Fundamental Algorithms

Chapter 6: Parallel Algorithms – The PRAM Model

Jan Křetínský

Winter 2017/18

Example: Parallel Sorting

Definition

Sorting is required to order a given sequence of elements, or more precisely:

Input : a sequence of n elements $a_1, a_2, \ldots, a_n$

Output : a permutation (reordering) $a'_1, a'_2, \ldots, a'_n$ of the input sequence, such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example: Parallel Sorting

Definition

Sorting is required to order a given sequence of elements, or more precisely:

Input : a sequence of n elements $a_1, a_2, \ldots, a_n$

Output : a permutation (reordering) $a'_1, a'_2, \ldots, a'_n$ of the input sequence, such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

A naive(?) solution:

- pairwise comparison of all elements
- count "wins" for each element to obtain its position
- use one processor for each comparison!

A (Naive?) Parallel Example: AccumulateSort

```
AccumulateSort (A:Array[1..n]) {

    Create Array P[1..n] of Integer;
    // all P[i]=0 at start

    for 1 <= i,j <= n and i<j do in parallel {
        if A[i] > A[j]
        then P[i] := P[i]+1
        else P[j] := P[j]+1;
    }

    for i from 1 to n do in parallel {
        A[ P[i]+1 ] := A[i];
    }
}
```

AccumulateSort – Discussion

Implementation:

- do all $\binom{n}{2}$ comparisons at once and in parallel
- use $\binom{n}{2}$ processors
- count “wins” for each element; then move them to their respective “rank”
- complexity: $T_{\mathrm{AS}} = \Theta(1)$ on $n(n-1)/2$ processors

AccumulateSort – Discussion

Implementation:

- do all $\binom{n}{2}$ comparisons at once and in parallel
- use $\binom{n}{2}$ processors
- count "wins" for each element; then move them to their respective "rank"
- complexity: $T_{\text{AS}} = \Theta(1)$ on $n(n-1)/2$ processors

Assumptions:

- all read accesses to A can be done in parallel
- increments of P[i] and P[j] can be done in parallel
- second for-loop is executed after the first one (on all processors)
- all moves A[P[i]] := A[i] happen in one atomic step (no overwrites due to sequential execution)

Example: Parallel Searching

Definition (Search Problem)

Input: a set A of n elements $\in \mathcal{A}$, and an element $x \in \mathcal{A}$.
Output: The (smallest) index $i \in \{1, \ldots, n\}$ with $x = A[i]$.

Example: Parallel Searching

Definition (Search Problem)

Input: a set A of n elements $\in \mathcal{A}$, and an element $x \in \mathcal{A}$.
Output: The (smallest) index $i \in \{1, \ldots, n\}$ with $x = A[i]$.

An immediate solution:

- use n processors
- on each processor: compare x with $A[i]$
- return matching index/indices i

Simple Parallel Searching

```
ParSearch(A:Array[1..n], x:Element) : Integer {
    for i from 1 to n do in parallel {
        if x = A[i] then return i;
    }
}
```

Simple Parallel Searching

```
ParSearch(A:Array[1..n], x:Element) : Integer {
    for i from 1 to n do in parallel {
        if x = A[i] then return i;
    }
}
```

Discussion:

- Can all n processors access x simultaneously?
  → **exclusive** or **concurrent** read
- What happens if more than one processor finds an x?
  → **exclusive** or **concurrent** write (of multiple returns)
- general approach: parallelisation by "competition"

Towards Parallel Algorithms

First Problems and Questions:

- parallel read access to variables possible?
- parallel write access (or increments?) to variables possible?
- are parallel/global copy statements realistic?
- how do we synchronise parallel executions?

Towards Parallel Algorithms

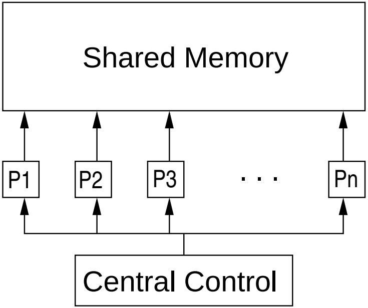
First Problems and Questions:

- parallel read access to variables possible?
- parallel write access (or increments?) to variables possible?
- are parallel/global copy statements realistic?
- how do we synchronise parallel executions?

