

PAL: Parallel Algorithms

Important Notes About These Notes

These notes were written by me, Mark Ormesher, during my revision for Computer Science exams at King's College London. I am publishing them here to help my classmates and other students in the years below me. To the best of my knowledge they are correct.

- These notes are **not endorsed** by King's College London, the module lecturers, or any member of College staff.
- These notes are **not checked** by any qualified individual, and I offer no guarantee as to how accurate or complete these notes are. They are offered for free, as-is, and I accept no responsibility for how you choose to use them.
- These notes were relevant for my year (2016/17) but **the content for your course may be different**. Check your lecture slides, syllabi, tutorial guides, etc.
- These notes were produced for my own personal use (it's how I like to study and revise). That means that some annotations may be irrelevant to you, and some topics might be skipped entirely.
- Feel free to **share** these notes, however please only share a link to the repo (see the link below), not individual files.

Notes are originally from **<https://github.com/markormesher/CS-Notes>** and remain the copyright-protected work of Mark Ormesher.

List of PAL Algorithms

Multi-threading

- Loops
- Tree traversal
- Exponentiation
- Horner's method
- Merge sort (and fast merge)

PRAM EREW

- Fan-in array binary operation (sum, product, min, etc.)
- Broadcast
- Array membership (partial)
- Array membership (full)
- Array prefix sums

PRAM CR*W

- List ranking (CW or EW)
- Root finding (CW or EW)
- Matrix * vector multiplication (CW or EW)
- Matrix * matrix multiplication (CW or EW)
- Array sum (CW-priority)
- Array min/max (CW-any)
- Array membership (find index with CW-priority, find true/false with CW-arbitrary)
- Array sorting (CW must "stack")

Graph Interconnected Model

- 1D mesh even/odd transposition sort
- 2D mesh snake sort
- 2D mesh broadcast
- 2D mesh array min/max
- 2D mesh membership
- 2D mesh array prefix sums

Distributed Networks

- Directed ring leader election
- General leader election
- General breadth-first search tree finding

Graphs

- Sequential graph components
- Parallel graph components
- Parallel minimum spanning tree (Boruvka's method)

Multi-Threading Loops

Input: indexes i and j , $i \leq j$
some code to execute, exec .

Output: run exec for all indexes from i to j , inclusive

func $\text{paraLoop}(i, j, \text{exec})$:

```
if ( $i == j$ ):  
    exec( $i$ )  
} Base case: ran exec for the  
single index  $i$  ( $j$  would also  
work).  
  
else:  
    k =  $i + \text{floor}((j-i)/2)$  } Find the mid-point between  $i$   
                                and  $j$ .  
    spawn  $\text{paraLoop}(i, k, \text{exec})$  } Do half of the range  
                                on a new thread...  
     $\text{paraLoop}(k+1, j, \text{exec})$  } ... and half on this thread.  
    sync } Make sure everything finishes.
```

The runtime is $\Theta(\log_2 n)$ if n is the range from i to j .

Note: the overall runtime is $O(m \log_2 n)$ where m is the runtime of exec .

Multi-Threading Tree Traversal

Input: binary tree root node, t .

Output: pre-, in- or post-order traversal of the tree rooted at t .

func paraTraversal (t):

if ($t == \text{null}$): } Base case.
return []

$m = t.\text{data}$ } Visit this node.

$l = \text{spawn paraTraversal} (t.\text{left})$ } Traverse the left sub-tree on a new thread.

$r = \text{paraTraversal} (t.\text{right})$ } Traverse the right sub-tree on this thread.

sync } Wait for both traversals to finish.

For pre-order:
return $[m, l, r]$ } Implicit array flattening is assumed.

For in-order:
return $[l, m, r]$

For post-order:
return $[l, r, m]$

The span is $\Theta(h)$ where h is the height of the tree. For a balanced tree with n nodes, $h = \log_2 n$.

Multi-Threading Exponentiation.

Input: integers n and k , $k \geq 1$

Output: n^k

func paraExp(n, k):

if ($k == 1$):
 return n

} Base case.

$j = \text{floor}(k/2)^*$

} We will compute two halves of
the problem separately.

$a = \text{spawn paraExp}(n, j)$

} The first half runs on a
new thread...

$b = \text{paraExp}(n, k-j)$

} ... and the second half runs on
this thread. Note " $k-j$ " to avoid
rounding errors from the point
marked *.

sync

} Wait for both halves to finish.

return $a * b$

} Re-combine and return the result.

The span is $\Theta(\log_2 k)$, assuming multiplication as an $O(1)$ operation.

Multi-Threading Horner's Method (Polynomial Evaluation)

Input: polynomial in array form, a Eg. $3x^2 + 4x + 9 = [9, 4, 3]$
value for x , x
indexes s and e

Output: value of the polynomial between the powers s and e inclusive.

Note: initially, $s = 0$ and $e = \text{highest power in polynomial}$

func paraHorner (a, x, s, e):

if ($s == e$): } Base case.
 return $a[s]$

$m = s + \text{floor}((e-s)/2)$ } Find the midpoint of s and e .

$a = \text{spawn paraHorner}(a, x, s, m)$ } Compute the range s to m
on a new thread.

$b = \text{paraHorner}(a, x, m+1, e)$ } Compute the range $m+1$ to e
on this thread

sync } Wait for both halves to finish.

$\exp = m - s + 1$ } b will be too small by a factor
of x^{\exp} ...

return $a + (b * x^{\exp})$ } ... so we shift it up
accordingly.

