COMP90025 Parallel and Multicore Computing

Prefix Sum

Aaron Harwood

School of Computing and Information Systems
The University of Melbourne

2019 Semester II

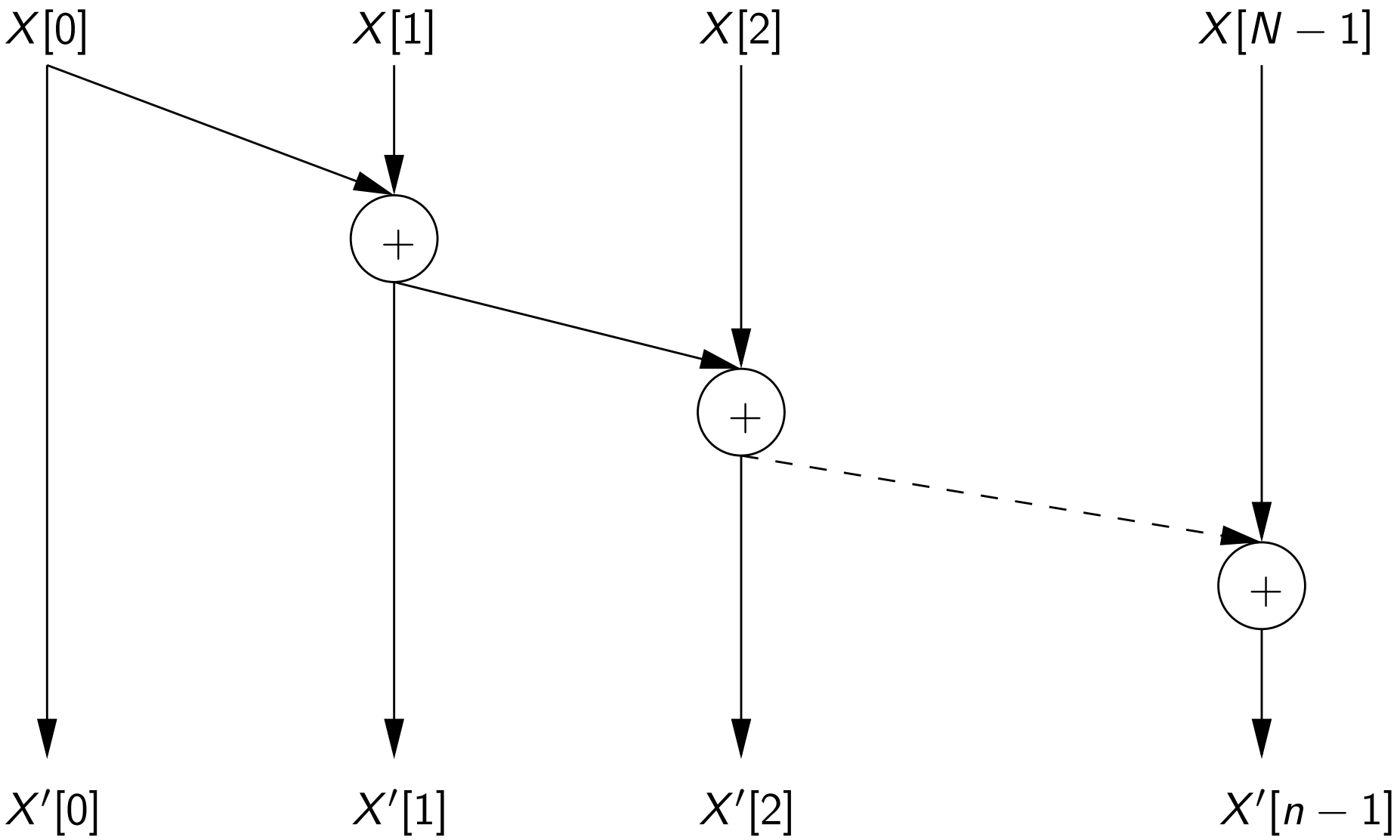
Prefix problem

The prefix problem requires computing the prefix sum (or other dyadic operation) on an array of elements:

```
int X[n];
⋮

for(i=1;i<n;i++)
    X[i] = X[i-1] + X[i];
```

The data dependency suggests flow dependence in this computation.


X[0]
X[1]
X[2]
X[N − 1]
+
+
+
X′[0]
X′[1]
X′[2]
X′[n − 1]

- The sequential prefix problem has a *size* of $n-1$ because there are $n-1$ operations in total.
- The depth of the sequential prefix problem is $n-1$ because it requires $n-1$ time steps to complete.
- It is possible to reduce the depth of the prefix problem, but only at the expense of increasing the size.

Upper/Lower parallel prefix algorithm

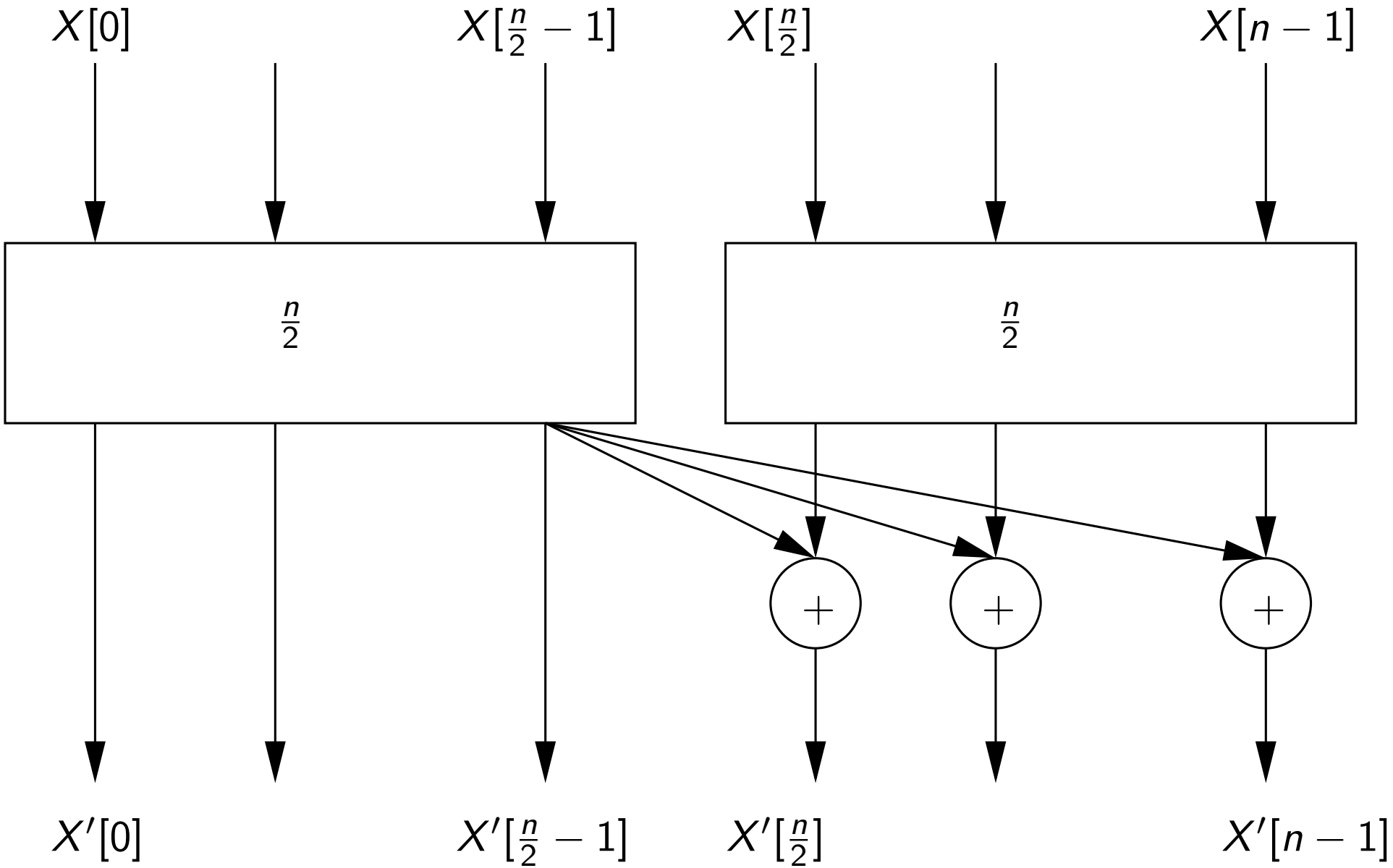
The upper/lower algorithm uses a *divide and conquer* approach to recursively divide the problem into smaller sub-problems. The results of the sub-problems are then combined to give the final output.
Divide and conquer is a fundamental approach for obtaining parallelism.
The upper/lower algorithm divides the lower half of the elements from the upper half of the elements. The prefix sum is independently computed on the lower and upper halves by recursively applying the divide and conquer.
The output from the highest element of the lower half is added to each output of the upper half.

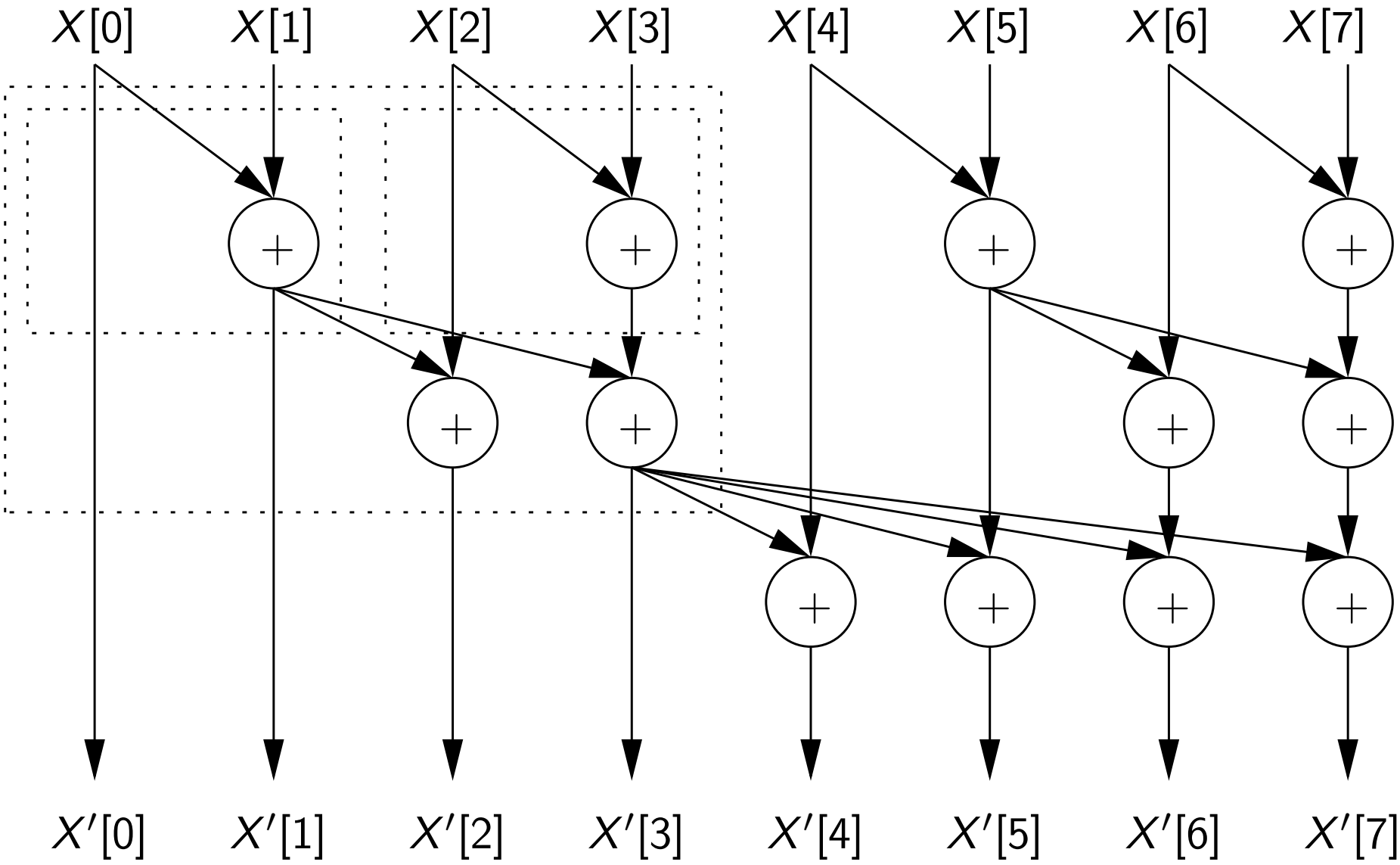
Description: For array $X = [x_0 \dots x_{n-1}]$ returns an array $S = [s_0 \dots s_{n-1}]$ of sums where $s_i = \sum_{j=0}^{i} x_j$.

Analysis: $\Theta(\log n)$

Processors: n

1: **procedure** $\textsc{UpperLower}^{\diamondsuit}_{\text{CREW}}(X, n)$
2: **if** $n = 1$ **then**
3: $S[0] \leftarrow X[0]$
4: **return** S
5: **for** $i \leftarrow 0$ **to** $n - 1$ **do in parallel**
6: $\frac{n}{2}$ **processor array** $0 \dots \frac{n}{2} - 1$ **does**
7: $S\left[0 \dots \frac{n}{2} - 1\right] \leftarrow \textsc{UpperLower}^{\diamondsuit}_{\text{CREW}}(X\left[0 \dots \frac{n}{2} - 1\right], \frac{n}{2})$
8: $\frac{n}{2}$ **processor array** $\frac{n}{2} \dots n - 1$ **does**
9: $S\left[\frac{n}{2} \dots n - 1\right] \leftarrow \textsc{UpperLower}^{\diamondsuit}_{\text{CREW}}(X\left[\frac{n}{2} \dots n - 1\right], \frac{n}{2})$
10: **for** $i \leftarrow \frac{n}{2}$ **to** $n - 1$ **do in parallel**
11: **processor** i **does**
12: $S[i] \leftarrow S[i] + S\left[\frac{n}{2} - 1\right]$ ▷ Concurrent read requirement
13: **return** S


$X[0]$
$X[\frac{n}{2}-1]$
$X[\frac{n}{2}]$
$X[n-1]$
$\frac{n}{2}$
$\frac{n}{2}$
+
+
+
$X'[0]$
$X'[\frac{n}{2}-1]$
$X'[\frac{n}{2}]$
$X'[n-1]$


X[0]
X[1]
X[2]
X[3]
X[4]
X[5]
X[6]
X[7]
X′[0]
X′[1]
X′[2]
X′[3]
X′[4]
X′[5]
X′[6]
X′[7]

Let $P^{ul}(n)$ denote the upper/lower prefix algorithm on n inputs.

Theorem

Let $S(n)$ be the size of $P^{ul}(n)$, then for $n = 2^t$, $S(2^t) = t2^{t-1} = \frac{n}{2}\log_2 n$.

Proof.

