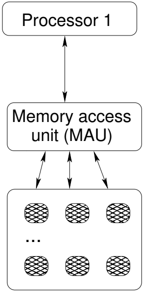
CS 467/567: Introduction to Parallel Algorithms

Stefan D. Bruda

Winter 2020

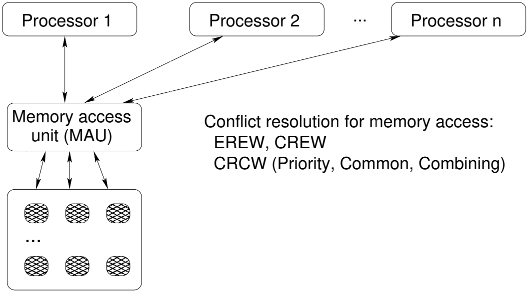
- The Random Access Machine (RAM)



- Programming language: pseudocode

PARALLEL MODELS: THE PRAM

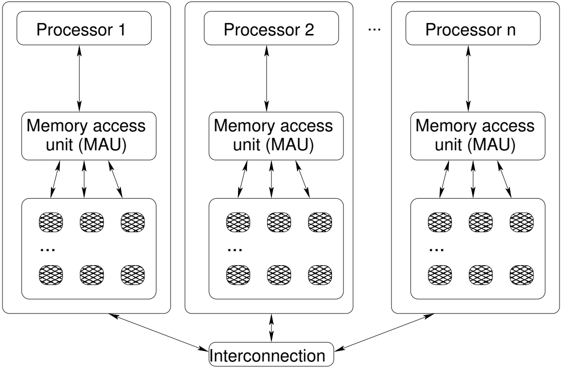
- The Parallel Random Access Machine (PRAM)



Conflict resolution for memory access:
EREW, CREW
CRCW (Priority, Common, Combining)

- Programming language: pseudocode
  - Extra statement:
    for $i = 1$ **to** n **do in parallel** $\{$ statements parameterized on processor p_i $\}$

- The Interconnection network



- Programming language: pseudocode
  - Extra statements: **send** and **receive** (via point-to-point connections only)

- We charge one time unit for each elementary computation step (like in the sequential case)
- We also charge for moving data from one processor to another = routing steps
    - Generally the cost of moving data depends on the distance between processors
- Routing cost for shared memory:
    - Uniform analysis: constant time for memory access
    - Discriminating analysis: $O(\log M)$ time for accessing one word in memory of size M
- Routing for interconnection networks: $O(1)$ time per direct link traversed

- Putting all these costs together we obtain the running time $t : \mathbb{N} \to \mathbb{N}$
    - Usually worst case analysis

- Measures of parallel performance: speedup $S_p : \mathbb{N} \to \mathbb{N}$, efficiency $E_p : \mathbb{N} \to \mathbb{N}$, and cost $c_p : \mathbb{N} \to \mathbb{N}$

$$S_p = \frac{t_1}{t_p} \qquad E_p = \frac{S_p}{p} \qquad c_p = p \times t_p$$

  - $t_p : \mathbb{N} \to \mathbb{N}$ is the time taken by the p-processor algorithm being analyzed to solve the problem
  - $t_1 : \mathbb{N} \to \mathbb{N}$ is the time taken by the best known sequential algorithm to solve the same problem
  - Speedup and efficiency are usually (but not always) invariable with the input size

Theorem (Speedup theorem)

In the classical theory of parallel algorithms $S_p \leqslant p$ and so $E_p \leqslant 1$

- A parallel algorithm with $S_p = p$ (or $E_p = 1$, or $c_p = t_1$) is optimal
- If $S_p = O(1)$ then the running time of the parallel algorithm is just as bad as the running time of a sequential algorithm
  - This is believed to happen to all the P-complete problems

Performance of parallel algorithms cont'd

- Another important measure is the slowdown = effect on running time of reducing the number of processors

Theorem (Slowdown theorem)

In the classical theory of parallel algorithms if a certain computation can be performed with p processors in time t_p and with $q < p$ processors in time t_q then $t_p \leqslant t_q \leqslant t_p + pt_p/q$

- Number of processors also important
  - Number of processors can or cannot be optimal
    - It is possible that an analysis of the algorithm reveals that a number of processors are idle most of the time and so can be discarded without affecting the performance
  - Sometimes the optimal running time can only be achieved with a certain number of processors
  - Sometimes reducing the number of processors below a certain threshold results in an unacceptable slowdown

- Processors capable of performing the usual logic and arithmetic operations on $O(\log n)$-sized words but having only a constant number of internal registers
- The processors are connected to each other as vertices in a directed acyclic graph
  - Vertices with no incoming edges are input processors
  - Vertices with no outgoing edges are output processors
- The processors can be viewed as aligned into columns, one column per distance from the input nodes
  - It is convenient (though not strictly necessary) to have all the output vertices in the rightmost column
- Performance measures for combinational circuits:
  - The depth of the circuit (or number of columns)
  - The width of the circuit (the number of processors in the largest column)

