Lecture: Systolic Arrays II

- Topics: algorithms for addition/multiplication, solving a system of linear equations, graph algorithms, sort

- Schedule for next week:
  - Tuesday:
  - Thursday:

Motivation

- How can accelerator computations be performed efficiently?
  - Basic adds, multiplies, convolutions
  - While you may not directly use the algorithm as shown, it might inspire other applications of the same technique
- Broaden our knowledge beyond dot products
- Last class: using arrays of simple processors (systolic arrays) to perform sorting and matrix multiplication
- This class: algorithms for addition and multiplication on linear arrays/trees, solving a system of linear equations, graph algorithms

Adding Two N-bit Numbers

- Propagating the carry takes N steps – can we do better?
- Recall the generate, propagate, and stop signals

```
   a =  1010
   b =  0011
 ---------------
   c =  0010
        psgp
```

s g p p g p s s p p s g p p p s

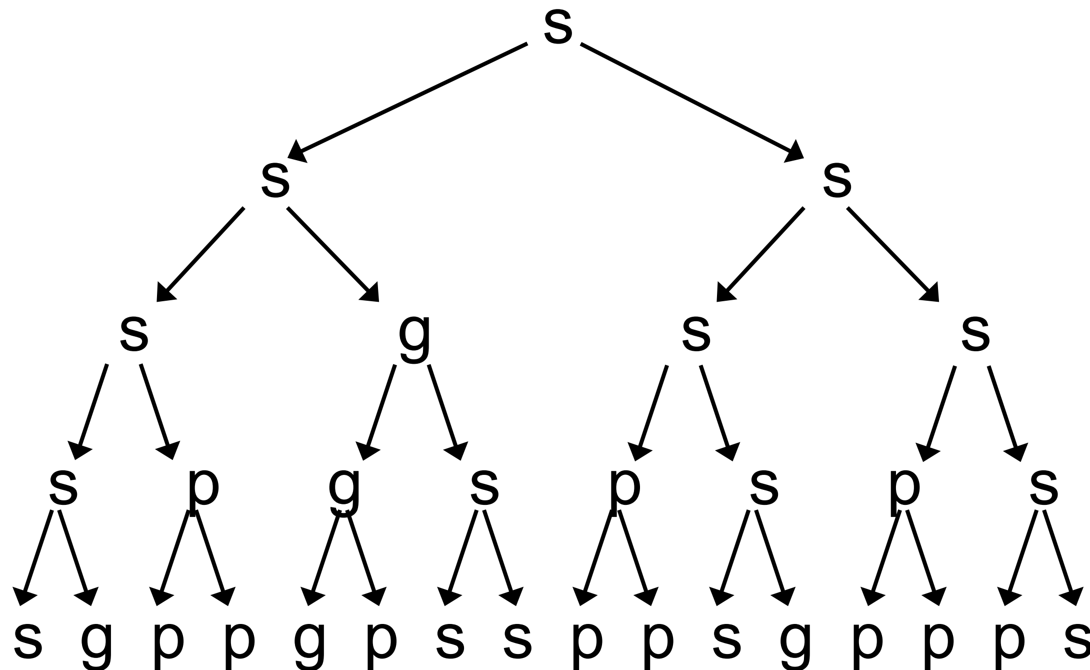
Carry =

Each carry depends on the leftmost non-p bit to the right

Using a Tree – Log(N)

Carry Lookahead

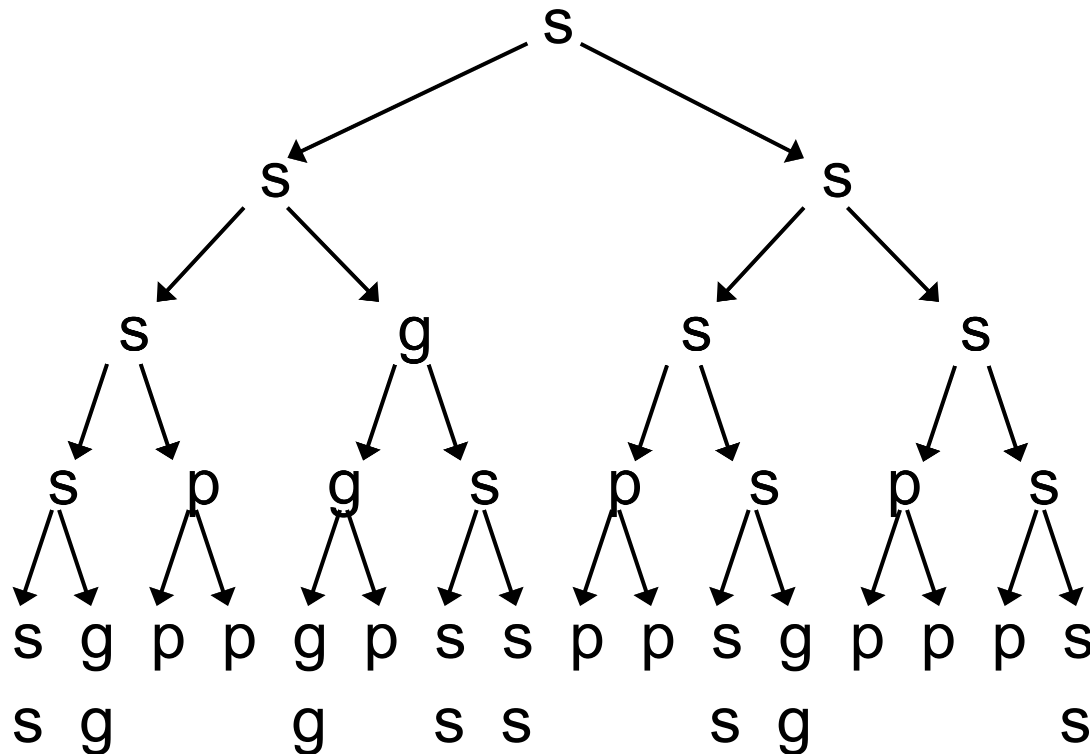
1. Propagate to the root: is the sub-tree generating a carry?
   node = value of its leftmost non-p child
   or p if both children are p



Using a Tree – Log(N)

