Introduction to PRAM

Oguz Kaya

Assistant Professor
Université Paris-Saclay and ParSys team at LISN, Orsay, France

Outline

1 Introduction to PRAM

Outline

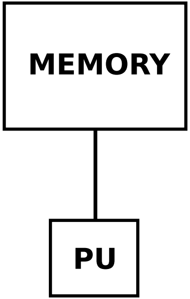
1. Introduction to PRAM

Objectives

- Revising sequential algorithms, RAM model, and complexity analysis
- Introducing the PRAM concept
- Explore and compare different PRAM variants
- Design and analyse some fundamental algorithms in PRAM model
- Analyze the complexity/performance of PRAM algorithms
- Brent and simulation theorems

RAM model

RAM (random access machine) model is an abstraction of computers



- Consists of a processing unit (PU) and an associated memory space.
- Access to each element in the memory is done in constant time.
- Each operation is performed in constant time in the PU.
- Equivalent to a turing machine (simulation)
- Belongs to the class "register machines"
- Not very realiste (L1/L2/L3/DRAM memory hierarchy) yet useful for developing algorithms that are "asymptotically optimal".
  - One should then adapt these algorithms to modern computer architectures, i.e. HPC.

Algorithms and complexity

Algorithm: A sequence of operations on a RAM machine, which aims to solve a given problem.

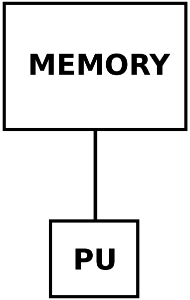
SUMARRAY(A[N])

```
1: sum = 0
2: for i = 1, ..., N do
3:     sum ← sum + A[i]
4: return sum
```

- Each memory access and arithmetic operation takes unit time.
- The **time complexity** of the algorithm is a function $T(N)$ of the problem size which represents the total number of operations it performs.
  - One can also consider its **space complexity** which represents the total memory space it uses in execution.

Complexity analysis

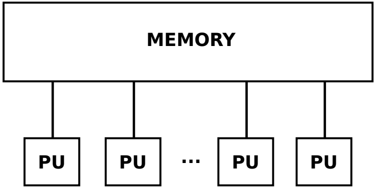
Many methods exist for complexity analysis:



- Nested loops: Counting techniques in the loop iteration domain
- Recursion / divide-and-conquer: Counting by hand, Master theorem
- Probabilistic analysis (i.e., quicksort)
- Amortized analysis (i.e., dynamic tables)
- These analysis could be performed for
  - the best case
  - the worst case
  - the average case

PRAM model

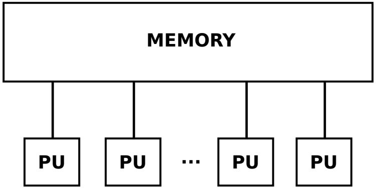
Parallel random access machine (PRAM) is an abstraction of parallel computers.



- Consists of P PUs and a memory space shared among all PUs.
- PUs
  - execute the same algorithm/code in a synchronous manner, but on different elements in the memory (i.e., à la SIMD).
  - can perform an aritmetic operation **simultaneously** in constant time.
  - can do reads/writes from/to different memory locations **simultaneously** in constant time.

PRAM model (cont.)

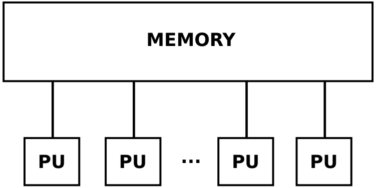
Parallel random access machine (PRAM) is an abstraction of parallel computers.



- Read/write on the same memory location?
    - This behavior defines different PRAM models.
- Number of PUs: Potentially infinite!
- Memory size: Potentially infinite!
- Communication cost: Completely ignored (done through fast shared memory implicitly)
    - PRAM is useful for developing "asymptotically optimal" parallel algorithms
    - One should then adapt these algorithms to modern parallel computer architectures.

Memory conflicts in PRAM

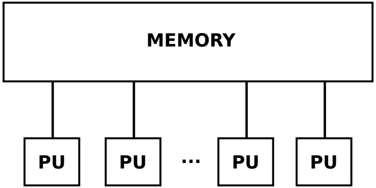
During the execution of a PRAM algorithm, multiple PUs could try to access to the same memory location (memory conflict).



- Possible resolutions?
- **CREW:** Concurrent Read Exclusive Write
  - Simultaneous reading of the same memory location is allowed, and takes constant time
  - Simultenous writing is forbidden (bug)
  - Standard PRAM model, closest to real machines

Memory conflicts in PRAM

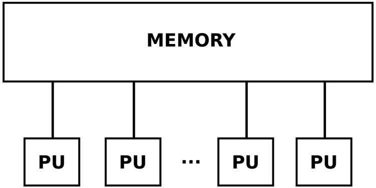
During the execution of a PRAM algorithm, multiple PUs could try to access to the same memory location (memory conflict).



- Possible resolutions?
- **CRCW:** Concurrent Read Concurrent Write
  - Simultaneous reading/writing from/to the same memory location is allowed, and takes constant time
  - The most powerful model
  - A "concurrency mode" is employed for simultenous writes:
    - Consistent mode: All PUs write the same value.
    - Arbitrary mode: One PUs value is retained randomly.
    - Priority mode: The PU with min/max index's value is retained.
    - Fusion mode: A commutative and associative operation is applied (+,*,min,max,and,or,xor,etc.)

Memory conflicts in PRAM

During the execution of a PRAM algorithm, multiple PUs could try to access to the same memory location (memory conflict).



- Possible resolutions?
- **EREW:** Exclusive Read Exclusive Write
  - Simultaneous read/write of the same memory location by multiple PUs is forbidden.
  - The most restrictive model
- **ERCW:** Exclusive Read, Concurrent Write
  - Does not exist. Why?
    - Does not make sense architecturally.

A first PRAM algorithm: Search in an array

Given an array $A[N]$ and a key e, find the unique element index in A that contains e.

SEARCHINDEX(A[N], e)

```
idx = 0
forall i ← 1 ... N in parallel do
    if e = A[i] then
        idx ← i
return idx
```

- Sequential algorithm/complexity?
  - Iterate over all array elements, $O(N)$.
- PRAM complexity?
  - $O(1)$ time complexity since all comparisons are done simultaneously.
  - Is EREW sufficient? Or do we need CREW, CRCW?
    - EREW OK as long as the element is unique.
    - What about reading the variable N?

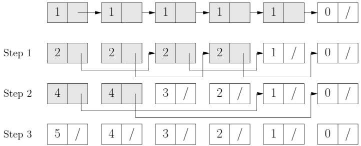
Another algorithm: List ranking

Given a linked list $next[N]$, find the distances from the end of the list for each node.

- $next[i] = \mathbf{Nil} \implies d[i] = 0$
- $next[i] = j \neq \mathbf{Nil} \implies d[i] = d[j] + 1$
- Sequential algorithm? Complexity?
    - Recursion + memoisation in $O(N)$.

List ranking (cont.)

How to parallelize in PRAM?



- Idea: Recursive pointer jumping
  - First, each pointer jump adds 1 distance
  - Then, as we jump over pointers, we add the distance of that pointer.
  - Continue until all pointers point to the end of list.
  - How many iterations until convergence?
    - At each jump, either jump distance of a pointer doubles, or it points to the end of list.
    - In $\log_2 N$ itérations, all pointers point to the end of list, with correct jump distance $d[i]$.

List ranking (cont.)

How to write the PRAM algorithm?

```
RANK_COMPUTATION(L)
  forall i in parallel do                                { Initialization }
    if next[i] = Nil then d[i] ← 0 else d[i] ← 1
  while there exists a node i such that next[i] ≠ Nil do   { Main loop }
    forall i in parallel do
      if next[i] ≠ Nil then
        d[i] ← d[i] + d[next[i]]
        next[i] ← next[next[i]]
```

- Initialization is done in $O(1)$.
- Computation terminates in $\log_2 N$ iterations using N PUs.

List ranking (cont.)

Read/write conflicts in lines 7, 8?

1 $\textsc{Rank\_Computation}(L)$

2   **forall** i **in parallel do** { *Initialization* }

3   **if** $next[i] = $ **Nil then** $d[i] \leftarrow 0$ **else** $d[i] \leftarrow 1$

4   **while** there exists a node i such that $next[i] \neq$ **Nil do** { *Main loop* }

5   **forall** i **in parallel do**

6   **if** $next[i] \neq$ **Nil then**

7         $d[i] \leftarrow d[i] + d[next[i]]$

8         $next[i] \leftarrow next[next[i]]$

- We consider it as a sequence of synchronous operations:
  - $temp1[i] \leftarrow d[next[i]]$
  - $temp2[i] \leftarrow d[i] + temp1[i]$
  - $d[i] \leftarrow temp2[i]$
- Asymptotically the same!
- No read nor write conflicts! EREW is sufficient.

List ranking (cont.)

How to know when to stop?

```
RANK_COMPUTATION(L)
  forall i in parallel do                                    { Initialization }
    if next[i] = Nil then d[i] ← 0 else d[i] ← 1
  while there exists a node i such that next[i] ≠ Nil do     { Main loop }
    forall i in parallel do
      if next[i] ≠ Nil then
        d[i] ← d[i] + d[next[i]]
        next[i] ← next[next[i]]
```

- CRCW PRAM?
  - Write *done* $\leftarrow (next[i] = \textbf{Nil})$ at the end of while
  - For write conflicts, use either "min" or "and" fusion mode
- CREW PRAM?
  - Each process sets $done[i] \leftarrow (next[i] = \textbf{Nil})$ at the end of loop
  - Run the reduction algorithm to compute the global *done* ($O(\log N)$ CREW, à venir).

List ranking (cont.)

How to know when to stop?

```
RANK_COMPUTATION(L)
  forall i in parallel do                                  { Initialization }
    if next[i] = Nil then d[i] ← 0 else d[i] ← 1
  while there exists a node i such that next[i] ≠ Nil do    { Main loop }
    forall i in parallel do
      if next[i] ≠ Nil then
        d[i] ← d[i] + d[next[i]]
        next[i] ← next[next[i]]
```

- EREW PRAM?
  - Each process sets $done[i] \leftarrow (next[i] = \mathbf{Nil})$ at the end of loop
  - Run the reduction algorithm to compute the global *done* ($O(\log N)$ CREW, à venir).
  - Run a broadcast algorithm to copy the global *done* in each *done*[*i*] ($O(\log N)$ en EREW, à venir).
  - Complexity? $O(\log N \log N) = O(\log^2 N)$?
    - Perform this check every $\log N$ itérations.
    - Amortized cost per iteration is $O(1)$.

Prefix sum computation on a list

Given a linked list of values $(x_1, \dots, x_N)$ and a binary associative operation $\otimes$, compute the sequence $(y_1, \dots, y_N)$ such that $y_k = x_1 \otimes \cdots \otimes x_k$

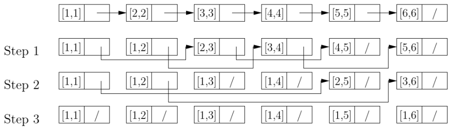


FIGURE 1.3: Example execution of the prefix computation algorithm. $[i, j]$ denotes $x_i \otimes x_{i+1} \otimes \ldots \otimes x_j$ for $i \leqslant j$.

- x is in a linked list form
  - $x[i]$ gives the value
  - $next[i]$ gives the index to the next element
- Use the pointer jumping technique
  - Each element i applies itself to its $next[i]$ before jumping
  - $O(\log N)$ iterations maximum

Prefix sum computation on a list (cont.)

Given a linked list of values $(x_1, \ldots, x_N)$ and a binary associative operation $\otimes$, compute the sequence $(y_1, \ldots, y_N)$ such that $y_k = x_1 \otimes \cdots \otimes x_k$

```
PREFIX_COMPUTATION(L)
  forall i in parallel do                                  { Initialization }
    y[i] ← x[i]
  while there exists a node i such that next[i] ≠ Nil do   { Main loop }
    forall i in parallel do
      if next[i] ≠ Nil then
        y[next[i]] ← y[i] ⊗ y[next[i]]
        next[i] ← next[next[i]]
```

- x is in a linked list form
  - $x[i]$ gives the value
  - *next*$[i]$ gives the index to the next element
- Use the pointer jumping technique
  - Each element i applies itself to its *next*$[i]$ before jumping
  - $O(\log N)$ iterations maximum
  - What type of PRAM is needed?

