Advanced Techniques for Combinatorial Algorithms: Parallel Algorithms

Gianluca Della Vedova

Univ. Milano – Bicocca
https://gianluca.dellavedova.org

April 30, 2020

RAM model

- Random Access Memory

RAM model

- Random Access Memory
- One processor

RAM model

- Random Access Memory
- One processor
- sequential algorithms

RAM model

- Random Access Memory
- One processor
- sequential algorithms
- Flat memory

RAM model

- Random Access Memory
- One processor
- sequential algorithms
- Flat memory
- Infinite memory

PRAM model

- Parallel RAM

PRAM model

- Parallel RAM
- p RAMs

PRAM model

- Parallel RAM
- p RAMs
- Shared memory

PRAM model

- Parallel RAM
- p RAMs
- Shared memory
- Synchronized (running on the same clock)

PRAM model

- Parallel computation is rapidly becoming a dominant theme in all areas of computer science and its applications. It is likely that, within a decade, virtually all developments in computer architecture, systems programming, computer applications and the design of algorithms will be taking place within the context of parallel computation.

PRAM model

- Parallel computation is rapidly becoming a dominant theme in all areas of computer science and its applications. It is likely that, within a decade, virtually all developments in computer architecture, systems programming, computer applications and the design of algorithms will be taking place within the context of parallel computation.
- Karp, R M. and Ramachandran, V. Chapter 17. Parallel Algorithms for Shared-Memory Machines. Handbook of Theoretical Computer Science: Algorithms and complexity, Volume 1. 1990.

PRAM model

- Selim Akl, Parallel Computation: Models and Methods, Prentice Hall, 1997.
- Selim Akl, Design & Analysis of Parallel Algorithms, Prentice Hall, 1989.
- Cormen, Leiserson, and Rivest, Introduction to Algorithms, 1st edition, 1990, McGraw Hill and MIT Press, Chapter 30 on parallel algorithms.
- Joseph JaJa, An Introduction to Parallel Algorithms, Addison Wesley, 1992.

PRAM model

- It's a MODEL!

PRAM model

- It's a MODEL!
- MIMD (Multiple Instruction Multiple Data)

PRAM model

- It's a MODEL!
- MIMD (Multiple Instruction Multiple Data)
- Processor ID

PRAM model

- It's a MODEL!
- MIMD (Multiple Instruction Multiple Data)
- Processor ID
- No communication cost

PRAM model

- It's a MODEL!
- MIMD (Multiple Instruction Multiple Data)
- Processor ID
- No communication cost
- Shared memory, same access time

PRAM model

	Read	Write
Exclusive	ER	EW
Concurrent	CR	CW

- Different accesses

PRAM model

	Read	Write
Exclusive	ER	EW
Concurrent	CR	CW

- Different accesses
- CRCW is better than EREW.

PRAM model

	Read	Write
Exclusive	ER	EW
Concurrent	CR	CW

- Different accesses
- CRCW is better than EREW.
- But how much?

PRAM model

	Read	Write
Exclusive	ER	EW
Concurrent	CR	CW

- Different accesses
- CRCW is better than EREW.
- But how much?
- There are different CRCW models:

PRAM model

	Read	Write
Exclusive	ER	EW
Concurrent	CR	CW

- Different accesses
- CRCW is better than EREW.
- But how much?
- There are different CRCW models:
    - Common CRCW: concurrent writes if same value from all processors

PRAM model

	Read	Write
Exclusive	ER	EW
Concurrent	CR	CW

- Different accesses
- CRCW is better than EREW.
- But how much?
- There are different CRCW models:
  - Common CRCW: concurrent writes if same value from all processors
  - Priority CRCW: highest priority processor wins

Efficient Algorithm

- $t(n) =$ polylogarithmic time

Efficient Algorithm

- $t(n) =$ polylogarithmic time
- $p(n) =$ polynomial number of processors

Efficient Algorithm

- $t(n) =$ polylogarithmic time
- $p(n) =$ polynomial number of processors
- NC

Efficient Algorithm

- $t(n) =$ polylogarithmic time
- $p(n) =$ polynomial number of processors
- NC
- NC $\subseteq$ P

Efficient Algorithm

- $t(n)$ = polylogarithmic time
- $p(n)$ = polynomial number of processors
- NC
- NC $\subseteq$ P
- Hardness = P-complete problems