The recursion depth and span are $O(\log_2 n)$ for a polynomial with n terms. The overall runtime is $O(\log_2^2 n)$ if exponentiation (marked *) can be done in logarithmic time with the parallel method.

Multi-Threading Array Merge.

Input: Two sorted arrays, a and b .

Output: The sorted merge of a and b .

func paraMerge(a, b):

$$m = \text{floor}((a.\text{length} + 1) / 2) \quad \} \text{Mid-position in } a.$$

$$x = a[m] \quad \} x \text{ is the pivot value.}$$

$$\begin{aligned} a_{\text{Low}} &= a[1 \dots m-1] \\ a_{\text{High}} &= a[m+1 \dots \text{end}] \end{aligned} \quad \} \text{Split } a \text{ into values above } m \text{ and values below } m.$$

- All values in a_{Low} are $\leq x$
- All values in a_{High} are $\geq x$

$$q = \text{findPositionWithBinarySearch}(b, x) \quad \} \text{Find the position of } x \text{ in } b, \text{ or its closest value above. below. above.}$$

$$\begin{aligned} b_{\text{Low}} &= b[1 \dots q-1] \\ b_{\text{High}} &= b[q \dots \text{end}] \end{aligned} \quad \} \begin{array}{l} \text{- All values in } b_{\text{Low}} \text{ are } \leq x \\ \text{- All values in } b_{\text{High}} \text{ are } \geq x \end{array}$$

$l = \text{spawn ParaMerge}(a_{\text{Low}}, b_{\text{Low}}) \quad \} \text{Merge the Low end in one new thread.}$

$r = \text{paraMerge}(a_{\text{High}}, b_{\text{High}}) \quad \} \text{Merge the high end in this thread.}$

sync $\quad \} \text{Wait for both merges to finish.}$

return [l, x, r] $\quad \} \text{Return the recombined merged set.}$
Note: implicit array flattening.

The recursion depth and span are $\Theta(\log_2 n)$, each level of which uses an $\Theta(\log_2 n)$ binary search.

The overall runtime is $\Theta(\log^2 n)$.

Multi-Threading Array Merge Sort

Input: array, a
start and end indexes, s and e

Output: a, sorted between s and e inclusive.

Note: initially, s=1 and e=array length

func paraMergeSort (a, s, e):

if (s == e): } Base case.
return a [s]

m = s + floor ((e-s)/2) } Mid-point of s and e.

l = spawn paraMergeSort (a, s, m) } Sort the left side on a
new thread.

r = paraMergeSort (a, m+1, e) } Sort the right side on this
thread.

Sync } Wait for both sorts to finish.

return paraMerge (l, r) } Recombine the sorted halves
with the fast parallel merge.

The recursion depth and span are $\Theta(\log_2 n)$, each of which
uses an $\Theta(\log_2^2 n)$ merge.

The overall runtime is $\Theta(\log_2^3 n)$.

PRAM-EREW Binary Fan-in

Input: n -item array, a

Output: cumulative value in $a[1]$ of operation \oplus

Note: $n = 2^k$

works for any binary associative operation ($+$, $*$, min, max, etc.).
To avoid destroying the input, copy to a duplicate array
first in one parallel step.

for h from 1 to k , do: \leftarrow Loop 1

for i from 1 to $n/2^h$, in parallel do: \leftarrow Loop 2

$$a[i] = a[2i] \oplus a[2i-1]$$

Example: $a = [4, 2, 1, 0, 9, 5, 4, 1]$ operation = addition

• Loop 1, $h=1$ Loop 2, $i = 1$ to 4

$$a = [4, 2, 1, 0, 9, 5, 4, 1]$$

$i=1$ $i=2$ $i=3$ $i=4$

$$a = [6, 1, 14, 5, 9, 5, 4, 1]$$

• Loop 1, $h=2$ Loop 2, $i = 1$ to 2

$$a = [6, 1, 14, 5, 9, 5, 4, 1]$$

$i=1$ $i=2$

$$a = [7, 19, 14, 5, 9, 5, 4, 1]$$

• Loop 1, $h=3$ Loop 2, $i = 1$ to 1

$$a = [7, 19, 14, 5, 9, 5, 4, 1]$$

$i=1$

$$a = [26, 19, 14, 5, 9, 5, 4, 1]$$

The runtime is $\Theta(\log_2 n)$, assuming the operation \oplus is an $O(1)$ action.

For loop 1, in the first iteration loop 2 will need $n/2$ processors.
In the second iteration loop 2 will need $n/4$ processors, then $n/8$,
 $n/16$, etc.

Loop 1 →

Loop 2 →

$$[1, 2, 3] \oplus [1, 2, 3] = [1, 10]$$

$$[1, 2, 3, 0, 0, 1, 2, 4] = 0 \quad : \text{aligned}$$

$$1+0=1, \text{Loop 1} = 1 = \text{Loop 2}$$

$$[1, 2, 3, 0, 0, 1, 2, 4] = 0$$

$$[1, 2, 3, 0, 0, 1, 2, 4] = 0$$

$$1+1=2, \text{Loop 1} = 2 = \text{Loop 2}$$

$$[1, 2, 3, 0, 2, 4, 1, 8] = 0$$

$$[1, 2, 3, 0, 2, 4, 1, 8] = 0$$

$$1+1=2, \text{Loop 1} = 2 = \text{Loop 2}$$

$$[1, 2, 3, 0, 2, 4, 1, 8] = 0$$

$$[1, 4, 2, 3, 0, 2, 4, 1, 8] = 0$$

PRAM - EREW Array Broadcast

Input: empty n -item array, a value x to broadcast, x

Output: $a[i] = x$ for all $1 \leq i \leq n$.

