm*

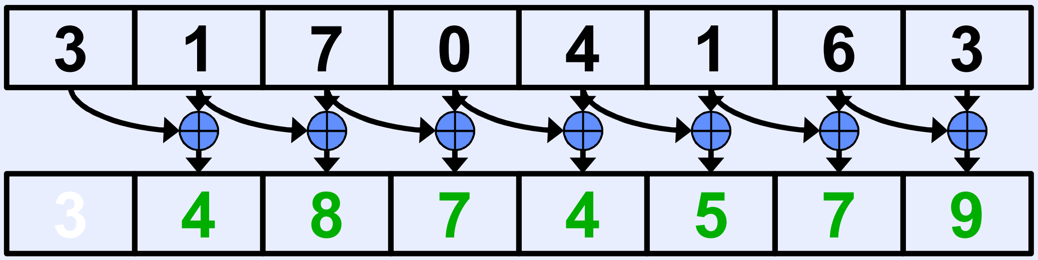
Parallel Inclusive Scan (Hillis and Steele)

A Scan Algorithm

3	1	7	0	4	1	6	3

Assume array has shared scope

A Scan Algorithm – Preview

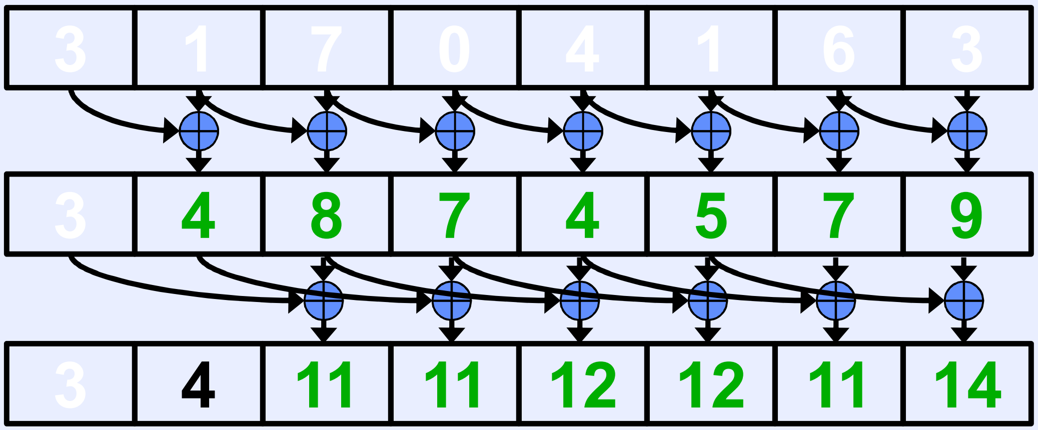


Iteration 0, *n-1* threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value