Parallel models: Combinational circuits

- Processors capable of performing the usual logic and arithmetic operations on $O(\log n)$-sized words but having only a constant number of internal registers
- The processors are connected to each other as vertices in a directed acyclic graph
  - Vertices with no incoming edges are input processors
  - Vertices with no outgoing edges are output processors
- The processors can be viewed as aligned into columns, one column per distance from the input nodes
  - It is convenient (though not strictly necessary) to have all the output vertices in the rightmost column
- Performance measures for combinational circuits:
  - The depth of the circuit (or number of columns)
  - The width of the circuit (the number of processors in the largest column)
- The combinational circuit represents the unfolded computation of an "usual" parallel machine (depth = running time; width = number of processors; cost = depth $\times$ width)

- Problem: Given an array x with n values, find all the prefix sums $s_i = \sum_{k=0}^{i} x_i$, $0 \leqslant i < n$, where the summation is done according to an associative binary operation $\circ$

  Algorithm RAM_PREFIX $(x_{0 \ldots n-1})$ **returns** $s_{0 \ldots n-1}$:

  1. $s_0 \leftarrow x_0$
  2. **for** $i = 1$ **to** $n - 1$ **do**:
     1. $s_i \leftarrow s_{i-1} \circ x_i$

- Problem: Given an array x with n values, find all the prefix sums $s_i = \sum_{k=0}^{i} x_i$, $0 \leqslant i < n$, where the summation is done according to an associative binary operation $\circ$

  Algorithm RAM_PREFIX $(x_{0\ldots n-1})$ **returns** $s_{0\ldots n-1}$:

  1. $s_0 \leftarrow x_0$
  2. **for** $i = 1$ **to** $n-1$ **do**:
     1. $s_i \leftarrow s_{i-1} \circ x_i$

  Algorithm PRAM_PREFIX $(x_{0\ldots n-1})$ **returns** $s_{0\ldots n-1}$:

  1. **for** $i = 0$ **to** $n-1$ **do in parallel**:
     1. $s_i \leftarrow x_i$
  2. **for** $j = 0$ **to** $\log n - 1$ **do**:
     1. **for** $i = 2^j$ **to** $n-1$ **do in parallel**:
     2. $s_i \leftarrow s_{i-2^j} \circ s_i$

- Sequential time: $t_1(n) = O(n)$ (also a lower bound); parallel time: $t_n(n) = O(\log n)$
- Cost: $c_n(n) = O(n \log n)$ (PRAM_Prefix is not optimal)

An optimal PRAM algorithm for prefix computations

- We exploit the associativity of $\circ$
- Let $k = \log n$ and $m = n/k$ (rounded); we use an m-processor algorithm
- All the processors P_i, $0 \leqslant i < m$ use in parallel RAM_PREFIX to compute he prefix sums $s_{ik}, s_{ik+1}, \ldots, s_{(i+1)(k-1)}$, where $s_{ik+j} = x_{ik} \circ x_{ik+1} \circ \cdots \circ x_{ik+j}$
  - $O(k) = O(\log n)$ time
- Now PRAM_PREFIX is used on all the processors to compute the prefix sum of the sequence $\langle s_{k-1}, s_{2k-1}, \ldots, s_{n-1} \rangle$; the result is put back into $\langle s_{k-1}, s_{2k-1}, \ldots, s_{n-1} \rangle$
  - At the end of this step s_{ik-1} will be replaced with $s_{k-1} \circ s_{2k-1} \circ \cdots \circ s_{ik-1}$
  - $O(\log m) = O(\log(n/\log n))$ time
- Fnally, all processors P_i, $1 \leqslant i < m$ perform sequentially $s_{ik+j} \leftarrow s_{ik-1} \circ s_{ik+j}$ for all $0 \leqslant j \leqslant k-2$
  - Executed sequentially by all processors (except P_0)
  - $O(k) = O(\log n)$ time

An optimal PRAM algorithm for prefix computations (cont'd)

- $t(n) = O(\log n) + O(\log(n/\log n)) + O(\log n) = O(\log n)$ and so $c(n) = O(n)$
- The algorithm also illustrated how an m-processor PRAM can be made to run an algorithm designed to run on n processors, $n > m$
  - This "self-simulation" is extremely useful in practice
  - It shows how to solve a problem with less that the number of processors required theoretically
- A certain storage overhead is necessary for this algorithm as opposed to the previous
  - If optimality is not a concern (e.g., we have n processors anyway) then the original algorithm is preferable

WHY PREFIX COMPUTATIONS?

