CS256
Applied Theory of Computation
Parallel Computation IV
John E Savage
Overview

- PRAM
- Work-time framework for parallel algorithms
- Prefix computations
- Finding roots of trees in a forest
- Parallel merging
- Parallel partitioning
- Sorting
- Simulating CREW on EREW PRAM
PRAM Model

- The PRAM is an abstract programming model
  - Processors operate synchronously, reading from memory, executing locally, and writing to memory.
PRAM Model

- Four types: EREW, ERCW, CREW, CRCW
  - R – read, W – write, E – exclusive, C – common

- Can Boolean functions be computed quickly?
  - How can a function be represented?
  - Can we use concurrency to good advantage?
  - Is this use of concurrency realistic?

- Good source: Intro to Parallel Algs by Jaja
Matrix Multiplication on PRAM

Input:  n×n matrices A and B  
Output: n×n matrix C, local vars. C’(i,j,l), n

begin  
Compute C(i,j,l) = A(i,l)B(l,j)  
For h = 1 to log₂ n do  
  if (l ≤ n) then C(i,j,l) := C(i,j,2l-1) + C’(i,j,2l)  
  If (l = 1) then C(i,j) := C(i,j,l)  
End  

Running time on CREW is O(log n).  

Why is CREW necessary?
Measuring Performance of Parallel Programs

- Let $T(n)$ time and $P(n)$ processors be used on a parallel machine on a problem with $n$ inputs.

- **Cost**: $C(n) = P(n)T(n)$ is the time $\times$ processor product, or work, for problem on $n$ inputs.

- An equivalent serial algorithm will run in time $O(C(n))$.

- If $p \leq P(n)$ processors available, we can implement the algorithm in time $O(P(n)T(n)/p)$ or $O(C(n)/p)$ time.
Advantages of PRAM

- A body of algorithms exist for this shared memory model.
- The model ignores algorithmic details of synchronization and communication.
- It makes explicit the association between operations and processors.
- PRAM algorithms are robust – network-based algorithms can be derived from them.
- PRAM is MIMD model.
Work-Time Framework for Parallel Algorithms

- Informal guideline to algorithm performance on PRAM.
- Work-time framework exhibits parallelism.
  - Use `for l \leq i \leq u \text{pardo}` for parallel operations
  - Also allow serial straight-line and branching ops

- $W(n)$ (work) is total no. of ops on $n$ inputs
- $T(n)$ is the running time of algorithm
Work-Time Framework for Parallel Algorithms

Sum
Input: $n = 2^k$ inputs in array $A[n]$
Output: $\text{Sum } S = A(1) + A(2) + \ldots + A(n)$

begin
  Copy $A[n]$ to $B[n]$
  for $h = 1$ to $\log_2 n$ do
    for $1 \leq i \leq n/2^h$ pardo
      $B(i) := B(2i-1) + B(2i)$
    $S := B(1)$
end
Work-Time Framework for Parallel Algorithms

- **Rescheduling**: Generally through rescheduling the following bounds can be achieved.
  - If $p$ procs. available, time is $\leq W(n)/p + T(n)$.
  - Cost $C(n) = p(W(n)/p + T(n)) \leq O(W(n) + p \cdot T(n))$
When Is a Parallel Computation Optimal?

- If the smallest parallel work is about the same as the best serial time, the parallel machine cannot run substantially faster.
Prefix Computation – Powerful Operation with Many Applications

**Input:** $(x_1, \ldots, x_n)$, $n = 2^k$ elements of S.

**Output:** $(s_1, \ldots, s_n)$, $s_i = x_1 \ast \ldots \ast x_i$, $\ast$ an associative op

```plaintext
begin
    If $n = 1$ then ($s_1 := x_1$; exit)
    for $1 \leq i \leq n/2$ pardo
        $y_i = x_{2i-1} \ast x_{2i}$
    Compute prefix $(z_1, \ldots, z_{n/2})$ from $(y_1, \ldots, y_{n/2})$
    for $1 \leq i \leq n$ pardo
        i even $s_i := z_{i/2}$
        i = 1 $s_1 := x_1$
        i odd $s_i := z_{(i-1)/2} \ast x_i$
end
```
Prefix Computation

- Graphical illustration

![Graphical illustration of prefix computation with variables and indices.](image-url)
Prefix Sum

- We show algorithm does work $W(n) = O(n)$ and uses time $T(n) = O(\log n)$.
- The work and time to compute $(z_1, \ldots, z_{n/2})$ from $(y_1, \ldots, y_{n/2})$ are $W(n/2)$ and $T(n/2)$. Additional work to compute prefix sum is $bn$ for $b > 0$. Additional time is $a > 0$.
  - $W(n) = W(n/2) + bn$
  - $T(n) = T(n/2) + a$
- These imply $W(n) = O(n)$, $T(n) = O(\log n)$. 

