Fundamental Algorithms

Chapter 3: Parallel Algorithms – The PRAM Model

Michael Bader
Winter 2013/14
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 {
}
}
AccumulateSort – Discussion

Idea:

• do all \( \binom{n}{2} \) comparisons at once and in parallel
• use \( \binom{n}{2} \) processors
• count “wins” for each element to obtain its position
• complexity: \( T_{AS} = \Theta(1) \) 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)
  (no overwrites due to sequential execution)
Example: Parallel Searching

Definition (Search Problem)

**Input:** a set $A$ of $n$ elements $\in A$, and an element $x \in A$.

**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;
    }
}

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)
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

exclusive read

exclusive write

concurrent read

concurrent write
The PRAM Models (2)

SIMD

- Underlying principle for parallel hardware architecture: strict single instruction, multiple data (SIMD)
  ⇒ All parallel instructions of a parallelized loop are performed synchronously (applies even to simple if-statements)
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

5

5 5

5 5 5 5

5 5 5 5 5 5 5 5

5 5 5 5 5 5 5 5 5 5
Broadcast on the PRAM – Implementation

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

    for i from 0 to k−1 do
        for j from 2^i+1 to 2^(i+1) do in parallel {
        }
}

Complexity:

- \( T(n) = \Theta(\log n) \) on \( \frac{n}{2} \) processors
Minimum Search on the PRAM – “Binary Fan-In”

4  7  3  9  5  6  10  8

4  3  5  8

3  5

3
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
      else
      end if;
  }
  return A[1];
}

Complexity: \( T(n) = \Theta(\log n) \) on \( \frac{n}{2} \) processors
“Binary Fan-In” (2)

**Comment** Concerned about synchronous if-statement (guaranteed by SIMD assumptions)?

⇒ Modify 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.

**Straightforward serial implementation:**

```plaintext
Prefix ( A: Array [1..n] ) {
    // in-place computation:
    for i from 2 to n do {
    }
}
```
The Prefix Problem – Divide and Conquer

Idea:

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

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 Scheme on a CREW PRAM
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 {
            }
    }

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];
        }
}

Comment: