CSc 8530
Parallel Algorithms

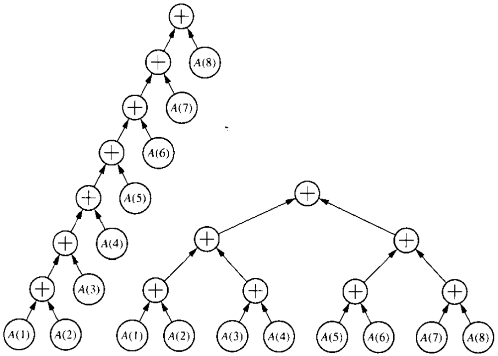
Spring 2019

January 24th, 2019

Dags for parallel processing

- We can represent computations using dags
    - Nodes with zero in-degree are inputs (also called **leafs**)
    - Nodes with zero out-degree are outputs (also called **roots** or **sinks**)
    - For simplicity, here we assume all internal vertices have in-degree ≤ 2
- Each node represents an $O(1)$ (constant-time) operation
- This model is best-suited for numerical computations
- For simplicity, we will assume no loops (what the book strangely calls branching)
    - We can always unroll a loop by duplicating it the appropriate number of times
- Node order represents **precendence**
    - What operations must come before and after

Parallel sums



- Two dags for computing the sum S of the $n = 2^k$ elements of an array A
- Note how the depths (max distance from leafs to root) differ significantly between the two choices: $O(n)$ vs $O(\log(n))$

Dag model

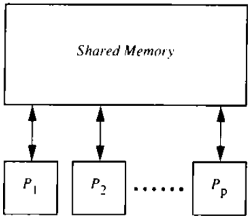
- Input nodes have $t_i = 0$ and no processor is allocated to them
- The sequence $\{(j_i, t_i) \,|\, i \in N\}$ is an execution **schedule**
    - With p processors
    - N is the number of nodes in the dag
- The time to execute a particular schedule is $\max_{i \in N} t_i$
- The **parallel complexity** is

$$T_p(n) = \min \{\max_{i \in N} t_i\}$$

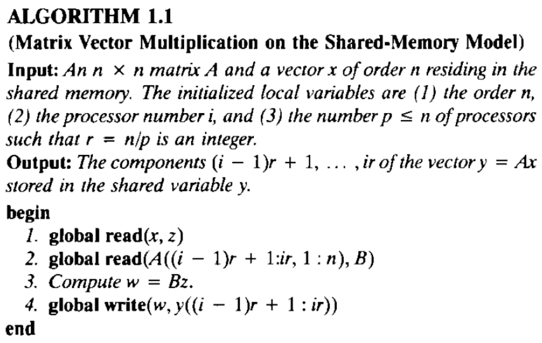
- The minimum is taken over all possible schedules with p processors
- The depth of the dag is a lower bound on $T_p(n)$, for any p

The shared-memory model

- A natural extension of the sequential RAM model
- Many processors have access to a single, **shared memory** unit (also called **global memory**)
- Each processor also has its own **local memory**
- Processors communicate by exchanging data through the shared memory
- Each processor is indexed by a unique id



Example: matrix-vector multiplication – pseudocode



ALGORITHM 1.1
(Matrix Vector Multiplication on the Shared-Memory Model)

Input: *An $n \times n$ matrix A and a vector x of order n residing in the shared memory. The initialized local variables are (1) the order n, (2) the processor number i, and (3) the number $p \leq n$ of processors such that $r = n/p$ is an integer.*

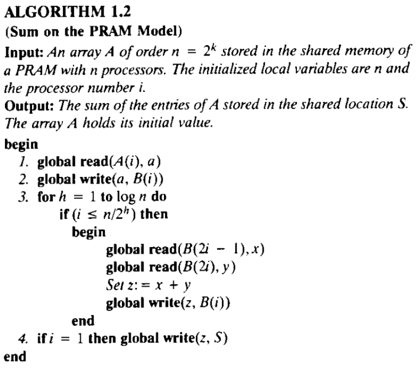
Output: *The components $(i-1)r+1, \ldots, ir$ of the vector $y = Ax$ stored in the shared variable y.*

```
begin
   1. global read($x, z$)
   2. global read($A((i-1)r+1:ir, 1:n), B$)
   3. Compute $w = Bz$.
   4. global write($w, y((i-1)r+1:ir)$)
end
```