Note: $n = 2^k$

$a[1] = x$ } Copy to the first cell manually.

for h from 0 to $k-1$, do:

for i from 2^h+1 to 2^{h+1} , in parallel do:

} We will work on the "top half" of increasing powers of two. First just 2, then 3 and 4, then 5 to 8, etc.

$a[i] = a[i-2^h]$

} Copy the value from the corresponding position in the "bottom half" of the current power of 2.

Example:

$$x = 5$$

$$a = [-, -, -, -, -, -, -, -]$$

Init:

$$a = [5, -, -, -, -, -, -, -] *$$

$$\begin{matrix} h=0 \\ i=2 \text{ to } 2 \end{matrix}$$

$$a = [5, 5, -, -, -, -, -, -]$$

$$\begin{matrix} h=1 \\ i=3 \text{ to } 4 \end{matrix}$$

$$a = [5, 5, 5, 5, -, -, -, -]$$

$$\begin{matrix} h=2 \\ i=5 \text{ to } 8 \end{matrix}$$

$$a = [5, 5, 5, 5, 5, 5, 5, 5]$$

The runtime is $\Theta(\log_2 n)$. In each iteration of the outer loop, the inner loop requires 1, 2, 4, 8... $n/2$ processors.

PRAM EREW Array Membership (Partial)

Input: n-item array, a

n-item array, b , containing the value to check, x

Output: $b[i] = c$ iff $a[i] = x$

$b[i] = \text{inf}$ iff $a[i] \neq x$

for i from 1 to n , in parallel do: } One processor per index

if ($a[i] == b[i]$): } Store i for matching positions.
 $b[i] = c$

else: } Store inf for non-matching positions.
 $b[i] = \text{inf}$

Example: In: $a = [2, 5, 8, 4, 6, 5, 1]$

$b = [5, 5, 5, 5, 5, 5, 5]$

Out: $b = [\infty, 2, \infty, \infty, \infty, 6, \infty]$

Execution takes one parallel step (i.e. $\Theta(1)$) and requires n processors.

PRAM CREW Array Membership (Full)

Input: n -item array, a
item to search for, x

Output: $b[1] = \text{lowest index of } x \text{ in } a, \text{ or infinity.}$

Note: $n = 2^k$

- ① {
 $b = \text{empty } n\text{-item array}$ } This will store n copies of x , for the membership check.
 $\text{Array Broadcast}(b, x)$ } Create n copies of x .
 - ② {
 $\text{Array Membership Partial}(a, b)$ } Now b will contain the matching indexes in a .
 - ③ {
 $\text{Array Binary Fan In}(b, \min)$ } Find the smallest value in b , which is written to $b[1]$.
-

Example: Init: $a = [0, 1, 4, 1]$
 $x = 1$

① $b = [1, 1, 1, 1]$

② $b = [\infty, 2, \infty, 4]$

③ $b = [\infty, 2, \infty, 4]$

$b = [2, 4, \infty, 4]$

$b = [2, 4, \infty, 4]$

Answer

PRAM CREW Array Prefix Sums

Input: n values in $a[n \dots 2n-1]$ of array a

Eg. 4 values in $a = [-, -, -, 1, 2, 3, 4]$

Output: prefix sums in $b[n \dots 2n]$

Eg. $b = [-, -, -, 1, 3, 6, 10]$

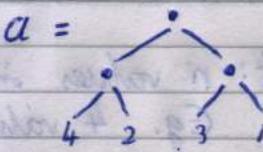
Note: $n = 2^k$

- ① { for h from $k-1$ to 0, do:
- for i from 2^h to $2^{h+1}-1$, in parallel do:
- $a[i] = a[2i] + a[2i+1]$
- } Move up through each level of the tree, starting above the leaves.
- } Make each value the sum of its children. Root will be total sum.
- ② { $b = \text{empty array, same size as } a$
- $b[1] = a[1]$
- } Manually copy the root.
- ③ { for h from 1 to k , do:
- for i from 2^h to $2^{h+1}-1$, in parallel do:
- if (i is even):
 $b[i] = b[i/2] - a[i+1]$
- } Move back down the layers of the tree, starting at the second and ending at the leaves.
- } Left-child value = parent - right sibling.
- else:
 $b[i] = b[(i-1)/2]$
- } Right-child value = parent

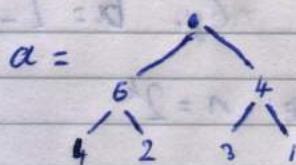
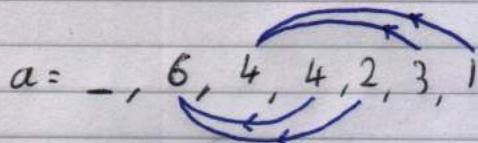
The runtime is $\Theta(\log_2 n)$.

