Parallel Random-Access Machines

Marc Moreno Maza

University of Western Ontario, London, Ontario (Canada)

CS3101

Plan

1. The PRAM Model
2. Performance counters
3. Handling Shared Memory Access Conflicts: PRAM submodels
4. Simulation of large PRAMs on small PRAMs
5. Comparing the Computational Power of PRAM Submodels
6. More PRAM algorithms and exercies

Plan

1. The PRAM Model
2. Performance counters
3. Handling Shared Memory Access Conflicts: PRAM submodels
4. Simulation of large PRAMs on small PRAMs
5. Comparing the Computational Power of PRAM Submodels
6. More PRAM algorithms and exercies

The RAM Model

Recall

The *Random Access Machine* is a convenient model of a sequential computer. Its features are as follows.

- The *computation unit* executes a user defined program.
- It uses a *read-only input tape* and a *write-only output tape*.
- The RAM has an unbounded number of local *memory cells*.
- Each memory cell can hold an integer of *unbounded size*.
- The *instruction set* includes operations for: moving data between memory cells, comparisons and conditional branching, additions, subtractions, multiplications.
- Execution starts with the first instruction of the program and terminates when an `Halt` instruction is reached.
- Each operation takes one *time unit* regardless of the the operand sizes.

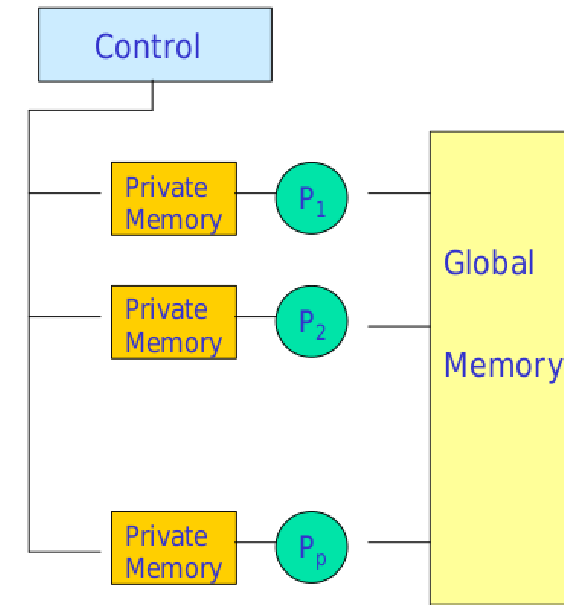
The PRAM Model: Definition (1/6)

Architecture

The *Parallel Random Access Machine* is a natural generalization of RAM. It is also an idealization of a *shared memory machine*. Its features are as follows.

- It holds an *unbounded collection of RAM processors* $P_0, P_1, P_2, \ldots$ **without tapes**.
- It holds an *unbounded collection of shared memory cells* $M[0], M[1], M[2], \ldots$
- Each processor P_i has its own (unbounded) local memory (register set) and P_i knows its index i.
- Each processor P_i can access any shared memory cell $M[j]$ in *unit time*, unless there is a conflict (see further).

The PRAM Model: Definition (2/6)



The PRAM Model: Definition (3/6)

Program execution (1/2)

- The input of a PRAM program consists of n items stored in $M[0], \ldots, M[n-1]$.
- The output of a PRAM program consists of n' items stored in n' memory cells, say $M[n], \ldots, M[n+n'-1]$.
- A PRAM instruction executes in a 3-phase cycle:
  1. **Read** (if needed) from a shared memory cell,
  2. **Compute** locally (if needed),
  3. **Write** in a shared memory cell (if needed).
- All processors execute their 3-phase cycles **synchronously**.
- **Special assumptions** have to be made in order to resolve shared memory access conflicts.
- The only way processors can exchange data is by writing into and reading from memory cells.

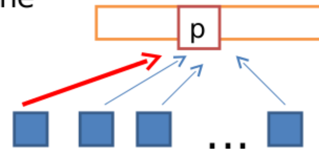
The PRAM Model: Definition (4/6)

Program execution (2/2)

- P_0 has a special *activation register* specifying the maximum index of an active processor:
  1. Initially, only P_0 is active; it computes the number of required active processors,
  2. Then, P_0 loads this number in the activation register,
  3. The corresponding processors start executing their programs.
- Computations proceed until P_0 halts, at which time all other active processors are halted.
- **Parallel time complexity** = the time for P_0's computations.
- **Parallel space complexity** = the maximum number of shared memory cells in use during the computations.

