COL380

Introduction to
Parallel & Distributed Programming

Agenda

- Formal Models of Parallel Computational
  - ➡ Actor model
  - ➡ PRAM model

Formal Models of Computation

- Simplify specifying, reasoning, analyzing algorithms
- Must abstract away many details
  - ➡ Should predict computability
- Should track performance
- General classes
  - ➡ Shared Memory vs Distributed Memory
  - ➡ Synchronous vs Asynchronous

Concurrent/Parallel Computation Models

- Actors
- Communicating Sequential Processes
- Parallel Random Access Machine
- Bulk Synchronous Parallel computation

Actors Model

- Actors: autonomous computing agents
  - ➡ No shared state; interact with each other using messages
    - Asynchronous, Lossless, Unordered
    - Have a (addressed) mailbox for communication (allows buffering)
- Actors process messages in their mailbox, in response:
  - ➡ sends zero or more messages
  - ➡ creates zero or more new actors
  - ➡ changes its own local state (impacts the next message, ‘local computation’)

[See: Hewitt, Bishop, Steiger,"A Universal Modular Actor Formalism for Artificial Intelligence," IJCAI 1973]

CSP

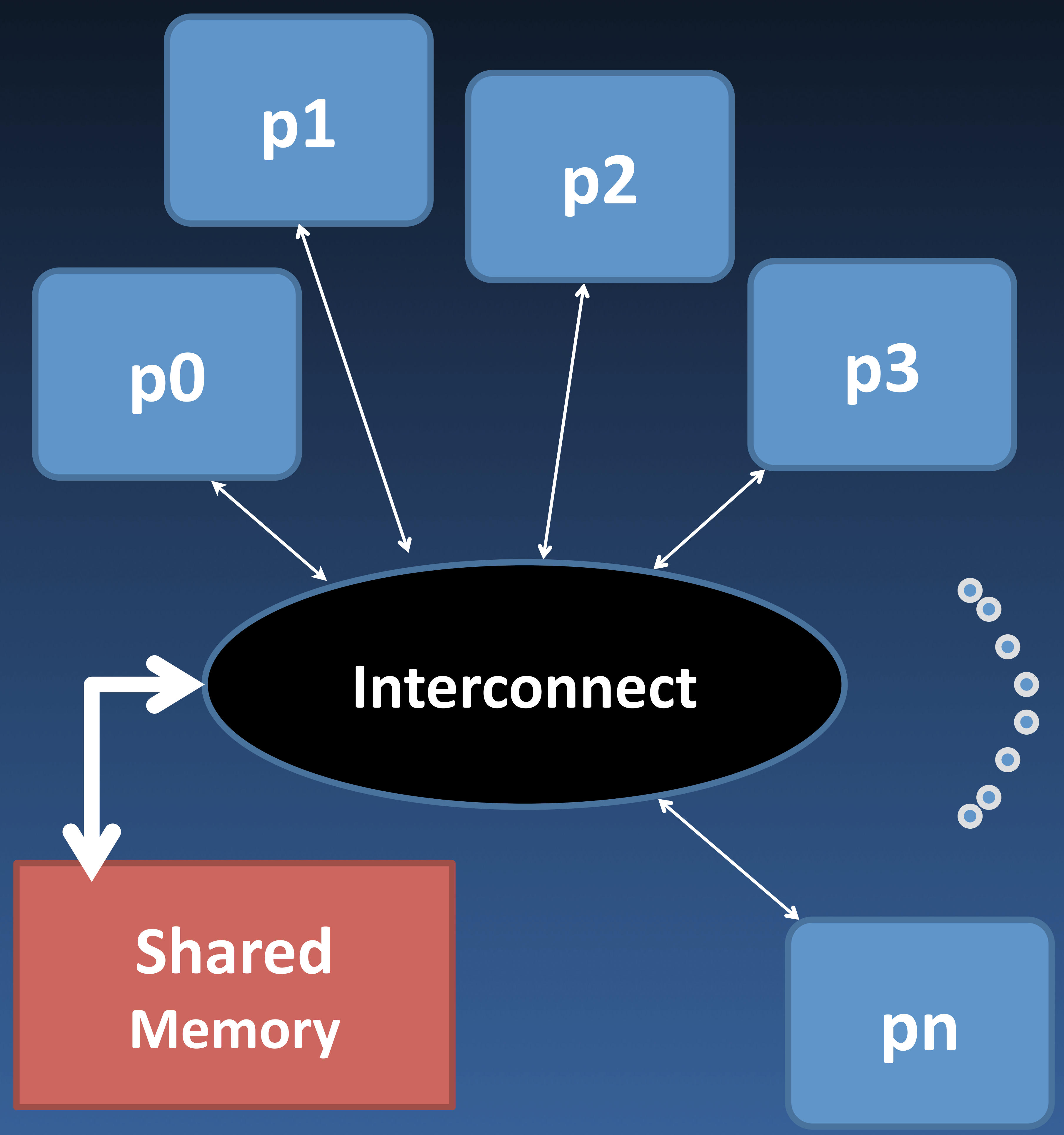
- Compose sequential processes passing messages
  - ➡ Synchronous: send completes when message received (and vice versa)
  - ➡ Processes names known to senders (used for send and receive)
- Sender?gotvalue II Recipient!somevalue
- Guard; sender?P -> {post arrival code}
  - ➡ Wait for predicate to become true (or fail if it becomes false)

```
[ Sender1? msg -> process(msg, Q) ||
  Sender2? msg -> process(msg, R)) ||
  Sender3? msg -> process(msg, S)    ]
```