For the base case when $t = 1$, $S(2) = 1 \times 2^{1-1} = 1$ which is true. Assume that $S\left(2^i\right) = i2^{i-1}$ for $i > 1$. Now for $n = 2^{i+1}$,

$$\begin{aligned} S\left(2^{i+1}\right) &= S\left(2^i\right) + S\left(2^i\right) + 2^i \\ &= 2\,S\left(2^i\right) + 2^i \\ &= i2^i + 2^i \\ &= (i+1)2^i. \end{aligned}$$

Substituting $i' = i + 1$ yields $S\left(2^{i'}\right) = i'2^{i'-1}$ which by inductive hypothesis is true and this completes the proof. □

- If n is not a power of two then one prove bounds (using padding), e.g. that $(t-1)2^{t-2} \leq S(n) \leq t2^{t-1}$ for $2^{t-1} < n < 2^t$ and $t > 0$.
- Similarly the depth of $P^{ul}(n)$ for $n = 2^t$ is shown to be $t = \log_2 n$.
- If as many as $n/2$ processors are available then the operation can be completed in $\log_2 n$ steps. Furthermore, using $p = \frac{n}{2}$ processors:
  - speedup is $\frac{n-1}{\log_2 n}$.
  - efficiency is $\frac{2(n-1)}{n \log_2 n} \times 100\%$
  - work is $\frac{n}{2} \log_2 n$ compared to $(n-1)$ for sequential algorithm, which is not optimal, but efficient.

Odd/Even parallel prefix algorithm

- The odd/even parallel prefix algorithm, P^{oe}, also uses a divide and conquer approach.
- The algorithm divides n inputs into groups whose indices are odd and even, respectively.
- The recursion continues to halve the number of inputs until reaching a base case.

Description: For array $X = [x_0 \dots x_{n-1}]$ returns an array $S = [s_0 \dots s_{n-1}]$ of sums where $s_i = \sum_{j=0}^{i} x_j$.

Analysis: $\Theta(\log n)$

Processors: n

1: **procedure** $\textsc{OddEven}^{\diamondsuit}_{\text{EREW}}(X, n)$
2: **if** $n \leq 4$ **then**
3: $S[0] \leftarrow X[0]$
4: **for** $i \leftarrow 1$ **to** n **do**
5: $S[i] = X[i] + S[i-1]$
6: **return** S
7: **for** $i \leftarrow 0$ **to** $n-1$ **do in parallel**
8: **processor** $i \mid i \bmod 2 = 0$ **does**
9: $S[i] \leftarrow X[i]$
10: **processor** $i \mid i \bmod 2 = 1$ **does**
11: $P\left[\left\lfloor \frac{i}{2} \right\rfloor\right] \leftarrow X[i-1] + X[i]$

12: $\frac{n}{2}$ **processor array** $i \mid i \bmod 2 = 1, 0 < i \leq n-1$ **does**

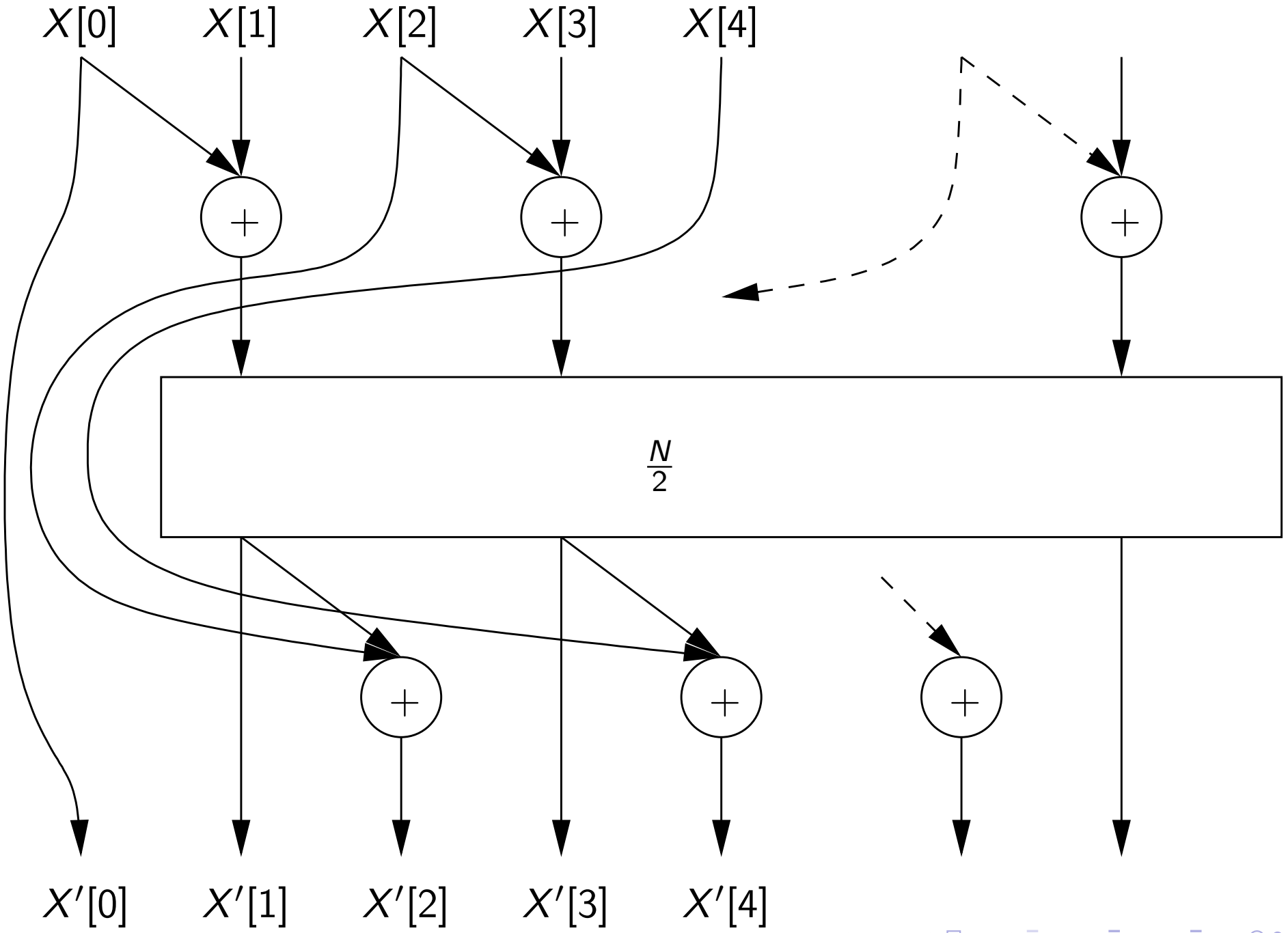
13: $A \leftarrow \textsc{OddEven}^{\diamondsuit}_{\text{EREW}}(P, \frac{n}{2})$