Carry Lookahead

2. Leaves: remember the first non-p value you're going to see

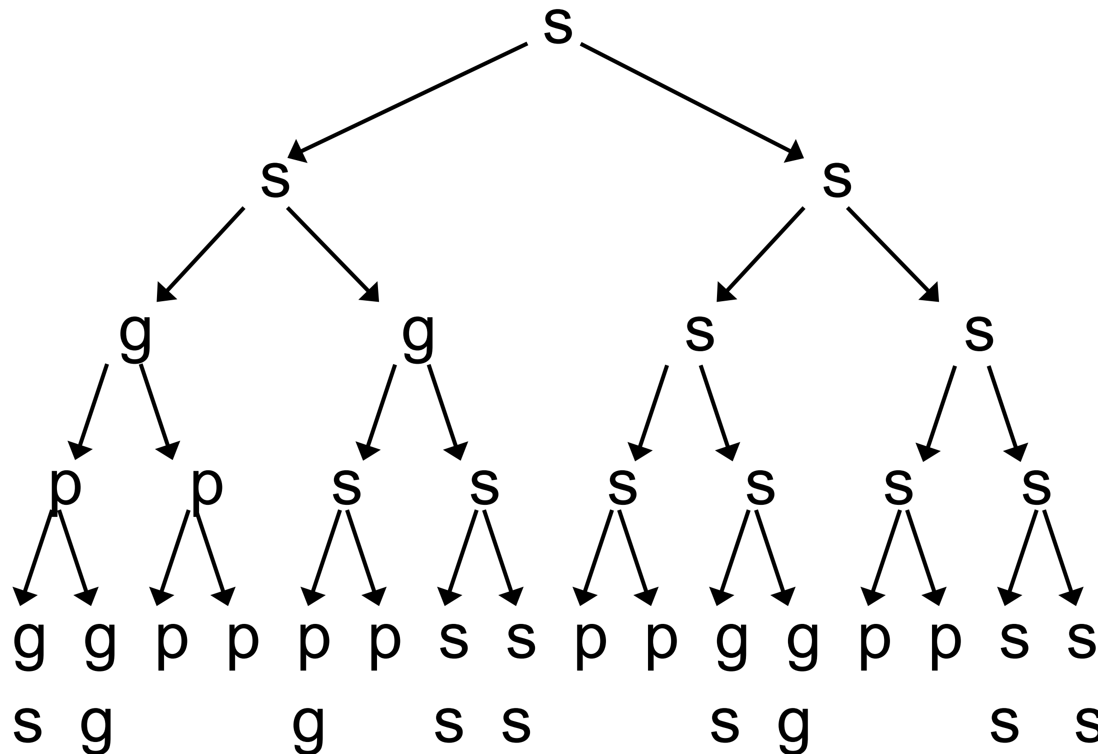


Remember: s g g s s s g s

Using a Tree – Log(N)

Carry Lookahead

3. As values are propagated to the root, copy the right child to the left child

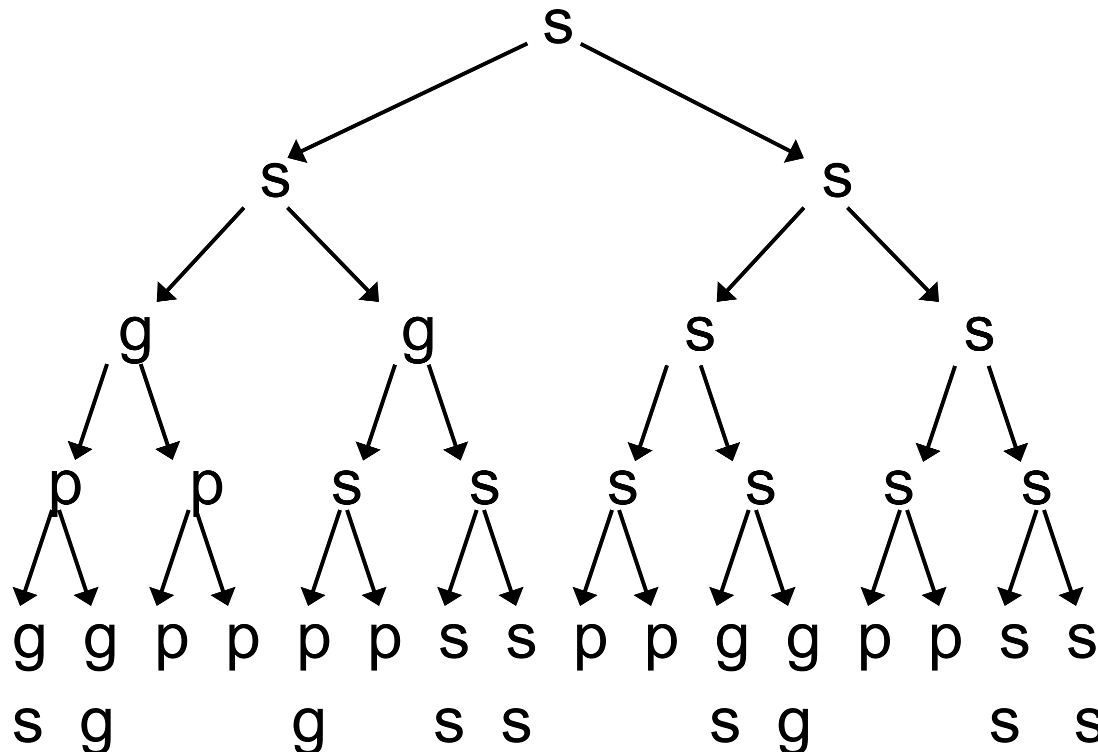


Remember:

Using a Tree – Log(N)

Carry Lookahead

4. The root is the carry of the final sum. We'll swap it out with s (since there is no carry-in from the right non-existent subtree)

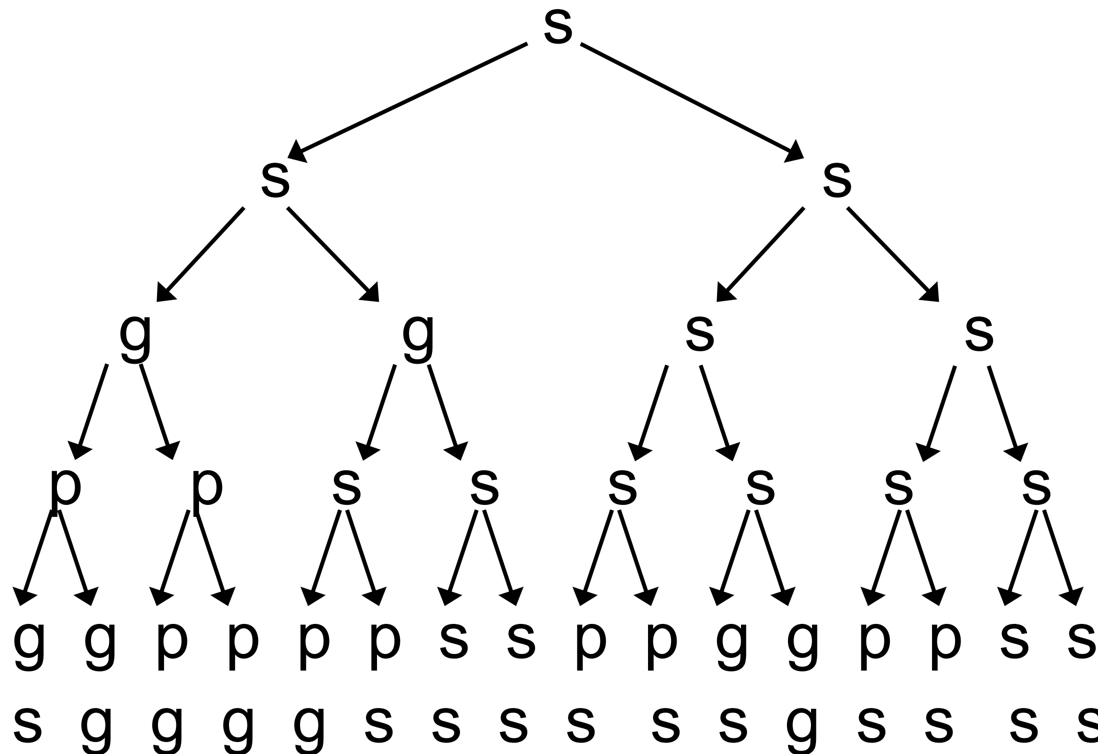


Remember:

Using a Tree – Log(N)

Carry Lookahead

5. Start pushing a node to its children. Note that leaves are remembering the first non-p value they see.



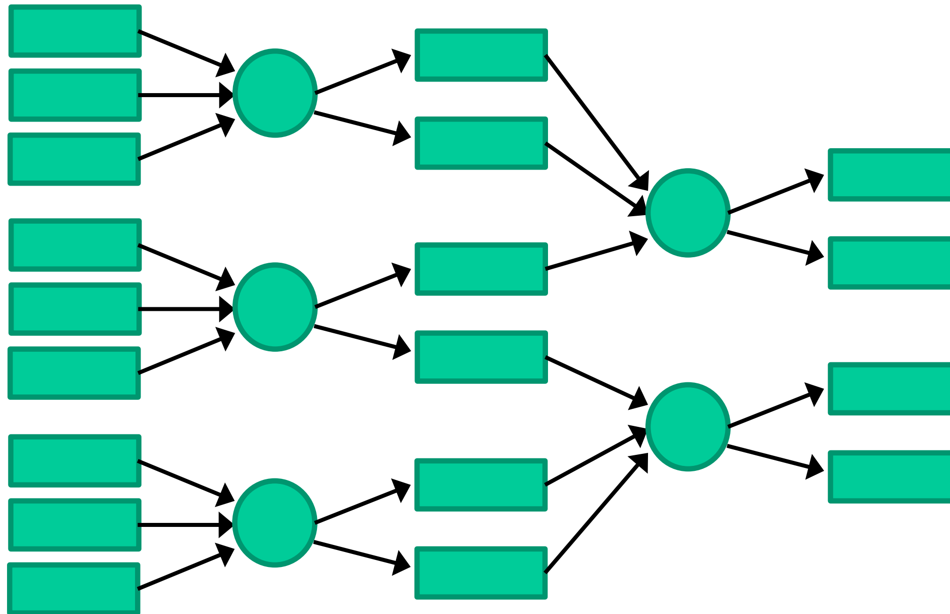
Adding N k-bit Numbers

- Can build a tree of adders (DaDianNao); logN additions; each addition is O(logk); therefore O(logN * logk)
- Instead, we'll do it in O(logN + logk) – called a Wallace Tree – used in the SNN vs. ANN study (MICRO'15)
- Based on the insight that adding 3 bits gives us 2 bits, i.e., we can add 3 numbers to produce two numbers.

A:		1	0	1	1	0	0	0	1
A':		0	1	1	0	1	1	0	1
A'':		1	0	1	0	0	1	0	0
D:		0	1	1	1	1	0	0	0
C:	1	0	1	0	0	1	0	1	

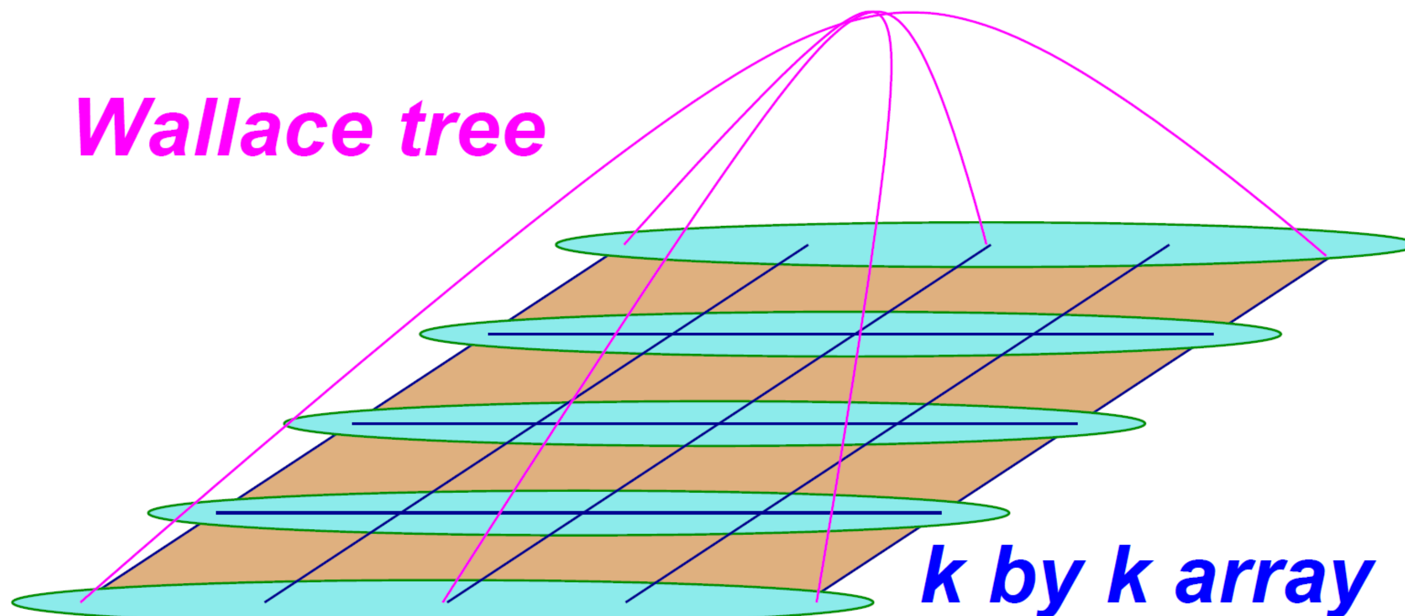
Wallace Tree

- Each level shrinks 3 inputs to 2 inputs in a single cycle; $\log_{3/2}N$ levels; adding the last 2 numbers takes logk



Multiplication of Two N-bit Numbers

- See typical multiplication example on next slide
- Essentially, adding N 2N-bit numbers – we know that's do-able in O(logN) time, but requires more than O(N) hardware (O(N^2) work)
- We'll try to do it with O(N) hardware in O(N) time (4N-1 steps)



Multiplication Example

Multiplying 7x6 = 42

```
            1   1   1
            1   1   0
    -------------------
            0   0   0
        1   1   1
    1   1   1
-----------------------
1   0   1   0   1   0
```

