

COMP 322: Parallel and Concurrent Programming

Lecture 37: Algorithms Based on Parallel Prefix (Scan) Operations

Mack Joyner
mjoyner@rice.edu

<http://comp322.rice.edu>



Formalizing Parallel Prefix: Scan operations

- The *i-scan* operation is an inclusive parallel prefix sum operation.
- The scan operator was introduced in APL in the 1960's, and has been popularized recently in more modern languages, most notably the NESL project in CMU



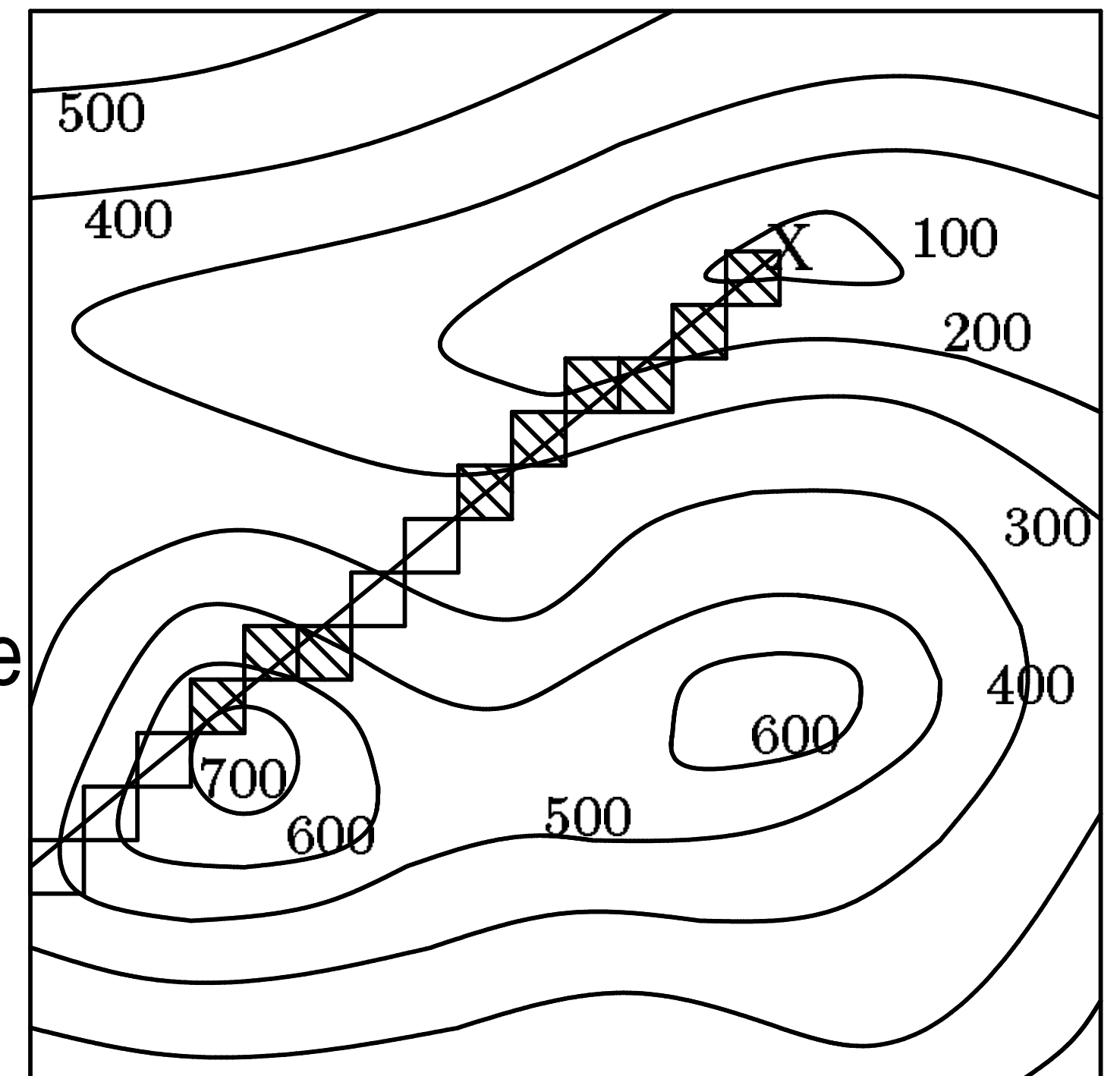
Formalizing Parallel Prefix: Scan operations

- The *scan* operation is an exclusive parallel prefix sum operation. It takes a binary associative operator \oplus with identity I , and a vector of n elements, $[a_0, a_1, \dots, a_{n-1}]$, and returns the vector $[I, a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{n-2})]$.
- An scan can be generated from a i-scan by shifting the vector right by one and inserting the identity. Similarly, the i-scan can be generated from the scan by shifting left, and inserting at the end the sum of the last element of the scan and the last element of the original vector.