Simulations

- EREW PRAM can simulate CRCW PRAM

Simulations

- EREW PRAM can simulate CRCW PRAM
- Time multiplied by $O(\log p(n))$

Optimal Algorithm

- work $w(n) \leq t(n)p(n)$

Optimal Algorithm

- work $w(n) \leq t(n)p(n)$
- $t(n) =$ polylogarithmic time

Optimal Algorithm

- work $w(n) \leq t(n)p(n)$
- $t(n) =$ polylogarithmic time
- $w(n) = O(T(n))$, where $T(n) =$ time complexity of best known sequential algorithm

Algorithms

Sum of elements of an array

Algorithm 1: Sum

if $n = 1$ **then**
  return $x[1]$
else
  return $\text{Sum}(\{x[2i-1] + x[2i] : 1 \leq i \leq n/2\})$

Algorithm 2: Iterative Sum

for $i \leftarrow 1$ **to** n *in parallel* **do**
  $B[i] \leftarrow x[i]$
for $k \leftarrow 1$ **to** $(\log_2 n) - 1$ **do**
  for $i \leftarrow 1$ **to** 2^{k-1} *in parallel* **do**
    $B[i] \leftarrow B[i] + B[i+1]$
return $\text{Sum}(\{x[2i-1] + x[2i] : 1 \leq i \leq n/2\})$

Prefix sum problem (PRAM)

- Input

Prefix sum problem (PRAM)

- Input
- Sequence $\langle x_1, \ldots, x_n \rangle$ of elements

Prefix sum problem (PRAM)

- Input
- Sequence $\langle x_1, \ldots, x_n \rangle$ of elements
- Associative operation $+$

Prefix sum problem (PRAM)

- Input
- Sequence $\langle x_1, \dots, x_n \rangle$ of elements
- Associative operation $+$
- Output

Prefix sum problem (PRAM)

- Input
- Sequence $\langle x_1, \ldots, x_n \rangle$ of elements
- Associative operation $+$
- Output
- $S = \langle S_1 \ldots, S_n \rangle$, with $S_i = x_1 + \cdots + x_i$

Prefix sum problem (PRAM)

- Input
- Sequence $\langle x_1, \ldots, x_n \rangle$ of elements
- Associative operation $+$
- Output
- $S = \langle S_1 \ldots, S_n \rangle$, with $S_i = x_1 + \cdots + x_i$
- trivial sequential algorithm

Prefix sum

Algorithm 3: PrefixSum

if $n = 1$ **then**
  return (x_1)

for $i \leftarrow 1$ **to** $n/2$ **do**
  $y_i \leftarrow x_{2i-1} + x_{2i}$;

$S^* = \text{PrefixSum}([y_1, \ldots, y_{n/2}])$;
/* $S^*_j = x_1 + \cdots + x_{2j}$ */

for $i \leftarrow 1$ **to** n **do**
  if i *is even* **then**
    $S_i \leftarrow S^*_{i/2}$;
  else
    $S_i \leftarrow S^*_{i/2} + x_i$

return $(S_1, \ldots, S_n)$

Prefix sum

Algorithm 4: PrefixSum

```
if n = 1 then
    return (x_1)
for i ← 1 to n/2 in parallel do
    y_i ← x_{2i−1} + x_{2i};
S* = PrefixSum([y_1, ..., y_{n/2}]);
/* S*_j = x_1 + ··· + x_{2j}                */
for i ← 1 to n in parallel do
    if i is even then
        S_i ← S*_{i/2};
    else
        S_i ← S*_{i/2} + x_i
return (S_1, ..., S_n)
```

Prefix sum

- EREW

Prefix sum

- EREW
- $O(\log n)$ time, $O(n)$ processors

Prefix sum

- EREW
- $O(\log n)$ time, $O(n)$ processors
- $w(n) = O(n)$

Prefix sum

- EREW
- $O(\log n)$ time, $O(n)$ processors
- $w(n) = O(n)$
- $O(n/\log n)$ processors are enough

Prefix sum

- EREW
- $O(\log n)$ time, $O(n)$ processors
- $w(n) = O(n)$
- $O(n/\log n)$ processors are enough

Brent's scheduling principle

Prefix sum

- EREW
- $O(\log n)$ time, $O(n)$ processors
- $w(n) = O(n)$
- $O(n/\log n)$ processors are enough