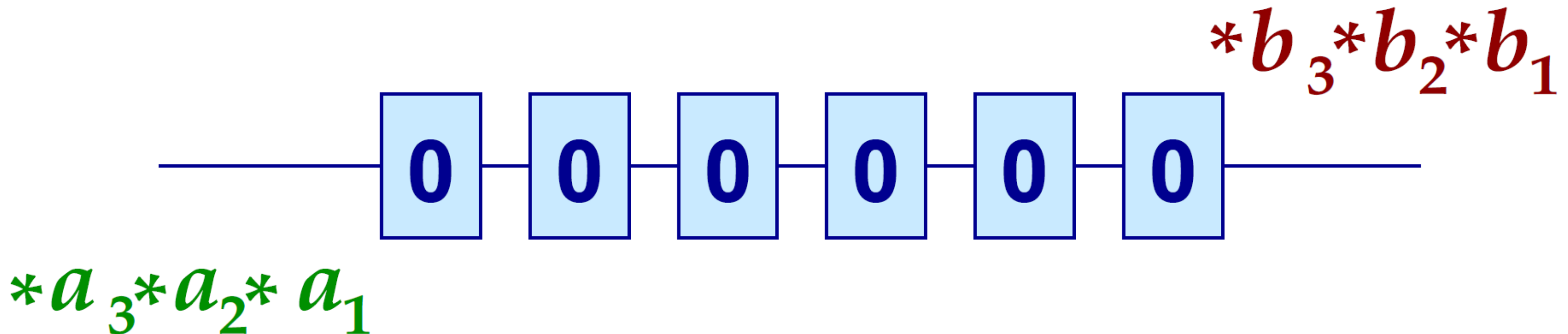
			b_3	b_2	b_1
			a_3	a_2	a_1
			a_1b_3	a_1b_2	a_1b_1
		a_2b_3	a_2b_2	a_2b_1	
	a_3b_3	a_3b_2	a_3b_1		
p_6	p_5	p_4	p_3	p_2	p_1

$$y_p = \sum_{i+j=p+1} a_i b_j.$$

This is the definition of a 1D conv.

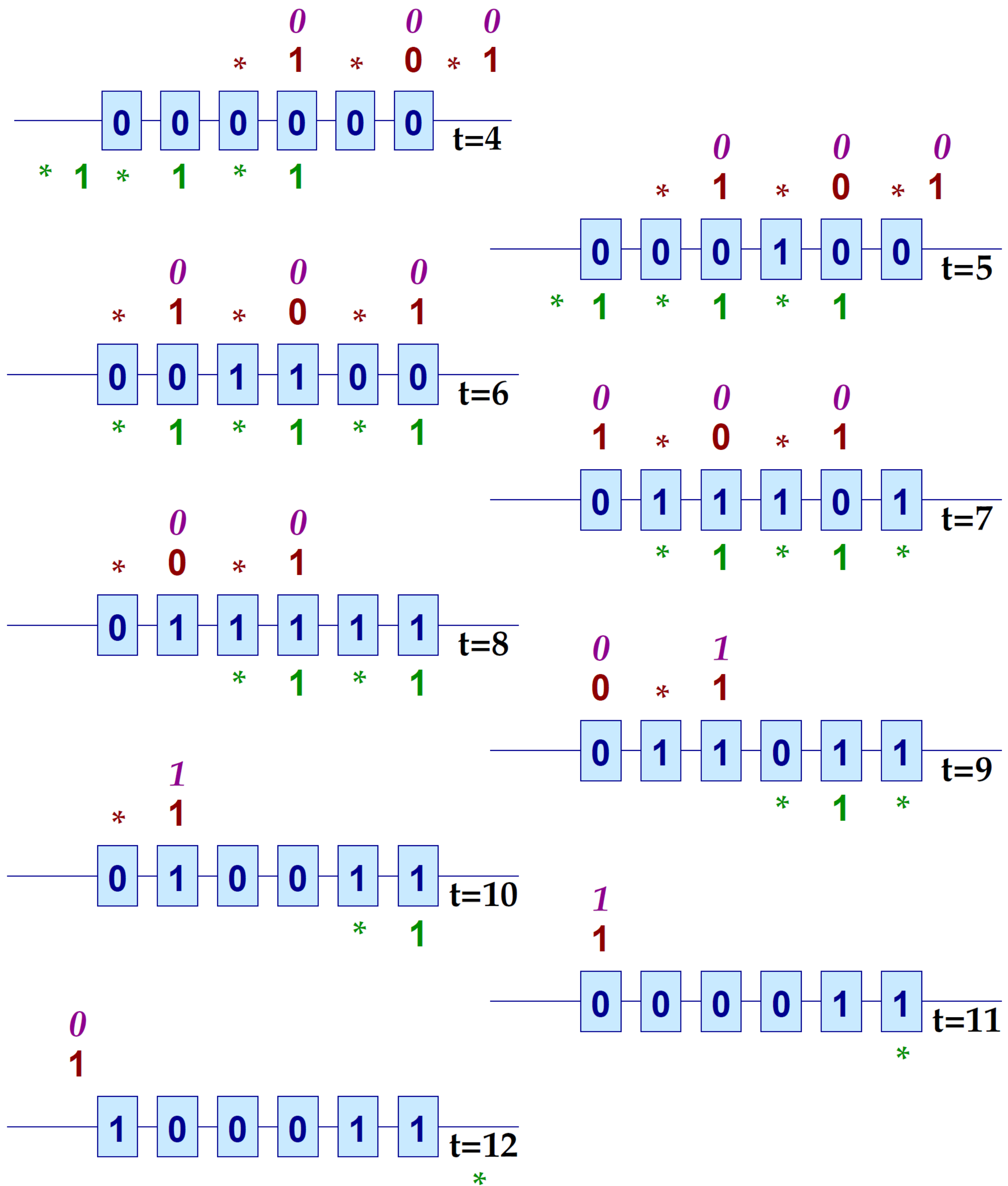
1D Convolution

- Need 2N processing elements to multiply 2 N-bit numbers
- Inputs are fed from left and right with gaps (why gaps?)
- Each processing element accumulates the corresponding bit of the product
- The generated carry has to also move leftward



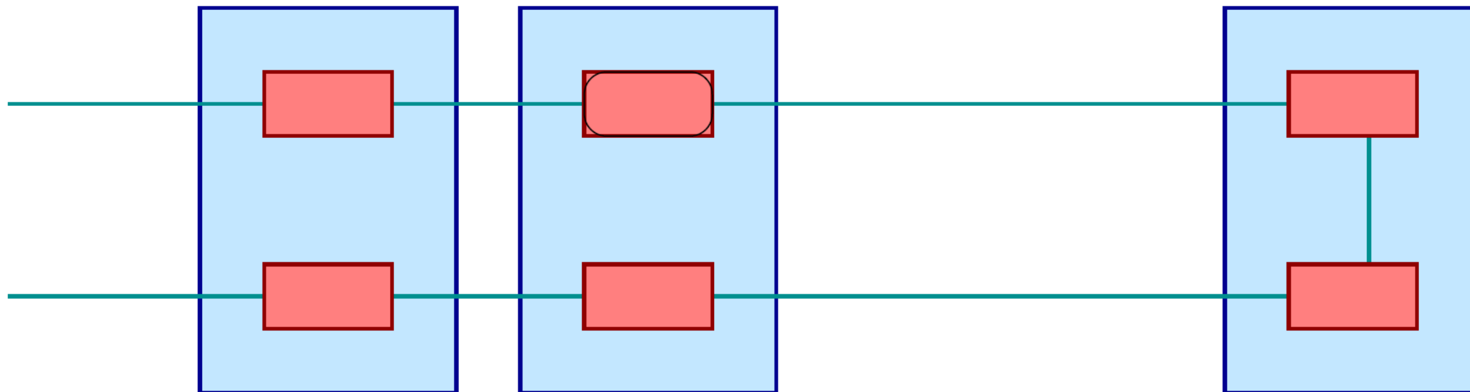
Example

Convolution of
5 (101) and 7 (111)