Line-of-Sight Problem

- Problem Statement: given a terrain map in the form of a grid of altitudes and an observation point, X , on the grid, find which points are visible along a ray originating at the observation point. Note that a point on a ray is visible if and only if no other point between it and the observation point has a greater vertical angle.
- Define $\text{angle}[i]$ = angle of point i on ray relative to observation point, X (can be computed from altitudes of X and i)
- A max scan on $\text{angle}[*]$ returns to each point the maximum previous angle.
- Each point can compare its angle with its max scan value to determine if it will be visible or not



Segmented Inclusive Scan

Goal: Given a data vector and a flag vector as inputs, compute independent i-scans on segments of the data vector specified by the flag vector.

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

a	=	[5	1	3	4	3	9	2	6]
f	=	[1	0	1	0	0	0	1	0]
segmented +-scan	=	[5	6	3	7	10	19	2	8]
segmented max-scan	=	[5	5	3	4	4	9	2	6]



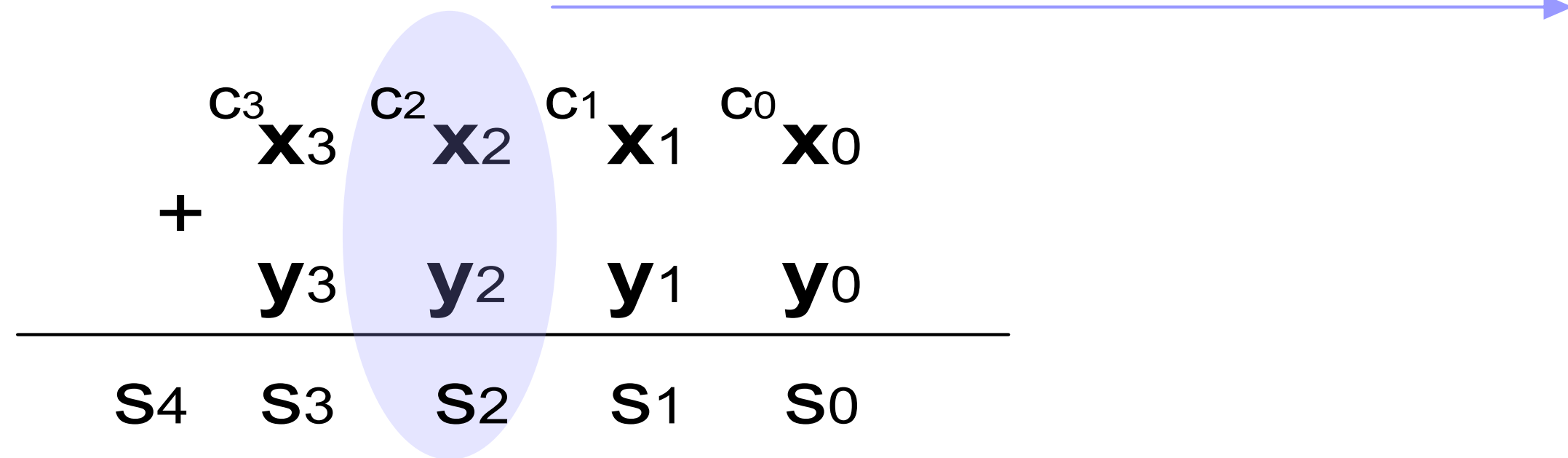
Using Segmented Scan for Quicksort

```
procedure quicksort(keys)
  seg-flags[0] ← 1
  while not-sorted(keys)
    pivots ← seg-copy(keys, seg-flags)
    f ← pivots <=> keys
    keys ← seg-split(keys, f, seg-flags)
    seg-flags ← new-seg-flags(keys, pivots, seg-flags)
```