The PRAM Model: Definition (5/6)

- P_0 places the number of processors (p) in the designated shared-memory cell
  - each active P_i, where $i < p$, starts executing
  - $O(1)$ time to activate
  - all processors halt when P_0 halts



- Active processors explicitly activate additional processors via FORK instructions
  - tree-like activation
  - $O(\log p)$ time to activate

i processor will activate a processor *2i* and a processor *2i+1*

The PRAM Model: Definition (6/6)

Summary of main assumptions

- Inputs/Outputs are placed in the shared memory
- Memory cell stores an arbitrarily large integer
- Each instruction takes unit time
- Instructions are synchronized across the processors

PRAM complexity measures

- for each individual processor
  - *time:* number of instructions executed
  - *space:* number of memory cells accessed
- PRAM machine
  - *time:* time taken by the longest running processor
  - *hardware:* maximum number of active processors

The PRAM Model: Remarks

The PRAM Model is attractive for designing parallel algorithms:

- It is natural: the number of operations executed per one cycle on p processors is at most p.
- It is strong: any processor can read or write any shared memory cell in unit time.
- It is simple: ignoring any communication or synchronization overhead.

This natural, strong and simple PRAM model can be used as a benchmark: If a problem has no feasible (or efficient) solution on a PRAM then it is likely that it has no feasible (or efficient) solution on any parallel machine.

- The PRAM model is an idealization of existing shared memory parallel machines.
- The PRAM ignores lower level architecture constraints (memory access overhead, synchronization overhead, intercommunication throughput, connectivity, speed limits, etc.)

Constrained PRAM Models (1/2)

A *small-memory PRAM* satisfies the axioms of a PRAM except that is has a bounded number of shared memory cells.

- A *m-cell PRAM* is a small-memory PRAM with m shared memory cells.
- If the input (or output) data set exceeds the capacity of the shared memory, then this data can be distributed evenly among the registers of the processors.
- Limiting the amount of shared memory corresponds to restricting the amount of information that can be communicated between processors in one step.
- For example, a distributed memory machine with processors interconnected by a shared bus can be modeled as a PRAM with a single shared memory.

Constrained PRAM Models (2/2)

- A *small PRAM* satisfies the axioms of a PRAM except that is has a bounded number of processors.
- A *p-processor PRAM* is a small PRAM with $p + 1$ processors (counting P_0).

Plan

1. The PRAM Model
2. Performance counters
3. Handling Shared Memory Access Conflicts: PRAM submodels
4. Simulation of large PRAMs on small PRAMs
5. Comparing the Computational Power of PRAM Submodels
6. More PRAM algorithms and exercies

Performance counters (1/8)

Recall

The *Parallel Time*, denoted by $T(n, p)$, is the time elapsed

- from the start of a parallel computation to the moment where the last processor finishes,
- on an input data of size n,
- and using p processors.

$T(n, p)$ takes into account

- computational steps (such as adding, multiplying, swapping variables),
- routing (or communication) steps (such as transferring and exchanging information between processors).

Performance counters (2/8)

Example 1

Parallel search of an item x

- in an unsorted input file with n items,
- in a shared memory with p processors,
- where any cell can be accessed by only one processor at a time.

Broadcasting x costs $O(\log(p))$, leading to

$$T(n, p) = O(\log(p)) + O(n/p).$$

Performance counters (3/8)

Definition

- The *parallel efficiency*, denoted by $E(n,p)$, is

$$E(n,p) = \frac{SU(n)}{pT(n,p)},$$

where $SU(n)$ is a lower bound for a sequential execution. Observe that we have $SU(n) \leq p\,T(n,p)$ and thus $E(n,p) \leq 1$.

- One also often considers the *speedup factor* defined by

$$S(n,p) = \frac{SU(n)}{T(n,p)}.$$

Performance counters (4/8)

Remark

Reasons for inefficiency:

- large communication latency compared to computational performances (it would be better to calculate locally rather than remotely)
- too big overhead in synchronization, poor coordination, poor load distribution (processors must wait for dependent data),
- lack of useful work to do (too many processors for too little work).

Performance counters (5/8)

The *Work* is defined by $W(n,p) = a_{t_{\mathrm{start}}} + \cdots + a_{t_{\mathrm{end}}}$ where a_t is the number of active processors a time t.

- A *data-processor iso-efficiency function* is an asymptotically maximal function f_1 such that for all $p_0 > 0$ there exists n_0 such that for $n \geq n_0$ we have $E(n, f_1(n)) \geq E(n_0, p_0)$.
- A *processor-data iso-efficiency function* is an asymptotically minimal function f_2 such that for all $n_0 > 0$ there exists p_0 such that for $p \geq p_0$ we have $E(f_2(p), p) \geq E(n_0, p_0)$.
- The iso-efficiency function f_2 quantifies the growth rate of the problem size, required to keep the efficiency fixed while increasing the number of processors. It reflects the ability of a parallel algorithm to maintain a constant efficiency. A large iso-efficiency function f_2 indicates poor scalability, whereas a small one indicates that only a small increment in the problem size is sufficient for efficient exploitation of newly added processors.

Performance counters (6/8)

Example 2

Consider the following problem: summing n numbers on a small PRAM with $p \leq n$ processors. With the assumption that every "basic" operation runs in unit time, we have $SU(n) = n$.

- Each processor adds locally $\lceil \frac{n}{p} \rceil$ numbers.
- Then the p partial sums are summed using a *parallel binary reduction* on p processors in $\lceil \log(p) \rceil$ iterations.
- Thus, we have: $T(n,p) \in O(\frac{n}{p} + \log(p))$.
- Elementary computations give

$$f_1(n) = \frac{n}{\log(n)} \quad \text{and} f_2(p) = p\log(p).$$

Performance counters (7/8)

Example 3 (1/2)

Consider a tridiagonal linear system of order n:

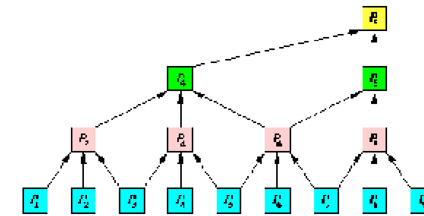
$$
\begin{array}{llllll}
\cdots & \cdots & \cdots & & & \cdots \\
a_{i-2}x_{i-2} & + b_{i-1}x_{i-1} & + c_i x_i & & = & e_{i-1} \\
 & a_{i-1}x_{i-1} & + b_i x_i & + c_{i+1}x_{i+1} & = & e_i \\
 & & a_i x_i & + b_{i+1}x_{i+1} \quad + c_{i+2}x_{i+2} & = & e_{i+1} \\
 & & \cdots & \cdots \quad \cdots & & \cdots
\end{array}
$$

For every even i replacing x_i with $-\frac{e_i - c_{i+1}x_{i+1} - a_{i-1}x_{i-1}}{b_i}$ leads to another tridiagonal system of order $n/2$:

$$
\begin{array}{llllll}
\cdots & \cdots & \cdots & & & \cdots \\
A_{i-3}x_{i-3} & + B_{i-1}x_{i-1} & + C_{i+1}x_{i+1} & & = & E_{i-1} \\
 & A_{i-1}x_{i-1} & + B_{i+1}x_{i+1} & + C_{i+3}x_{i+3} & = & E_{i+1} \\
 & \cdots & \cdots & \cdots & & \cdots
\end{array}
$$

Performance counters (8/8)

Example 3 (2/2)



- the number of processors, here $p = n$, can be set such that
- the number of parallel steps, here $O(\log n)$, is known and small,
- processors activity (scheduling) is easy to organize,
- data-communication is not intensive.

Plan

1 The PRAM Model
2 Performance counters
3 Handling Shared Memory Access Conflicts: PRAM submodels
4 Simulation of large PRAMs on small PRAMs
5 Comparing the Computational Power of PRAM Submodels
6 More PRAM algorithms and exercies

Handling Shared Memory Access Conflicts (1/18)

Definition

EREW (Exclusive Read Exclusive Write): No two processors are allowed to read or write the same shared memory cell simultaneously.

CREW (Concurrent Read Exclusive Write): Simultaneous reads of the same memory cell are allowed, but no two processors can write the same shared memory cell simultaneously.

PRIORITY CRCW (PRIORITY Concurrent Read Conc. Write):

- Simultaneous reads of the same memory cell are allowed.
- Processors are assigned fixed and distinct priorities.
- In case of write conflict, the processor with highest priority is allowed to complete WRITE.

Handling Shared Memory Access Conflicts (2/18)

Definition

ARBITRARY CRCW (ARBITRARY Concurrent Read Conc. Write):

- Simultaneous reads of the same memory cell are allowed.
- In case of write conflict, one randomly chosen processor is allowed to complete WRITE.
- An algorithm written for this model should make no assumptions about which processor is chosen in case of write conflict.