Reality vs. Theory:

- on real hardware: probably lots of restrictions (e.g., no parallel reads/writes; no global operations on or access to memory)
- in theory: if there were no such restrictions, how far can we get?
- or: for different kinds of restrictions, how far can we get?

The PRAM Models



Concurrent or Exclusive Read/Write Access:

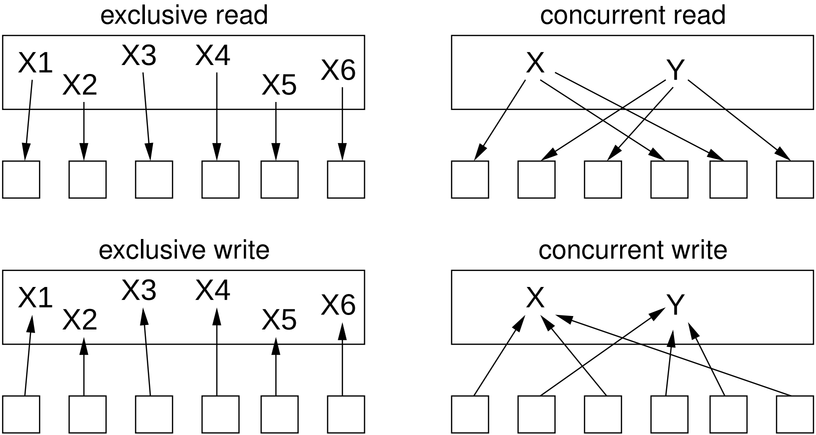
EREW exclusive read, exclusive write

CREW concurrent read, exclusive write

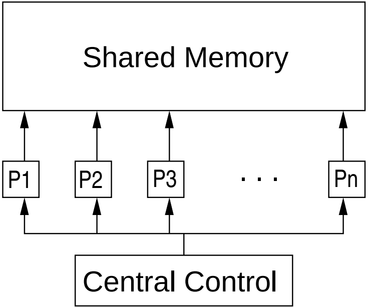
ERCW exclusive read, concurrent write

CRCW concurrent read, concurrent write

Exclusive/Concurrent Read and Write Access



The PRAM Models (2)



SIMD

- Underlying principle for parallel hardware architecture: strict single instruction, multiple data (SIMD)

$\Rightarrow$ All parallel instructions of a parallelized loop are performed synchronously (applies even to simple if-statements)

Loops and If-Statements in PRAM Programs

Lockstep Execution of parallel for:

- Parallel for-loops (i.e., with extension **in parallel**) are executed "in lockstep".
- Any instruction in a parallel for-loop is executed at the same time (and "in sync") by all involved processors.
- If an instruction consists of several substeps, all substeps are executed in sync.
- If an if-then-else statement appears in a parallel for-loop, all processors first evaluate the comparison at the same time. Then, all processors on which the condition evaluates as **true** execute the then branch. Finally, all processors on which the condition evaluates to **false** execute the else branch.

Lockstep Example:

```
for i from 1 to n do in parallel {
    if U[i] > 0
    then F[i] := (U[i]−U[i−1]) / dx
    else F[i] := (U[i+1]−U[i]) / dx
    end if
}
```

- First, all processors perform the comparison U[i]>0
- All processors where U[i]>0 then compute F[i]; note that first all processors read U[i] and then all processors read U[i−1] (substeps!); hence, there is no concurrent read access!
- Afterwards, the else-part is executed in the same manner by all processors with U[i]<=0

Parallel Search on an EREW PRAM

ToDos for exclusive read and exclusive write:

- avoid exclusive access to x
  $\Rightarrow$ replicate x for all processors ("broadcast")
- determine smallest index of all elements found:
  $\Rightarrow$ determine minimum in parallel

Parallel Search on an EREW PRAM

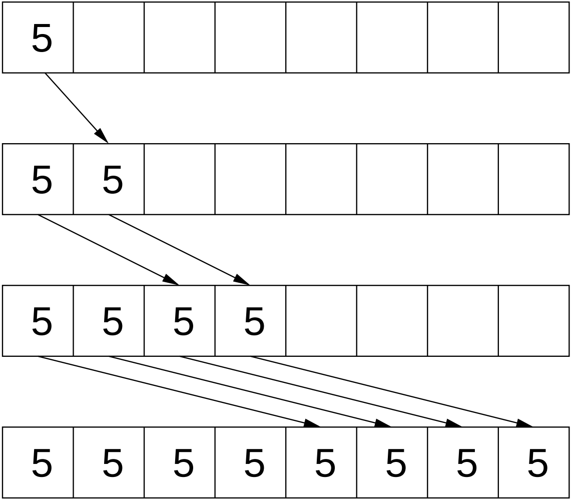
ToDos for exclusive read and exclusive write:

- avoid exclusive access to x
  $\Rightarrow$ replicate x for all processors (“broadcast”)
- determine smallest index of all elements found:
  $\Rightarrow$ determine minimum in parallel

Broadcast on the PRAM:

- copy x into all elements of an array X[1..n]
- note: each processor can only produce one copy per step

Broadcast on the PRAM – Copy Scheme



Broadcast on the PRAM – Implementation

```
BroadcastPRAM( x:Element, A:Array[1..n]) {
    // n assumed to be 2^k
    // Model: EREW PRAM

    A[1] := x;
    for i from 0 to k-1 do
        for j from 2^i+1 to 2^(i+1) do in parallel {
            A[j] := A[j-2^i];
        }
}
```

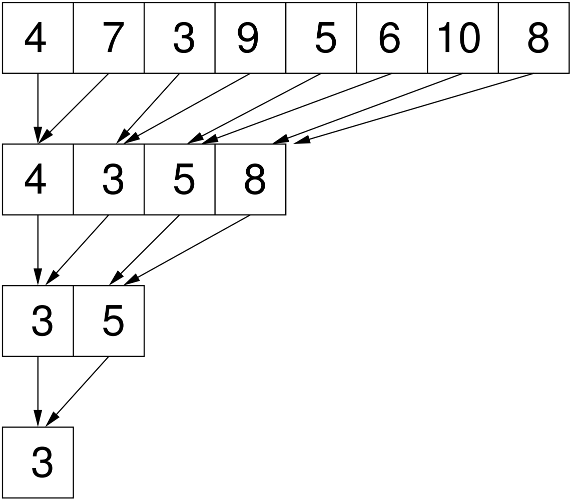
Broadcast on the PRAM – Implementation

```
BroadcastPRAM( x:Element, A:Array[1..n]) {
    // n assumed to be 2^k
    // Model: EREW PRAM

    A[1] := x;
    for i from 0 to k-1 do
        for j from 2^i+1 to 2^(i+1) do in parallel {
            A[j] := A[j-2^i];
        }
}
```

Complexity:

- $T(n) = \Theta(\log n)$ on $\frac{n}{2}$ processors

Minimum Search on the PRAM – "Binary Fan-In"



Minimum on the PRAM – Implementation

```
MinimumPRAM( A:Array[1..n]) : Integer {
    // n assumed to be 2^k
    // Model: EREW PRAM

    for i from 1 to k do
        for j from 1 to n/(2^i) do in parallel {
            if A[2*j-1] < A[2*j]
            then A[2*j] := A[2*j-1];
            end if;
            A[j] := A[2*j];
        }
    return A[1];
}
```

Minimum on the PRAM – Implementation

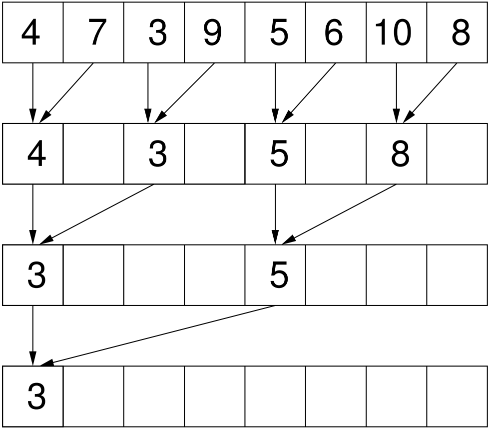
```
MinimumPRAM( A:Array[1..n]) : Integer {
    // n assumed to be 2^k
    // Model: EREW PRAM

    for i from 1 to k do
        for j from 1 to n/(2^i) do in parallel {
            if A[2*j-1] < A[2*j]
            then A[2*j] := A[2*j-1];
            end if;
            A[j] := A[2*j];
        }
    return A[1];
}
```

Complexity: $T(n) = \Theta(\log n)$ on $\frac{n}{2}$ processors

“Binary Fan-In” (2)

Comment Concerned about synchronous copy statement?