A Scan Algorithm – Preview

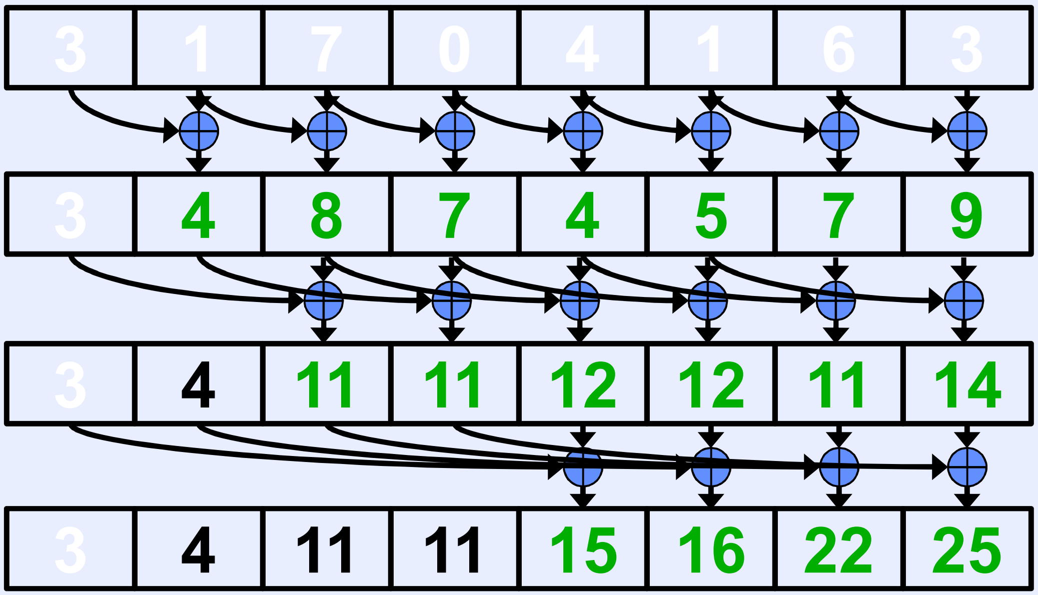


Iteration 1, *n-2* threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *offset* elements away to its own value

A Scan Algorithm – Preview



Iteration *i*, *n*-2*^i* threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *offset* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

A Scan Algorithm – Preview

3	4	11	11	15	16	22	25

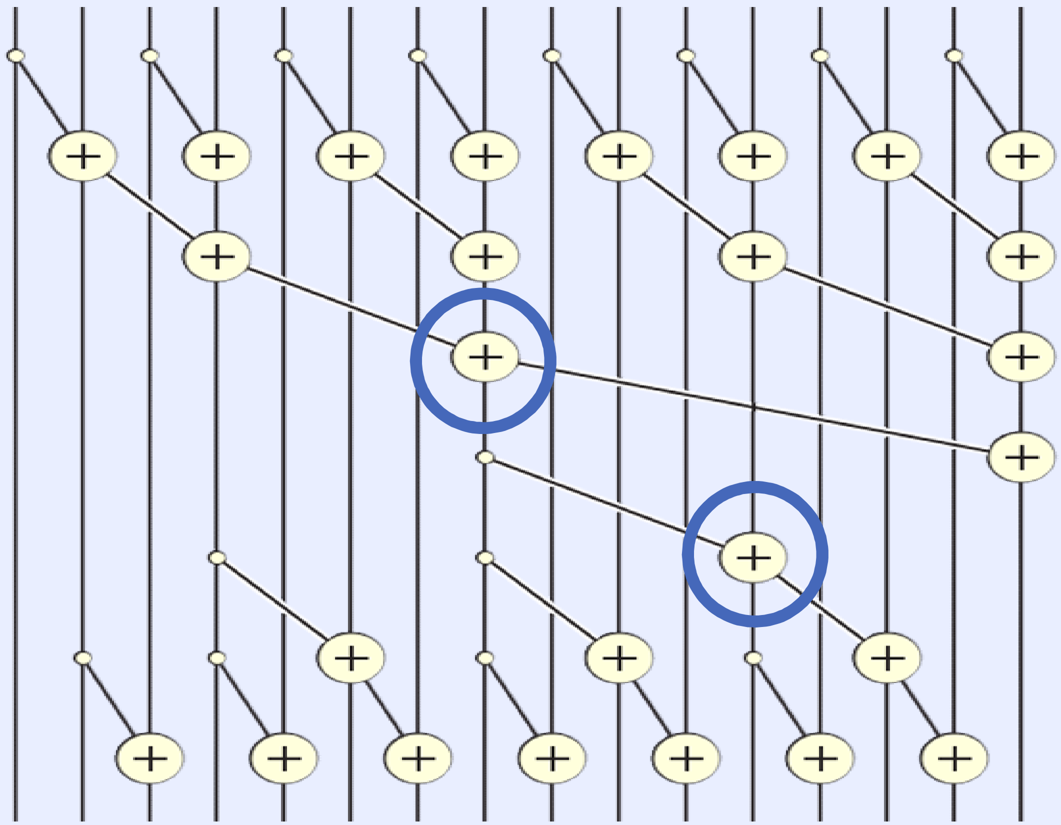
- We have an inclusive scan result
- End of iteration N, array will contain the sum of 2^N input elements at and before the location.
- Input X[0] → output Y[0]
- First iteration → Each position except Y[0] receives the sum of current content and its left element.
- Second iteration → Except Y[0],Y[1], receive the sum of its current content and content 2 elements away.