Example

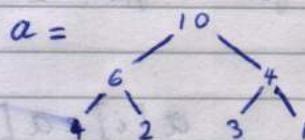
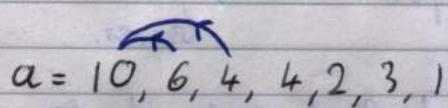
$$a = \underline{\quad}, \underline{\quad}, 4, 2, 3, 1$$



① $h=1, i=2 \text{ to } 3$

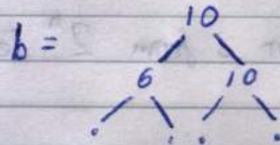
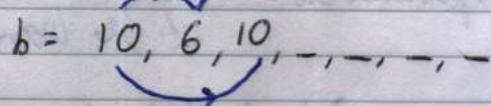


② $h=0, i=1 \text{ to } 1$



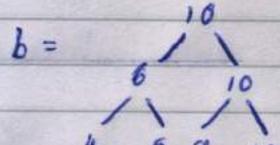
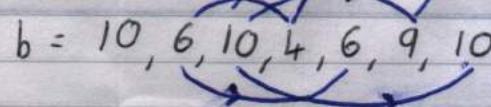
③ $b = 10, \underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}, \underline{\quad}$

$$a = 10, 6, 4, 4, 2, 3, 1$$



④ $h=2, i=4 \text{ to } 7$

$$a = 10, 6, 4, 4, 2, 3, 1$$



Result: $b = 10, 6, 10, \underline{4, 6, 9, 10}$

PRAM CRW List Ranking

($\log \max$)

Input: parent array of n -item linked list, p

Output: distance array, d , such that $d[i]$ gives i 's distance from the root

Note: $n = 2^k$

① { for i from 1 to n , do: } Start the root at distance = 0,
 if ($p[i] = i$): all others at distance = 1.
 else:
 $d[i] = 1$

for h from 1 to k , do: } $k = \log_2(n)$, which is the maximum
 number of steps needed to point every
 node to the root by parent pointer-
 jumping.

for i from 1 to n , in parallel do: } At every node...

② if ($p[i] := p[p[i]]$): } If it hasn't been pointer-jumped to
 the root yet...

③ $d[i] = d[i] + d[p[i]]$ } Add the distance of the
④ $p[i] = p[p[i]]$ parent, then "jump past" the
 parent.

Runtime is $\Theta(\log n)$, requiring $O(n)$ processors at each iteration.

Example:

$$p = 5 \ 8 \ 1 \ 6 \ 2 \ 3 \ 7 \ 7$$

$$d = 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ \} \textcircled{1}$$

$$h=1, i = 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8$$

$$pp = 2 \ 7 \ 5 \ 3 \ 8 \ 1 \ 7 \ 7 \ \} \textcircled{2}$$

$$d = 2 \ 2 \ 2 \ 2 \ 2 \ 2 \ 0 \ 1 \ \} \textcircled{3}$$

$$p = 2 \ 7 \ 5 \ 3 \ 8 \ 1 \ 7 \ 7 \ \} \textcircled{4}$$

$$h=2, i = 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8$$

$$pp = 7 \ 7 \ 8 \ 5 \ 7 \ 2 \ 7 \ 7 \ \textcircled{2}$$

$$d = 4 \ 2 \ 4 \ 4 \ 3 \ 4 \ 0 \ 1 \ \textcircled{3}$$

$$p = 7 \ 7 \ 8 \ 5 \ 7 \ 2 \ 7 \ 7 \ \textcircled{4}$$

$$h=3, i = 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8$$

$$pp = 7 \ 7 \ 7 \ 7 \ 7 \ 7 \ 7 \ 7 \ \textcircled{2}$$

$$d = 4 \ 2 \ 5 \ 7 \ 3 \ 6 \ 0 \ 1 \ \textcircled{3}$$

↑
Answer.

PRAM CRW Root Finding

symmetric

Input: forest of 1 or more trees stored in the parent array p

Output: root $_i$, r_i for all nodes.

Note: $n = 2^k$

for i from 1 to n , in parallel do:
 for h from 1 to k , do:
 $p[i] = p[p[c]]$

} Use pointer jumping to flatten all trees.

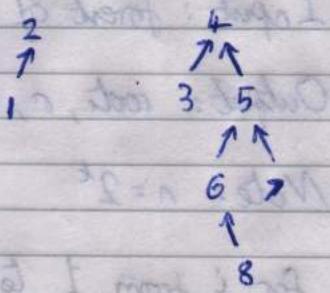
$r[c] = p[i]$ } Store the roots for all nodes.

Runtime is $\Theta(\log_2 n)$, requiring $O(n)$ processors at each iteration.

Example

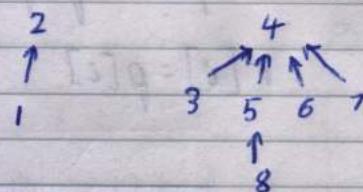
Init:

$$p = 2 \ 2 \ 4 \ 4 \ 4 \ 5 \ 5 \ 6$$



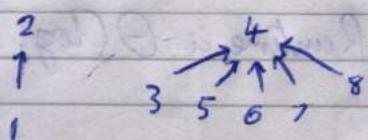
Loop 1:

$$p = 2 \ 2 \ 4 \ 4 \ 4 \ 4 \ 4 \ 5$$



Loop 2:

$$p = 2 \ 2 \ 4 \ 4 \ 4 \ 4 \ 4 \ 4$$



Loop 3:

No change

Output:

$$r = 2 \ 2 \ 4 \ 4 \ 4 \ 4 \ 4 \ 4$$

PRAM CR*W Matrix * Vector Multiplication

Input: $n \times n$ -matrix, A
 n -vector, B

$$\begin{bmatrix} 2 & 1 & 2 & 1 \\ 1 & 1 & 0 & 1 \\ 3 & 2 & 1 & 2 \\ 2 & 2 & 0 & 1 \end{bmatrix} = B \quad \begin{bmatrix} 1 & 4 & 2 & 1 \\ 1 & 0 & 4 & 3 \\ 2 & 1 & 2 & 2 \\ 2 & 0 & 1 & 2 \end{bmatrix} = A$$

