COMP 322: Fundamentals of Parallel Programming

Lecture 37: Algorithms based on Parallel Prefix (Scan) operations (contd)

Instructors: Vivek Sarkar, Mack Joyner
Department of Computer Science, Rice University
{vsarkar, mjoyner}@rice.edu

Acknowledgements:

- Book chapter on "Prefix Sums and Their Applications", Guy E. Blelloch, CMU
- Slides on "Parallel prefix adders", Kostas Vitoroulis, Concordia University

http://comp322.rice.edu/

Recap of Parallel Prefix and Scan

- In Lecture 13, we learned how to compute the parallel prefix sum (set of partial sums), X, for an input array A of size *n* with *WORK* = *O(n)* and *CPL* = *O(log n)*
  - *For P processors, T_P = O(n/P + log (P))*
- In Lecture 36, we learned the scan operator which generalizes parallel prefix sum to parallel prefix for any binary associative operator, ⊕ (need not be restricted to xor).
- Specifically, scan takes a binary associative operator ⊕, and an array of n elements $[A_0, A_1, ..., A_{n-1}]$ as input, and returns array $[A_0, (A_0 \oplus A_1), ..., (A_0 \oplus A_1 \oplus ... \oplus A_{n-1})]$ as output. The output from scan can also be specified as a recurrence:

$$x_i = \begin{cases} a_0 & i = 0 \\ x_{i-1} \oplus a_i & 0 < i < n, \end{cases}$$

- From Lecture 13, we see that scan can also be performed with *WORK* = *O(n)* and *CPL* = *O(log n)*
- Note that any constant sequence of scan operators can also be performed with *WORK* = *O(n)* and *CPL* = *O(log n)*

Parallelizing Prefix Sum (Lecture 13)

Observation: each prefix sum can be decomposed into reusable terms of power-of-2-size e.g.

$$\begin{aligned} X[6] &= A[0]+A[1]+A[2]+A[3]+A[4]+A[5]+A[6] \\ &= (A[0]+A[1]+A[2]+A[3])+(A[4]+A[5])+A[6] \end{aligned}$$

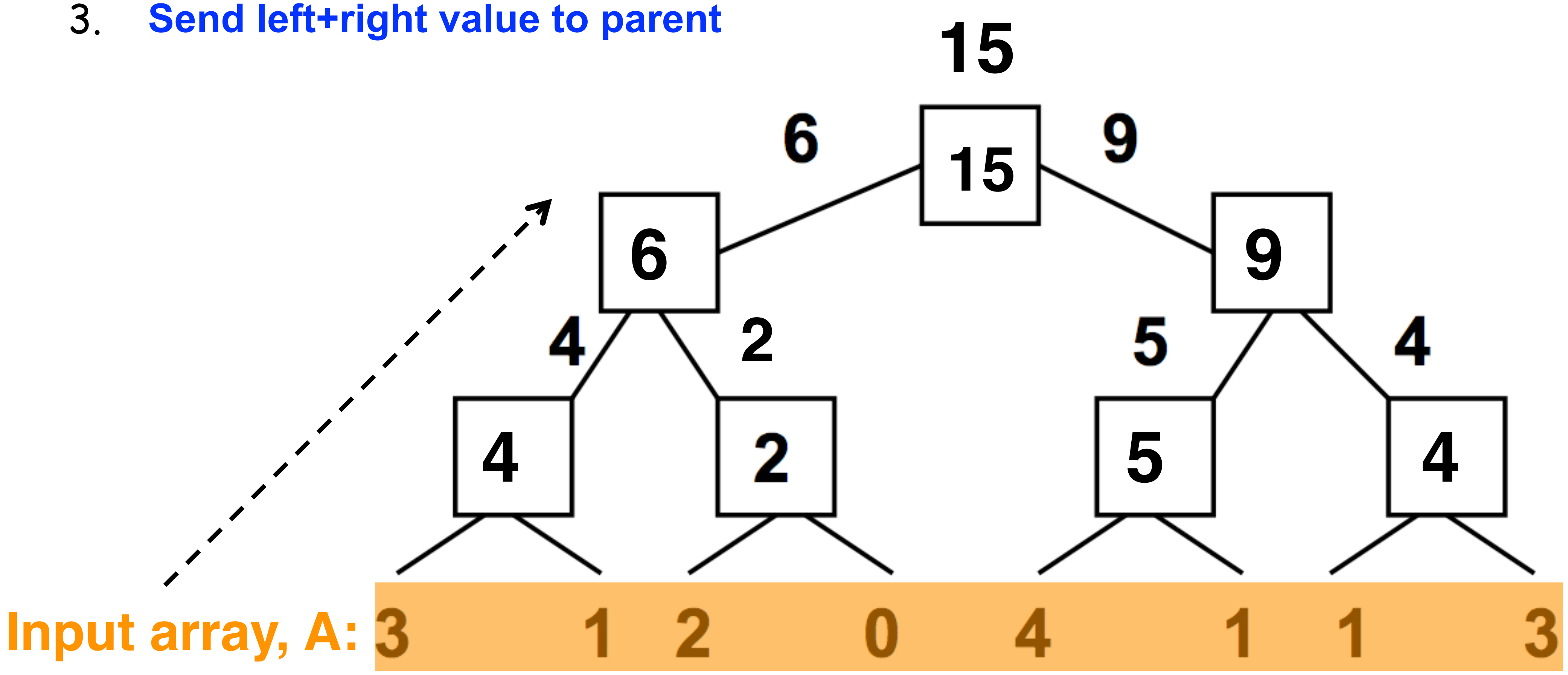
Approach:

- Combine reduction tree idea from Parallel Array Sum with partial sum idea from Sequential Prefix Sum
- Use an “upward sweep” to perform parallel reduction, while storing partial sum terms in tree nodes
- Use a “downward sweep” to compute prefix sums while reusing partial sum terms stored in upward sweep

Parallel Prefix Sum: Upward Sweep (while calling scan recursively)

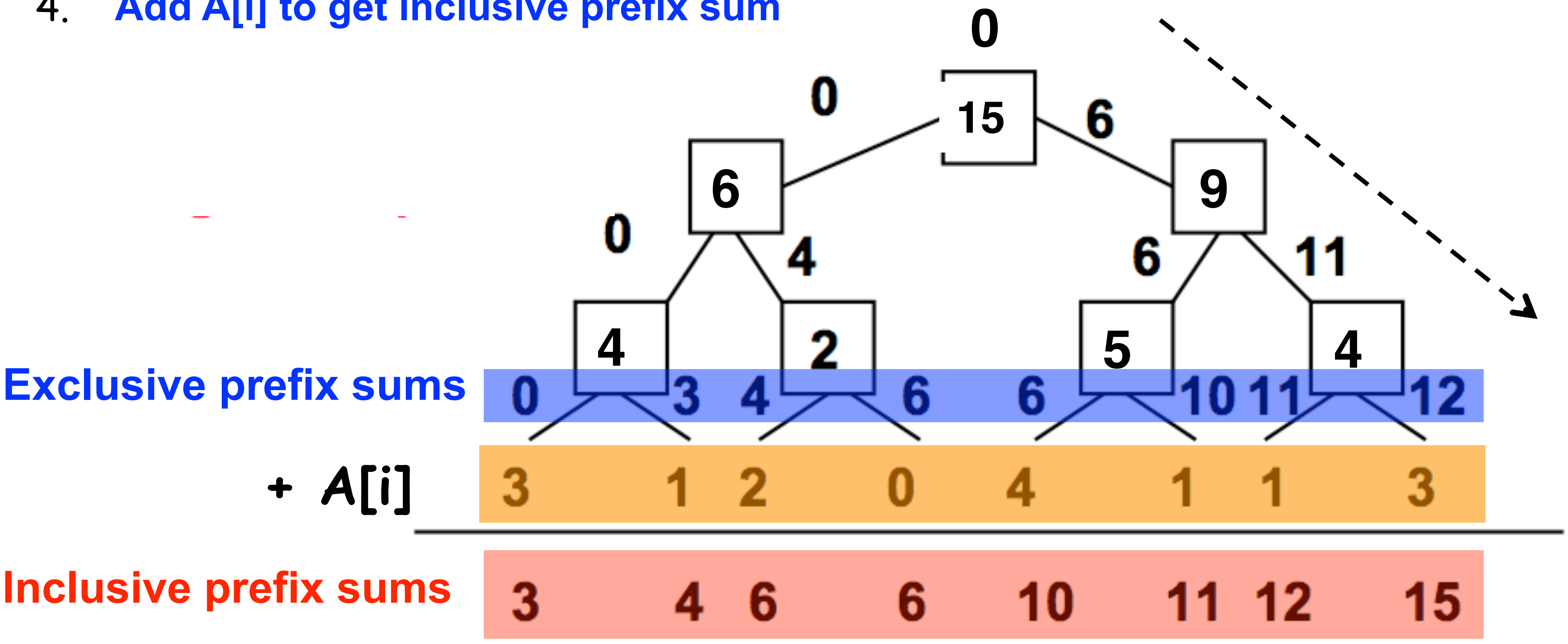
Upward sweep is just like Parallel Reduction, except that partial sums are also stored along the way