Scan - OpenMP Parallelism

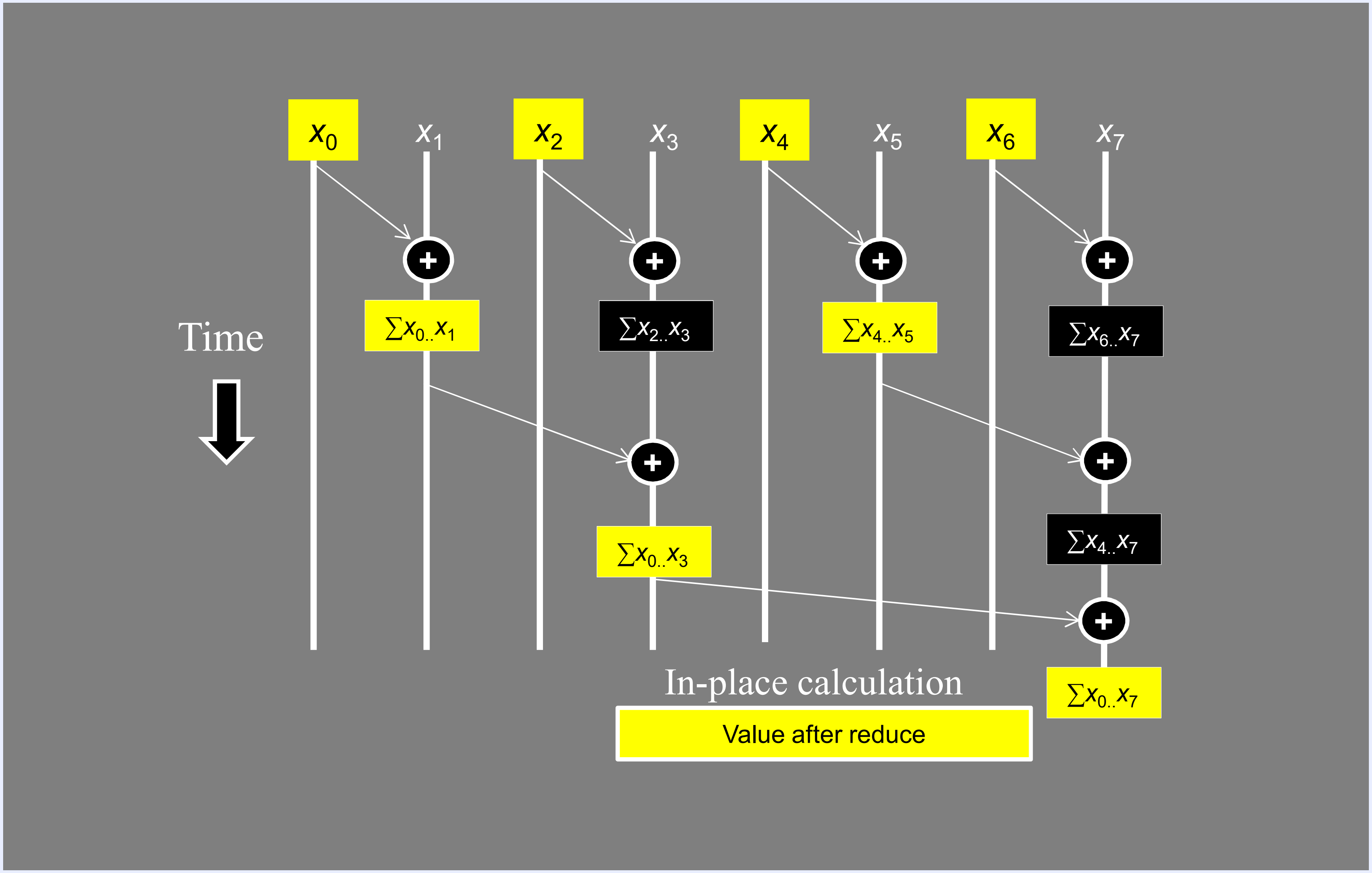
```
 for(i = 1; i <= log_of_N; i++)
  {
    two_i_1 = 1 << (i-1);
    two_i = 1 << i;
    out = 1 - out;
    in = 1 - out;
    #pragma omp parallel for private(j)
shared(scanSum, in, out)
    for(j = 0; j < N; j++)
    {
      if(j >= two_i_1)
        scanSum[out][j] = scanSum[in][j] +
scanSum[in][j - two_i_1];
      else
        scanSum[out][j] = scanSum[in][j];
    }
  }
```