Brent's scheduling principle

- w work on p processors in time t

Prefix sum

- EREW
- $O(\log n)$ time, $O(n)$ processors
- $w(n) = O(n)$
- $O(n/\log n)$ processors are enough

Brent's scheduling principle

- w work on p processors in time t
- $p_1 < p$ (use fewer processors)

Prefix sum

- EREW
- $O(\log n)$ time, $O(n)$ processors
- $w(n) = O(n)$
- $O(n/\log n)$ processors are enough

Brent's scheduling principle

- w work on p processors in time t
- $p_1 < p$ (use fewer processors)
- time $\lfloor w/p_1 \rfloor + t$, work w (more time, same work)

Find Maximum

Instance

An array A of n integers

Find Maximum

Instance

An array A of n integers

Question

Find the largest element in A.

Find Maximum

Instance

An array A of n integers

Question

Find the largest element in A.

Goal

Fastest algorithm

Find Maximum

Algorithm 5: Find1. Find Maximum in an Array A

for $i \leftarrow 1$ **to** n *in parallel* **do**
    $B[i] \leftarrow$ true;
for $i \leftarrow 1$ **to** n *in parallel* **do**
    for $j \leftarrow 1$ **to** n *in parallel* **do**
        if $A[i] < A[j]$ *or* $A[i] = A[j]$ *and* $i < j$ **then**
            $B[i] \leftarrow$ false;
for $i \leftarrow 1$ **to** n *in parallel* **do**
    if $B[i]$ **then**
        Return $A[i]$

Time? Work?

Find Maximum

Algorithm 6: Find1. Find Maximum in an Array A

```
for i ← 1 to n in parallel do
    B[i] ← true;
for i ← 1 to n in parallel do
    for j ← 1 to n in parallel do
        if A[i] < A[j] or A[i] = A[j] and i < j then
            B[i] ← false;
for i ← 1 to n in parallel do
    if B[i] then
        Return A[i]
```

Time? Work? $T(n) = O(1)$, $W(n) = O(n^2)$

Find Maximum

Algorithm 7: Find2. Find Maximum in an Array A

if $n > 16$ **then**
    for $i \leftarrow 1$ **to** $\sqrt{n}$ *in parallel* **do**
        $B[i] \leftarrow$Find2$(A[1 + \lfloor (i-1)/\sqrt{n} \rfloor : \lfloor i/\sqrt{n} \rfloor])$;
    Find1(B);
else
    Find1(A);

$T(n)$
$W(n)$

Find Maximum

Algorithm 8: Find2. Find Maximum in an Array A

```
if n > 16 then
    for i ← 1 to √n in parallel do
        B[i] ← Find2(A[1 + ⌊(i − 1)/√n⌋ : ⌊i/√n⌋]);
    Find1(B);
else
    Find1(A);
```

$T(n) \leq T(\sqrt{n}) + c_1 \Rightarrow T(n) = O(\log\log n)$
$W(n) \leq \sqrt{n}W(\sqrt{n}) + c_2 n \Rightarrow W(n) = O(n \log\log n)$

Find Maximum

Algorithm 9: Find3. Find Maximum in an Array A

for $i \leftarrow 1$ **to** $n/\log\log n$ *in parallel* **do**
  $B[i] \leftarrow \min(A[1 + \lfloor (i-1)\log\log n \rfloor : \lfloor i/\log\log n \rfloor]);$
Find2(B);

$T(n) =$
$W(n) =$

Find Maximum

Algorithm 10: Find3. Find Maximum in an Array A

for $i \leftarrow 1$ **to** $n/\log\log n$ *in parallel* **do**
  $B[i] \leftarrow \min(A[1 + \lfloor (i-1)\log\log n \rfloor : \lfloor i/\log\log n \rfloor]);$
Find2(B);

$T(n) = O(\log\log n)$
$W(n) = O(n)$

Pointer Jumping

- Problem: given a single-link list L, propagate the value of the last element to the entire list

---

foreach $L[i]$ *in parallel* **do**
    for $k \leftarrow 1$ **to** $\log_2 n$ **do**
        if $next(i) \neq NIL$ **then**
            $next[i] \leftarrow next[next[i]]$
    $value[i] \leftarrow value[next[i]]$

List Ranking

- Problem: given a list L, find the position of each element in L

Algorithm 11: List Ranking via pointer jumping

foreach $L[i]$ *in parallel* **do**
    if $next(i) = NIL$ **then**
        $rank[i] \leftarrow 0$
    else
        $rank[i] \leftarrow 1$
    for $k \leftarrow 1$ **to** $\log_2 n$ **do**
        $rank[i] \leftarrow rank[i] + rank[next[i]]$;
        $next[i] \leftarrow next[next[i]]$

List Ranking

Proof

List Ranking

Proof

- At iteration k:

List Ranking

Proof

- At iteration k:
- if $next[i] \neq NIL$ then $rank[i] = 2^k$

List Ranking

Proof

- At iteration k:
- if $next[i] \neq NIL$ then $rank[i] = 2^k$
- if $next[i] = NIL$ then $rank[i]$ is the distance between $L[i]$ and the end of the list

List Ranking

Proof

- At iteration k:
- if $next[i] \neq NIL$ then $rank[i] = 2^k$
- if $next[i] = NIL$ then $rank[i]$ is the distance between $L[i]$ and the end of the list
- $next[i] = NIL$ for the last 2^k elements of L

Binary trees

- Problem: to determine depth of each node

Binary trees

- Problem: to determine depth of each node
- parent, left child, right child

Binary trees

- Problem: to determine depth of each node
- parent, left child, right child
- 3 processors for each node

Binary trees

- Problem: to determine depth of each node
- parent, left child, right child
- 3 processors for each node
- $O(n)$-time sequential algorithm

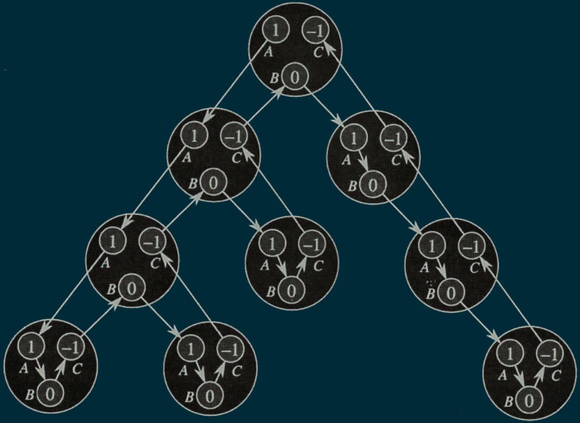
Binary trees

- Problem: to determine depth of each node
- parent, left child, right child
- 3 processors for each node
- $O(n)$-time sequential algorithm
- Algorithm 1: Level-wise visit, each node in parallel

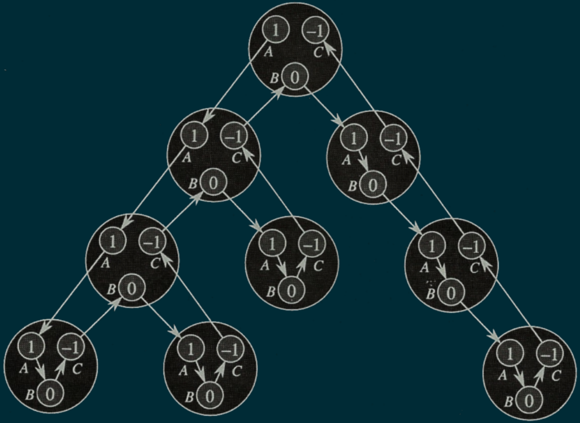
Binary trees

- Problem: to determine depth of each node
- parent, left child, right child
- 3 processors for each node
- $O(n)$-time sequential algorithm
- Algorithm 1: Level-wise visit, each node in parallel
- $t(n) =$height (not good)