(Selection)

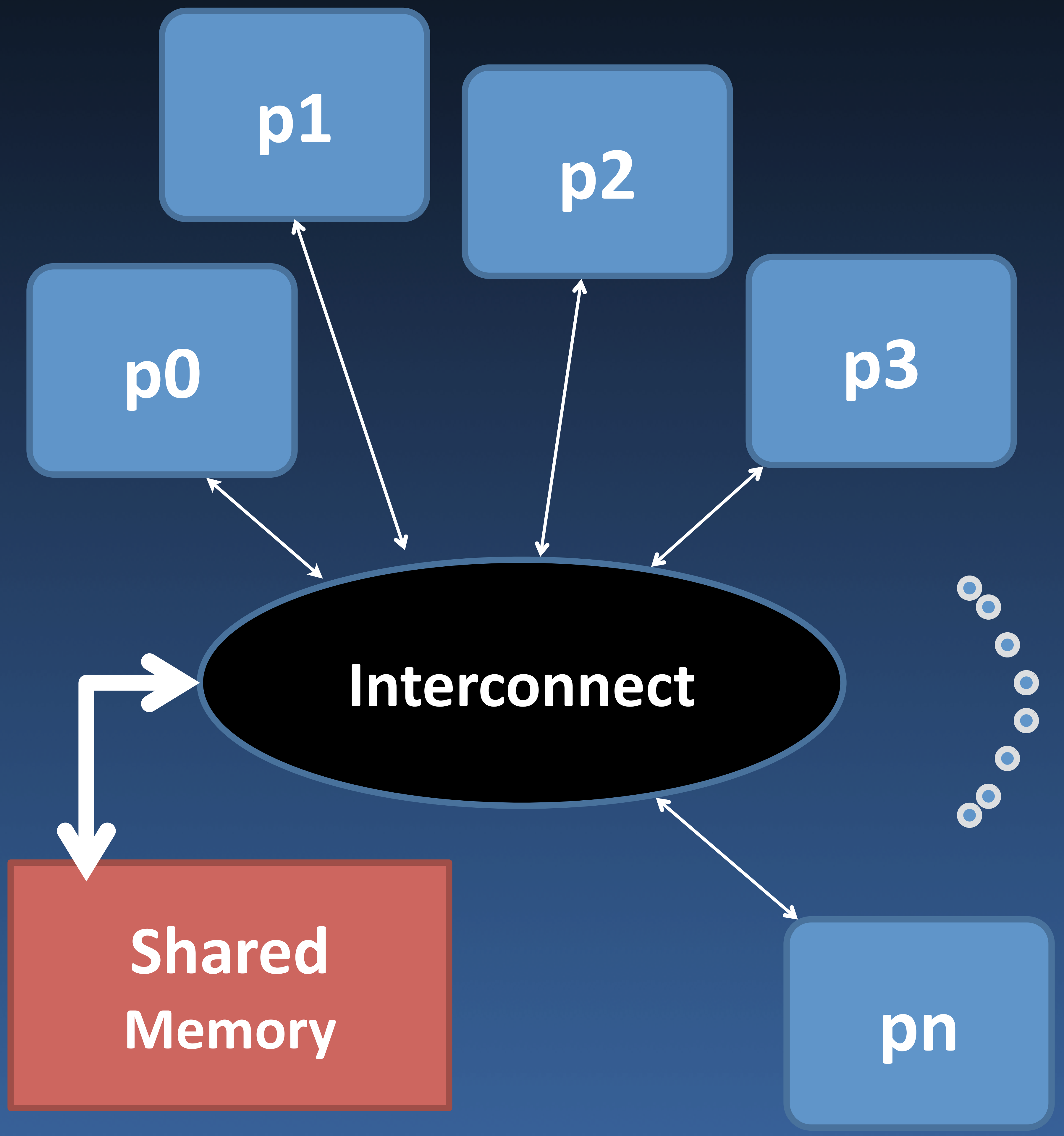
[See Hoare,"Communicating sequential processes," CACM 21 (8), 1978]

