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 by 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.

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 paraLoop (i, j, exec):
    if (i == j):
        exec (i)                    } Base case: run exec for the single index i (j would also work).

    else:
        k = i + floor ((j-i)/2)     } Find the mid-point between i and j.

        spawn paraLoop (i, k, exec)     } Do half of the range on a new thread...

        paraLoop (k+1, j, 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 == null):                          } Base case.
        return []

    m = t.data                               } Visit this node.

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

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

    sync                                     } Wait for both traversals to finish.

    For pre-order:                           } Implicit array flattening is
    return [m, l, r]                           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):                    } Base case.
        return n

    j = floor(k/2)*                 } We will compute two halves of
                                      the problem separately.

    a = spawn paraExp(n, j)         } The first half runs on a
                                      new thread...

    b = paraFib(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 value for x, x indexes s and e — Eg. $3x^2 + 4x + 9 = [9, 4, 3]$

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

Note: initially, $s = 0$ and e = highest power in polynomial

```
func paraHorner(a, x, s, e):
    if (s == e):                    } Base case.
        return a[s]

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

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

    b = 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 $\Theta(\log_2 n)$ for a polynomial with n terms. The overall runtime is $\Theta(\log_2^2 n)$ 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 = floor((a.length + 1) / 2)      } Mid-position in a.
    x = a[m]                           } x is the pivot value.
    aLow  = a[1...m-1]                 } Split a into values above m
    aHigh = a[m+1...end]               } and values below m.
                                         - All values in aLow are ≤ x
                                         - All values in aHigh are ≥ x
    q = findPositionWithBinarySearch(b, x)  } Find the position of x
                                              in b, or its closest value
                                              ~~above~~. ~~below~~. above.
    bLow  = b[1...q-1]                 } - All values in bLow are < x
    bHigh = b[q...end]                 } - All values in bHigh are ≥ x
    l = spawn ParaMerge(aLow, bLow)    } Merge the low end in
                                         one new thread.
    r = paraMerge(aHigh, bHigh)        } Merge the high end in
                                         this thread.
    sync                               } Wait for both merges to finish.
    return [l, x, r]                   } 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 $\Theta(\log_2^2 n)$.

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: ← Loop 1

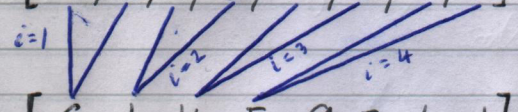
  for i from 1 to $n/2^k$, in parallel do: ← Loop 2

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

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

- Loop 1, h = 1 Loop 2, i = 1 to 4

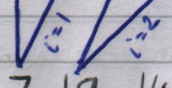
a = [4, 2, 1, 0, 9, 5, 4, 1]



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

- Loop 1, h = 2 Loop 2, i = 1 to 2

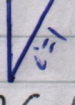
a = [6, 1, 14, 5, 9, 5, 4, 1]



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]



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.

PRAM-EREW Array Broadcast

Input: empty n-item array, a
value 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.

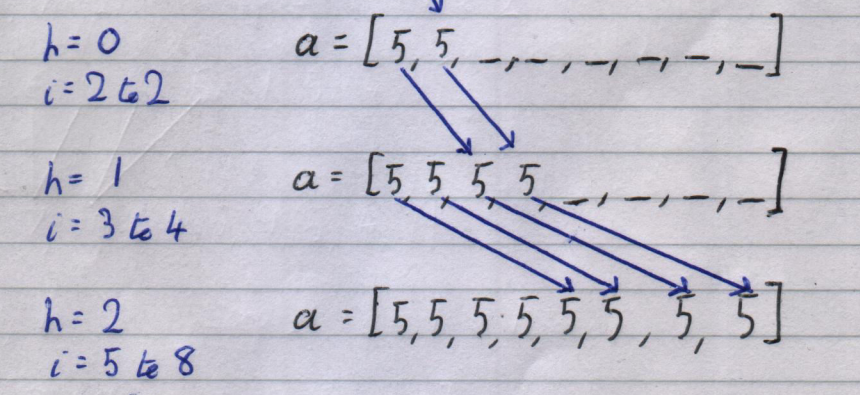
$a[i] = a[i - 2^h]$ } Copy the value from the corresponding position the "bottom half" of the current power of 2.

Example:

$x = 5$

$a = [\_, \_, \_, \_, \_, \_, \_, \_]$

Init: $a = [5, \_, \_, \_, \_, \_, \_, \_]$ *



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] = i iff a[i] = x
b[i] = inf iff a[i] ≠ 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] = i
    else:                                 } Store inf for non-matching positions.
        b[i] = inf
```

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

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

Out: b = [∞, 2, ∞, ∞, ∞, 6, ∞]

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

PRAM EREW Array Membership (Full)

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

Output: $b[1]$ = Lowest index of x in a, or infinity.

Note: $n = 2^k$

① { b = empty n-item array } This will store n copies of x, for the membership check.
Array Broadcast (b, x) } Create n copies of x.

② { Array Membership Partial (a, b) } Now b will contain the matching indexes in a.

③ { 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 EREW Array Prefix Sums

Input: n values in $a[n \ldots 2n-1]$ of array a
Eg. 4 values in $a = [-, -, -, 1, 2, 3, 4]$

Output: prefix sums in $b[n \ldots 2n-]$
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 it children. Root will be total sum.

② b = 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):
      $b[i] = b[i/2] - a[i+1]$
    else:
      $b[i] = b[(i-1)/2]$

} Move back down the layers of the tree, starting at the second and ending at the leaves.

} Left-child value = parent - right sibling.

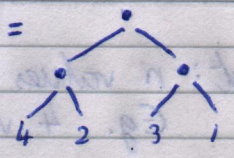
} Right-child value = parent

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

Example

a = _, _, _, 4, 2, 3, 1

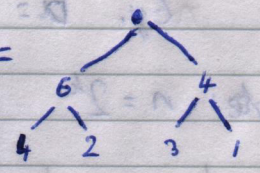
a =



① h = 1, i = 2 to 3

a = _, 6, 4, 4, 2, 3, 1

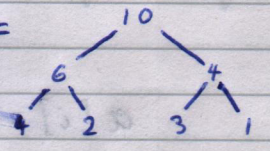
a =



② h = 0, i = 1 to 1

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

a =



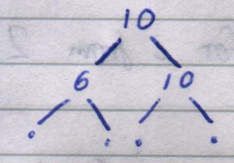
③ b = 10, _, _, _, _, _, _

④ h = 1, i = 2 to 3

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

b = 10, 6, 10, _, _, _, _

b =

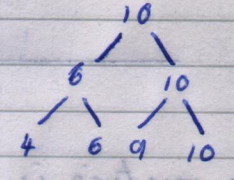


⑤ h = 2, i = 4 to 7

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

b = 10, 6, 10, 4, 6, 9, 10

b =



Result: b = 10, 6, 10, 4, 6, 9, 10

PRAM CR*W List Ranking

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 in parallel
```
for i from 1 to n, do:
    if (p[i] == i):
        d[i] = 0
    else:
        d[i] = 1
```
} Start the root at distance = 0, all others at distance = 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]]`
④ `            p[i] = p[p[i]]`
} Add the distance of the parent, then "jump past" the parent.

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

Example:

p = 5 8 1 6 2 3 7 7

d = 1 1 1 1 1 1 0 1 } ①

---

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

pp = 2 7 5 3 8 1 7 7 } ②

d = 2 2 2 2 2 2 0 1 } ③

p = 2 7 5 3 8 1 7 7 } ④

---

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

pp = 7 7 8 5 7 2 7 7 ②

d = 4 2 4 4 3 4 0 1 ③

p = 7 7 8 5 7 2 7 7 ④

---

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

pp = 7 7 7 7 7 7 7 7 ②

d = 4 2 5 7 3 6 0 1 ③

↑ Answer.

PRAM CR*W Root Finding

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

Output: root, r, for all nodes.