Euler tour

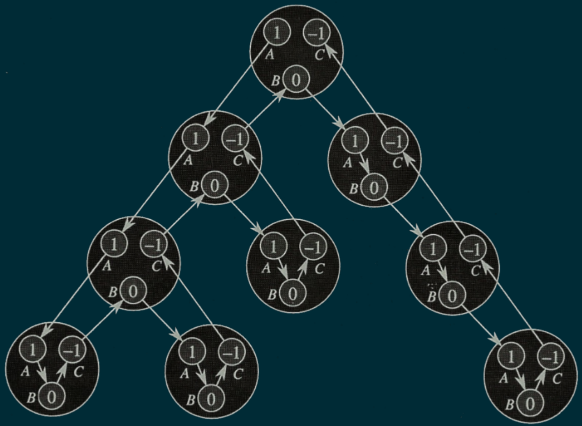


Euler tour

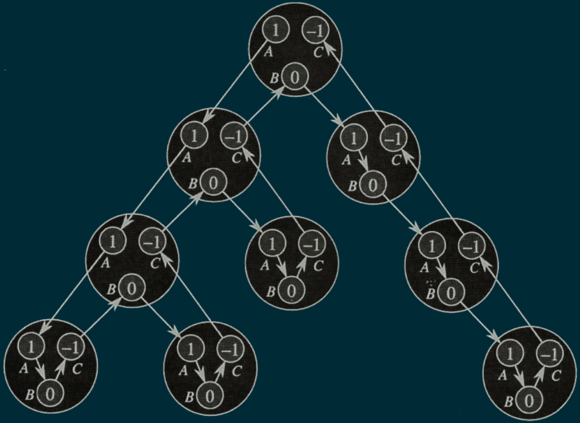


Depth = prefix sum

Size of all subtrees



Size of all subtrees



replace -1 with 0, difference between third and first prefix sums

Matrix multiplication

- $C = AB$, simpler case A, B square matrices

Matrix multiplication

- $C = AB$, simpler case A, B square matrices
- embarassingly parallel

Matrix multiplication

- $C = AB$, simpler case A, B square matrices
- embarassingly parallel
- $C[i,j] = \sum_{k \leq n} A[i,k]B[k,j]$

Matrix multiplication

- $C = AB$, simpler case A, B square matrices
- embarassingly parallel
- $C[i,j] = \sum_{k \leq n} A[i,k]B[k,j]$
- $O(\log n)$ time, $O(n^3 / \log n)$ processors

Graph Algorithms

- depth-first visit

Graph Algorithms

- depth-first visit
- No NC algorithm

Graph Algorithms

- depth-first visit
- No NC algorithm
- breadth-first visit

Graph Algorithms

- depth-first visit
- No NC algorithm
- breadth-first visit
- $O(n^{2.37})$ processors

Graph Algorithms

- depth-first visit
- No NC algorithm
- breadth-first visit
- $O(n^{2.37})$ processors
- Euler tour

Connected components

Instance

Undirected graph $G = (V, E)$

Connected components

Instance

Undirected graph $G = (V, E)$

Data structure

Connected components

Instance

Undirected graph $G = (V, E)$

Data structure

1. $M(v, w)$ adjacency matrix

Connected components

Instance

Undirected graph $G = (V, E)$

Data structure

1. $M(v, w)$ adjacency matrix
2. $R(v) \leftarrow v$ representative. All vertices in the same connected components have the same representative.

Connected components

Instance

Undirected graph $G = (V, E)$

Data structure

1. $M(v, w)$ adjacency matrix
2. $R(v) \leftarrow v$ representative. All vertices in the same connected components have the same representative.
3. $C[v, w]$ connected components with representative v and w can be merged

Connected components

Algorithm 12: ConnectedComponents

for $\log_2 n$ *times* **do** *hookings*
    foreach *edge* (v, w) *such that* $R[v] \neq R[w]$ **do**
        if $R[v] < R[w]$ **then**
            $C[R[v], R[w]] \leftarrow \text{true};$
    foreach *vertex* v *such that* $R[v] = v$ **do**
        $R[v] \leftarrow \max w : C[R[v], R[w]] \text{ is true};$
    for $i \leftarrow 1$ *to* $\log_2 n$ **do** *parallel pointer jumping*
        foreach *vertex* v **do**
            $R[v] \leftarrow R[R[v]];$

Minimum Spanning Tree

Problem

Given an undirected edge-weighted connected graph $G = (V, E)$, find a minimum-weight subset $T \subseteq E$ such that T is a tree spanning V.

Minimum Spanning Tree

Problem

Given an undirected edge-weighted connected graph $G = (V, E)$, find a minimum-weight subset $T \subseteq E$ such that T is a tree spanning V.

Lemma

Let $G = (V, E)$ be an undirected graph, let (V_1, V_2) be a bipartition of V, let T be a minimum spanning tree of G, and let e be the lightest edge connecting V_1 and V_2. Then $e \in T$.

Additional Bibliography on PRAM

- A Survey of Parallel Algorithms for Shared-Memory Machines
  http://techreports.lib.berkeley.edu/accessPages/CSD-88-408.html
- Vishkin, Uzi (2009), Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques. http://www.umiacs.umd.edu/users/vishkin/PUBLICATIONS/classnotes.pdf