PRAM Model



- Synchronous, Shared Mem
  - ➡ *Arbitrary* number of cells
- Arbitrary number of processors, each:
  - ➡ has local memory (*Arbitrary* number of cells)
  - ➡ knows its ID
  - ➡ can access a shared memory location in *constant* time

PRAM Model



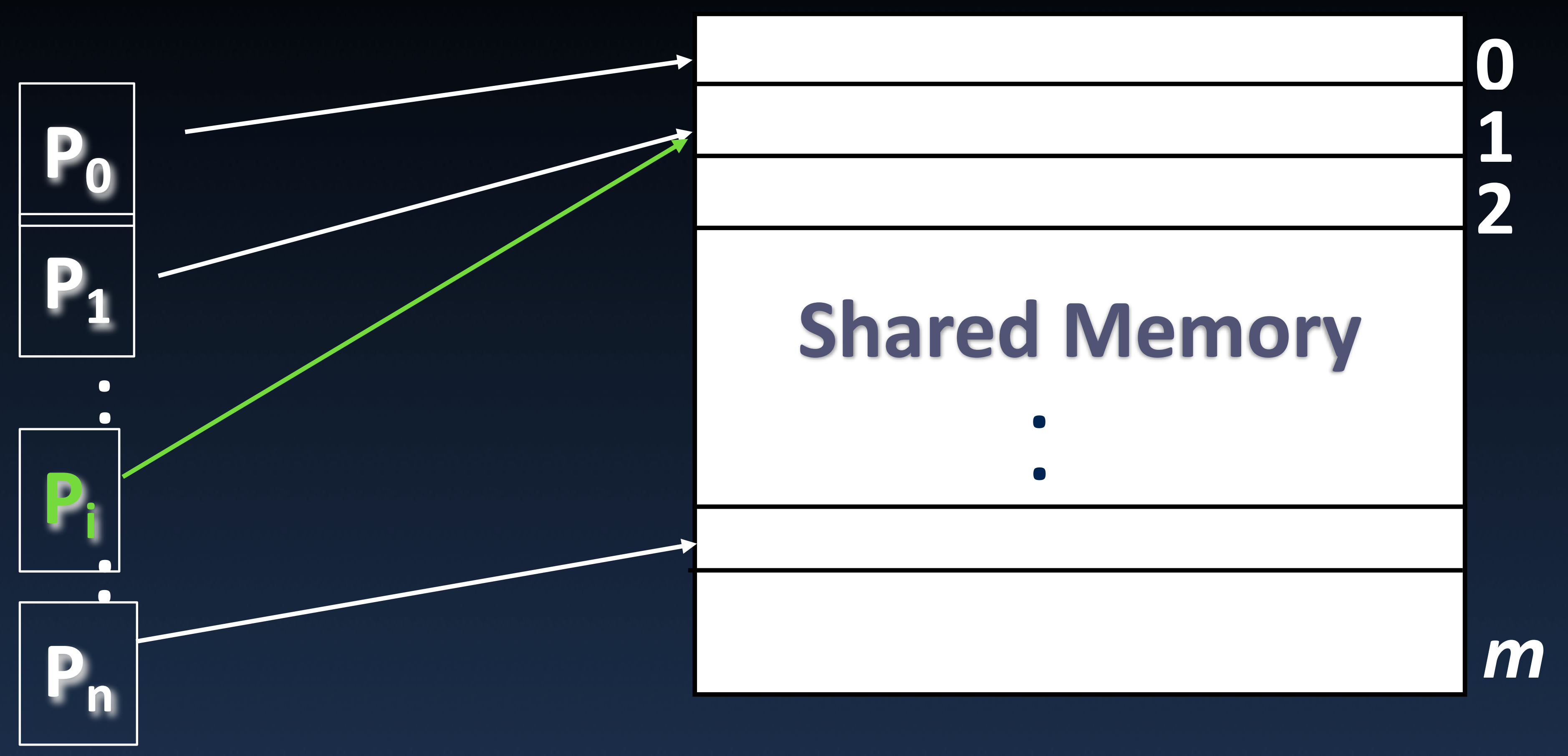
- Synchronous, Shared Mem
  - ➡ *Arbitrary* number of cells
- Arbitrary number of processors, each:
  - ➡ has local memory (*Arbitrary* number of cells)
  - ➡ knows its ID
  - ➡ can access a shared memory location in *constant* time