Analysis

- 4N-1 steps because a 2N input has to go across 2N PEs
- But every PE is idle in alternate cycles – can improve efficiency by interleaving two multiplications or by wrapping the linear array into two rows



- This is also one approach for computing a 1D convolution; extending to 2D and 3D convolutions is not straightforward

Solving Systems of Equations

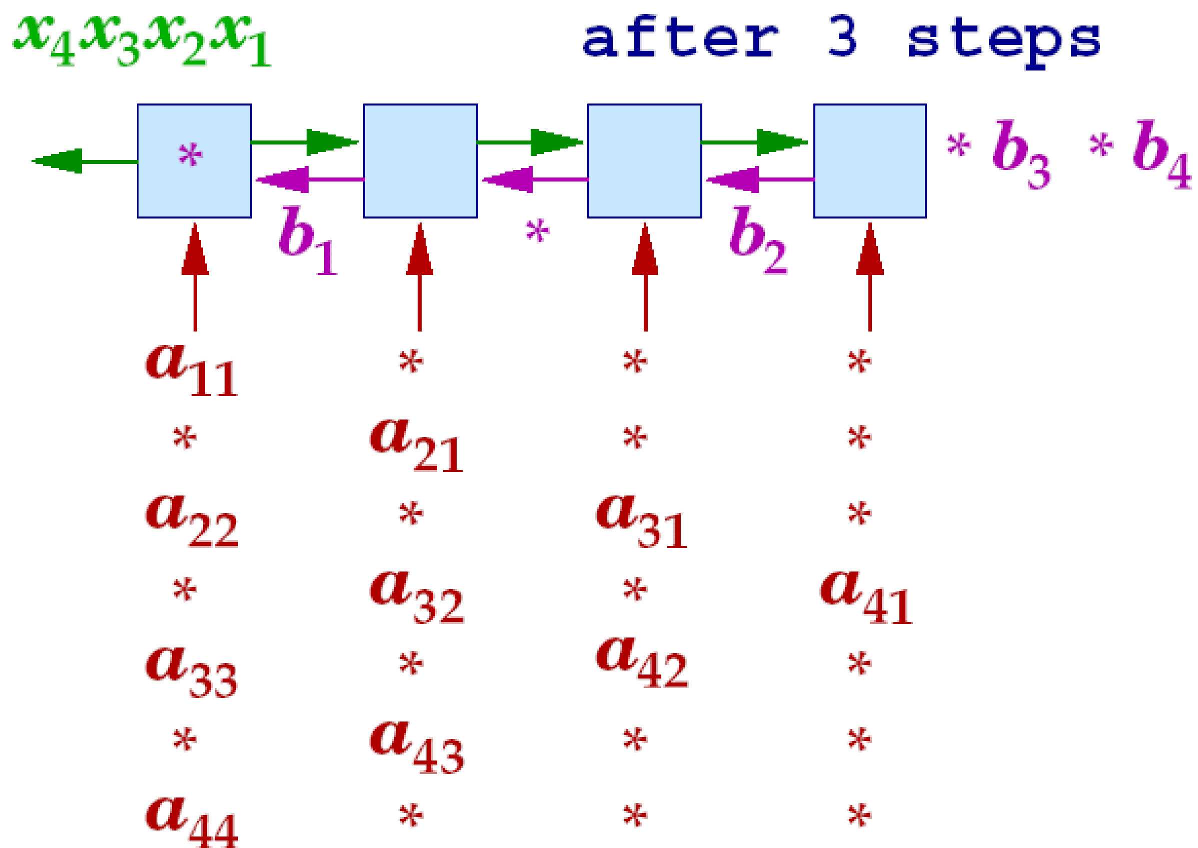
- Given an N x N lower triangular matrix A and an N-vector *b*, solve for *x*, where A*x* = *b* (assume solution exists)

$a_{11}x_1 = b_1$
$a_{21}x_1 + a_{22}x_2 = b_2$, and so on…

Define $t_1 =_{\text{def}} b_1$, $t_i =_{\text{def}} b_i - \sum_{j=1}^{i-1} a_{ij}x_j, 2 \leq i \leq N$. Then $x_i = t_i/a_{ii}$.

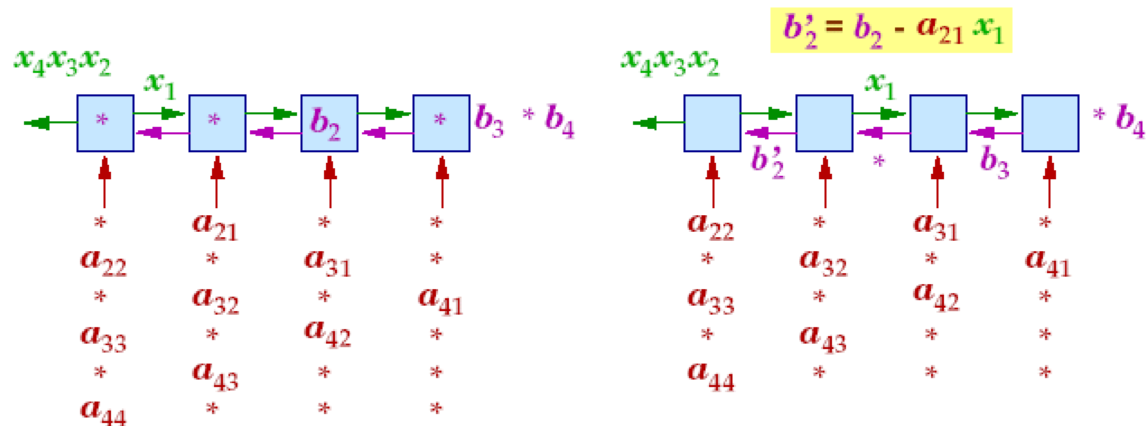
Equation Solver

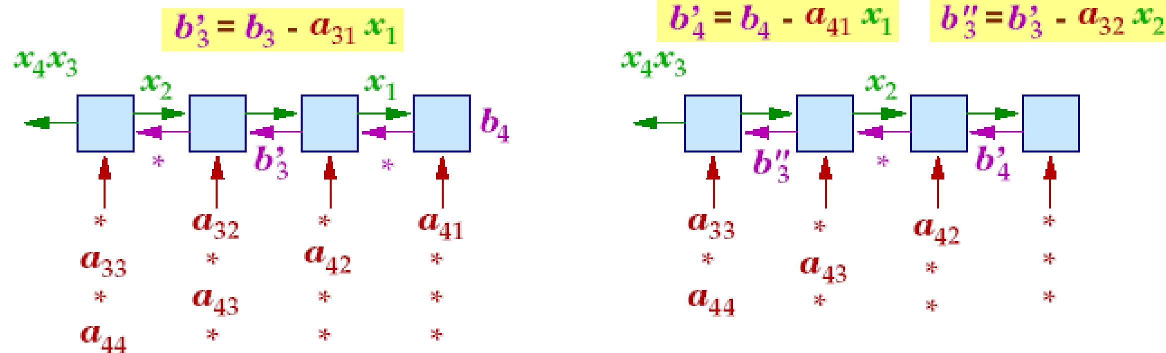
Define $t_1 =_{\text{def}} b_1$, $t_i =_{\text{def}} b_i - \sum_{j=1}^{i-1} a_{ij} x_j, 2 \leq i \leq N$. Then $x_i = t_i / a_{ii}$.