Key	=	[6.4	9.2	3.4	1.6	8.7	4.1	9.2	3.4]
Seg-Flags	=	[1	0	0	0	0	0	0	0]
Pivots	=	[6.4	6.4	6.4	6.4	6.4	6.4	6.4	6.4]
F	=	[=	>	<	<	>	<	>	<]
Key ← split(Key, F)	=	[3.4	1.6	4.1	3.4	6.4	9.2	8.7	9.2]
Seg-Flags	=	[1	0	0	0	1	1	0	0]
Pivots	=	[3.4	3.4	3.4	3.4	6.4	9.2	9.2	9.2]
F	=	[=	<	>	=	=	=	<	=]
Key ← split(Key, F)	=	[1.6	3.4	3.4	4.1	6.4	8.7	9.2	9.2]
Seg-Flags	=	[1	1	0	1	1	1	1	0]



Binary Addition



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

A **stage** of the addition is the set of \mathbf{x} and \mathbf{y} bits being used to produce the appropriate sum and carry bits. For example the highlighted bits \mathbf{x}_2 , \mathbf{y}_2 constitute **stage 2** which generates carry \mathbf{c}_2 and sum \mathbf{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: $g_i = x_i \cdot y_i$

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:

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

- Output is a new vector of pairs:

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

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

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

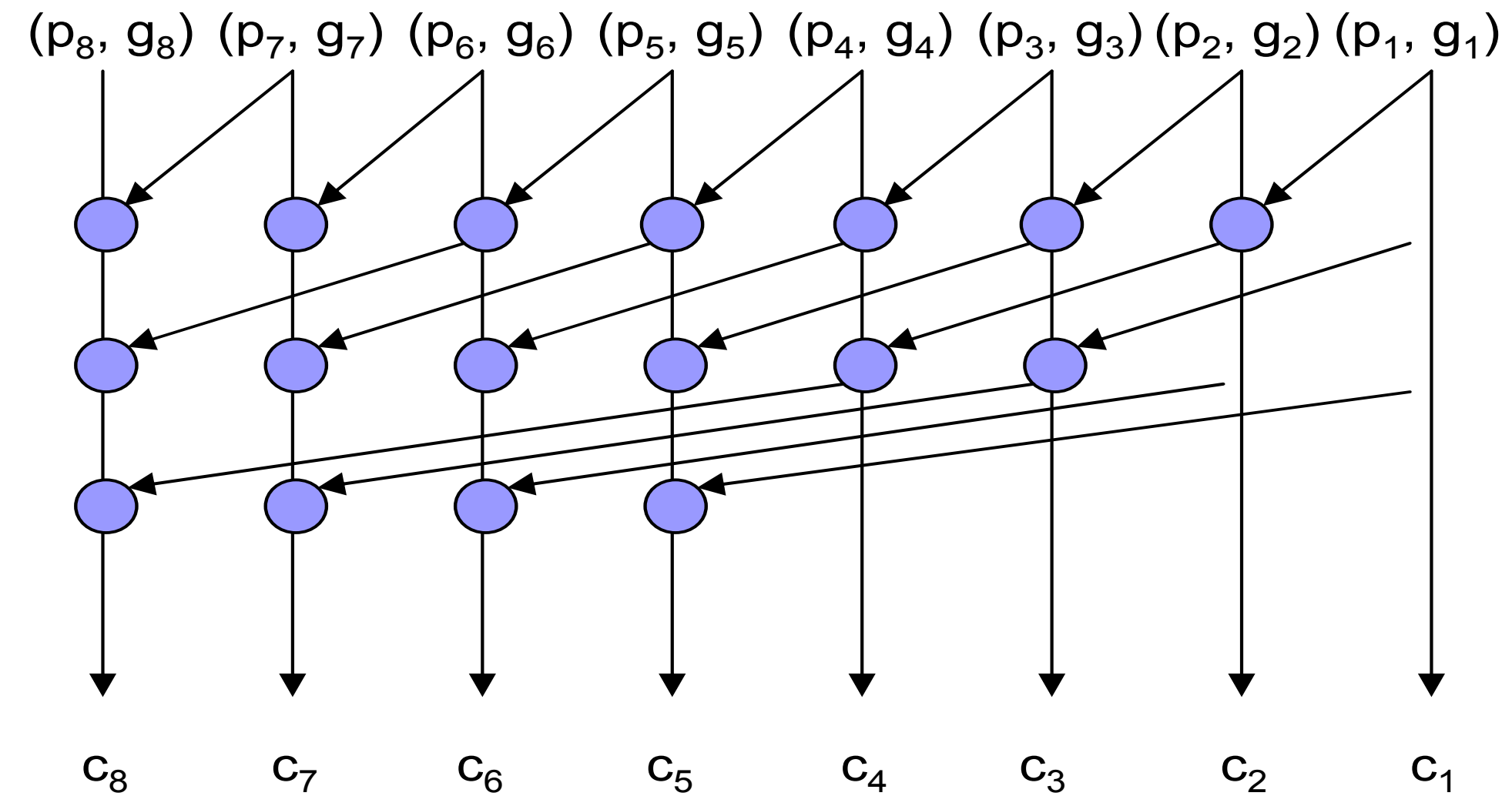
Where:

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

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

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

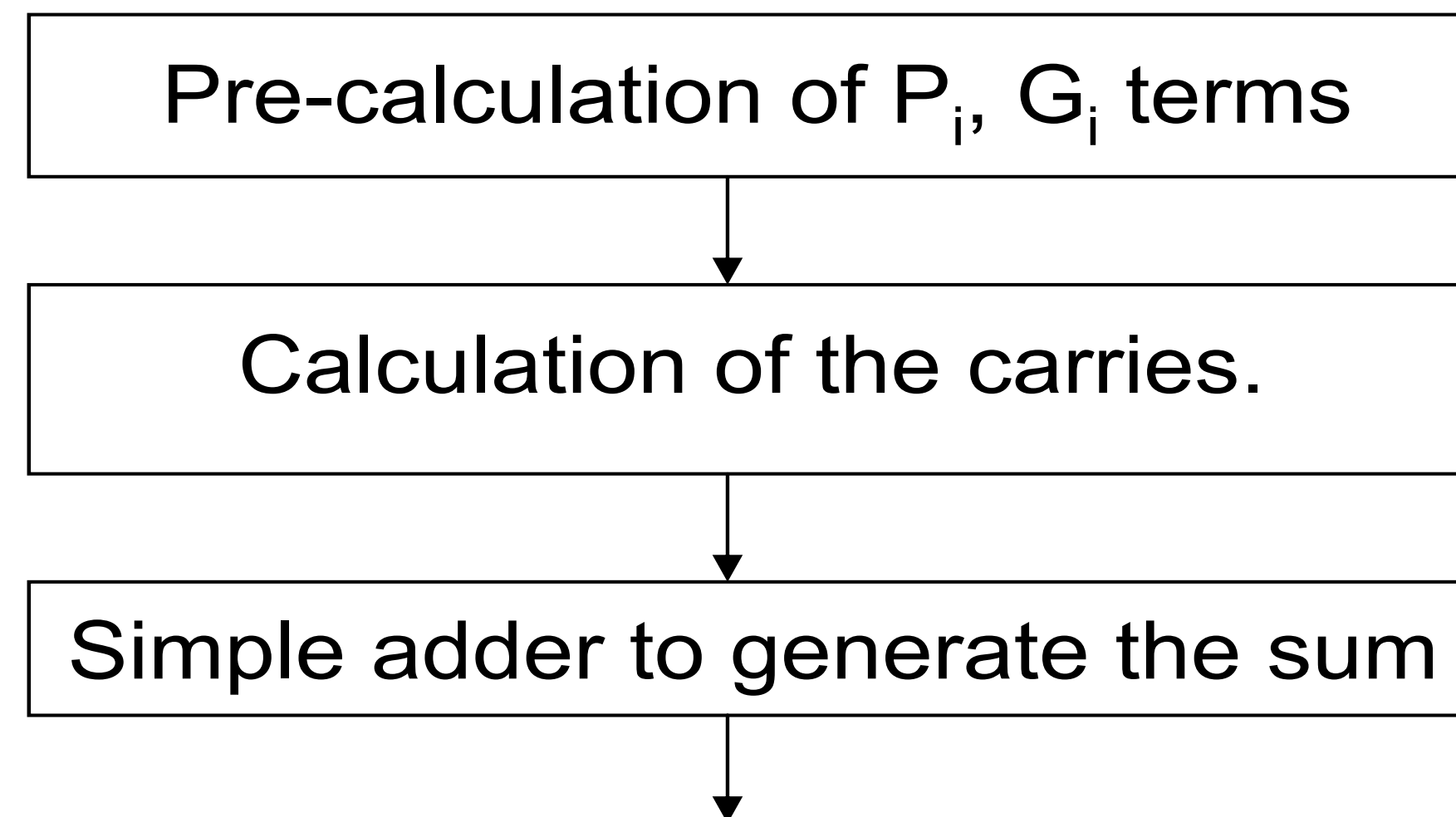
1973: Kogge-Stone adder



- 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:



- 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.