Note: $n = 2^k$

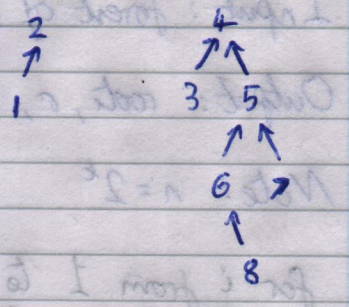
```
for i from 1 to n, in parallel do:        } Use pointer jumping to flatten
    for h from 1 to k, do:                  all trees.
        p[i] = p[p[i]]

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

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

Example

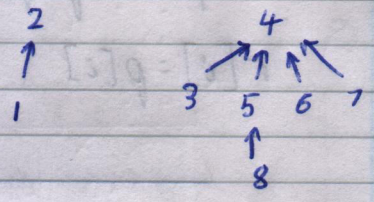
Init:

p = 2 2 4 4 4 5 5 6



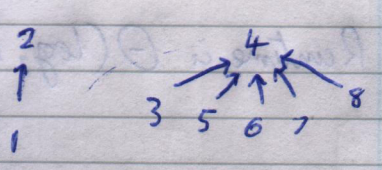
Loop 1:

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

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

Note: $n = 2^k$

① for r, c from 1 to n, in parallel do:
    $T[r, c] = A[r, c] \times B[c]$
} 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.

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

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

The runtime is $\Theta(\log_2 n)$ and the algorithm requires $\Theta(n^2)$ processors.

Example

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

① $$T=\begin{bmatrix}1\times2&2\times1&4\times3&1\times2\\5\times2&4\times1&0\times3&1\times2\\8\times2&2\times1&1\times3&2\times2\\2\times2&1\times1&9\times3&3\times2\end{bmatrix}=\begin{bmatrix}2&2&12&2\\10&4&0&2\\16&2&3&4\\4&1&27&6\end{bmatrix}$$

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

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

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

PRAM CR*W Matrix*Matrix Multiplication

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

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

Each cell in the $n \times n$ output matrix is the sum of n multiplications. 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 $T[r,c]$.

② 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 $n/2^h$, 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 fan in.

Q = empty $n \times n$ 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 $\Theta(\log_2 n)$ with n^3 processors.

Example:

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

① :
$$T = \begin{bmatrix} [1\times5,\ 2\times7] & [1\times6,\ 2\times8] \\ [3\times5,\ 4\times7] & [3\times6,\ 4\times8] \end{bmatrix} = \begin{bmatrix} [5,14] & [6,16] \\ [15,28] & [18,32] \end{bmatrix}$$

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:     } Compute the sum in
    s = sta[i]                           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]

Note: $n = 2^k$

```
m = empty n-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 "loses".

```
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 runtime is $\Theta(1)$, requiring $\Theta(n)$ processors 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 = empty n-item array

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

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 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 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.

```
for i from 1 to n, in parallel do:
    b[w[i]] = a[i]
```

Copy all values to their correct positions in the output, using the knowledge that w[i] contains the correct index for a[i].

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

Example

a = 2 1 4 6 2 8 } Initial setup.

w = 1 1 1 1 1 1 }

w = 2 1 2 2 2 2 } i = 1, j = 2 to 6
w[1] = 2 is now fixed

w = 2 1 3 3 3 3 } i = 2, j = 3 to 6
w[2] = 1 is now fixed

w = 2 1 4 4 3 4 } i = 3, j = 4 to 6
w[3] = 4 is now fixed

w = 2 1 4 5 3 5 } i = 4, j = 5 to 6
w[4] = 5 is now fixed

w = 2 1 4 5 3 6 } i = 5, j = 6 to 6
w[5] = 3 is now fixed

Output phase:

i			
1	→ b[w[1]] = a[1]	→	b[2] = 2
2	→ b[w[2]] = a[2]	→	b[1] = 1
3	→ b[w[3]] = a[3]	→	b[4] = 4
4	→ b[w[4]] = a[4]	→	b[5] = 6
5	→ b[w[5]] = a[5]	→	b[3] = 2
6	→ b[w[6]] = a[6]	→	b[6] = 8

} 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)
    else:
        for each odd 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.

On odd cycles, odd processors are compared with their right neighbour.