Equation Solver Example

- When an *x, b,* and *a* meet at a cell, *ax* is subtracted from *b*
- When *b* and *a* meet at cell 1, *b* is divided by *a* to become *x*





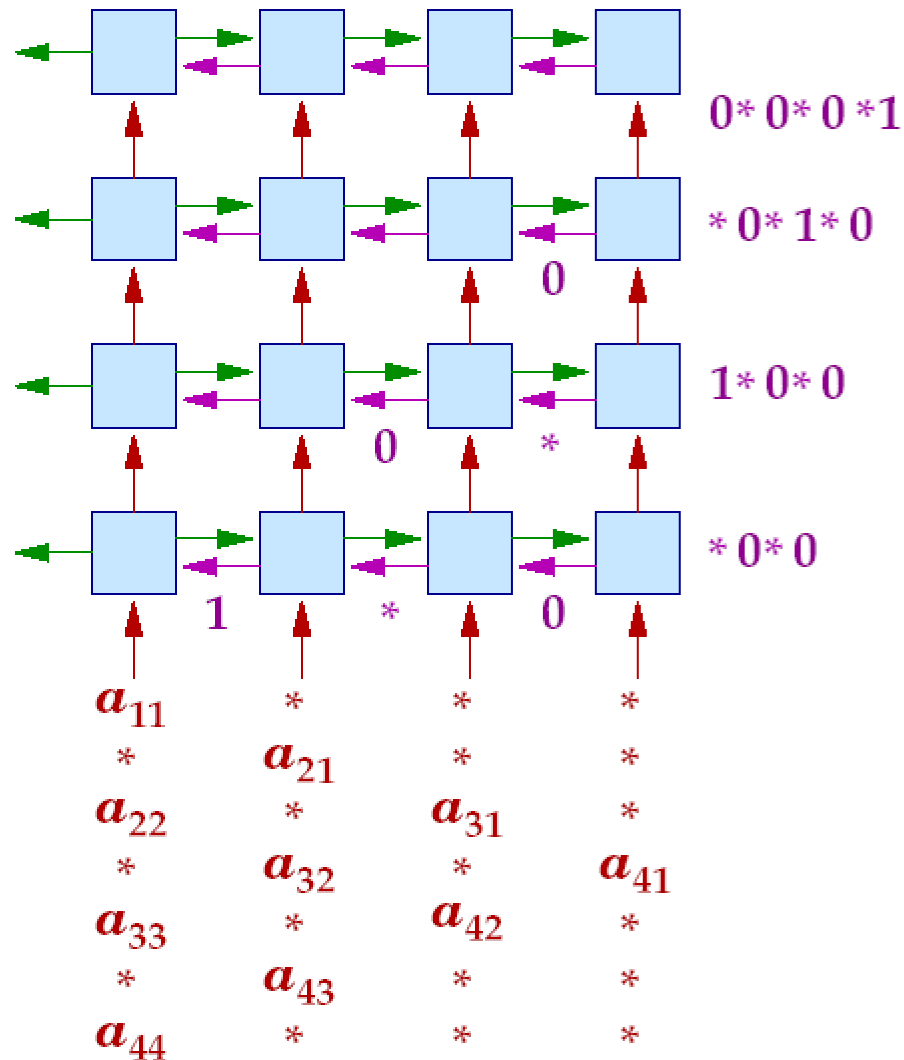
Complexity

- Time steps = 2N – 1
- Speedup = O(N), efficiency = O(1)
- Note that half the processors are idle every time step – can improve efficiency by solving two interleaved equation systems simultaneously or by wrapping the linear array

Inverting Triangular Matrices

- Finding X, such that AX = I, where A is a lower triangular matrix
- For each row j, $A x_j = e_j$, where e_j is the jth unit vector (0,…, 0, 1, 0,…, 0) and x_j is the jth row of matrix X
- Simple extension of the earlier algorithm – it can be applied to compute each row individually

Inverting Triangular Matrices



Gaussian Elimination

- Solving for x, where Ax=b and A is a nonsingular matrix
- Note that $A^{-1}Ax = A^{-1}b = x$; keep applying transformations to A such that A becomes I ; the same transformations applied to b will result in the solution for x
- Sequential algorithm steps:
  - Pick a row where the first (i^{th}) element is non-zero and normalize the row so that the first (i^{th}) element is 1
  - Subtract a multiple of this row from all other rows so that their first (i^{th}) element is zero
  - Repeat for all i

Sequential Example

$$\begin{pmatrix} 2 & 4 & -7 \\ 3 & 6 & -10 \\ -1 & 3 & -4 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 6 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & -7/2 \\ 3 & 6 & -10 \\ -1 & 3 & -4 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} 3/2 \\ 4 \\ 6 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & -7/2 \\ 0 & 0 & 1/2 \\ -1 & 3 & -4 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} 3/2 \\ -1/2 \\ 6 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & -7/2 \\ 0 & 0 & 1/2 \\ 0 & 5 & -15/2 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} 3/2 \\ -1/2 \\ 15/2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 & -7/2 \\ 0 & 5 & -15/2 \\ 0 & 0 & 1/2 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} 3/2 \\ 15/2 \\ -1/2 \end{pmatrix}$$

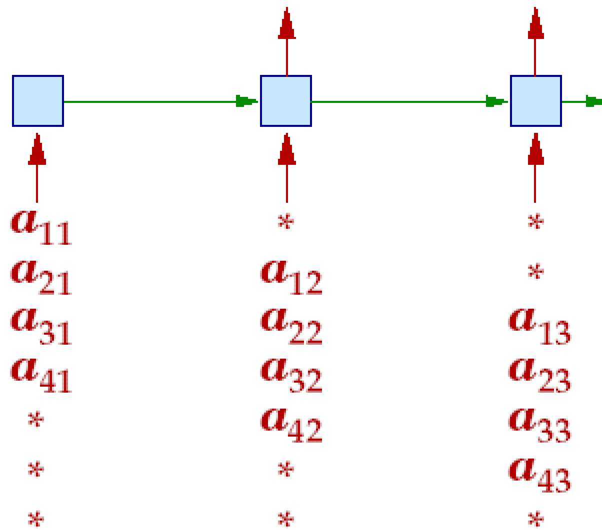
$$\begin{pmatrix} 1 & 2 & -7/2 \\ 0 & 1 & -3/2 \\ 0 & 0 & 1/2 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} 3/2 \\ 3/2 \\ -1/2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & -1/2 \\ 0 & 1 & -3/2 \\ 0 & 0 & 1/2 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} -3/2 \\ 3/2 \\ -1/2 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & -1/2 \\ 0 & 1 & -3/2 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} -3/2 \\ 3/2 \\ -1 \end{pmatrix}$$

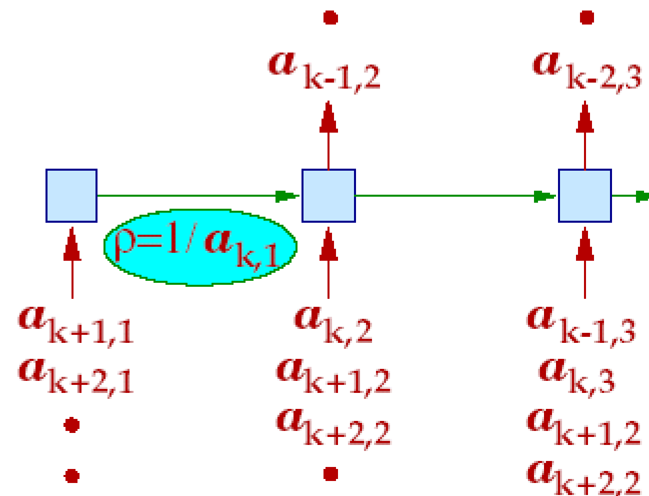
$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} -2 \\ 0 \\ -1 \end{pmatrix}$$

Algorithm Implementation

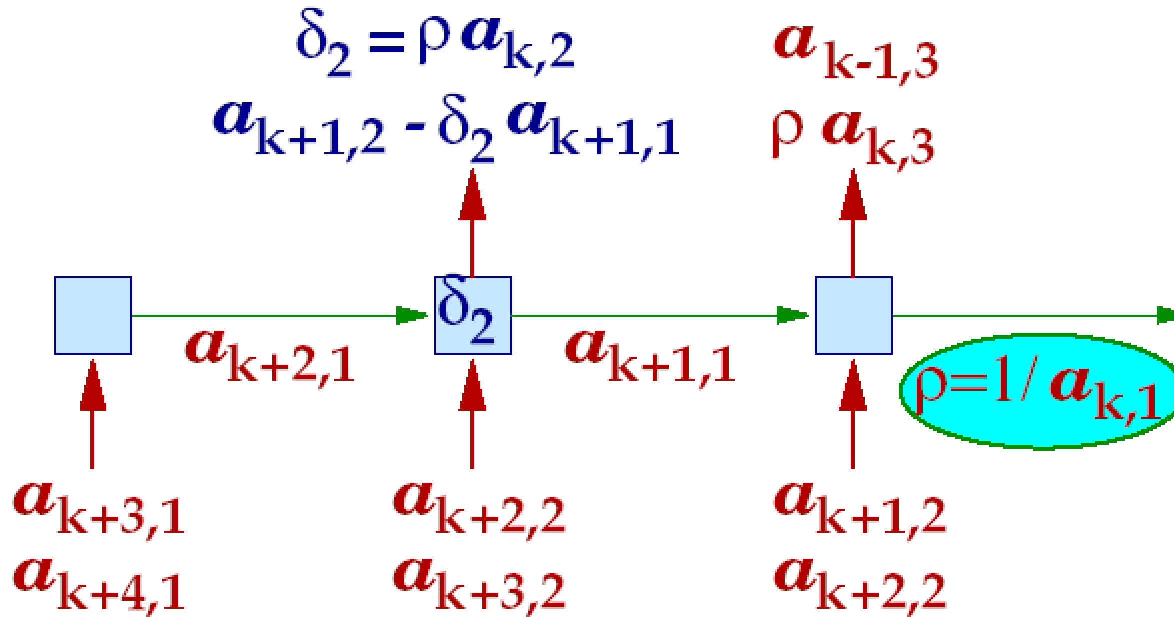


- The matrix is input in staggered form
- The first cell discards inputs until it finds a non-zero element (the pivot row)

- The inverse ρ of the non-zero element is now sent rightward
- ρ arrives at each cell at the same time as the corresponding element of the pivot row



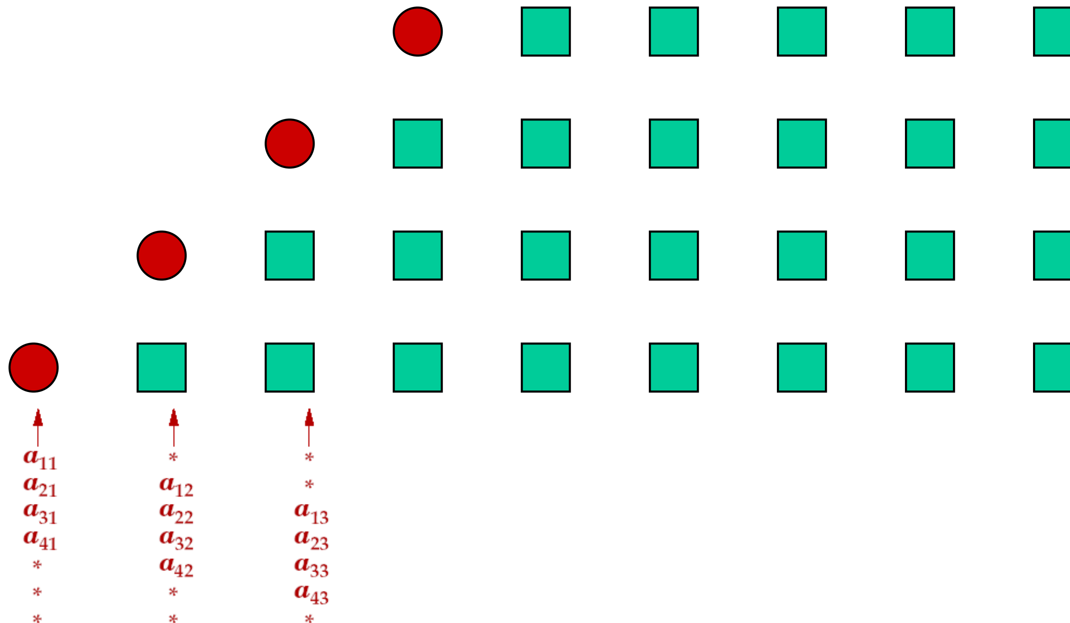
Algorithm Implementation



- Each cell stores $\delta_i = \rho\, a_{k,I}$ – the value for the normalized pivot row
- This value is used when subtracting a multiple of the pivot row from other rows
- What is the multiple? It is $a_{j,1}$
- How does each cell receive $a_{j,1}$? It is passed rightward by the first cell
- Each cell now outputs the new values for each row
- The first cell only outputs zeroes and these outputs are no longer needed

Algorithm Implementation

- The outputs of all but the first cell must now go through the remaining algorithm steps
- A triangular matrix of processors efficiently implements the flow of data
- Number of time steps?
- Can be extended to compute the inverse of a matrix