Unrealistic?

Can be often simulated

Subodh Kumar

PRAM Model Steps



- At each time-step each P_i can:
  1. read some memory cell
  2. perform a local computation step
  3. write a memory cell (Read and write are in two phases)
     - ➡ Co-access may be restricted
- Thus, a pair of processors P_i and P_j can communicate in two steps
  - ➡ constant time

More about PRAM

- Inputs/Outputs are placed at designated addresses
  - ➡ Technically also a ‘start’ protocol to activate processors
- Each instruction takes O(1) time
- Processors are synchronous
  - ➡ Asynchronous PRAM models exist as well
- Cost analysis:
  - ➡ Cost, Work, Time (taken by the longest running processor)
  - ➡ Maximum number of active processors and memory cells

Shared-Memory Access Models

- EREW (Exclusive Read Exclusive Write)
  - ➡ Only one processors may read or write any given location in a step
- CREW (Concurrent Read Exclusive Write)
  - ➡ Many processors can simultaneously read a location, but only one may write
- CRCW (Concurrent Read Concurrent Write)
  - ➡ Many processors can read/write the same memory location
- ERCW (Exclusive Read Concurrent Write)
  - ➡ Not commonly used

Concurrent Write (CW)

- Priority CW
  - ➡ Higher priority processor (normally lower index) wins
- Common CW
  - ➡ Succeeds only if all writes have the same value
- Arbitrary/Random CW
  - ➡ One of the values is randomly chosen

EREW ≤ CREW ≤ Common ≤ Arbitrary ≤ Priority

Less powerful — More powerful

Subodh Kumar

Parallel Addition

p = n; B[i] = A[i]

p0 p1 p2 p3 p4 p5 p6 p7

p = p/2; if(i<p) B[i] = B[2i]+B[2i+1]

p0 p1 p2 p3

p = p/2; if(i<p) B[i] = B[2i]+B[2i+1]

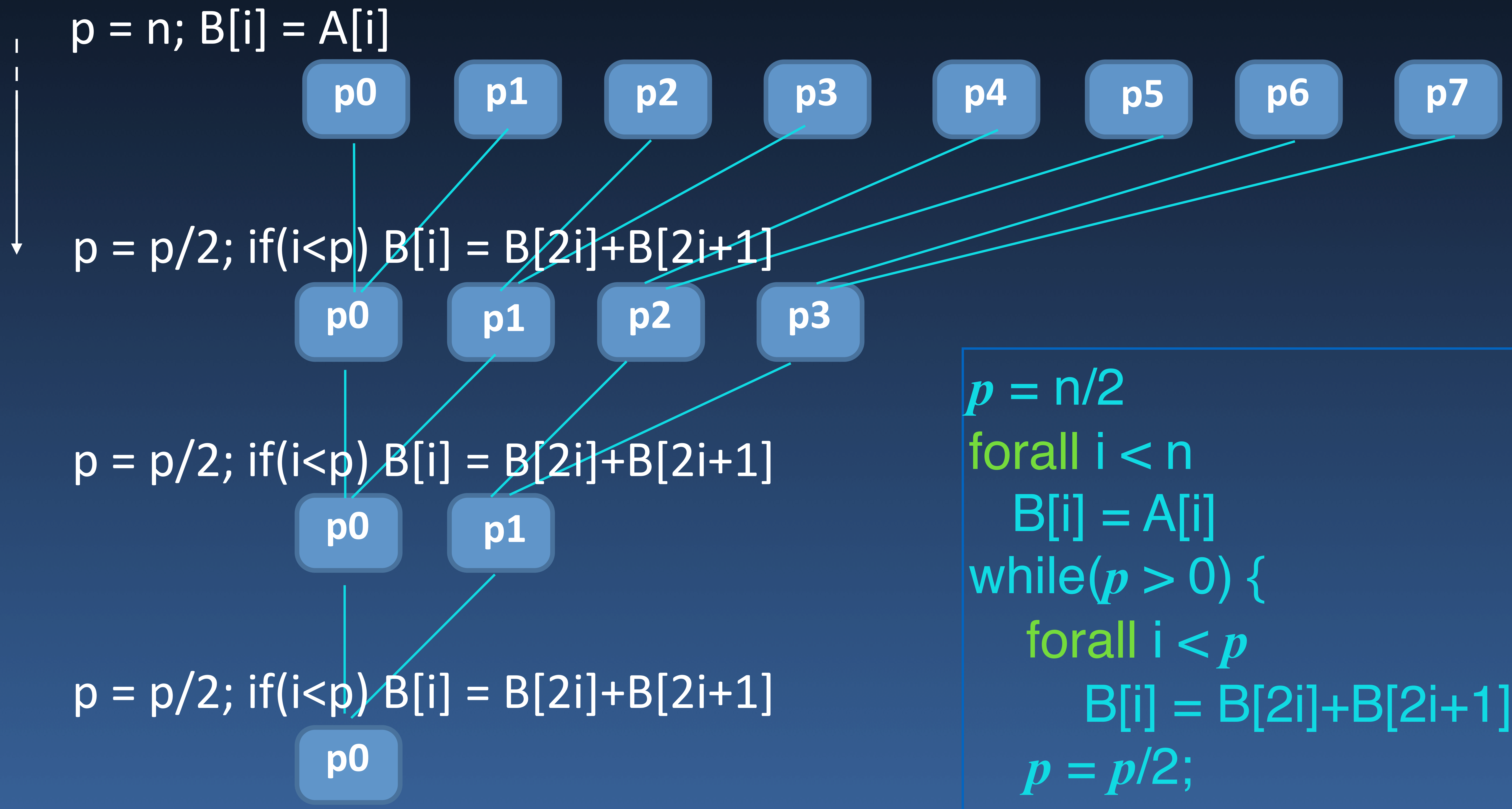
p0 p1

p = p/2; if(i<p) B[i] = B[2i]+B[2i+1]