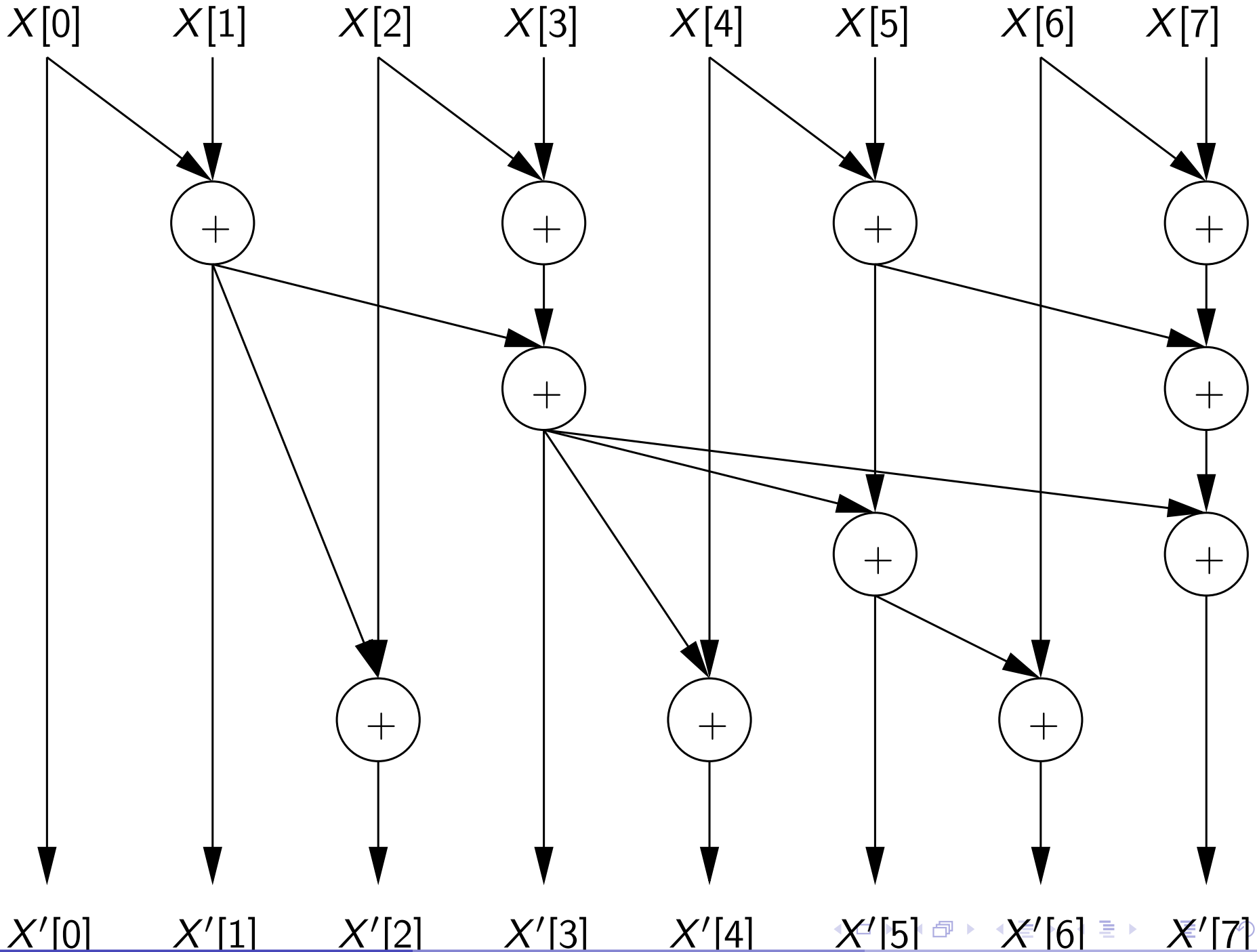
14: **for** $i \leftarrow 2$ **to** $n-1$ **do in parallel**

15: **processor** $i \mid i \bmod 2 = 0$ **does**

16: $S[i] \leftarrow S[i] + A\left[\frac{i}{2} - 1\right]$

17: **return** S


X[0]
X[1]
X[2]
X[3]
X[4]
N/2
X′[0]
X′[1]
X′[2]
X′[3]
X′[4]


X[0]
X[1]
X[2]
X[3]
X[4]
X[5]
X[6]
X[7]
X′[0]
X′[1]
X′[2]
X′[3]
X′[4]
X′[5]
X′[6]
X′[7]

- Clearly the construction of $P^{oe}(n)$ adds 2 levels of operations to the depth for each time that the number of inputs are halved.
- When $n = 4$, the construction of $P^{oe}(4)$ finishes with 2 levels of operations.
- Considering only when $n = 2^t$, the depth of $P^{oe}\left(2^t\right)$ is

$$\sum_{i=3}^{t} 2 + 2 = 2t - 2 = 2\log_2 n - 2, t \geq 2.$$

- Similarly for size, the construction of $P^{oe}\left(2^t\right)$ is

$$\sum_{i=1}^{t}(2^i - 1) = 2^{t+1} - t - 2$$

$$= 2n - \log_2 n - 2, t \geq 0.$$

This is significantly better than size of $P_{ul}(n)$, but still the work on $p = \Theta(n)$ processors is $\Theta(n \log n)$ which is not optimal.

Optimal parallel prefix sum algorithm

To be done in class.

Ladner and Fischer's parallel prefix algorithm

- A more sophisticated parallel prefix algorithm that allows finer tradeoffs between depth and size was proposed by Ladner and Fischer in 1980.
- Their work actually defines a class of algorithms called $P_j(n)$ for $j \geq 0$.
- Here we examine the fundamental construction that uses $P_0(n)$ and $P_1(n)$ only. Both the upper/lower and even/odd constructions are used.
- Basically, the odd/even construction is used to construct $P_1(n)$ from $P_0(n/2)$, and the upper/lower construction is used to define $P_0(n)$ in terms of both $P_1(n/2)$ and $P_0(n/2)$.