COMMON CRCW (COMMON Concurrent Read Conc. Write):

- Simultaneous reads of the same memory cell are allowed.
- In case of write conflict, all processors are allowed to complete WRITE iff all values to be written are equal.
- An algorithm written for this model should make sure that this condition is satisfied. If not, the algorithm is illegal and the machine state will be undefined.

Handling Shared Memory Access Conflicts (3/18)

Example 4: concurrent search

- Consider a p-processor PRAM with $p < n$.
- Assume that the shared memory contains n distinct items and P_0 owns a value x.
- The goal is to let P_0 know whether x occurs within the n items.

Concurrent search EREW PRAM algorithm

(a) P_0 broadcasts x to $P_1, P_2, \ldots, P_p$ in $O(\log(p))$ steps using binary broadcast tree.

(b) Every processor $P_1, P_2, \ldots, P_p$ performs local searches on (at most) $\lceil n/p \rceil$ items, hence in $\lceil n/p \rceil$ steps.

(c) Every processor defines a Boolean flag `Found`. The final answer is obtained by a *parallel reduction*, that is by, means of a binary tree.

This leads to $T(n, p) = O(\log(p) + \lceil n/p \rceil)$.

Handling Shared Memory Access Conflicts (4/18)

Concurrent search CREW PRAM algorithm

A similar approach, but $P_1, P_2, \ldots, P_p$ can read x in $O(1)$. However, the final reduction is still in $\log(p)$, leading again to $T(n, p) = O(\log(p) + \lceil n/p \rceil)$.

Concurrent search COMMON PRAM algorithm

Now, the final step takes $O(1)$. Indeed, those processors with their flag Found equal to `true` can write simultaneously to the same memory cell initialized to `false`. Hence, we have $T(n, p) = O(\lceil n/p \rceil)$.

Handling Shared Memory Access Conflicts (5/18)

Example 5: statement

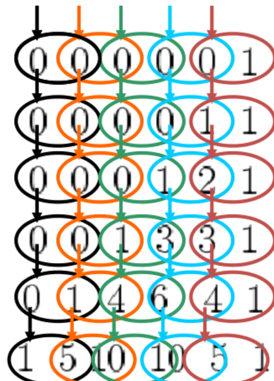
On a CREW-PRAM Machine, what does the pseudo-code do?

```
A[1..6] := [0,0,0,0,0,1];
for each 1 <= step <= 5 do
    for each 1 <= i <= 5 do in parallel
        A[i] := A[i] + A[i+1]; // done by processor #i
print A;
```

Handling Shared Memory Access Conflicts (6/18)

Example 5: solution

- No data races occur thanks to the execution model (the 3-phasis cycle) and CREW handling.
- On an actual computer, there would be data races and an uncertain result, that is, a non-deterministic answer.



Handling Shared Memory Access Conflicts (7/18)

Example 6: statement

Write an EREW-PRAM Program for the following task:

- Given $2n$ input integer number compute their maximum

Handling Shared Memory Access Conflicts (8/18)

Example 6: solution

Input: $2n$ integer numbers stored in $M[1], \ldots, M[2n]$, where $n \geq 2$ is a power of 2.

Output: The maximum of those numbers, written at $M[2n+1]$.

Program:

```
Active Proocessors P[1], ...,P[n];
step := 0;
jump := 2^step;
while jump <= n do {
  // id the index of one of the active processor
  if (id mod jump = 0)
      M[2 * id] := max(M[2 * id], M[2 * id - jump]);
  step := step + 1;
  jump := 2^step;
}
if (id = n) then M[2n+1] := M[2n];
```


Handling Shared Memory Access Conflicts (9/18)

Example 7: statement

This is a follow-up on Example 6.

1. What is $T(2n, n)$? $SU(n)$? $S(2n, n)$?
2. What is $W(2n, n)$? $E(2n, n)$?
3. Propose a variant of the algorithm for an input of size n using p processors such that we have $S(n, p) = 50\%$?

Handling Shared Memory Access Conflicts (10/18)

Example 7: solution

1. $\log(n)$, n, $n/\log(n)$.
2. $n\log(n)$, $n/(n\log(n))$.
3. Algorithm:
   1. Use $p := n/\log(n)$ processors, instead of n.
   2. Make each of these p processors computes serially the maximum of $\log(n)$ numbers. This requires $\log(n)$ parallel steps and has total work n.
   3. Run the previous algorithm on these p "local maxima". This will take $\log(p) \in O(\log(n))$ steps with a total work of $p\log(p) \in O((n/\log(n))\log(n))$.
   4. Therefore the algorithm runs in at most $2\log(n)$ parallel steps and uses $n/\log(n)$ processors. Thus, we have $S(n, p) = 50\%$.