- Analysis:
  - Steps 1 and 2 transfer $O(n^2/p)$ values from the shared memory into each processor
  - Step 3 requires $O(n^2/p)$ arithmetic operations
  - Step 4 stores n/p numbers from local to shared memory

Example: sum on the PRAM model

- Given an array A with $n = 2^k$ values and a PRAM with p processors
- We wish to compute $S = A[1] + A[2] + \ldots A[n]$
- A parallel implementation will run fastest with n processors
- However, not all processors are needed at every iteration

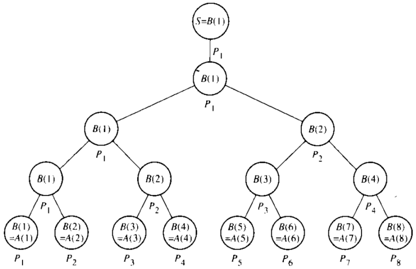


ALGORITHM 1.2
(Sum on the PRAM Model)
Input: *An array A of order $n = 2^k$ stored in the shared memory of a PRAM with n processors. The initialized local variables are n and the processor number i.*
Output: *The sum of the entries of A stored in the shared location S. The array A holds its initial value.*

```
begin
  1. global read($A(i)$, $a$)
  2. global write($a$, $B(i)$)
  3. for $h = 1$ to $\log n$ do
        if ($i \leq n/2^h$) then
          begin
              global read($B(2i - 1)$, $x$)
              global read($B(2i)$, $y$)
              Set $z := x + y$
              global write($z$, $B(i)$)
          end
  4. if $i = 1$ then global write($z$, $S$)
end
```

Example: sum on the PRAM model

- Given an array A with $n = 2^k$ values and a PRAM with p processors
- We wish to compute $S = A[1] + A[2] + \dots A[n]$
- A parallel implementation will run fastest with n processors
- However, not all processors are needed at every iteration



PRAM variations

- PRAM variants differ in how they handle simultaneous access to the same location in shared memory
    - **Exclusive read exclusive write (EREW)**
    - **Concurrent read exclusive write (CREW)**
    - **Concurrent read concurrent write (CRCW)**
- Furthermore, we have three subtypes of CRCW:
    - **Common** CRCW PRAM
        - Allows concurrent writes only when all processors attempt to write *the same value*
    - **Arbitrary** CRCW PRAM
        - Allows an arbitrary processor to succeed
    - **Priority** CRCW PRAM
        - Assumes processors have a priority (based on their ids)
        - The lowest id wins
- EREW, CREW, and, CRCW differ slightly in their computational power
    - i.e., in the space of functions they can theoretically compute

PRAM simplifications

- In the book and our slides, we will omit details concerning memory-access operations
- e.g., an instruction such as $A = B + C$ will really mean:
  1. **global read**(B, x)
  2. **global read**(C, y)
  3. $z = x + y$
  4. **global write**(z, A)

The network model

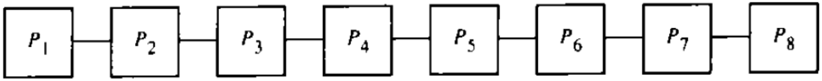
- A **network** is a graph $G = (V, E)$
  - The nodes V are the processors
  - The edges E are two-way communication links between processors
- There is **no** shared memory
  - Each processor does have local memory
- The model can be either **synchronous** or **asynchronous**
- **send**(X, i) instruction: sends X to processor P_i (and continue executing the next instruction immediately)
- **receive**(Y, j) operation: wait for Y from processor P_j (and suspend execution until data is received)