Output: n -vector, $Q = A \times B$

Note: $n = 2^k$

① { for r, c from 1 to n , in parallel do: } For each vector value, multiply it by each value in the corresponding column of the matrix and store the result in the temporary matrix T .

for r from 1 to n , in parallel do: } For each row in parallel:

② { for h from 1 to k , do: } Add up the row using binary fan-in.
 for c from 1 to $n/2^k$, in parallel do:
 $T[r, c] = T[r, 2c] + T[r, 2c-1]$

③ { $Q = \text{empty } n\text{-vector}$ } Copy the sums to the output vector.
 for r from 1 to n , in parallel do:
 $Q[r] = T[r, 1]$

The runtime is $\Theta(\log n)$ and the algorithm requires $\Theta(n^2)$ processors.

Example

$$A = \begin{bmatrix} 1 & 2 & 4 & 1 \\ 5 & 4 & 0 & 1 \\ 8 & 2 & 1 & 2 \\ 2 & 1 & 9 & 3 \end{bmatrix} \quad B = \begin{bmatrix} 2 \\ 1 \\ 3 \\ 2 \end{bmatrix}$$

$$\textcircled{1} \quad T = \begin{bmatrix} 1 \times 2 & 2 \times 1 & 4 \times 3 & 1 \times 2 \\ 5 \times 2 & 4 \times 1 & 0 \times 3 & 1 \times 2 \\ 8 \times 2 & 2 \times 1 & 1 \times 3 & 2 \times 2 \\ 2 \times 2 & 1 \times 1 & 9 \times 3 & 3 \times 2 \end{bmatrix} = \begin{bmatrix} 2 & 2 & 12 & 2 \\ 10 & 4 & 0 & 2 \\ 16 & 2 & 3 & 4 \\ 4 & 1 & 27 & 6 \end{bmatrix}$$

$$\textcircled{2} \quad h=1 \quad T = \begin{bmatrix} 4 & 14 & 12 & 2 \\ 14 & 2 & 0 & 2 \\ 18 & 7 & 3 & 4 \\ 5 & 33 & 27 & 6 \end{bmatrix}$$

$$\textcircled{2} \quad h=2 \quad T = \begin{bmatrix} 18 & 14 & 12 & 2 \\ 16 & 2 & 0 & 2 \\ 25 & 7 & 3 & 4 \\ 38 & 33 & 27 & 6 \end{bmatrix}$$

$$\textcircled{3} \quad Q = \begin{bmatrix} 18 \\ 16 \\ 25 \\ 38 \end{bmatrix}$$

PRAM CR*W Matrix*Matrix Multiplication

Input: $n \times n$ -matrices, A and B

$$\begin{bmatrix} 3 & 2 \\ 8 & 5 \end{bmatrix} = B \quad \begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix} = A$$

Output: $n \times n$ -matrix, $Q = A \times B$

Note: $n = 2^k$

① { for r, c, p from 1 to n , in parallel do: } Each cell in the $n \times n$ output matrix is the sum of n multiplication. This step does n^3 multiplications in parallel, storing results in the temporary 3D array T . Think of T as a 2D array of lists, where the list at $T[r, c]$ is the list of values that will be summed for position (r, c) in the output matrix. $T[r, c, p]$ is position p in the "list" at $T[r, c]$.

$$T[r, c, p] = A[r, p] \times B[p, c]$$

for r, c from 1 to n , in parallel do: } For each output matrix position...

for h from 1 to k , do

for p from 1 to $\frac{n}{2^h}$, in parallel do:

$$T[r, c, p] = T[r, c, 2p] + T[r, c, 2p-1]$$

} Add up the "list" for each cell using binary sum in.

$Q = \text{empty } n \times n \text{ matrix}$

for r, c from 1 to n , in parallel do:

$$Q[r, c] = T[r, c, 1]$$

} Copy the sums to the output matrix.

The runtime is $\Theta(n \log_2 n)$ with n^3 processors.

Example:

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad B = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

① $T = \begin{bmatrix} [1 \times 5, 2 \times 7] & [1 \times 6, 2 \times 8] \\ [3 \times 5, 4 \times 7] & [3 \times 6, 4 \times 8] \end{bmatrix} = \begin{bmatrix} [5, 14] & [6, 16] \\ [15, 28] & [18, 32] \end{bmatrix}$

② $T = \begin{bmatrix} [19, 14] & [22, 16] \\ [43, 28] & [50, 32] \end{bmatrix}$

③ $Q = \begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$

PRAM CRCW-Priority Array Sum

Input: n -item array, a

Output: sum in $a[1]$

$s = 0$

} Initial sum

for i from 1 to n , in parallel do:

$s = s + a[i]$

} Compute the sum in
one parallel step

$a[1] = s$

} Write output.

The runtime is $\Theta(1)$ with $\Theta(n)$ processors.

PRAM CRCW Array Minimum/Maximum

Input: n -item array, a

Output: min/max of a in $a[1:n]$

Note: $n = 2^k$

$m = \text{empty } n\text{-item array}$
for i from 1 to n , in parallel do:
 $m[i] = 0$

} m will store 1 for all
"losers" and a single zero
for the "winner".