Evaluating the performance of PRAM algorithms

For a problem $A(n)$ of size n and corresponding sequential and parallel algorithms, we define the following metrics for evaluation:

- $T_{seq}(n)$: Sequential execution time
- $T_{par}(n, p)$: Parallel execution time using P PUs
- $C_p(n) = pT_{par}(p, n)$: Parallel execution cost
- $W_p(n)$: Total work performed in parallel (total number of operations across all PUs)
- If $W_p(n) = O(T_{seq})$, PRAM algorithm is **work-optimal**.
- $D(n) = \lim_{p->\infty} T_{par}(n, p)$: Depth of the parallel algorithm
- $T_{seq}(n) \leq W_p(n) \leq C_p(n)$: Parallelism potentially gives an overhead
- $S_p(n) = \frac{T_{seq}(n)}{T_{par}(p,n)}$: Speedup
- $E_p(n) = \frac{S_p(n)}{p} = \frac{T_{seq}(n)}{pT_{par}(p,n)}$: Efficiency

A simple simulation

Let A be an algorithm whose execution time is t using PRAM with p PUs. Then, A can be simulated with a same type PRAM with $p' \leq p$ PUs in $O(\frac{tp}{p'})$ time. The cost of the algorithm on this latter PRAM (having p' PUs) is at most $2x$ the cost of the initial PRAM with p PUs.

- Each "big" step of the PRAM can be simulated in $\lceil \frac{p}{p'} \rceil$ substeps.
- There are at most t such substeps, thus $t' = O(\frac{p}{p'}t) = O(\frac{tp}{p'})$.
- New cost :

$$C_{p'} = t'p' \leq \lceil \frac{p}{p'} \rceil p't \leq (\frac{p}{p'} + 1)p't = pt(1 + \frac{1}{p'}) = C_p(1 + 1/p') \leq 2C_p$$

- **Corollary:** The cost of a parallel execution of a PRAM algorithm is asymptotically the same or superior to the best sequential algorithm
- If the cost of PRAM algorithm is asymptotically equivalent to its sequential time, then the algorithm is called **efficient**.

Brent theorem

Let A be an algorithm that performes m operations in t time using PRAM (with a certain, potentially infinite, number of PUs). Then, A can be simulated with a PRAM of the same type having p PUs in time $O(\frac{m}{p} + t)$.

- Let m_i be the number of operations performed in the step i ($\sum_{i=1}^{t} m_i = m$).
- Each coarse step of PRAM having m_i operations could be simulated in $\lceil \frac{m_i}{p'} \rceil$ time using a PRAM with p PUs.
- $$t' = \sum_{1}^{t} \lceil \frac{m_i}{p} \rceil \leq \sum_{1}^{t} (\frac{m_i}{p} + 1) = \frac{m}{p} + t$$

Comparison between the power of PRAM models

Is there a problem that one can solve asymptotically faster using CRCW PRAM instead of CREW PRAM

```
COMPUTE_MAXIMUM(A, n)
    forall i ∈ {1, ..., n} in parallel do
        m[i] ← True
    forall i, j ∈ {1, ..., n}^2, i ≠ j in parallel do
        if A[i] < A[j] then m[i] ← False
    forall i ∈ {1, ..., n} in parallel do
        if m[i] = True then max ← A[i]
    return max
```

- Find the maximum of an array $A[N]$
- Can be performed in $O(1)$ time using CRCW
- CREW requires at least $O(\log_2 n)$ due to reduction

Comparison between the power of PRAM models (cont.)

Is there a problem that one can solve asymptotically faster using CREW PRAM instead of EREW PRAM

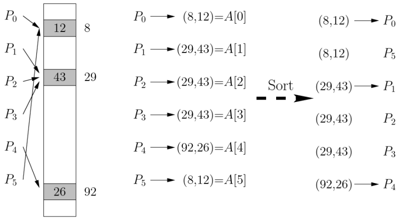
SEARCH(A[n], e)

1: $idx = 0$
2: **forall** $i = 1, \ldots, N$ **in parallel do**
3:     **if** $e = A[i]$ **then**
4:         $idx \leftarrow i$
5: **return** idx

- Search an element e in an array (with unique elements) $A[N]$
- Could be performed in $O(1)$ using CREW
- EREW requires at least $O(\log_2 n)$ for diffusing e to all PUs.

Simulation theorem

The execution time of a CRCW PRAM algorithm using P PUs is at most $O(\log N)$ times faster than the best EREW PRAM algorithm using P PUs for the same problem



(a) CRCW write access

(b) EREW simulation

- To simulate each step of CRCW using EREW in consistent mode
- For each write at an index, put (index, value) pair in an array $A[P]$.
- Sort $A[P]$ ($O(\log P)$ in EREW PRAM)
- Perform the write for $A[0]$, then for $A[i]$ if $A[i] \neq A[i-1]$.

References

Contact

Oguz Kaya
Université Paris-Saclay and LRI, Paris, France
oguz.kaya@lri.com
www.oguzkaya.com