Graph Algorithms

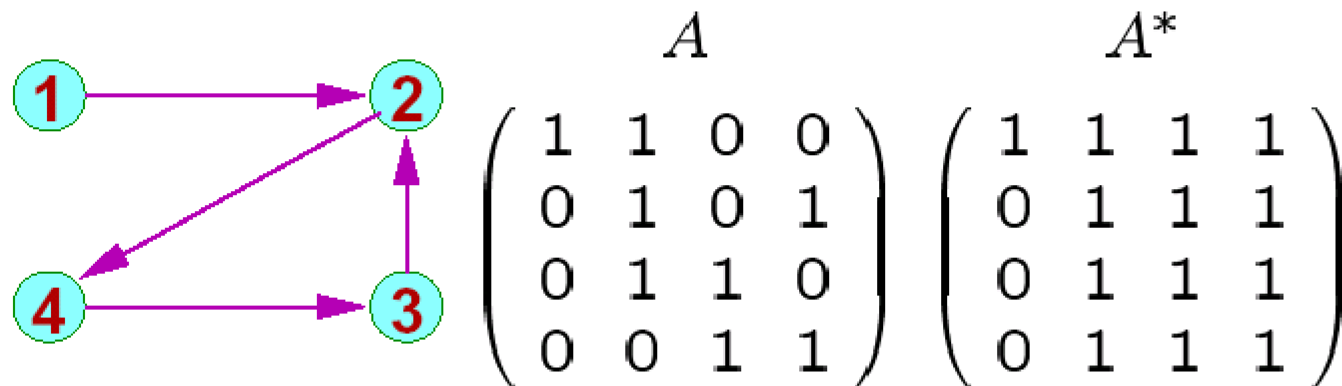
$G = (V, E)$: a directed graph, $V = \{1, \ldots, N\}$

The *adjacency matrix* $A = (a_{ij})$ of G is

$$a_{ij} = \begin{cases} 1 & \text{if either } (i,j) \in E \text{ or } i = j, \\ 0 & \text{otherwise.} \end{cases}$$

The transitive closure of G is $G^* = (V, E^*)$,

$$E^* = \{(i,j) \mid j \text{ is reachable from } i \text{ in } G\}.$$



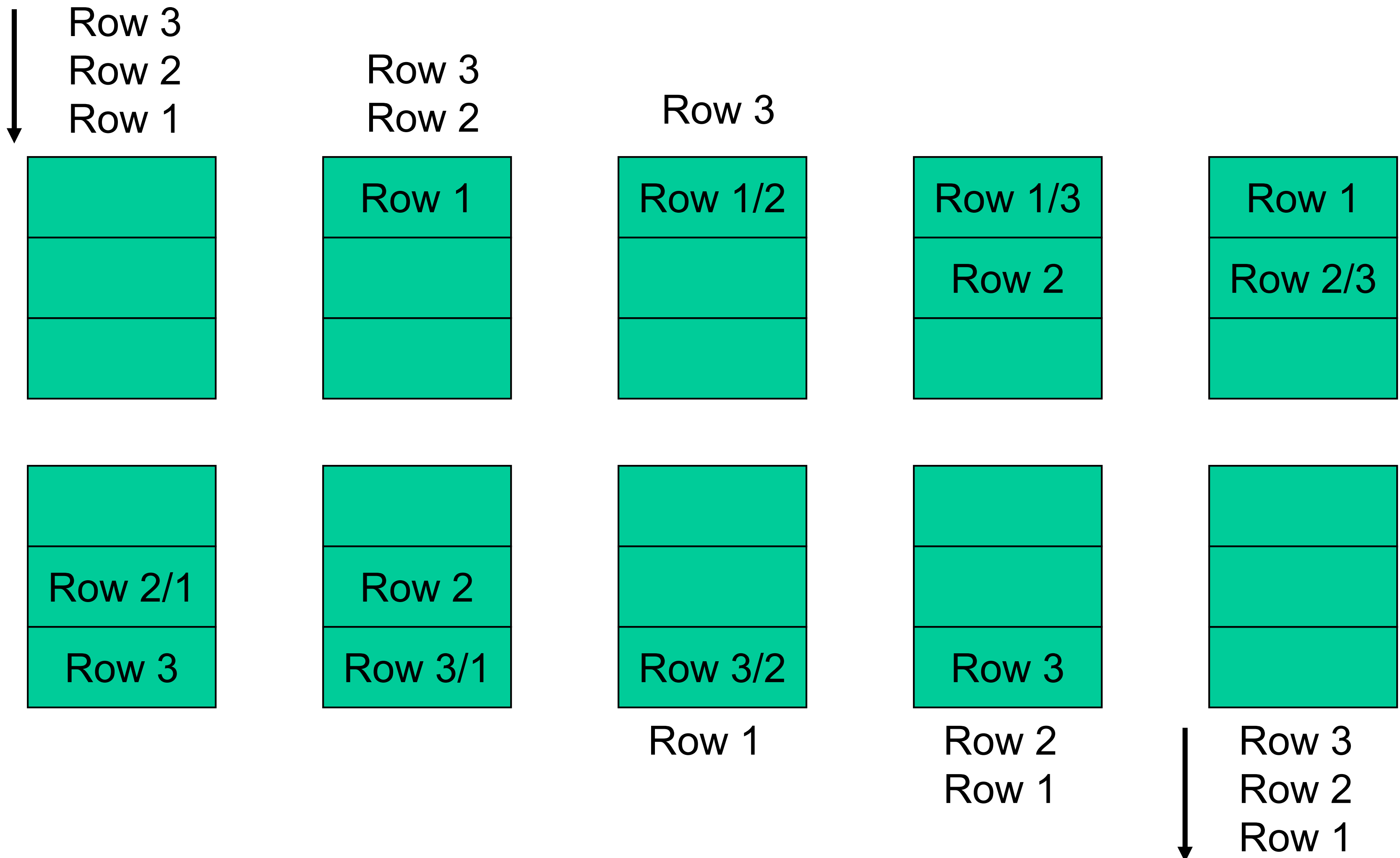
Floyd Warshall Algorithm

$A^{(k)} =_{\text{def}} (a_{ij}^{(k)})$, where for each $k, 0 \leq k \leq N$, $a_{ij}^{(k)} = 1$ if j is reachable from i *passing through* only nodes $\leq k$ and 0 otherwise.

Then $A^{(N)} = A^*$, $A^{(0)} = A$, and for all $k \geq 1$,

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} \vee \left(a_{ik}^{(k-1)} \wedge a_{kj}^{(k-1)} \right).$$

Implementation on 2d Processor Array



Algorithm Implementation

- Diagonal elements of the processor array can broadcast to the entire row in one time step (if this assumption is not made, inputs will have to be staggered)

- A row sifts down until it finds an empty row – it sifts down again after all other rows have passed over it

- When a row passes over the 1st row, the value of a_{i1} is broadcast to the entire row – a_{ij} is set to 1 if $a_{i1} = a_{1j} = 1$ – in other words, the row is now the ith row of $A^{(1)}$

- By the time the kth row finds its empty slot, it has already become the kth row of $A^{(k-1)}$

Algorithm Implementation

- When the ith row starts moving again, it travels over rows a_k ($k > i$) and gets updated depending on whether there is a path from i to j via vertices $< k$ (and including k)

Shortest Paths

- Given a graph and edges with weights, compute the weight of the shortest path between pairs of vertices
- Can the transitive closure algorithm be applied here?

Shortest Paths Algorithm

$D = (d_{ij})$: the distance matrix, where $d_{ij} = \infty$ if there is no edge from i to j
Define $D^{(k)} = (d_{ij}^{(k)})$, where $d_{ij}^{(k)}$ is the length of the shortest path from i to j that passes through only nodes $\leq k$.

Then we have only to compute $D^{(N)}$. Note $D^{(0)} = D$ and for all $k \geq 1$,

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}).$$

The above equation is very similar to that in transitive closure

References

- "Introduction to Parallel Algorithms and Architectures," Leighton
- Figure credits: Mitsu Ogihara