* > for max
< for min
for i, j from 1 to n , in parallel do:
if ($a[i] > a[j]$):
 $m[j] = 1$

} Compare all pairs, writing 1s
for the "losers".

for i, j from 1 to n , in parallel do:
if ($m[i] = 0$ and $i < j$):
 $m[j] = 1$

} Compare all pairs again,
writing 1s for all positions
higher than each "winner". This
keeps only the lowest-index
"winner".

for i from 1 to n , in parallel do:
if ($m[i] = 0$):
Result = $a[i]$
IndexOfResult = i

} keep the remaining "winner".

The runtime is $\Theta(1)$, requiring $\Theta(n)$ processors.

The CW model does not matter, because only "1" is ever
written. Priority, arbitrary and common would all work.

PRAM CRCW Array Membership

Input: n -item array, a
item to check, x

Output: (Cw-priority) lowest index containing x , or inf
(Cw-arbitrary) some index containing x , or inf

Index = inf . } Initial state: not found.

for i from 1 to n , in parallel do: } Check every index in parallel.

if ($a[i] = x$): } Store a matching index
Index = i

Contains = (Index != inf) } This gives a boolean value that
works for Cw-priority and
Cw-arbitrary.

The routine is $\Theta(1)$, requiring $\Theta(n)$ processes to do $\Theta(n)$ parallel steps.

This algorithm does not work with Cw-common.

PRAM CRCW Array Sorting

Input: n -item array, a

Output: n -item array, b , containing the sorted contents of a

$w = \text{empty } n\text{-item array}$

for i from 1 to n , in parallel do:
 $w[i] = 1$

for i, j from 1 to n , in parallel do:

if ($i < j$):

if ($a[i] <= a[j]$):

$w[j] = w[j] + 1$

else:

$w[i] = w[i] + 1$

for i from 1 to n , in parallel do:

$b[w[i]] = a[i]$

} w will store, for each position i , the number of items smaller than or equal to $a[i]$. Everything starts with one equal: itself.

For every position, compare all higher positions. If $a[i]$ is less than or equal to $a[j]$, increase j 's score in w . Otherwise, increase i 's score.

The algorithm requires $O(n^2)$ processors and runs in $\Theta(1)$.

Example

$$a = 2 \ 1 \ 4 \ 6 \ 2 \ 8 \quad \left. \begin{array}{l} \\ \end{array} \right\} \text{Initial setup.}$$

$$w = 1 \ 1 \ 1 \ 1 \ 1 \ 1 \quad \left. \begin{array}{l} \\ \end{array} \right\}$$

$$w = 2 \ 1 \ 2 \ 2 \ 2 \ 2 \quad \left. \begin{array}{l} \} i=1, j=2 \text{ to } 6 \\ w[1]=2 \text{ is now fixed} \end{array} \right\}$$

$$w = 2 \ 1 \ 3 \ 3 \ 3 \ 3 \quad \left. \begin{array}{l} \} i=2, j=3 \text{ to } 6 \\ w[2]=1 \text{ is now fixed} \end{array} \right\}$$

$$w = 2 \ 1 \ 4 \ 4 \ 3 \ 4 \quad \left. \begin{array}{l} \} i=3, j=4 \text{ to } 6 \\ w[3]=4 \text{ is now fixed} \end{array} \right\}$$

$$w = 2 \ 1 \ 4 \ 5 \ 3 \ 5 \quad \left. \begin{array}{l} \} i=4, j=5 \text{ to } 6 \\ w[4]=5 \text{ is now fixed} \end{array} \right\}$$

$$w = 2 \ 1 \ 4 \ 5 \ 3 \ 6 \quad \left. \begin{array}{l} \} i=5, j=6 \text{ to } 6 \\ w[5]=3 \text{ is now fixed} \end{array} \right\}$$

Output phase:

$$\left. \begin{array}{l} \frac{i}{1} \rightarrow b[w[1]] = a[1] \rightarrow b[2] = 2 \\ 2 \rightarrow b[w[2]] = a[2] \rightarrow b[1] = 1 \\ 3 \rightarrow b[w[3]] = a[3] \rightarrow b[4] = 4 \\ 4 \rightarrow b[w[4]] = a[4] \rightarrow b[5] = 6 \\ 5 \rightarrow b[w[5]] = a[5] \rightarrow b[3] = 2 \\ 6 \rightarrow b[w[6]] = a[6] \rightarrow b[6] = 8 \end{array} \right\} b = 1 \ 2 \ 2 \ 4 \ 6 \ 8$$

1D Mesh Even/Odd Transposition Sort

Model: 1D mesh, M , with n processors

Input: n values, held as variable x on processors 0 to $n-1$

Output: n values, sorted, held on the same mesh.

for s from 0 to $n-1$; do:

if (s is even):

for each even processor label i , in parallel do:
compareExchange ($M(i).x, M(i+1).x$) } On even cycles, even
processors are compared with their right neighbour.

else:

for each odd processor label i , in parallel do:
compareExchange ($M(i).x, M(i+1).x$) } On odd cycles, odd
processors are compared with their right neighbour.

func compareExchange (p, q) } Put two items in ascending order.

if ($q < p$):

swap (p, q)

The routine is $O(n)$, because n steps are needed to guarantee that a value can be shuffled along the length of the mesh (eg. worst case scenario of a reverse sorted array).

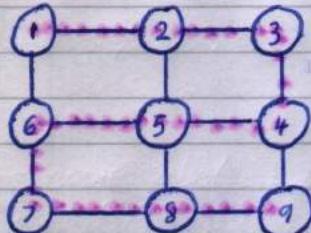
2D Mesh Snake Order Sort

Model: 2D mesh, M , with $q \times q$ processes.