```
func compareExchange(p, q)
    if (q < p):
        swap(p, q)
```

Put two items in ascending order.

The runtime 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).

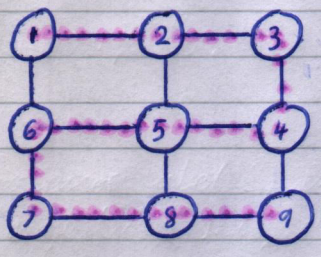
2D Mesh Snake Order Sort

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

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

Output: mesh values sorted in snake order.

Snake Order:



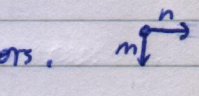
for s from 0 to ceil($\log_2 n$), do:

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

2D Mesh Broadcast

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

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-2, do:
    $M(0, c+1).x = M(0,c)$
} Spread value across the top row, left to right.

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

    for r from 0 to n-2, do:
        ~~$M(r, c+$~~
        $M(r+1, c) = M(r,c)$
    } Spread the value down each column, top to bottom.

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

2D Mesh Array Min/Max

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

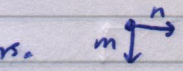
Input: nn values stored at variable x on processors in the mesh.

Output: min/max of the nn values.

```
for c from 0 to n-1, in parallel do:        } For each column in parallel...
    for r from n-2 to 0, do:                } 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.
        M(r,c).Temp = M(r+1,c).x
        M(r,c).x = min/max(Temp, x)
for c from n-2 to 0, do:                    } Work along the top row in the same fashion, from right to left, to bring the min/max into the top-left processor.
    M(0,c).Temp = M(0,c+1).x
    M(0,c).x = min/max(Temp, x)
return M(0,0).x
```

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

2D Mesh Membership

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

Input: nm 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 $n-1$, 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: } Store inf for non-matches.
      $M(r,c).s = inf$

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

return $M(0,0).s$

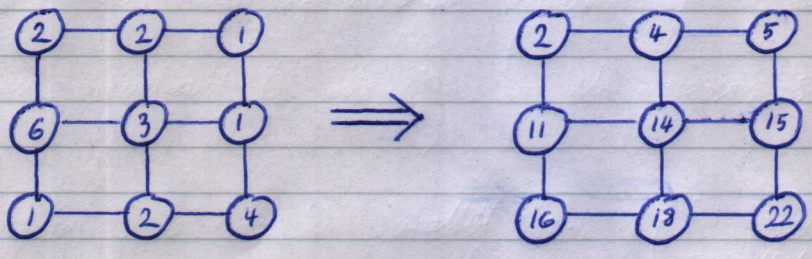
The runtime is $\Theta(n+m)$.

2D Mesh Array Prefix Sums

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

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

Output: prefix sums of the variables:



for r from 0 to n-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→2→3 becomes 1→3→6.

for r from 1 to n-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 n-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.maxId = n.id        } To start, each node assumes it has the max ID.

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

    send n.maxId to all neighbours    } Send the current max ID...

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

if (n.maxID == n.id):                 } A node is the leader iff it maintains
    status = "leader"                     its max ID after d
else:                                     cycles of "send and overwrite.
    status = "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.status = "not in tree"
    n.parent = "unknown"
    n.children = {}
```
} Initialise a default state for all nodes.

```
s.parent = s
s.status = "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.status == "not in tree"):
        select v = lowest ID of "join tree" senders

        n.parent = v

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

        send "join tree" to all neighbours
```
} 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 child status messages to all neighbours.

} Continue spreading the tree.

```
on receipt of "your child" message from v:
    add v to n.children
```
} Record children as they arrive.

```
when child status is known from all neighbours except parent:

    if (n.children == {}):
        send "terminated" to parent
    else:
        when "terminated" is received from all children:
            send "terminated" to parent.
```
} Tell the parent we are done when all neighbours have child states and all children (if any) have finished.

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

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 = lowest-index node in n      } Pick the lowest-index node,
    s[c] = {c}                      } create a set for it, then remove
    n.remove(c)                     } it from the "working set".

    do:                             } Discover c's entire component
        fan out from c with BFS     } using repeated BFS until no
        add found nodes to s[c]     } more nodes are found.
        remove found nodes from n
    while (BFS finds more nodes)

return s
```

The runtime is $\Theta(|Nodes| + |Edges|)$, which can be close to $O(|Nodes|^2)$ 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
$N(v)$ = neighbours of v

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

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

  for each v in V, in parallel do: } 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.
    $minNS$ = min of $S(j)$ for each j in $N(v)$
    $S(v)$ = min (v, $minNS$)

  for each v in V, in parallel do: } Find the S-value root for all vertices.
    repeat $\log_2(|V|)$ times:
      $S(v) = S(S(v))$

The runtime is $\Theta(\log_2{}^2 |V|)$.

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.