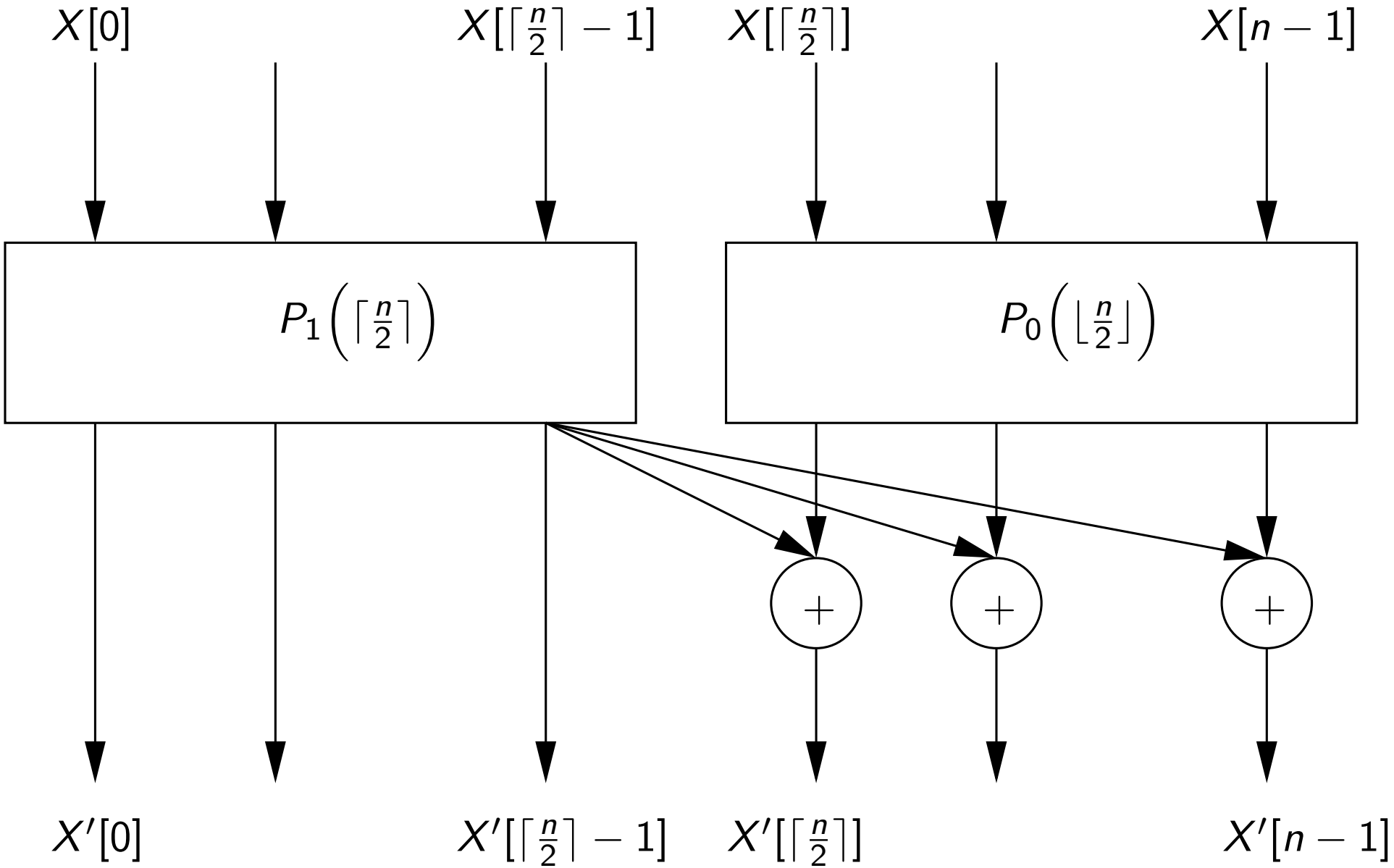


Figure: Definition of Ladner and Fischer's $P_0(n)$.

The example construction for 8 elements is equivalent to the upper/lower construction for 8 elements because odd/even on 4 elements is equivalent to upper/lower on 4 elements. A large construction is needed to see the difference.

The construction for $P_0(n)$, $n = 2^t$, maintains a depth of $\lceil \log_2 n \rceil$. This is due to the odd/even construction of $P_1(n)$ having one less operation than its depth in the right most element. The construction for $P_0(n)$ is using the upper/lower construction which maintains a minimum depth.

The size for $P_0(n)$ is approximately $4n - 4.96n^{0.69} + 1$. The proof is based on the Fibonacci sequence :-)

The size is by no means as easy to show. Could you guess what it is? Consider the basic size relationships:

$$S_0(n) = S_1\left(\lceil n/2 \rceil\right) + S_0\left(\lfloor n/2 \rfloor\right) + \lceil n/2 \rceil$$
$$S_j(n) = S_{j-1}\left(\lceil n/2 \rceil\right) + n - 1, j \geq 1, \text{even } n \geq 2$$
$$S_j(n) = S_{j-1}\left(\lceil n/2 \rceil\right) + n - 2, j \geq 1, \text{odd } n \geq 3$$

Theorem

If $n = 2^t$ *then*

$$S_0(n) = 4n - F(2+t) - 2F(3+t) + 1$$
$$S_1(n) = 3n - F(1+t) - 2F(2+t)$$

where $F(m)$ *is the m-th Fibonacci number.*

Note: $F(0) = 0$, $F(1) = 1$ and $F(m) = F(m-1) + F(m-2)$ for $m \geq 2$.

Proof.

For the base cases, $S_0(2^1) = 1 = 4 \times 2 - F(3) - 2F(4) + 1$ and $S_1(2^2) = 4 = 3 \times 4 - F(3) - 2F(4)$. Assume that for $t = k$

$$S_0(n) = 4n - F(2+k) - 2F(3+k) + 1$$
$$S_1(n) = 3n - F(1+k) - 2F(2+k).$$

Now show the result for $k+1$ using the size relationships deduced earlier:

$$\begin{aligned} S_0(2^{k+1}) &= 3 \times 2^k - F(1+k) - 2F(2+k) + \\ &\quad 4 \times 2^k - F(2+k) - 2F(3+k) + 1 + 2^k \\ &= 4 \times 2^{k+1} - F(3+k) - 2F(4+k) + 1, \end{aligned}$$

$$\begin{aligned} S_1(2^{k+1}) &= 4 \times 2^k - F(2+k) - 2F(3+k) + 1 + 2^{k+1} - 1 \\ &= 3 \times 2^{k+1} - F(2+k) - 2F(3+k). \end{aligned}$$

Substituting $k' = k+1$ yields the same form as our assumptions and the proof is complete. □

A well known asymptotic formula for the Fibonacci numbers is:

$$F(m) = \frac{\phi^m - \hat{\phi}^m}{\sqrt{5}}$$

where

$$\phi = \frac{1 + \sqrt{5}}{2}$$

and

$$\hat{\phi} = \frac{1 - \sqrt{5}}{2}.$$

For large m, $F(m) \approx \frac{\phi^m}{\sqrt{5}}$.
Consequently:

$$\begin{aligned} S_0(n) &\approx 4n - \frac{\phi^{k+2}}{\sqrt{5}} - 2\frac{\phi^{k+3}}{\sqrt{5}} + 1 \\ &= 4n - 4.9597n^{0.69424} + 1 \end{aligned}$$

Bounds on dyadic operations

It is easy to show that no matter how many processors we have, the depth of an arithmetic expression with n atoms is at least $\lceil \log_2 n \rceil$ (consider a tree with the root as the final answer and each node in the tree combining at most 2 atoms), and bounds the computation time from below.
Harder to prove, is that any expression with n atoms can be rearranged using associativity, commutativity, and distributivity so that the time required to evaluate the expression is no more than $\lceil 4 \log_2 n \rceil$ with the use of no more than $3n$ processors.
Such a rearrangement though takes $O(n \log_2 n)$ steps, so the computational overhead to rearrange the expression may be larger than the benefit gained.