Handling Shared Memory Access Conflicts (11/18)

Example 8: statement

Write a COMMON CRCW-PRAM Program for the following task:

- Given n input integer number compute their maximum.
- And such that this program runs essentially in constant time, that is, $O(1)$.

Handling Shared Memory Access Conflicts (12/18)

Example 8: solution

Input: n integer numbers stored in $M[1], \ldots, M[n]$, where $n \geq 2$.

Output: The maximum of those numbers, written at $M[n+1]$.

Program:

```
Active Proocessors P[1], ...,P[n^2];
// id the index of one of the active processor
if (id <= n)
    M[n + id] := true;
i := ((id -1) mod n) + 1;
j := ((id -1) quo n) + 1;
if (M[i] < M[j])
     M[n + i] := false;
if (M[n + i] = true)
     M[n+1] := M[i];
```


Handling Shared Memory Access Conflicts (13/18)

Example 9: statement

This is a follow-up on Example 6.

1. What is $T(n, n^2)$? $SU(n)$? $S(n, n^2)$?
2. What is $W(n, n^2)$? $E(n, n^2)$?

Handling Shared Memory Access Conflicts (14/18)

Example 9: solution

1. $O(1)$, n, n.
2. n^2, $1/n$.

Handling Shared Memory Access Conflicts (15/18)

Example 10: statement

Write an EREW-PRAM Program for the following task:

- Given two polynomials of degree less than $n-$, say $a = a_{n-1}x^{n-1} + \cdots + a_1 X + a_0$ and $b = b_{n-1}x^{n-1} + \cdots + b_1 X + b_0$ compute their product in parallel time $o(\log_2(n))$.
- We may make assumptions of the form "n is a power of 2".

Handling Shared Memory Access Conflicts (16/18)

Example 10: solution (1/3)

Input: Tow polynomials $a = a_{n-1}x^{n-1} + \cdots + a_1 X + a_0$ and $b = b_{n-1}x^{n-1} + \cdots + b_1 X + b_0$ such that `M[i]` holds a_{i-1} and `M[n+i]` holds b_{i-1} for $1 \leq i \leq n$ and n is a power of 2.

Output: Their product.

Program:

```
Active Proocessors P[1], ...,P[n^2];
// id the index of one of the active processor
i := ((id -1) mod n) + 1;
j := ((id -1) quo n) + 1;
M[2n + id] := M[i] * M[n + j];
....
```

The problem in the above code is that we have to sum up all `M[2n + i] * M[2n + j]` contributing to the same coefficient of the product. Indeed, we need to write these products in consecutive memory location to sum them conveniently.

Handling Shared Memory Access Conflicts (17/18)

Example 10: solution (2/3)

- Observe that $a \cdot b$ has $2n - 1$ coefficients.
- The number n_d of terms contributing to X^d satisfies
$$n_d = \begin{cases} d+1 & \text{for} \quad 0 \leq d \leq n-1, \\ 2n-d & \text{for} \quad n \leq d \leq 2n-2. \end{cases}$$
- Observe that $1 \leq n_d \leq n$ for all $0 \leq d \leq 2n-2$.
- For each $d \in \{0, \ldots, 2n-2\}$, we allocate n slots (we assume that the memory allocator initializes them to zero) to write the n_d terms contributing to X^d.
- More precisely, `M[(2 * n) + (d * n) + i + 1]` stores the product `M[i + 1] * M[n + j + 1]` if $d = i + j$.

Handling Shared Memory Access Conflicts (18/18)

Example 10: solution (3/3)

```
Active Proocessors P[1], ...,P[n^2];
// id the index of one of the active processor
i := ((id -1) mod n); j := ((id -1) quo n);
// Observe that i and j are now in 0..(n-1)
d := i+j;
M[(2 * n) + (d * n) + i + 1] := M[i+1] * M[n + j + 1];
```

- After this point n processors can work together on a parallel reduction for each d.
- Since $d \in \{0, \ldots, 2n-2\}$, each processor will participate to at most 2 parallel reductions.
- For simplicity, the code should make each processor work on 2 parallel reductions.
- Hence additional "zero" slots must be added.