The network model

- The processors of an asynchronous network coordinate their activities through **message passing**
  - A pair of processors need not be adjacent
  - **Routing** algorithms transmit a message through a network
- The topological properties of the network affects the system's processing capabilities:
  - **Diameter:** maximum distance between any two nodes
  - **Maximum degree:** of any node in G
  - **Node and edge connectivity:** the minimum number of nodes (edges) whose removal disconnects the graph
- We will briefly look at some representative topologies:
  - Linear array
  - 2D mesh
  - Hypercube

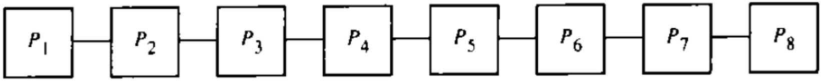
Linear array

- In a **linear array**, processor P_i is connected to P_{i-1} and P_{i+1}, if they exist
- In a **ring**, P_1 and P_p (the last processor) are connected to each other
- **Diameter:**
- **Maximum degree:**



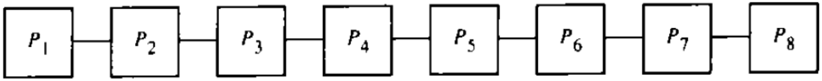
Linear array

- In a **linear array**, processor P_i is connected to P_{i-1} and P_{i+1}, if they exist
- In a **ring**, P_1 and P_p (the last processor) are connected to each other
- **Diameter:**
  - $p - 1$
- **Maximum degree:**



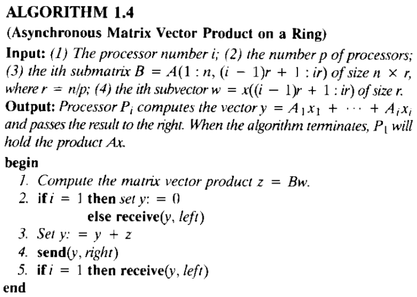
Linear array

- In a **linear array**, processor P_i is connected to P_{i-1} and P_{i+1}, if they exist
- In a **ring**, P_1 and P_p (the last processor) are connected to each other
- **Diameter:**
  - $p - 1$
- **Maximum degree:**
  - 2



Linear array – matrix-vector multiplication

- Assume we want to compute $y = Ax$
  - With $p \leq n$
  - and $r = p/n$ an integer
- First, partition:
  - $A = (A_1, A_2, \dots, A_p)$
  - $x = (x_1, x_2, \dots, x_p)$
- Then, compute $z_i = A_i x_i$ on each processor independently
- Finally, add up the sum $z = \sum_{i=1}^{p} z_i$



ALGORITHM 1.4

(Asynchronous Matrix Vector Product on a Ring)

Input: *(1) The processor number i; (2) the number p of processors; (3) the ith submatrix* $B = A(1:n, (i-1)r+1:ir)$ *of size* $n \times r$*, where* $r = n/p$*; (4) the ith subvector* $w = x((i-1)r+1:ir)$ *of size r.*

Output: *Processor* P_i *computes the vector* $y = A_1x_1 + \cdots + A_ix_i$ *and passes the result to the right. When the algorithm terminates,* P_1 *will hold the product Ax.*

```
begin
  1. Compute the matrix vector product $z = Bw$.
  2. if $i = 1$ then set $y := 0$
               else receive($y$, left)
  3. Set $y := y + z$
  4. send($y$, right)
  5. if $i = 1$ then receive($y$, left)
end
```

Linear array – matrix-vector multiplication

We split the computations as follows (for $p = n/2$):

$$\begin{aligned}
y_1 &= a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 + a_{1,4}x_4 + \ldots + a_{1,n-1}x_{n-1} + a_{1,n}x_n \\
y_2 &= a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 + a_{2,4}x_4 + \ldots + a_{2,n-1}x_{n-1} + a_{2,n}x_n \\
&\vdots \\
y_n &= a_{n,1}x_1 + a_{n,2}x_2 + a_{n,3}x_3 + a_{n,4}x_4 + \ldots + a_{n,n-1}x_{n-1} + a_{n,n}x_n
\end{aligned}$$

for processors P_1, P_2, …, P_p resp.