p0

Subodh Kumar

Parallel Addition

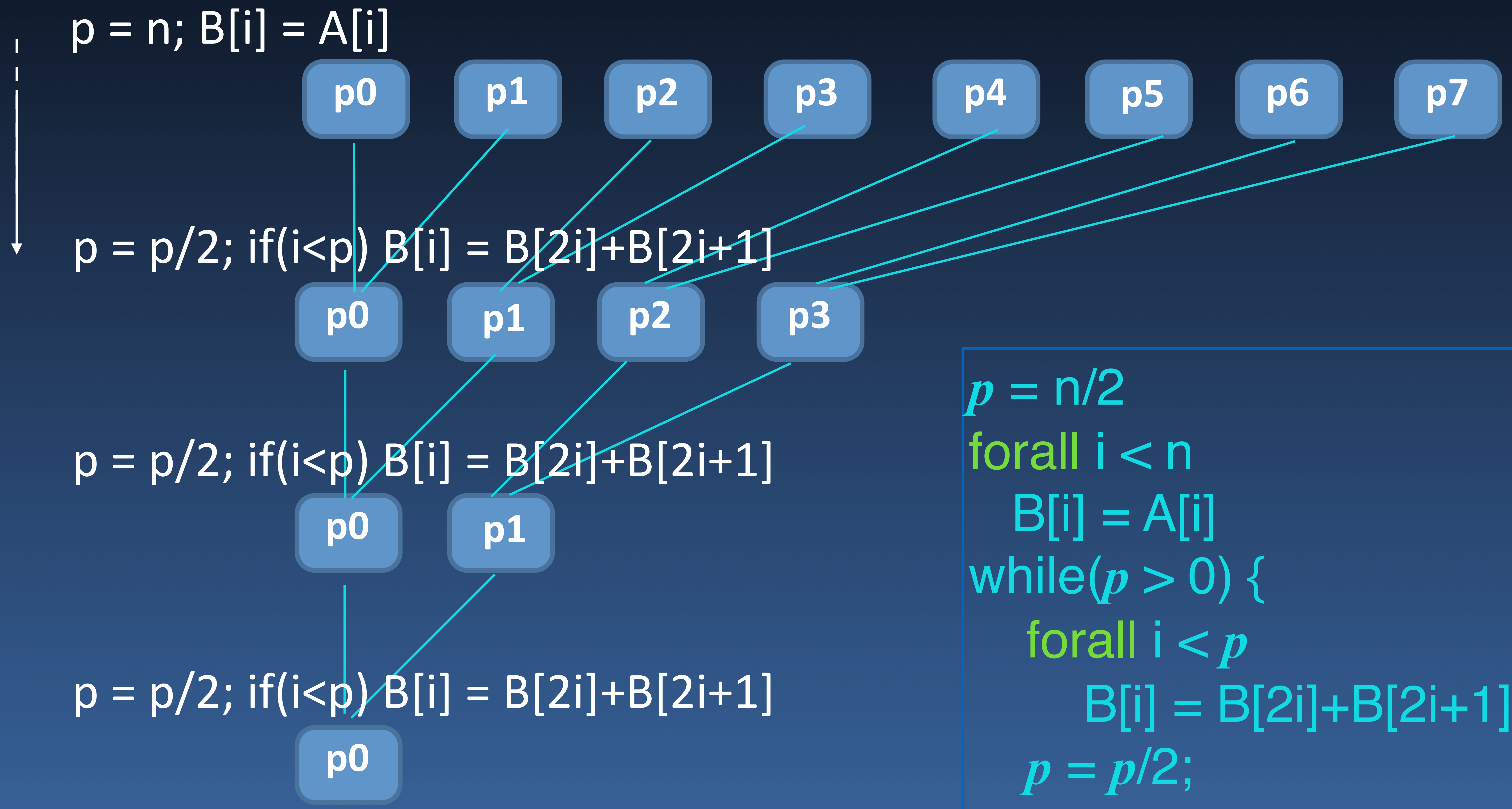



```
p = n/2
forall i < n
   B[i] = A[i]
while(p > 0) {
    forall i < p
        B[i] = B[2i]+B[2i+1]
    p = p/2;
}
```

(assumes n is a power of 2)

Subodh Kumar

Parallel Addition




```
p = n/2
forall i < n
   B[i] = A[i]
while(p > 0) {
   forall i < p
      B[i] = B[2i]+B[2i+1]
   p = p/2;
}
```