- Sequentially the prefix computation performs a “sweep” of the input sequence; such a sweep can be accomplished in many other ways (some times more efficient!)
- A parallel algorithm however performs the “sweep” in an optimal amount of time using prefix computations!
- Case in point: maximum sum subsequence – given a sequence of (not necessarily positive) integers $\langle x_0, x_1, \ldots, x_{n-1} \rangle$ find two indices u and v such that $x_u + \cdots + x_v$ is maximal

  Algorithm RAM_MAX_SUM $(x_{0\ldots n-1})$ **returns** u, v:

  1. $Maxseen \leftarrow x_0;\ u \leftarrow 0;\ v \leftarrow 0;\ Maxhere \leftarrow x_0;\ q \leftarrow 0$
  2. **for** $i = 0$ **to** n **do**:
     1. **if** $Maxhere \geqslant 0$ **then** $Maxhere \leftarrow Maxhere + x_i$
        else $Maxhere \leftarrow x_i;\ q \leftarrow i$
     2. **if** $Maxseen < Maxhere$ **then** $Maxseen \leftarrow Maxhere;\ u \leftarrow q;\ v \leftarrow i$

  - One traversal of the sequence, linear complexity, also remember CS 327

WHY PREFIX COMPUTATIONS? (CONT'D)

- A parallel algorithms solving the maximum sum subsequence cannot do this kind of traversal efficiently (the traversal is inherently sequential)
- We retort to prefix computations:

Input	x_i	-4	2	6	-1	-7	4	2	-1
Prefix sum	s_i	-4	-2	4	3	-4	0	2	1
Modified prefix sum	m_i	4	4	4	3	2	2	2	1
with max as $\circ$	a_i	2	2	2	3	6	6	6	7
$b_i \leftarrow m_i - s_i + x_i$	b_i	4	8	6	-1	-1	6	2	-1

  - $L \leftarrow \max_{0 \leqslant i < m} b_i \quad \Rightarrow \quad L = 8$ (modified prefix sum, as above)
  - u is the index at which L was found $\quad \Rightarrow \quad u = 1$
  - $v \leftarrow a_u \quad \Rightarrow \quad v = 2$
- Optimal algorithm for $n/\log n$ processors

- Problem: Given $n+1$ pairs of numbers (x_i, y_i), $0 \leqslant i \leqslant n$ such that $x_0 < x_1 < \cdots < x_n$, find a polynomial $h(x)$ such that $h(x_i) = y_i$, $0 \leqslant i \leqslant n$
- Newton's interpolation method:

$$\begin{aligned} h(x) &= y_0 + Y_{01}(x - x_0) + Y_{02}(x - x_0)(x - x_1) \\ &\quad + \cdots + Y_{0n}(x - x_0)(x - x_1)\cdots(x - x_n) \end{aligned}$$

$$\text{where } Y_{ii} = y_i \text{ and } Y_{i(i+j)} = \frac{Y_{i(i+j-1)} - Y_{(i+1)(i+j)}}{x_i - x_{i+j}}$$

  - Solving the recursion for Y_{0i}, $0 \leqslant i \leqslant n$ yields

$$Y_{0i} = \frac{y_0}{X_{01}X_{02}\cdots X_{0i}} + \frac{y_1}{X_{10}X_{12}\cdots X_{1i}} + \cdots + \frac{y_i}{X_{i0}X_{i1}\cdots X_{i(j-1)}}$$

    where $X_{ij} = x_i - x_j$ for all $i \neq j$
  - Denominators can be computed using prefix sum with the scalar multiplication operation
  - One prefix computation computes all the denominators for numerator y_j

- Problem: Given an array X of size n with some values therein labeled, bring all the labeled values into contiguous positions
- Sequential algorithm (optimal $O(n)$ time): Two pointers in the array q and r with initial values $q = 1$ and $r = n$
  1. q advances to the right if X_q is labeled
  2. r advances to the left if X_r is unlabeled
  3. X_q and X_r are switched whenever X_q is unlabeled and X_r is labeled

  The labeled elements are all in adjacent positions in the first part of the array as soon as $q \geqslant r$

- Problem: Given an array X of size n with some values therein labeled, bring all the labeled values into contiguous positions
- Sequential algorithm (optimal $O(n)$ time): Two pointers in the array q and r with initial values $q = 1$ and $r = n$
  1. q advances to the right if X_q is labeled
  2. r advances to the left if X_r is unlabeled
  3. X_q and X_r are switched whenever X_q is unlabeled and X_r is labeled

  The labeled elements are all in adjacent positions in the first part of the array as soon as $q \geqslant r$
- Parallel algorithm:
  1. Create a secondary array S of size n such that $S_i = 1$ if X_i is labeled and $s_i = 0$ otherwise
  2. Compute a prefix sum over S
  3. Move each labeled value X_i to index S_i

  $O(\log n)$ running time on $n/\log n$ processors (optimal)