Input: $n = q \times q$ values stored in the mesh M .

Output: mesh values sorted in snake order.

Snake Order:



for s from 0 to $\text{ceil}(\log_2 n)$, do:

if (s is even):

for all rows, in parallel do:
sort row with 1D mesh sort

else:

for all columns, in parallel do:
sort column with 1D mesh sort

The runtime is $\Theta(q \log_2 n)$, because each 1D mesh sort takes $\Theta(q)$.

2D Mesh Broadcast

Model: 2D mesh, M , with $n \times m$ processors. $\xrightarrow{m \uparrow^n}$

Input: value to broadcast at $M(0,0).x$

Output: $M(r, c).x = M(0,0).x$ for all $0 \leq r < n$, $0 \leq c < m$.

for c from 0 to $n - 1$, do: } Spread value across the top row,
 $M(0, c+1).x = M(0, c)$ left to right.

for c from 0 to $n - 1$, in parallel do: } For every column in parallel...

for r from 0 to $m - 1$, do: } Spread the value down each column,
 $M(r+1, c) = M(r, c)$ top to bottom.

The runtime is $\Theta(n+m)$.

2D Mesh Array Min/Max

Model: 2D mesh, M , with $n \times m$ processors. 

Input: nm values stored at variable x on processors in the mesh.

Output: min/max of the nm values.

for c from 0 to $n-1$, in parallel do: } for each column in parallel...

for r from $m-1$ to 0, do:

$$M(r, c).Temp = M(r+1, c).x$$

$$M(r, c).x = \min/\max(Temp, x)$$

} Bring the min/max to the top by starting one up from the bottom, reading the node below, then storing the min/max, until the top row is reached.

for c from $n-1$ to 0, do:

$$M(0, c).Temp = M(0, c+1).x$$

$$M(0, c).x = \min/\max(Temp, x)$$

} Work along the top row in the same fashion, from right to left, to bring the min/max into the top-left processor.

return $\overset{m}{\underset{n}{\mathbb{P}}}(0, 0).x$

The runtime is $\Theta(n+m)$.

2D Mesh Membership

Model: 2D mesh, M , with $n \times m$ processors. $m \xrightarrow{n}$

Input: $n \cdot m$ values stored in the mesh at memory variable x .
value to search for, y .

Output: Lowest index of processor containing y , or \inf .

2d-mesh-broadcast y into $M.y$ for all processors } Give each processor a
copy of y to read.

for r from 0 to $n-1$, in parallel do: } for each processor in
parallel...

for c from 0 to $m-1$, in parallel do:

if $(M(r,c).x == M(r,c).y)$: } Store the index of matching processors.

$$M(r,c).s = r \times n + c$$

else:

$$M(r,c).s = \inf$$

} Store \inf for non-matches.

2d-mesh-array-min on $M.s$ for all processors } find the smallest s
value.

return $M(0,0).s$

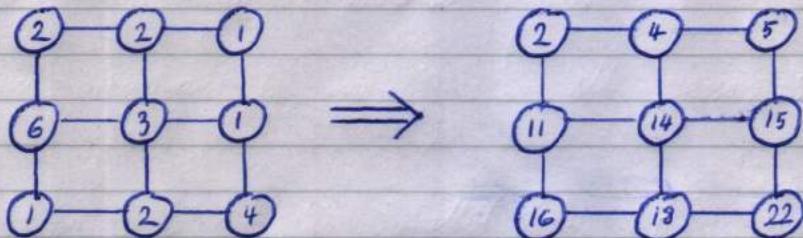
The runtime is $\Theta(n+m)$.

2D Mesh Array Prefix Sums

Model: 2D mesh, M , with $n \times m$ processors. $m \xrightarrow{n}$

Input: nm values stored as variable x in the mesh.

Output: prefix sums of the variables:



for r from 0 to $m-1$, in parallel do: } For each row in parallel:

$$M(r, 0).s = M(r, 0).x$$

for c from 1 to $n-1$, do

$$M(r, c).s = M(r, c).x + M(r, c-1).s$$

} Add up each row, left to right, so that $1 \rightarrow 2 \rightarrow 3$ becomes $1 \rightarrow 3 \rightarrow 6$.

for c from 1 to $m-1$, do:

$$M(r, n-1).s = M(r, n-1) + M(r-1, n-1)$$

} Add up the last column, top to bottom, in the same fashion. " $n-1$ " just gives the index of the last column.

for r from 1 to $m-1$, in parallel do:

for c from 0 to $n-2$, in parallel do:

$$M(r, c).s = M(r, c).s + M(r-1, n-1)$$

} On every row except the top row, for every cell except the rightmost, add the value of the rightmost cell from the row above.

The runtime is $\Theta(n+m)$.

Directed Ring Leader Election

Model: directed ring of message passing nodes.

Output: one node has status = "leader"
all other nodes have status = "not leader"

Note: the algorithm is as viewed at any node, n.

status = "unknown"
send ID to neighbour

} Initially no one knows their status, so everyone sends their ID as it might be the max.

while status = "unknown"

on receipt of message:

} Until a node learns its status, it listens for messages.

if (message == "leader"):
status = "not leader"
send "leader" to neighbour

} A leader was found elsewhere, so update status and pass the message on.

if (message is an ID):

if (message.id == ID):
status = "leader"
send "leader" to neighbour

} If a node receives its own ID then it is the leader, so it updates status and starts telling everyone.

if (message.id > ID)
send message.id to neighbour

} Drop lower IDs, but keep sending higher IDs around the ring.