Work Efficiency Considerations

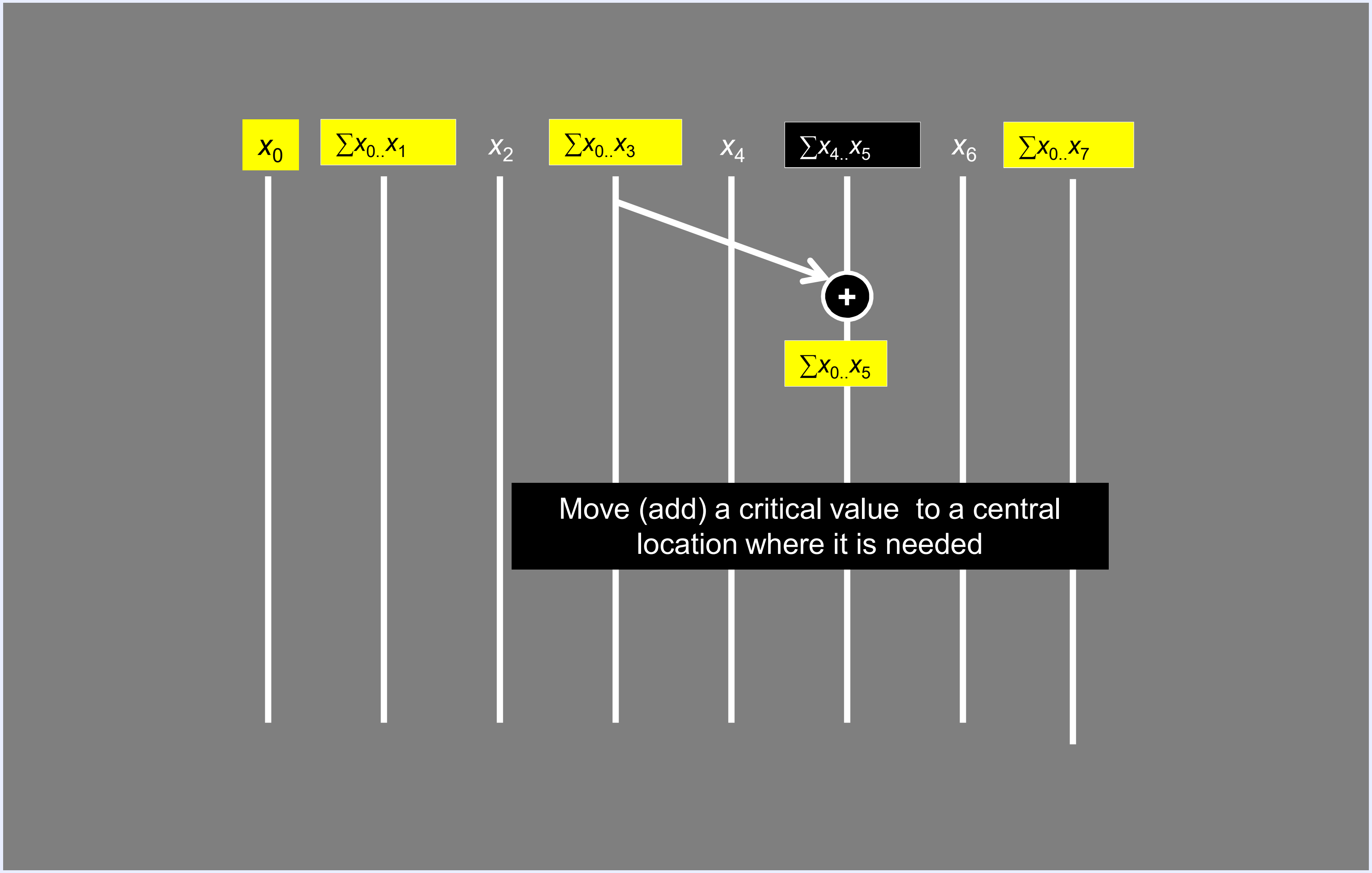
- This Scan executes log(n) parallel iterations
  - *The iterations do (n-1), (n-2), (n-4),..(n- n/2) adds each*
  - *Total adds: n * log(n) - (n-1) → O(n*log(n)) work*
- This scan algorithm is not work efficient
  - *Sequential scan algorithm does n adds*
  - *A factor of log(n) can hurt: 10x for 1024 elements!*
- A parallel algorithm can be slower than a sequential one when execution resources are saturated from low work efficiency
- This means we can complete a scan in O(lg n) time if we have O(n lg n) processors