Linear array – matrix-vector multiplication

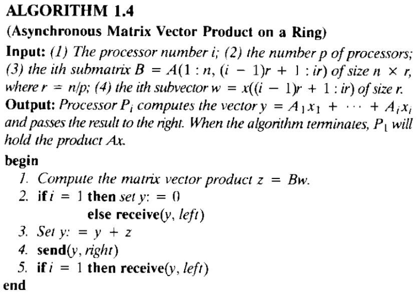
We split the computations as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} z_{1,1} \\ z_{2,1} \\ \vdots \\ z_{n,1} \end{bmatrix} + \begin{bmatrix} z_{1,2} \\ z_{2,2} \\ \vdots \\ z_{n,2} \end{bmatrix} + \ldots + \begin{bmatrix} z_{1,p} \\ z_{2,p} \\ \vdots \\ z_{n,p} \end{bmatrix}$$

for processors P_1, P_2, …, P_p resp.

Linear array – matrix-vector multiplication

- **Computation time:** $T_{comp} = O(n^2/p)$
    - Approx $\alpha(n^2/p)$ for some constant α
- However, P_1 has to wait until the $p-1$ partial sums have been transmitted to execute the last instruction
- **Communication time:** $T_{comm} = p * comm(n)$
    - $comm(n)$ is the time needed to transmit n numbers between adjacent processors



ALGORITHM 1.4

(Asynchronous Matrix Vector Product on a Ring)

Input: *(1) The processor number i; (2) the number p of processors; (3) the ith submatrix* $B = A(1:n, (i-1)r+1:ir)$ *of size* $n \times r$, *where* $r = n/p$; *(4) the ith subvector* $w = x((i-1)r+1:ir)$ *of size r.*
Output: *Processor* P_i *computes the vector* $y = A_1x_1 + \cdots + A_ix_i$ *and passes the result to the right. When the algorithm terminates,* P_1 *will hold the product Ax.*

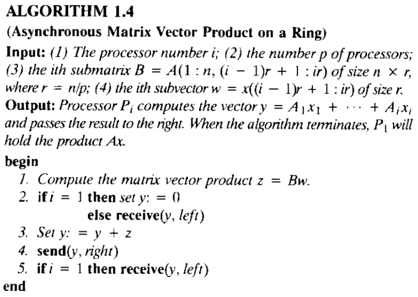
```
begin
   1. Compute the matrix vector product $z = Bw$.
   2. if $i = 1$ then set $y := 0$
              else receive($y$, left)
   3. Set $y := y + z$
   4. send($y$, right)
   5. if $i = 1$ then receive($y$, left)
end
```

Linear array – matrix-vector multiplication

- $comm(n) \approx \sigma + n\tau$
  - σ: startup time
  - τ: transfer rate
- Total execution time:

$$\begin{aligned} T &= T_{comp} + T_{comm} \\ &\approx \alpha(n^2/p) + p(\sigma + n\tau) \end{aligned}$$

- There is a trade-off between the two terms
- The sum is minimized when $\alpha(n^2/p) = p(\sigma + n\tau)$
  - Such that $p = n\sqrt{\alpha/(\sigma + n\tau)}$



ALGORITHM 1.4

(Asynchronous Matrix Vector Product on a Ring)

Input: *(1) The processor number i; (2) the number p of processors; (3) the ith submatrix $B = A(1:n, (i-1)r+1:ir)$ of size $n \times r$, where $r = n/p$; (4) the ith subvector $w = x((i-1)r+1:ir)$ of size r.*
Output: *Processor P_i computes the vector $y = A_1x_1 + \cdots + A_ix_i$ and passes the result to the right. When the algorithm terminates, P_1 will hold the product Ax.*

```
begin
   1. Compute the matrix vector product $z = Bw$.
   2. if $i = 1$ then set $y := 0$
                else receive($y$, left)
   3. Set $y := y + z$
   4. send($y$, right)
   5. if $i = 1$ then receive($y$, left)
end
```