$O(n)$ messages will be sent.

General Network Leader Election

Model: network of message-passing nodes.

Output: one node has status = "leader"
all other nodes have status = "not leader"

Note: algorithm is as viewed at any node, n
all nodes know the network diameter, d.

$n.\text{maxId} = n.\text{id}$ } To start, each node assumes it has the max ID.

for r from 1 to d , do: } Repeat "diameter" times.

send $n.\text{maxId}$ to all neighbours } Send the current max ID...

for all IDs received, i : } Then overwrite it if a higher
if ($i > n.\text{maxId}$): ID is received.
 $n.\text{maxId} = i$

if ($n.\text{maxId} == n.\text{id}$): } A node is the leader iff it maintains
status = "leader" its max ID ~~over~~ after d
else: cycles of "send and overwrite."
 $n.\text{status} = \text{"not leader"}$

d is the longest time required for the max ID to spread through the entire network.

General Network BFS Tree Finding

Model: network of message-forwarding nodes.

Output: every node knows its parent and children, forming a tree.

Note: algorithm is started from a specific node (such as the leader)

At the start node(s).

for all nodes, n :

$n.\text{status} = \text{"not in tree"}$	} Initialise a default state for all nodes.
$n.\text{parent} = \text{"unknown"}$	
$n.\text{children} = \{\}$	

$s.\text{parent} = s$
 $s.\text{status} = \text{"in tree"}$

} The start node, s , declares itself to be the root.	$\}$

send "join tree" to all neighbours } This starts the algorithm below.

At every node (n):

on receipt of "join tree" message:
if ($n.\text{status} == \text{"not in tree"}$):
select $v = \text{lowest ID of "join tree" sender}$
 $n.\text{parent} = v$

} When told to join a tree, pick the lowest ID of parents saying "join tree" if not already in one, then join that tree.	$\}$

send "your child" to v
send "not your child" to all neighbours except v

} Send child status messages to all neighbours.	$\}$

send "join tree" to all neighbours } Continue spreading the tree.

on receipt of "your child" message from v :
add v to $n.\text{children}$

} Record children as they arrive.	$\}$

when child status is known from all neighbours except parent:

} Tell the parent we are done when all neighbours have child status and all children (if any) have finished.	$\}$

if ($n.\text{children} == \{\}$):
send "terminated" to parent
else:
when "terminated" is received from all children:
send "terminated" to parent.

$O(e)$ messages will be sent, where e is the number of edges.

where you would go to buy it : John M.

seen 3 species, which are likely to visit the park: *Trochilus*

so now I show you what batch is available : 300

(2) $\lim_{x \rightarrow 0} \frac{\sin x}{x}$

Do my state things is another } when { in when the me
when "is" = what is
"something" = things in
{} = what is

3 Next we look at when $f(x) = 0$ {
for which x } \Rightarrow $x = \text{root of } f$

selected individuals at each site for analysis at a 'second' level.

(A) ~~2000~~ 2000 3A

• $v = \frac{1}{2} a t^2$

With this book f v of "this way" have
nothing to do with v those reading the st "this way" have

you're no mortal being { you're greater than me so there's no
difference at all

so far, it is not yet clear how much the most rapid evolution will be seen in the new species.

$\{ \cdot \} = \text{monoids} . n$) 33

Long & Astorius' bus

... "harmonie" mehr
... "harmonie" bzw.

Sequential Graph Components

Input: list of nodes, each knowing their outgoing edges.

Output: list of sets of nodes, each being a component.

Note: NOT PARALLEL! Included only for comparison.

n = list of nodes

s = list of sets

while (n is not empty):

$c = \text{lowest-index node in } n$
 $s[c] = \{c\}$
 $n. \text{remove}(c)$

} Pick the lowest-index node, create a set for it, then remove it from the "working set".

do:

fan out from C with BFS
add found nodes to $s[c]$
remove found nodes from n
while (BFS finds more nodes)

} Discover c 's entire component using repeated BFS until no more nodes are found.

return s

The runtime is $\Theta(|\text{Nodes}| + |\text{Edges}|)$, which can be close to $O(|\text{Nodes}|^2)$ for densely connected graphs.

Parallel Graph Components

Model: graph structure (vertices V and edges E)

Output: each vertex will know its super-vertex (the lowest label within its component).

Note: $S(v) = v$'s super-vertex
 $N(v) = \text{neighbours of } v$

for each v in V , in parallel do: } Each vertex starts as its own
 $S(v) = v$ super-vertex.

repeat $\log_2(|V|)$ times: } $\log_2(|V|)$ cycles needed for complete solution.

for each v in V , in parallel do:

$$\min NS = \min \{S(j) \text{ for each } j \in N(v)\}$$

$$S(v) = \min(v, \min NS)$$

} Find the minimum of the S-values held by all neighbours of each vertex, then overwrite each node's S-value if it is bigger than the found minimum.

for each v in V , in parallel do:

repeat $\log_2(|V|)$ times:

$$S(v) = S(S(v))$$

} Find the S-value root for all vertices.

The runtime is $\Theta(\log^2 |V|)$.

Parallel Minimum Spanning Tree (Boruvka's Method)

Model: graph structure

Output: MST for graph

Note: unique edge weights are required

Each vertex starts as its own single-member super-vertex.

While edges remain:

- Select the min weight edge out of each super-vertex
- Use selected edges to merge super-vertices
- Remove non-selected edges within super-vertices (self-loops)
- If multiple edges exist between components, remove all but the cheapest.
- Add all selected edges to the MST.