Plan

1. The PRAM Model
2. Performance counters
3. Handling Shared Memory Access Conflicts: PRAM submodels
4. Simulation of large PRAMs on small PRAMs
5. Comparing the Computational Power of PRAM Submodels
6. More PRAM algorithms and exercies

Simulation of large PRAMs on small PRAMs (1/3)

Proposition 1

Let $p' < p$. Then, any problem that can be solved on a p-processor PRAM in t steps can be solved on a p'-processor PRAM in $t' = O(tp/p')$ steps assuming the same size of shared memory.

Proof

In order to reach this result, each of the processors ${P'}_i$ of the p'-processor PRAM can simulate a group G_i of (at most) $\lceil p/p' \rceil$ processors of the p-processor PRAM as follows. Each simulating processor ${P'}_i$ simulates one 3-phase cycle of G_i by

1. executing all their READ instructions,
2. executing all their local COMPUTATIONS,
3. executing all their WRITE instructions.

One can check that, whatever is the model for handling shared memory cell access conflict, the simulating PRAM will produce the same result as the larger PRAM.

Simulation of large PRAMs on small PRAMs (2/3)

Proposition 2

Assume $m' < m$. Then, any problem that can be solved on a p-processor and m-cell PRAM in t steps can be solved on a $\max(p, m')$-processor and m'-cell PRAM in $t' = O(tm/m')$ steps.

Proof of Proposition 2 (1/2)

- Naturally, the idea is to use the register set of the processors of the m'-cell PRAM in order to compensate the diminution of shared memory.
- This is why it is necessary to assume that the m'-cell PRAM has at least m' processors. (After that, one can use Proposition 1 to save on processors.)
- Let $P_1, \ldots, P_p$ be the processors of the m-cell PRAM:
  - We use processors ${P'}_1, \ldots, {P'}_{m''}$ on the m'-cell PRAM to simulate $P_1, \ldots, P_p$ where $m'' = \max(p, m')$.
  - Moreover, we (mentally) group the m cells of the m-cell PRAM into m' continuous segments $S_1, \ldots, S_{m'}$ of size m/m'.

Simulation of large PRAMs on small PRAMs (2/3)

Proof of Proposition 2 (2/2)

- We use the register set of processor P'_i for simulating the segment S_i, for all $1 \leq i \leq m'$.
- We use the shared memory cell $M'[i]$, for $1 \leq i \leq m'$, on the m'-cell PRAM, as an auxiliary memory.

Simulation of one 3-phase cycle of the m-cell PRAM:

`READ`: for all $0 \leq k < m/m'$ repeat

1. for all $1 \leq i \leq m'$, the processor P'_i writes the value of the k-th cell of S_i into $M'[i]$;
2. for all $1 \leq i \leq p$, the processor P'_i reads from the share memory, **provided** that P_i would read its value at position congruent to k modulo m/m'.

`COMPUTE`: the local computation of P_i is simulated by P'_i, for all $1 \leq i \leq p$.

`WRITE`: Analogous to `READ`.

Plan

1. The PRAM Model
2. Performance counters
3. Handling Shared Memory Access Conflicts: PRAM submodels
4. Simulation of large PRAMs on small PRAMs
5. Comparing the Computational Power of PRAM Submodels
6. More PRAM algorithms and exercies

Comparing the Computational Power of PRAM Submodels

Remark

By PRAM submodels, we mean either EREW, CREW, COMMON, ARBITRARY or PRIORITY.

Definition

PRAM submodel A is computationally stronger than PRAM submodel B, written $\mathrm{A} \geq \mathrm{B}$, if any algorithm written for B will run unchanged on A in the same parallel time, assuming the same basic properties.

Proposition 3

We have:

$$\mathrm{PRIORITY} \geq \mathrm{ARBITRARY} \geq \mathrm{COMMON} \geq \mathrm{CREW} \geq \mathrm{EREW}.$$

Comparing the Computational Power of PRAM Submodels

Theorem 1

Any polylog time PRAM algorithm is robust with respect to all PRAM submodels.

Remark

- In other words, any PRAM algorithm which runs in polylog time on one submodel can be simulated on any other PRAM submodel and run within the same complexity class.
- This results from Proposition 3 and Lemma 2.
- Lemma 1 provides a result weaker than Lemma 2 but the proof of the former helps understanding the proof of the latter.