1. Receive values from left and right children
2. Compute left+right and store in box
3. Send left+right value to parent



Parallel Prefix Sum: Downward Sweep (while returning from recursive calls to scan)

1. Receive value from parent (root receives 0)
2. Send parent's value to LEFT child (prefix sum for elements to left of left child's subtree)
3. Send parent's value+ left child's box value to RIGHT child (prefix sum for elements to left of right child's subtree)
4. Add A[i] to get inclusive prefix sum



Worksheet #36 problem statement: Parallelizing the Split step in Radix Sort

The Radix Sort algorithm loops over the bits in the binary representation of the keys, starting at the lowest bit, and executes a split operation for each bit as shown below. The split operation packs the keys with a 0 in the corresponding bit to the bottom of a vector, and packs the keys with a 1 to the top of the same vector. It maintains the order within both groups. The sort works because each split operation sorts the keys with respect to the current bit and maintains the sorted order of all the lower bits. Your task is to show how the split operation can be performed in parallel using scan operations, and to explain your answer.

```
                          [101 111 011 001 100 010 111 010]
1.A =                     [5  7  3  1  4  2  7  2]
2.A⟨0⟩ =                  [1  1  1  1  0  0  1  0] //lowest bit
3.A←split(A,A⟨0⟩) =       [4  2  2  5  7  3  1  7]
4.A⟨1⟩ =                  [0  1  1  0  1  1  0  1] // middle bit
5.A←split(A,A⟨1⟩) =       [4  5  1  2  2  7  3  7]
6.A⟨2⟩ =                  [1  1  0  0  0  1  0  1] // highest bit
7.A←split(A,A⟨2⟩) =       [1  2  2  3  4  5  7  7]
```

Worksheet #36 solution: Parallelizing the Split step in Radix Sort

```
procedure split(A, Flags)
  I-down ← prescan(+, not(Flags)) // prescan = exclusive prefix sum
  I-up   ← rev(n - scan(+, rev(Flags)) // rev = reverse
  in parallel for each index $i$
    if (Flags[$i$])
      Index[$i$] ← I-up[$i$]
    else
      Index[$i$] ← I-down[$i$]
  result ← permute(A, Index)
```

A	=	[5	7	3	1	4	2	7	2]
Flags	=	[1	1	1	1	0	0	1	0]
I-down	=	[0	0	0	0	$\boxed{0}$	$\boxed{1}$	2	$\boxed{2}$]
I-up	=	[$\boxed{3}$	$\boxed{4}$	$\boxed{5}$	$\boxed{6}$	7	7	$\boxed{7}$	8]
Index	=	[3	4	5	6	0	1	7	2]
permute(A, Index)	=	[4	2	2	5	7	3	1	7]

FIGURE 1.9
The split operation packs the elements with a 0 in the corresponding flag position to the bottom of a vector, and packs the elements with a 1 to the top of the same vector. The permute writes each element of A to the index specified by the corresponding position in Index.