(assumes n is a power of 2)

- processors: n
- time: O(log n)
- Speed-up: n/(log n)
- Efficiency: 1/log(n)
- Cost: n log n
- Work: n

Subodh Kumar

Linear Search
p<n

- n input integers in n memory cells
- Does *x* exist in the input?
  - ➡ *x* is initially stored in a known shared memory location

<u>Algorithm</u>

step1: If p_0, answer = 0; broadcast n,x

step2: $\forall p_i$: search in i^{th} [n/p-size] block and {set flag f_i}

step3: If p_0, check if {any} flag is 1, and print answer

EREW	CREW	CRCW
• log(p)	• 1	• 1
• n/p	• n/p	• n/p
• log(p)	• log(p)	• 1

- Two parameters
  - ➡ $p(n)$, $t(n)$
- Generally, use work, $W(n)$
- If $W(n)$ similar, use $t(n)$
- Speedup/Scalability
  - ➡ Absolute: over best sequential algorithm
  - ➡ Relative: over the 1-processor implementation of the same algorithm

Performance Evaluation

- Two parameters
    - ➡ $p(n)$, $t(n)$
- Generally, use work, $W(n)$
- If $W(n)$ similar, use $t(n)$
- Speedup/Scalability
    - ➡ Absolute: over best sequential algorithm
    - ➡ Relative: over the 1-processor implementation of the same algorithm

➡ Work-optimal => work = O(serial complexity)

➡ $p(n)$ is hidden but important

- $W_1(n) = O(n)$; $t_1(n) = O(n)$
- $W_2(n) = O(n \log n)$ and $t_2(n) = O(\log n)$

Work Time Scheduling Principle

- Design algorithm in terms of
  - Total work done per 'time step': $W_i(n)$
  - $t(n)$ steps
- Total work done $W(n) = \sum W_i(n)$
- For each time step i:
  - divide the work $W_i(n)$ among p processors
    - Time $<= \sum \lceil (W_i(n)/p \rceil <= \lfloor W(n)/p) \rfloor + t(n)$
- Cost = $t(n,p) * p$

Work <= Cost. Cost optimality is more stringent.

Brent's Theorem

- Time taken by p processors:
  - ➡ $t(n,p) = O(W(n)/p + t(n))$
- Cost $= p * t(n,p) = O(W(n) + p * t(n))$
- Work = Cost if:
  - ➡ $W(n) + p * t(n) = O(W(n))$
  - ➡ Or, $p = O(W(n)/t(n))$

Notions of Optimality

- If sequentially optimal algorithm is $O(t'(n))$
  - Work done by Work-optimal parallel algorithm:
    - $O(t'(n))$ (with time $t(n)$).
  - Work-scheduling on p processors takes time:
    - $t(n,p) = O(t'(n)/p + t(n))$
  - Optimal speed-up: $t'(n)/t(n,p) = \theta(p)$, if
    - $[p^* t'(n)] / [t'(n)+p^* t(n)] = \theta(p)$
- Work-time optimal if:
  - $t(n)$ cannot be improved

Why PRAM?

- Easy to design, specify, analyze algorithms
    - Independent of machine details
- Fidelity of predicted performance
    - Not many surprises for shared-memory architecture
    - Partly successful for distributed memory
        - Memory-access and message latency can often be bounded
- Strong model
    - Possible to simulate on a wide variety of hardware
    - Poor PRAM solution often implies a hard problem

Why PRAM?

- Easy to design, specify, analyze algorithms
  - ➡ Independent of machine details
- Fidelity of predicted performance
  - ➡ Not many surprises for shared-memory architecture
  - ➡ Partly successful for distributed memory
    - Memory-access and message latency can often be bounded
- Strong model
  - ➡ Possible to simulate on a wide variety of hardware
  - ➡ Poor PRAM solution often implies a hard problem

- Fine-grained synchronization

Review

- PRAM model
    - ➡ EREW, CREW, CRCW variants
- Work-Time scheduling principle
- Work and Work-time optimality