Comparing the Computational Power of PRAM Submodels

Lemma 1

Assume PRIORITY CRCW with the priority scheme based trivially on indexing: lower indexed processors have higher priority. Then, one step of p-processor m-cell PRIORITY CRCW can be simulated by a p-processor mp-cell EREW PRAM in $O(\log(p))$ steps.

Comparing the Computational Power of PRAM Submodels

Proof of Lemma 1 (1/3)

Naturally, the idea is to

- store all the WRITE (or READ) *needs* for one cycle in memory
- evaluate their priorities
- execute the instruction of the winner

But there is a trap, we should avoid access conflict also during this simulation algorithm.

Comparing the Computational Power of PRAM Submodels

Proof of Lemma 1 (2/3)

(1) Each PRIORITY processor P_k is simulated by EREW processor P'_k, for all $1 \leq k \leq p$.

(2) Each shared memory cell $M[i]$, for all $i = 1, \ldots, m$, of PRIORITY
- is simulated by an array of p shared memory cells $M'[i,k]$, $k = 1, \ldots, p$ of EREW,
- $M'[i,1]$ plays the role of $M[i]$,
- $M'[i,2], \ldots, M'[i,p]$ are auxiliary cells used for resolving conflicts,
  - initially empty,
  - $M'[i,1], \ldots, M'[i,p]$ are regarded as the rows of a complete binary tree T_i with p leaves and height $\lceil \log(p) \rceil$; initially, they are regarded as the leaf row of T_i.

Comparing the Computational Power of PRAM Submodels

Proof of Lemma 1 (3/3)

(3) Simulation of a PRIORITY WRITE substep. Each EREW processor
- must find out whether it is the processor with lowest index within the group asking to write to the same memory cell, and if so,
- must become the group winner and perform the WRITE operation; the other processors of its group just fail and do not write.

This is done as follows:

1. For all $1 \leq k \leq p$ repeat: if P_k wants to write into $M[i]$, then P'_k turns **active** and becomes the k-th leaf of T_i.
2. Each active **left** processor stores its ID into the parent cell in its tree, marks it as **occupied** and remains active.
3. Each active **right** processor checks its parent cell: if it is empty, then it stores its ID there and remains active, otherwise it becomes inactive.
4. This is repeated one row after another from bottom to top in T_i, in $\lceil \log(p) \rceil$ iterations.
5. The process who managed to reach the root of T_i, becomes the winner and can WRITE.

Comparing the Computational Power of PRAM Submodels

Lemma 2

One step of PRIORITY CRCW with p processors and m shared memory cells by an EREW PRAM in $O(\log(p))$ steps with p processors and $m+p$ shared memory cells.

Comparing the Computational Power of PRAM Submodels

Proof of Lemma 2 (1/3)

1. Each PRIORITY processor P_k is simulated by EREW processor P'_k.
2. Each PRIORITY cell $M[i]$ is simulated by EREW cell $M'[i]$.
3. EREW uses an auxiliary array A of p cells.
4. If P_k wants to access $M[i]$, then processor P'_k writes pair (i,k) into $A[k]$.
5. If P_k does not want to access any PRIORITY cell, processor P'_k writes $(0,k)$ into $A[k]$.
6. All p processors sort the array A into **lexicographic order** using $(\log p)$-time parallel sort.

Comparing the Computational Power of PRAM Submodels

Proof of Lemma 2 (2/3)

1. Each P'_k appends to cell $A[k]$ a flag f defined as follows
   - $f=0$ if the first component of $A[k]$ is either 0 or it is the same as the first component of $A[k-1]$.
   - $f=1$ otherwise.
2. Further steps differ for simulation of WRITE or READ.

   PRIORITY WRITE:

   1. Each P'_k reads the triple (i,j,f) from cell $A[k]$ and writes it into $A[j]$.
   2. Each P'_k reads the triple (i,k,f) from cell $A[k]$ and writes into $M[i]$ iff $f=1$.

Comparing the Computational Power of PRAM Submodels

Proof of Lemma 2 (3/3)

PRIORITY READ:

1. Each P'_k reads the triple (i,j,f) from cell $A[k]$.
2. If $f=1$, it reads value v_i from $M[i]$ and overwrites the third component in $A[k]$ (the flag f) with v_i.
3. In at most $\log p$ steps, this third component is then distributed in subsequent cells of A until it reaches either the end or an element with a different first component.
4. Each P'_k reads the triple (i,j,v_i) from cell $A[k]$ and writes it into $A[j]$.
5. Each P'_k who asked for a READ reads the value v_i from the triple (i,k,v_i) in cell $A[k]$.