$\Rightarrow$ Modifiy stride!



Searching on the PRAM – Parallel Implementation

```
SearchPRAM( A:Array[1..n], x:Element) : Integer {
    // n assumed to be 2^k
    // Model: EREW PRAM

    BroadcastPRAM(x, X[1..n]);

    for i from 1 to n do in parallel {
        if A[i] = X[i]
        then X[i] := i;
        else X[i] := n+1;  // (invalid index)
        end if;
    }

    return MinimumPRAM(X[1..n]);
}
```

The Prefix Problem

Definition (Prefix Problem)

Input: an array A of n elements a_i.
Output: All terms $a_1 \times a_2 \times \cdots \times a_k$ for $k = 1, \ldots, n$.

$\times$ may be any associative operation.

The Prefix Problem

Definition (Prefix Problem)

Input: an array A of n elements a_i.
Output: All terms $a_1 \times a_2 \times \cdots \times a_k$ for $k = 1, \ldots, n$.

$\times$ may be any associative operation.

Straightforward serial implementation:

```
Prefix( A:Array[1..n]) {
   // in-place computation:
   for i from 2 to n do {
         A[i] := A[i-1]*A[i];
   }
```

The Prefix Problem – Divide and Conquer

Idea:

1. compute prefix problem for $A_1, \dots, A_{n/2}$
   $\rightarrow$ gives $A_{1:1}, \dots, A_{1:n/2}$
2. compute prefix problem for $A_{n/2+1}, \dots, A_n$
   $\rightarrow$ gives $A_{n/2+1:n/2+1}, \dots, A_{n/2+1:n}$
3. multiply $A_{1:n/2}$ with all $A_{n/2+1:n/2+1}, \dots, A_{n/2+1:n}$
   $\rightarrow$ gives $A_{1:n/2+1}, \dots, A_{1:n}$

The Prefix Problem – Divide and Conquer

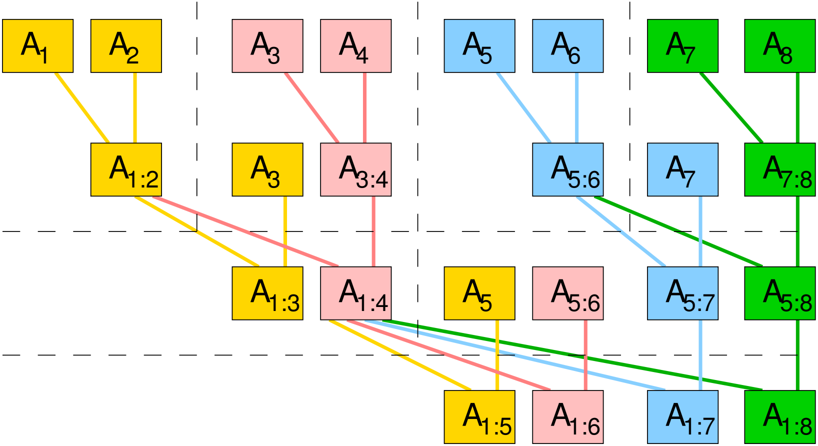
Idea:

1. compute prefix problem for $A_1, \dots, A_{n/2}$
   $\rightarrow$ gives $A_{1:1}, \dots, A_{1:n/2}$
2. compute prefix problem for $A_{n/2+1}, \dots, A_n$
   $\rightarrow$ gives $A_{n/2+1:n/2+1}, \dots, A_{n/2+1:n}$
3. multiply $A_{1:n/2}$ with all $A_{n/2+1:n/2+1}, \dots, A_{n/2+1:n}$
   $\rightarrow$ gives $A_{1:n/2+1}, \dots, A_{1:n}$

Parallelism:

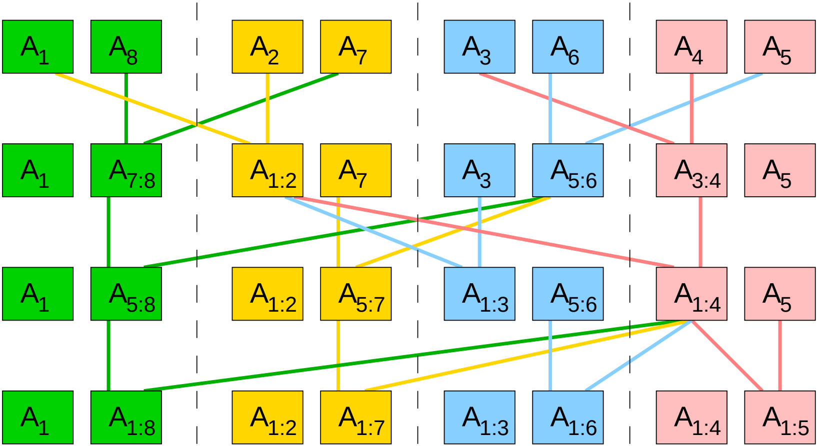
- steps 1 and 2 can be computed in parallel (divide)
- all multiplications in step 3 can be computed in parallel
- recursive extension leads to parallel prefix scheme

Parallel Prefix – Divide and Conquer



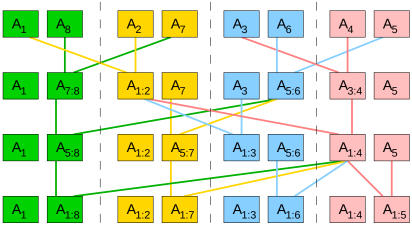
Parallel Prefix Scheme on a CREW PRAM

Additional Feature: In-Place Computation, Pin Elements to Cores



Outlook: Parallel Prefix on Distributed Memory

Consider scheme from previous slide:



Execution on Distributed Memory:

- Each color corresponds to one compute node
- Nodes cannot directly access matrices from a node with different colour
  $\rightarrow$ explicit data transfer (communication) required

Properties of the Distributed-Memory Parallel Prefix Scheme:

- In-place computation; A[1:n] will overwrite A[n]; all A[j:n] stored on the same node
- One of the two multiplied matrices is always local
- Still, $n/2$ outgoing messages from A[1:n/2] in the last step (bottleneck!)

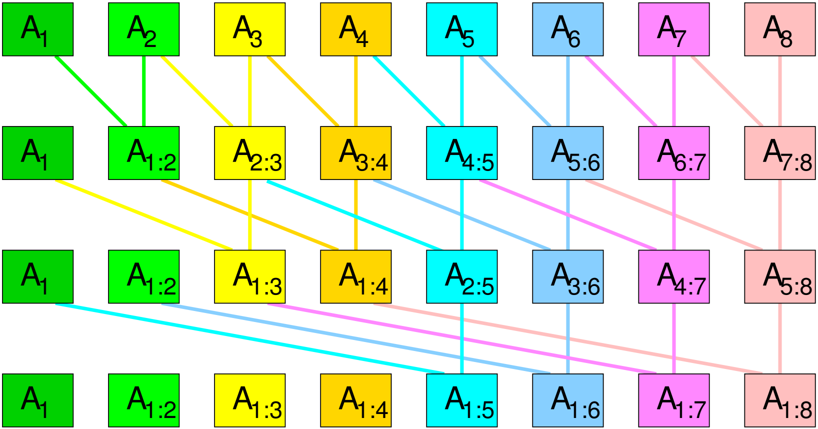
Parallel Prefix – CREW PRAM Implementation

```
PrefixPRAM( A:Array[1..n]) {
  // n assumed to be 2^k
  // Model: CREW PRAM (n/2 processors)

  for l from 0 to k−1 do
     for p from 2^l by 2^(l+1) to n do in parallel
        for j from 1 to 2^l do in parallel {
           A[p+j] := A[p]*A[p+j];
        }
}
```

Comments:

- p- and j-loop together: n/2 multiplications per l-loop
- concurrent read access to A[p] in the innermost loop

Parallel Prefix Scheme on an EREW PRAM



Parallel Prefix – EREW PRAM Implementation

```
PrefixPRAM( A:Array[1..n]) {
  // n assumed to be 2^k
  // Model: EREW PRAM (n-1 processors)

  for l from 0 to k-1 do
     for j from 2^l+1 to n do in parallel {
        tmp[j] := A[j-2^l];
        A[j] := tmp[j]*A[j];
     }
}
```

Comment:

- all processors execute tmp[j] := A[j-2ˆl] before multiplication!