Binary Addition

$$\begin{array}{rcccc} & {}^{c_3}x_3 & {}^{c_2}x_2 & {}^{c_1}x_1 & {}^{c_0}x_0 \\ + & & & & \\ & y_3 & y_2 & y_1 & y_0 \\ \hline s_4 & s_3 & s_2 & s_1 & s_0 \end{array}$$

This is the pen and paper addition of two 4-bit binary numbers **x** and **y**.
c represents the generated carries.
s represents the produced sum bits.

A **stage** of the addition is the set of **x** and **y** bits being used to produce the appropriate sum and carry bits. For example the highlighted bits **x_2**, **y_2** constitute **stage 2** which generates carry **c_2** and sum **s_2** .

Each stage *i* adds bits a_i, b_i, c_{i-1} and produces bits s_i, c_i
The following hold:

a_i	b_i	c_i	Comment:	Formal definition:	
0	0	0	The stage "kills" an incoming carry.	"Kill" bit:	$k_i = \overline{x_i + y_i}$
0	1	c_{i-1}	The stage "propagates" an incoming carry	"Propagate" bit:	$p_i = x_i \oplus y_i$
1	0	c_{i-1}	The stage "propagates" an incoming carry		
1	1	1	The stage "generates" a carry out	"Generate" bit:	$g_i = x_i \bullet y_i$

Binary Addition

a_i	b_i	c_i	Comment:	Formal definition:	
0	0	0	The stage "kills" an incoming carry.	"Kill" bit:	$k_i = \overline{x_i + y_i}$
0	1	c_{i-1}	The stage "propagates" an incoming carry	"Propagate" bit:	$p_i = x_i \oplus y_i$
1	0	c_{i-1}	The stage "propagates" an incoming carry		
1	1	1	The stage "generates" a carry out	"Generate" bit:	$g_i = x_i \bullet y_i$

The carry c_i generated by a stage *i* is given by the equation:

$$c_i = g_i + p_i \cdot c_{i-1} = x_i \cdot y_i + (x_i \oplus y_i) \cdot c_{i-1}$$

This equation can be simplified to:

$$c_i = x_i \cdot y_i + (x_i + y_i) \cdot c_{i-1} = g_i + a_i \cdot c_{i-1}$$

The "a_i" term in the equation being the "alive" bit.
The later form of the equation uses an OR gate instead of an XOR which is a more efficient gate when implemented in CMOS technology. Note that:

$$a_i = \overline{k_i}$$

Where k_i is the "kill" bit defined in the table above.

Binary addition as a prefix sum problem.

- We define a new operator: “ $\circ$ ”
- Input is a vector of pairs of ‘propagate’ and ‘generate’ bits:

$$(g_n, p_n)(g_{n-1}, p_{n-1})\ldots(g_0, p_0)$$

- Output is a new vector of pairs:

$$(G_n, P_n)(G_{n-1}, P_{n-1})\ldots(G_0, P_0)$$

- Each pair of the output vector is calculated by the following definition:

$$(G_i, P_i) = (g_i, p_i) \circ (G_{i-1}, P_{i-1})$$

Indicates that “o” is associative and amenable to parallel prefix algorithm

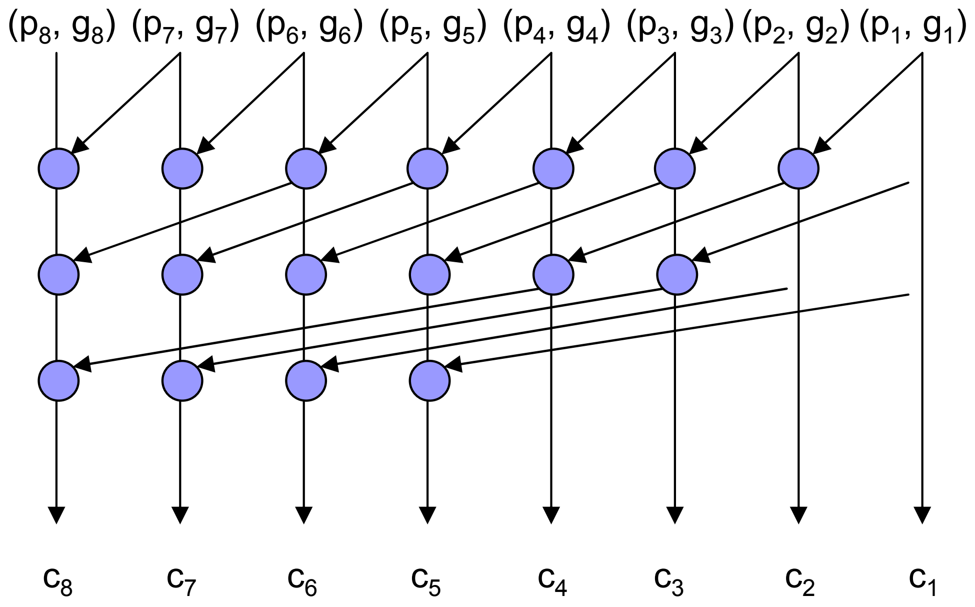
$$Where:$$

$$(G_0, P_0) = (g_0, p_0)$$

$$(g_x, p_x) \circ (g_y, p_y) = (g_x + p_x \cdot g_y, p_x \cdot p_y)$$

$$with \quad +, \cdot \quad being\ the\ OR, AND\ operations$$

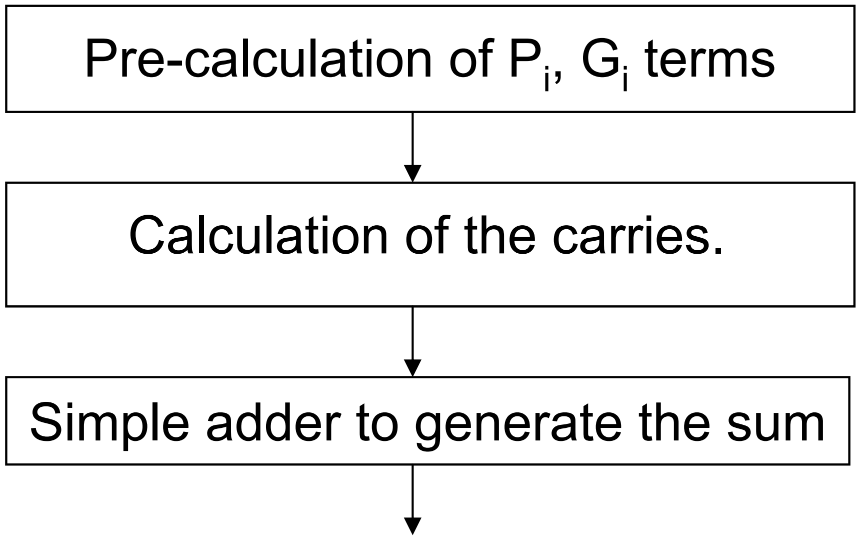
1973: Kogge-Stone adder


(p_8, g_8) (p_7, g_7) (p_6, g_6) (p_5, g_5) (p_4, g_4) (p_3, g_3) (p_2, g_2) (p_1, g_1)
c_8 c_7 c_6 c_5 c_4 c_3 c_2 c_1

- The Kogge-Stone adder has:
  - Low depth
  - High node count (implies more area).
  - Minimal fan-out of 1 at each node (implies faster performance).