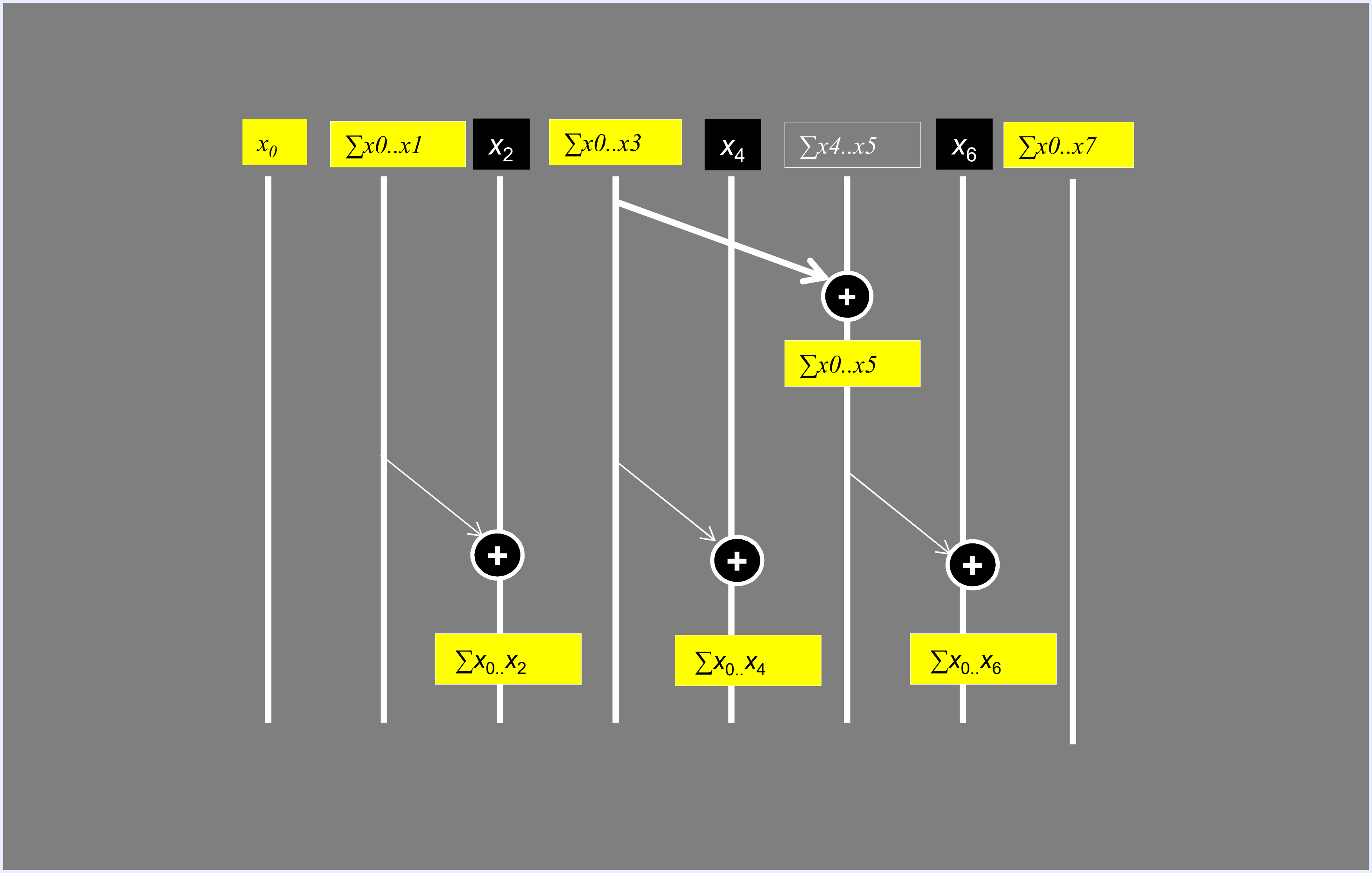
Parallel Scan - Reduction Phase



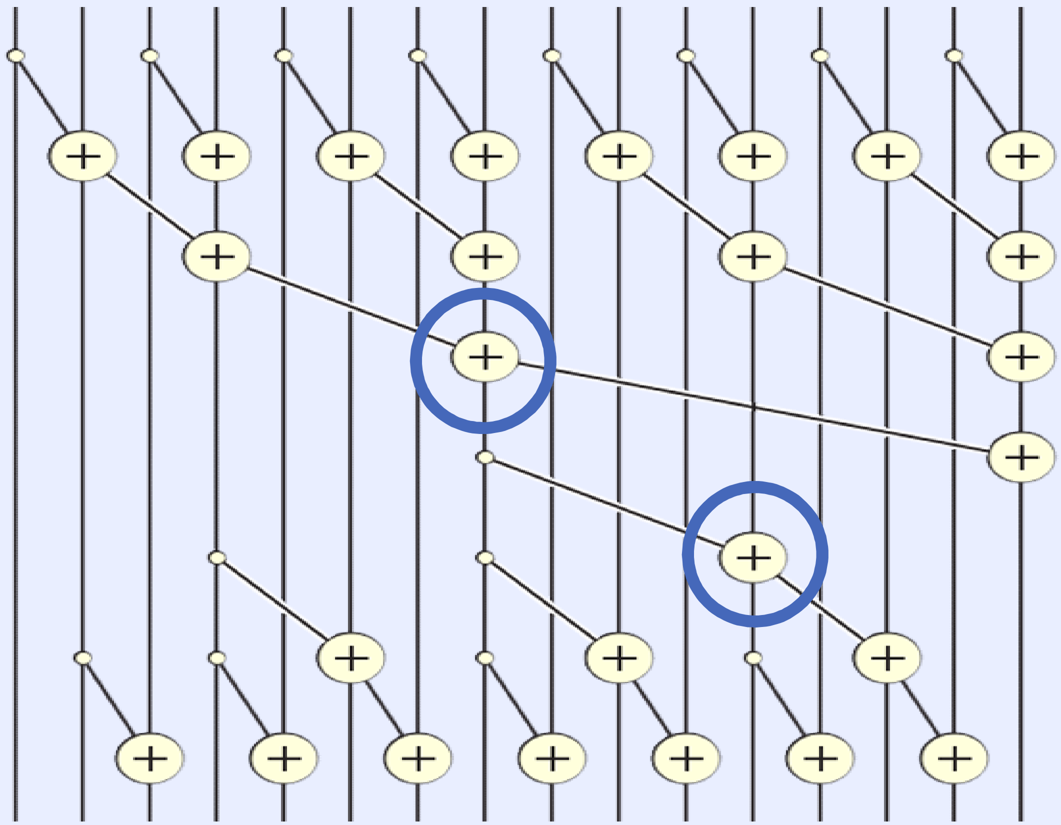
Parallel Scan - Post Reduction Reverse Phase



Parallel Scan - Post Reduction Reverse Phase



Final view



Work Efficient Parallel Scan (Blelloch 1990)

- Goal – build a scan algorithm that avoids the extra O(lg n) work
- Solution – make use of balanced binary trees
  - *Build a balanced binary tree on the input data and sweep the tree to compute the prefix sum*
- Consists of two parts
- Up Sweep
  - *Traverse tree from leaves to root computing partial sums during the traversal*
  - *The root node (last element in the array) will hold the sum of all elements*
- Down Sweep
  - *Traverse from root to leaves using the partial sums to compute the cumulative sums in place.*
  - *For the exclusive prefix sum, we replace the root with zero*

Blelloch Algorithm – Up Sweep


```
for(i = 0; i <= log_of_N_1; i++) {
  two_i_p1 = 1 << (i+1);
  two_i = 1 << i;

  for(j = 0; j < N; j+=two_i_p1)
    scanSum[j + two_i_p1 - 1] =
          scanSum[j + two_i - 1] +
          scanSum[j + two_i_p1 - 1];
}
```

Blelloch Algorithm – Down Sweep


```
scanSum[N-1] = 0;
for(i = log_of_N_1; i >= 0; i--) {
  two_i_p1 = 1 << (i+1);
  two_i = 1 << i;

  for(j = 0; j < N; j+=two_i_p1) {
    long t = scanSum[j + two_i - 1];
    scanSum[j + two_i - 1] =
               scanSum[j + two_i_p1 - 1];
    scanSum[j + two_i_p1 - 1] = t +
               scanSum[j + two_i_p1 - 1];
  }
}
```

Analysis

- This algorithm does complete the summation in O(n) operations, giving the appearance of efficiency.
- Actually, the algorithm requires 2*(n-1) additions and n-1 swaps to run to completion.
- For large arrays, this algorithm will indeed out-perform the Hillis and Steele Algorithm.

a work-efficient scan kernel

- Two-phased balanced tree traversal
  - *Aggressive re-use of intermediate results*

A Scan Algorithm – Preview

3	1	7	0	4	1	6	3

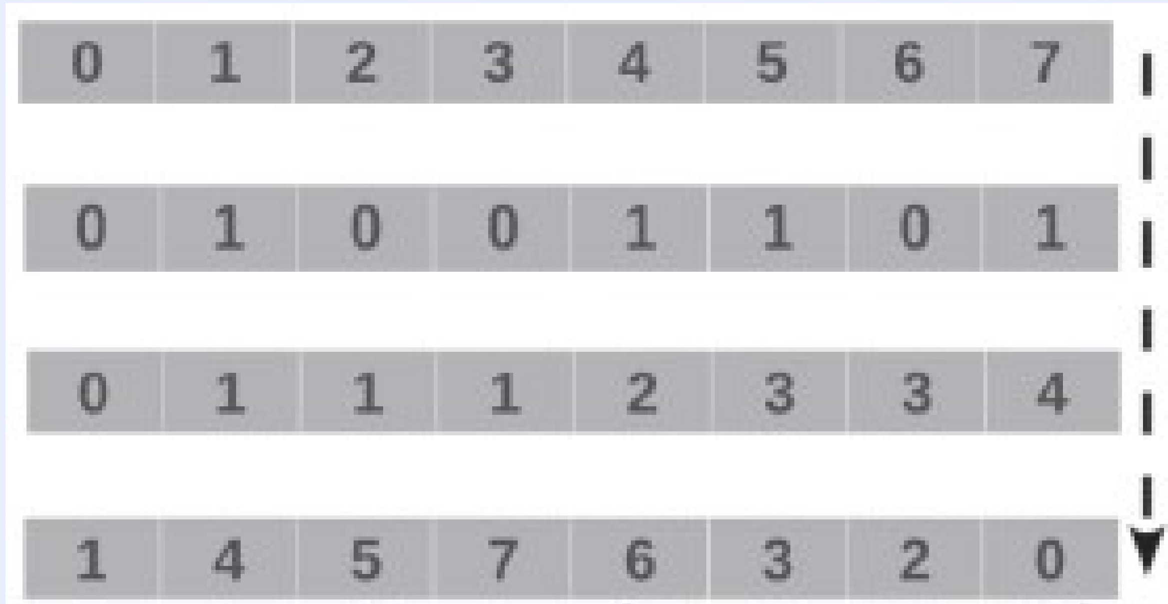
Assignment : Parallellelize using openMP.

Complexity of both the algorithms.

Speedup for different size of the input array.
Also scalability?

Problem

Old index



New index