Pointer jumping

- Sometimes we have data structures that are represented by pointer based data structures, where e.g. P is a list of nodes and $P[i] \in P$ is a pointer to a node in P.
- *Pointer jumping* is a well known technique to process such lists in parallel.
- Consider the following problems:
  - List Ranking: List P is a list of nodes where $P[i]$ points next node in the list for node i. Return the array R where $R[i]$ is the distance from node i to the head of the list.
  - Forest P is an array of nodes where $P[i]$ points to the parent of node i. Return the array S where $S[i]$ points to the root of node i.
  - Prefix Sum: List P is a list of nodes where $P[i] = (n, v)$ is a $(Next, Value)$ tuple, $Next$ pointing to the next node in the list for node i and $Value$ is a number being held by node i. Return the array S where $S[i]$ is the prefix sum from the head of the list.

Description: List P is a list of nodes where $P[i]$ points next node in the list for node i. Returns array R where $R[i]$ is the distance from node i to the head of the list.

Analysis: $\mathcal{O}(\log n)$ steps.

Processors: n

```
1:  procedure ListRank^◇_EREW(P, n)
2:      for i ← 0 to n − 1 do in parallel
3:          processor i does
4:              if P[i] = ∅ then
5:                  R[i] ← 0
6:              else
7:                  R[i] ← 1
8:              S[i] ← P[i]
9:              while S[i] ≠ ∅ do
10:                 R[i] ← R[i] + R[S[i]]
11:                 S[i] ← S[S[i]]
12:     return R
```

Description: Forest P is an array of nodes where $P[i]$ points to the parent of node i. Returns array S where $S[i]$ points to the root of node i.

Analysis: $\mathcal{O}(\log n)$ steps.

Processors: n

1: **procedure** $\textsc{FindRoots}^{\diamondsuit}_{\text{CREW}}(P, n)$
2: **for** $i \leftarrow 0$ **to** $n - 1$ **do in parallel**
3: **processor** i **does**
4: $S[i] \leftarrow P[i]$
5: **while** $S[i] \neq S[S[i]]$ **do**
6: $S[i] \leftarrow S[S[i]]$
7: **return** S

Description: List P is a list of nodes where $P[i] = (n, v)$ is a $(Next, Value)$ tuple, $Next$ pointing to the next node in the list for node i and $Value$ is a number being held by node i. Returns array S where $S[i]$ is the prefix sum from the head of the list.

Analysis: $\mathcal{O}(\log n)$ steps.

Processors: n

1: **procedure** $\textsc{ListPrefixSum}^{\diamondsuit}_{\text{EREW}}(P, n)$
2: **for** $i \leftarrow 0$ **to** $n - 1$ **do in parallel**
3: **processor** i **does**
4: $R[i] \leftarrow P[i]_{Value}$
5: $S[i] \leftarrow P[i]_{Next}$
6: **while** $S[i] \neq \emptyset$ **do**
7: $R[i] \leftarrow R[i] + R[S[i]]$
8: $S[i] \leftarrow S[S[i]]$
9: **return** R

Description: List P is a list of nodes where $P[i] = (n, v)$ is a $(Next, Value)$ tuple, $Next$ pointing to the next node in the list for node i and $Value$ is a number being held by node i. Returns array S where $S[i]$ is the prefix sum from the head of the list.

Analysis: $\mathcal{O}\left(\frac{n}{p} + \log p\right)$ steps.

```
1:  procedure ListPrefixSum*_EREW(P, n, p)
2:      for i ← 0 to p − 1 do in parallel
3:          processor i does
4:              S[i] ← P[i]_Value
5:              c ← i                          ▷ The current node for processor i
6:              R[i] ← ∅        ▷ Prepare array of pointers in reverse direction
7:              for j ← 0 to log n − 1 do
8:                  p ← c
9:                  c ← P[c]_Next
10:                 if c ≠ ∅ then
11:                     R[c] ← p     ▷ Build pointers in the reverse direction
```

12: $e \leftarrow c$ ▷ Remember the start of the next list
13: $c \leftarrow p$
14: **for** $j \leftarrow 0$ **to** $\log n - 2$ **do**
15: $c \leftarrow R[c]$
16: $S[c] \leftarrow S[c] + S[P[c]_{Next}]$
17: $T[p - i - 1] \leftarrow (e, S[i])$ ▷ Prepare the shorter list of length $\frac{n}{\log n}$
18: **all processors do**
19: $X \leftarrow \text{ListPrefixSum}^{\diamondsuit}_{\text{EREW}}(T, p)$
20: **for** $i \leftarrow 0$ **to** $p - 1$ **do in parallel**
21: **processor** i **does**
22: $c \leftarrow i$
23: **for** $j \leftarrow 0$ **to** $\log n - 1$ **do**
24: $S[c] \leftarrow S[c] + T[p - i - 1]$
25: $c \leftarrow P[c]_{Next}$
26: **return** S