Summary

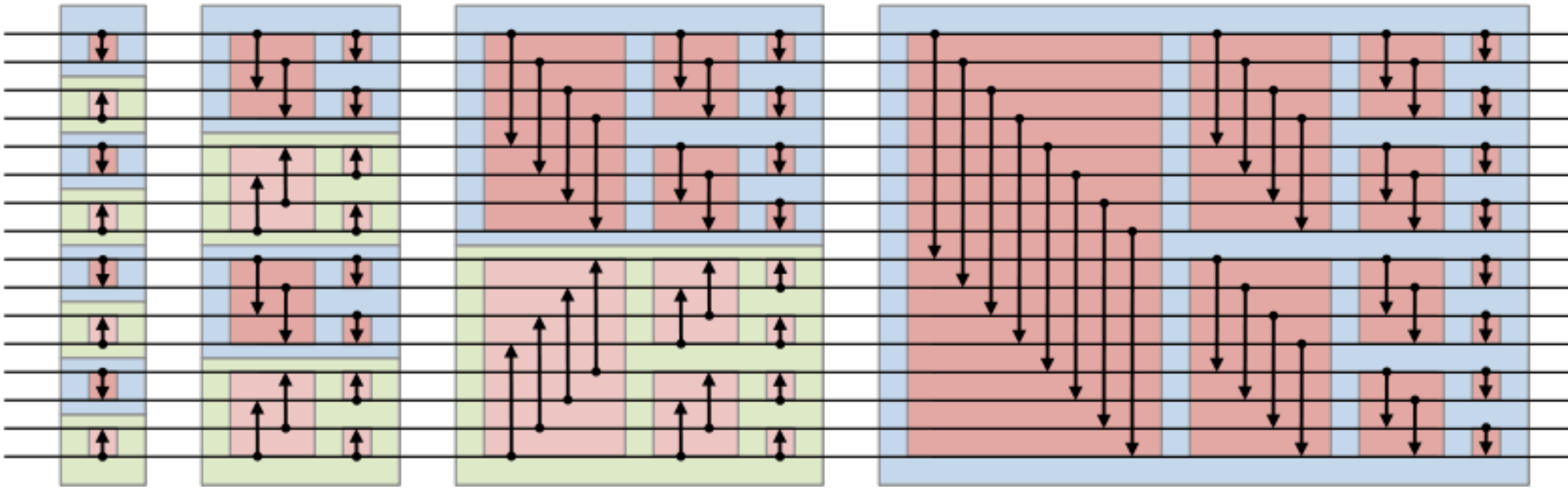
- A parallel prefix adder can be seen as a 3-stage process:


Pre-calculation of P_i, G_i terms
Calculation of the carries.
Simple adder to generate the sum

- There exist various architectures for the carry calculation part.
- Trade-offs in these architectures involve the
  - area of the adder
  - its depth
  - the fan-out of the nodes
  - the overall wiring network.

Parallel Algorithms, Computation Graphs, and Circuits

- Today's lecture shows that parallel algorithms, computation graphs, and circuits represent different approaches to *parallel computational thinking*
- A parallel algorithm unfolds into a computation graph when executing
- A circuit represents an "unrolled" computation graph in hardware e.g., see bitonic sorting network in https://en.wikipedia.org/wiki/Bitonic_sorter



And the same principles can be applied to distributed computing too, e.g., MPI_Scan

```
main(int argc, char **argv){
  int mype, ierr;
  ierr = MPI_Init(&argc, &argv);
  ierr = MPI_Comm_rank(WCOMM, &mype);

  double l = 0.5;
  double exp_sum = 0;
  double exp_pdf_i = 0.0;
  double exp_cdf_i = 0.0;
  double DIV_CONST = 2.0;
  int i;

  for(i = 0; i < NUMPTS; i++)
  {
    if (i == mype)
    {
      exp_pdf_i = l*exp(-l * ((double) i) / DIV_CONST);
    }
  }
//Calculate the cumulative frequency histogram (exp_cdf_i) from exp_pdf_i
  ierr = MPI_Scan( \
    &exp_pdf_i, &exp_cdf_i, 1, MPI_DOUBLE, MPI_SUM, WCOMM);

  for (i = 0; i < NUMPTS; i++)
  {
    if (i == mype)
    {
      printf("process %d: cumulative sum = %lf\n", \
             mype, exp_cdf_i);
    }
  }
```

Announcements & Reminders

- HW5 is now available
    - Due April 21st, with automatic extension until May 1st
- We are making an optional quiz available for Unit 10
    - Also due April 21st, with automatic extension until May 1st
    - Optional, but we will pick the best 9 of your 10 quiz scores
    - As always, please post any issues with quiz questions on Piazza
- Two more lectures + grand finale at 3pm on Friday
    - April 19th, 1pm: GPU Computing
    - April 21st, 1pm: Course review (scope of Exam 2) in class
    - April 21st, 3pm: COMP 322 grand finale in DH 3092 (group office hours time slot, but cake will be served!)