Plan

1. The PRAM Model
2. Performance counters
3. Handling Shared Memory Access Conflicts: PRAM submodels
4. Simulation of large PRAMs on small PRAMs
5. Comparing the Computational Power of PRAM Submodels
6. More PRAM algorithms and exercies

Parallel scan (1/5)

- Another common and important data parallel primitive.
- This problem seems inherently sequential, but there is an efficient parallel algorithm.
- Applications: sorting, lexical analysis, string comparison, polynomial evaluation, stream compaction, building histograms and data structures (graphs, trees, etc.) in parallel.

Parallel scan (2/5)

- Let S be a set, let $+ : S \times S \to S$ be an associative operation on S with 0 as identity. Let $A[0 \cdots n-1]$ be an array of n elements of S.
- Tthe *all-prefixes-sum* or *inclusive scan* of A computes the array B of n elements of S defined by

$$B[i] = \begin{cases} A[0] & \text{if} \quad i = 0 \\ B[i-1] + A[i] & \text{if} \quad 0 < i < n \end{cases}$$

- The *exclusive scan* of A computes the array B of n elements of S:

$$C[i] = \begin{cases} 0 & \text{if} \quad i = 0 \\ C[i-1] + A[i-1] & \text{if} \quad 0 < i < n \end{cases}$$

- An exclusive scan can be generated from an inclusive scan by shifting the resulting array right by one element and inserting the identity.
- Similarly, an inclusive scan can be generated from an exclusive scan.
- We shall focus on exclusive scan.

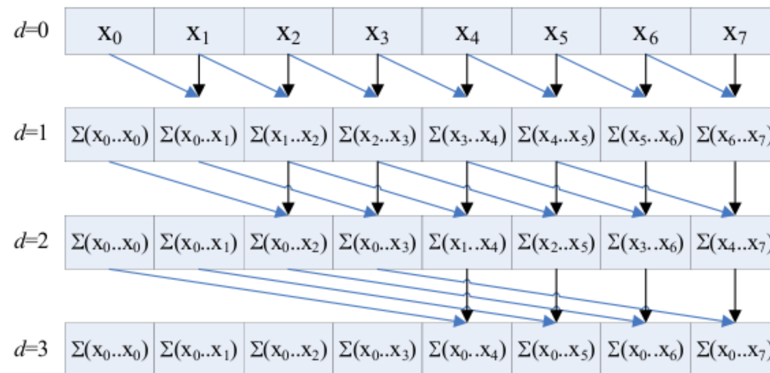
Parallel scan (3/5)

Here's a sequential algorithm for the exclusive scan.

```
void scan( float* output, float* input, int length)
{
    output[0] = 0; // since this is a prescan, not a scan
    for(int j = 1; j < length; ++j)
    {
        output[j] = input[j-1] + output[j-1];
    }
}
```

Parallel scan (4/5)

- Write a CREW algorithm for parallel scanning that would implement the principle used in the following example.



- Analyze its efficiency.

Parallel scan (5/5)

Input: Elements located in $M[1], \ldots, M[n]$, where n is a power of 2.

Output: The n prefix sums located in $M[n+1], \ldots, M[2n]$.

Program:

```
Active Proocessors P[1], ...,P[n]; // id the active proc
for d := 1 to log(n) do
if d is odd then
  if id >= 2^d then
     M[n + id] := M[id] + M[id - 2^(d-1)]
  else
     M[n + id] := M[id]
  end if
else
  if id >= 2^d then
     M[id] := M[n + id] + M[n + id - 2^(d-1)]
  else
     M[id] := M[n + id]
  end if
end if
if d is even then M[n + id] := M[id]
```


Mysterious algorithm (1/5)

- What does the following CREW-PRAM algorithm compute?

Input: n elements located in $M[1], \ldots, M[n]$, where $n \geq 2$ is a power of 2.

Output: Some values in located in $M[n+1], \ldots, M[2n]$.

Program:

```
Active Proocessors P[1], ...,P[n];
// id the index of one of the active processor
M[n + id] := M[id];
M[2 n + id] := id + 1;
for d := 1 to log(n) do
  if M[2 n + id] < n then {
     j := M[2 n + id];
     M[n + j] := M[n + id] + M[n + j];
     M[2 n + id] := M[2 n + j];
  }
}
```

- Analyze its efficiency.

Mysterious algorithm (2/4)

Mysterious algorithm (3/4)

Mysterious algorithm (4/4)