Examples of Prefix Computations

- Integer addition and multiplication
- Max and min
- Boolean AND and OR
- \texttt{copy\_right}: \ a \ @ \ b = a
Prefix Computations

**Segmented prefix computation**

- Provide two n-tuple vectors:
  - Elements \((x_1, \ldots, x_n)\) from \(S\) and \((f_1, \ldots, f_n)\), \(f_j\) in \(B = \{0, 1\}\).
- Output: \(y_i = x_i\) if \(f_j = 1\) else \(y_i = x_i \ast y_{i-1}\).
- Alternate input: \(((x_1, y_1), \ldots, (x_n, y_n))\)
- New operation: \& : \((S \times B)^2 \rightarrow S\)
- \& is associative, i.e.
  - \((x_1, y_1) \& (x_2, y_2) \& (x_3, y_3)) = ((x_1, y_1) \& (x_2, y_3)) \& (x_3, y_3)\)
Finding Roots of a Tree in a Forest

- **Pointer doubling** in a linked list: replace the pointer from node $i$ to its successor $P(i)$ by a pointer from the successor, namely $P(P(i))$.
- Given a forest of rooted trees, find roots $\{S(i)\}$. A root $r$ has $S(r) = r$.

```
begin
  for 1 ≤ i ≤ n pardo
    S(i) := P(i)
    while S(i) ≠ S(S(i)) do
      S(i) := S(S(i))
```

Parallel Merging

- Two sorted sequences of length $n$ can be merged in $O(\log n)$ time on $O(n)$ operations.
- **Goal:** Compute $\text{rank}(A:B)$, the rank elements of $A$ in $B$, two sets of distinct elements.
- **Approach:** rank individual elements of $A$ in $B$. Rank $x$ in $A$ using *binary search* of $x$ in $A$. 
Parallel Partitioning on CREW

- Given sorted arrays of distinct elements $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_m)$, where $\log_2 m$ and $k(m) = m/\log_2 m$ are integers, partition $B$ into $k$ segments of length $\log_2 m$, $B_1, \ldots, B_k$.
- Partition $A$ into $k(m)$ contiguous segments $A_1, \ldots, A_{k(m)}$ such that elements in $A_i$ and $B_i$ are less than all elements in $A_{i+1}$ and $B_{i+1}$ and greater than all elements in $A_{i-1}$ and $B_{i-1}$.
- Procedure: Compute $j(i) = rank(b_i \log m : A)$ and use to define $A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)})$. 

Parallel Partitioning on CREW

1. \( j(0) := 0, j(k(m)) := n \)
2. \( \text{for } 1 \leq i \leq k(m)-1 \pardo j(i) := \text{rank}(b_{i \log m} : A) \) (use binary search)
3. \( \text{for } 0 \leq i \leq k(m)-1 \pardo B_i = (b_{i \log m + 1}, \ldots, b_{(i+1) \log m}) \)
   \( A_i = (a_{j(i)+1}, \ldots, a_{j(i+1)}) \)
Parallel Partitioning

**Theorem** Parallel partitioning of sorted lists $A = (a_1, \ldots, b_n)$ and $B = (b_1, \ldots, b_m)$ takes $O(\log n)$ time and $O(n+m)$ operations.

**Proof** Step 1. takes $O(1)$ sequential time.
Step 2. takes $O(\log n)$ parallel time. It uses $O((\log n) \times (m/\log m)) = O(n+m)$ operations.
Step 3. takes $O(1)$ parallel time and uses linear number of operations.
Merging

- Merging of the segments $A_i$ and $B_i$ will produce sorted list. However, length of $A_i$ may be large. To reduce running time, in parallel partition the pairs $(B_i,A_i)$ so that segments have length at most $\log_2 n$.

- Merging of two sorted lists of length $n$ on CREW takes time $O(\log n)$ and $O(n)$ operations.

- Merging time can be reduced to $O(\log \log n)$ on CREW by clever using of ranking.
Sorting

- Merge sort on lists of length 1, 2, 4, 8, etc. can be done in time $O(\log^2 n)$ using the partitioning algorithm mentioned above. It can be improved to $O(\log n \log \log n)$ on CREW.
Simulating CRCW on EREW PRAM

- The CRCW PRAM can be simulated on the EREW PRAM in time that is larger by a factor of the time to sort a list of \( p \) items, as we show.
- Consider a reading cycle. Processor \( p_j \) puts into location \( M_j \) value \( (a_{j,j}) \) indicating it wishes to read from location \( a_j \). Sort pairs. \( p_i \) reads location \( M_i \) on first step and loc. \( i-1 \) on second. If they hold different addresses, \( p_i \) reads from address given in \( M_i \) and puts it into \( M_i \).
Simulating CRCW on EREW PRAM

- Using a segmented *copy-right* prefix computation (see book), copy value at location \(a\) to all locations \(M_k\) that request it. This step takes \(O(\log p)\) time for \(p\)-processor PRAM. Sorting takes more time.

- To write, repeat except let \(p_j\) store value \(v\) to be written in \(M_j\) and let first (arbitrary) value be written.