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.

Notes are originally from **https://github.com/markormesher/CS-Notes** and remain the

Contents

Recap: Algorithms

An algorithm is a **well-defined finite set** of rules that describe a **sequence of primitive operations** that can be applied to some **input** to produce some **output** in a finite amount of time.

Any algorithmic solution to a problem will include **designing** an algorithm and then **analysing its performance**. For some algorithms, parallelism can be used to improve performance.

Runtime of Algorithms

The runtimes and memory consumption of algorithms is usually stated with **Big-O** notation, but **Big-Θ** and **Big-Ω** may also be used. These are the same notations that has been covered in other notes and as such will not be covered here, other than a short summary:

- If f and g are functions of n, then $f = O(g)$ means that there is a constant $c > 0$ such that $f(n) <= c \times g(n)$ (i.e. f grows no faster than g)
- **Big-O** (big-oh) is an **upper** bound ('grows no faster than...')
- **Big-Θ** (big-theta) is a **tight** bound ('grows at...')
- **Big-Ω** (big-omega) is a **lower** bound ('grows at least as fast as...')
- $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$ (or they are roughly within a constant of each other)

Introduction

Parallelism is everywhere and can be exploited to improve performance:

- Global scale: huge computational grids, such as Folding@Home
- Super computers: simulations, modelling, etc.
- Desktop scale: multi-code CPUs and GPUs
- Specialised hardware for processes like encryption, hashing, etc.

Studying Parallel Algorithms

To study and consider parallel computing we will need **four things**:

- **Machine models** to define what operations are available at a fairly abstract level.
- **Cost models** to define the costs of these operations in terms of resources we care about (time and memory, usually).
- **Analysis techniques** to help us to map from algorithms to costs with reasonable accuracy.
- **Metrics** that let us discriminate between costs (e.g. speed vs. accuracy).

Programming Models

There are different programming models for parallel algorithms that reflect differences in underlying architectures:

- The **shared address space** model allows threads to interact directly through shared memory spaces. Care is needed to avoid race conditions and other problems. We will consider two variations of this model: **multi-threading** and **PRAM**.
- The **message passing** model gives each process its own address space and allows results to be sent between processes using messages. We will consider a **graph interconnection network** model.

Multi-Threading (Shared Address Space Model)

We consider the operations `spawn` and `sync` without worrying about the underlying hardware that enables them.

- `spawn` creates an instance of some process to run in parallel.
- `sync` waits for spawned processes to stop and allows results to be collated.

Many languages support these functions already and allow the creation of separately runnable processes called threads. Exact scheduling and parallelism is left to the OS - we consider this model at a higher, logical level.

PRAM (Shared Address Space Model)

Parallel Random Access Machine

In the PRAM model, **multiple processors act synchronously** to execute the same command on **different sets of data** (SIMD = Same Instruction, Multiple Data).

There are several **read/write possibilities**, from any mix of ER, EW, CR, CW (exclusive/-concurrent read/write). An EREW algorithm can be very different to a CRCW algorithm.

Graph Interconnected Network (Messaging Model)

In this model, a graph is represented as a set of **nodes/vertices (processors)** and **edges (links)**: $G = (V, E)$. Each processor has **its own memory**, and again a **SIMD** structure is followed. Some important notation:

- Each node i in the set V is the processor P_i.
- Each edge (i, j) in the set E is a two-way link between the processors P_i and P_j.
- $P \cdot X$ is the value of the variable X at the node (processor) P.
- P_i and P_j can communicate directly only if joined by an edge (i, j).

Sequential, Parallel, Distributed?

Parallel vs. Sequential

Some things are inherently sequential (like a train journey), whereas some things are inherently parallel (like a marathon). **Divide and conquer** algorithms can often be parallelised, but not always.

Parallel vs. Distributed

- **Parallel** systems have some central controller.
- **Distributed** systems have no central controller and cooperation is needed.

Multi-Threading Model

The multi-threading model uses the operations **`spawn`** and **`sync`** without worrying about the underlying hardware that enables them.

- **`spawn`** creates an instance of some process to run in parallel.
- **`sync`** waits for spawned processes to stop and allows results to be collated.

Many languages support these functions already and allow the creation of separately runnable processes called threads. Exact scheduling and parallelism is left to the OS - we consider this model at a **higher, logical level**.

Example: Parallelised Fibonacci Calculation

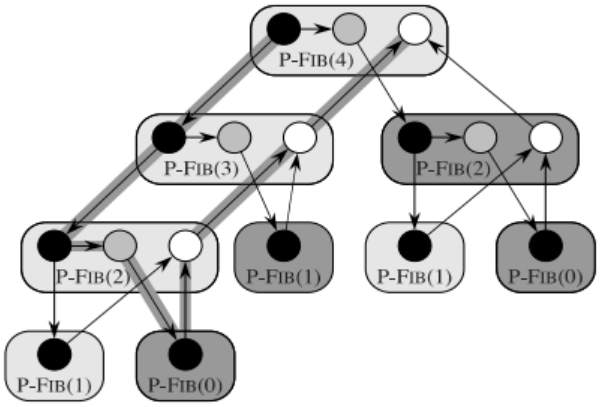
The 'dumb' method is the slow, recursive version:

```
1 fun recFib(n):
2     if (n <= 1) return n
3     return recFib(n — 1) + recFib(n — 2)
```

This can be parallelised by running the two recursive calls in parallel, so they can work side by side. This **does not reduce the work** to be done, it just gets it done faster.

```
1 fun parFib(n):
2     if (n <= 1) return n
3     x = spawn parFib(n — 1)
4     y = parFib(n — 2)
5     sync
6     return x + y
```

The process call diagram for `pFib(4)` is shown below:



The diagram forms a **directed, acyclic graph** (a.k.a. a **computation DAG**) with a **single source** and **single sink** node at the root.

- Black nodes: entry point of a spawned process
- Grey nodes: work done within the active process
- White nodes: synced results

The **critical path** is the longest path through the whole system (shaded in the diagram above).

Complexity Measures for Multi-Threading

T_p = the time to execute on p processors.

The **span** S or T_∞ = the number of vertices on the critical path.

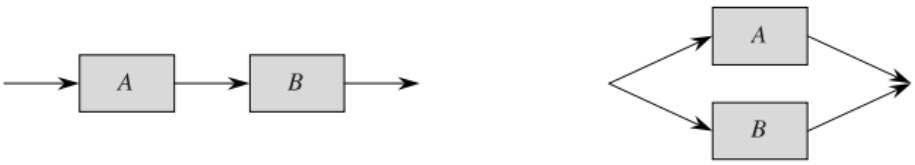
The **work** W or T_1 = the time to execute on one processor.

Speed up = T_1/T_p (i.e. how much faster it runs on p processors).

Parallelism = T_1/T_∞ (i.e. the maximum possible speed up).

Note: the **span** represents the unavoidable sequential work that must be done. The **work** is defined as the number of vertices in the **computation DAG**.

Work vs. span: when parallelised, work remains the same, but the span will reduce.



In the first case, work is $W = A + B$ and the span is $S = A + B$. In the second case, work is the same, but the span is now $S = max(A, B)$.

Example: Adding 1, n Times

This trivial example reduces to $S(n) = n$. We can **assume n will be some power of 2**.

The sequential approach is obvious, and the recursive approach is as follows:

```
1 fun recAdd(n):
2     if (n <= 1) return n
3     return recAdd(n / 2) + recAdd(n / 2)
```

The recursive approach can be parallelised as before:

```
fun recAdd(n):
    if (n <= 1) return n
    x = spawn recAdd(n / 2)
    y = recAdd(n / 2)
    sync
    return x + y
```

Assuming $n = 2^m$...

$T_1(n) = \Theta(n)$ because the actual work will comprise n additions.

$$\begin{aligned} T_\infty(n) &= \Theta(1) + max(T_\infty(n/2), T_\infty(n/2)) \\ &= \Theta(1) + T_\infty(n/2) \\ &= (m-1) \times \Theta(1) + T_\infty(n/2^m) \\ &= m \times \Theta(1) \\ &= m \end{aligned}$$

Parallelism is $T_1(n)/T_\infty(n) = \Theta(n/log_2(n))$. This is am almost linear speed-up over the initial sequential algorithm.

Example: Computing Squares

$$\begin{aligned} S(n) &= 1^2 + 2^2 + 3^2 + ... + n^2 \\ &= n^2 + (n-1)^2 + ... + 1 \end{aligned}$$

```
fun parSqaure(n):
    if (n == 1) return 1
    x = spawn parSquare(n − 1)
    y = n * n
    sync
    return x + y
```

Here, the spawned task does almost all of the work. On the **computation DAG**, all nodes are on the critical path, so this won't speed anything up. This is therefore **not parallel computing**; it is sequential computing *disguised* as parallel.

$T_1(n) = n$

$$\begin{aligned} T_\infty(n) &= \Theta(1) + max(1, T_\infty(n-1)) \\ &= \Theta(n) \end{aligned}$$

Parallelism is $n/n = 1$ (i.e. useless parallelisation).

Speed-up Bounds for p Processors

Reminder: speed-up = T_1/T_p for p processors.

How much faster can something be made to run? What are the bounds on $T_p(n)$?

A very crude bound is $T_p \geq T_1/p$ if the **work is split perfectly** between p processors. Any better than this would infer that a processor is running faster than the one used for T_1, making the comparison pointless.

In reality, it's **hard to split work perfectly**. This gives the bound $p \times T_p \geq T_1$ because there will always be some **overhead** and **non-perfect sharing** of work.

If p **is very large**, this lower bound is inaccurate. In any problem there will be some sequential work (indicated by the **span**), which affects the real lower bound. We need better accuracy.

Greedy Scheduling Principle

A greedy scheduler allocates work to any processor **as soon as it becomes free**.

If a computation is ran on a greedy scheduler, T_p is bounded by $T_p \leq (W/p) + S$. The lower bound is $T_p \geq max(W/p, S)$.

W/p assumes work can be allocated so that all p processors finish at the same time, therefore:

$$max(\frac{W}{p}, S) \leq T_p \leq \frac{W}{p} + S$$

Increasing p reduces W/p, but S does not change. If $W/p < S$, then S dominates the equation and we are wasting resources because some processors will lie empty, waiting to be used. **There is no value in reducing W/p below S.**

PRAM Model

In the PRAM model, **multiple processors act synchronously** to execute the same command on **different sets of data** (SIMD = Same Instruction, Multiple Data).

There are several **read/write possibilities**, from any mix of ER, EW, CR, CW (exclusive/-concurrent read/write). An EREW algorithm can be very different to a CRCW algorithm. Various **CW protocols** exist:

- The simplest is `priority` - the lowest ID process has priority.
- Another is `arbitrary` - a process is picked at random.
- Another is `common` - the write is allowed only if the values are the same.

CW can create a problem: if all processes are waiting to write, the program is sequential.

Example: Binary Fan-In Array Sum on EREW-PRAM

$A[1..n]$ = n numbers. We want to put the sum of the n numbers into $A[1]$.

We can halve the array size in **one parallel step** by adding the values at locations $2i$ and $2i - 1$, then putting them into location i. Overwriting data isn't a problem, because this is a **SIMD** system - every process reads their cells, performs the addition and write the result at the same time (read, read, add, write). This pattern is known as a **binary fan-in** and works for **any associative binary operation**.

```
1  fun arrSum—PRAM—EREW:
2      Input: n numbers stored in the array A[1..n]
3      Output: sum of A[1..n], written into A[1]
4      Note: n = 2^k; k = log_2(n)
5
6      for i from 1 to n, in parallel do:
7          B[i] = A[i]
8
9      for h from 1 to k, do:
10         for i from 1 to (n / 2^h), in parallel do:
11             B[i] = B[2i] + B[2i — 1]
12
13     A[1] = B[1]
```

On line 10, the i runs from 1 to $n/2$, then $n/4$, then $n/8$, etc.

Analysis

- Line 7: one round of parallel $\Theta(1)$ time

- Line 11: $k = log_2(n)$ rounds of parallel $\Theta(1)$ time
- Line 13: $\Theta(1)$ time

Example: Array Sum CRCW-PRAM with Priority-CW

```
1  fun arrSum—PRAM—EREW:
2      Input: n = 2^k numbers stored in the array A[1..n]
3      Output: sum of A[1..n], written into A[1]
4      Note: k = log_2(n)
5      Note: CW rule is PRIORITY—CW
6
7      S = 0
8      for i from 1 to n, in parallel do:
9          S = S + A[i]
10     A[1] = S
```

This works if you consider each processor to execute each parallel line in one step. After $S = 0$, this gives:

- $S = S + A[1] = A[1]$
- $S = S + A[2] = A[1] + A[2]$
- $S = S + A[3] = A[1] + A[2] + A[3]$
- etc.

Analysis

- Line 7: $\Theta(1)$ time
- Line 9: one round of parallel $\Theta(1)$ time
- Line 10: $\Theta(1)$ time

Greedy Work-Time Scheduling Principle

This is the PRAM equivalent of the greedy scheduling principle defined for the multi-threading model.

We know that $T_\infty(n) \leq T_p(n)$ and we assume that p processors can do p units of work in one time step. Therefore:

$$T_\infty(n) \leq T_p(n) \leq \frac{W(n)}{p} + T_\infty(n)$$

There are $W(n)$ operations to be done, and we assume that p processors can do p units of work in one step, giving $W(n)/p$ time steps, plus $T_\infty(n)$ unavoidable sequential steps.

Example 1

An algorithm has $T_\infty = 50$ and $W = 10,000$ on some input. How do increasing p values affect the runtime bounds?

- $50 \leq T_{200} \leq 100$
- $50 \leq T_{1000} \leq 60$
- $50 \leq T_{10000} \leq 51$
- $50 \leq T_{100000} \leq 51$

Once $W(n)/p$ drops below $T_\infty(n)$, there is no value to increasing p.

Example 2

An algorithm has $T_\infty(n) = \Theta(n^{2/3})$ and $W(n) = \Theta(n)$. What is the maximum useful p?

$$T_\infty(n) \leq T_p(n) \leq \frac{W(n)}{p} + T_\infty(n)$$

$$\Theta(n^{2/3}) \leq T_p(n) \leq \frac{\Theta(n)}{p} + \Theta(n^{2/3})$$

The maximum useful p is $n^{1/3}$. When p is this large, $\frac{\Theta(n)}{p} = \Theta(n^{2/3})$, which gives the best possible upper bound; any more processors and resources would be wasted, because the constant $T_\infty(n) = \Theta(n^{2/3})$ begins to dominate the equation.

PRAM Algorithms

Matrix $*$ Vector Multiplication

Matrix and vector multiplications are good candidates for parallel computation because each element of the result can be computed independently and only associative operations are used.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

The sequential algorithm has a runtime of $\Theta(n^2)$:

```
fun matrixVectorMult—SEQ:
    Input: n * n matrix A, n * 1 vector B
    Output: vector Y = A * B

    for r from 1 to n, do:
        Y[r] = 0
        for c from 1 to n, do:
            Y[r] = Y[r] + A[r, c] * B[c]
```

This can be parallelised by computing all of the multiplications in one parallel step, using the parallel addition algorithm (*see more: Example: Binary Fan-In Array Sum on EREW-PRAM, page 11*) to add up each row ($log_2(n)$ parallel steps), then copying into the result (one parallel step).

```
fun matrixVectorMult—PRAM—CREW:
    Input: n * n matrix A, n * 1 vector B
    Output: n * 1 vector Y = A * B
    Note: n = 2^k; k = log_2(n)

    for r, c from 1 to n, in parallel do:
        C[r, c] = A[r, c] * B[c]

    for i from 1 to n, in parallel do:
        for h from 1 to k, do:
            for j from 1 to (n / 2^h), in parallel do:
                C[i, j] = C[i, 2j] + C[i, 2j — 1]

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

With enough processors, this reduces the runtime from $\Theta(n^2)$ to $\Theta(log_2(n))$.

Matrix $*$ Matrix Multiplication

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} =$$

$$\begin{bmatrix} aj + bm + cp & ak + bn + cq & al + bo + cr \\ dj + em + fp & dk + en + fq & dl + eo + fr \\ gj + hm + ip & gk + hn + iq & gl + ho + ir \end{bmatrix}$$

```
fun matrixMatrixMult—PRAM—CREW:
    Input: n * n matrix A, n * n matrix B
    Output: n * n matrix Y = A * B
    Note: n = 2^k; k = log_2(n)

    for i, j, p from 1 to n, in parallel do:
        C[i, j, p] = A[i, p] * B[p, j]

    for i, j from 1 to n, in parallel do:
        for h from 1 to k, do:
            for p from 1 to (n / 2^h), in parallel do:
                C[i, j, p] = C[i, j, 2l] + C[i, j, 2l — 1]

    for i, j from 1 to n, in parallel do:
        Y[i, j] = C[i, j, 1]
```

This still has a runtime of $\Theta(log_2(n))$ with n^3 processors or more.

Binary Tree Algorithms

Recap: Binary Tree Basics

- The **height** h of a binary tree is the maximum distance from the root to any leaf.
- If the tree is **complete** then h will be the same for all leaves.
- A complete binary tree of height h has $2^{h+1} - 1$ nodes, of which 2^h as leaves and 2^{h+1} are internal nodes.
- If there are T leaves then the total tree size is $2T - 1$.
- Tree nodes are labelled as follows:

- The root node has index 1.
- For any node at index i, its left child has index $2i$ and its right child has index $2i+1$.

Array Membership Algorithm EREW (Intro)

To achieve a complete PRAM-EREW array membership checking method, we will use the following components:

- Broadcast (array filling)
- Parallel Array Membership
- Binary Fan-in Array Minimum

The resulting algorithm will take an input array and a single element, and return the lowest index at which that element occurs in the array (or ∞ if it doesn't). The trivial solution runs in $O(n)$; this solution will run in $\Theta(log_2(n))$.

Broadcast (Array Filling) EREW

Because we are building an **ER**EW algorithm, the single element we are searching for must be replicated n times, so that all parallel processes can read it at the same time. Typically this would take $\Theta(n)$, defeating the purpose of the parallelisation, but it can be done in $\Theta(log_2(n))$.

```
1 fun arrBroadcast—PRAM—EREW:
2     Input: an n—item array A, a number X to be broadcast to the array A
3     Output: A[i] = n for i = 1..n
4     Note: n = 2^k; k = log_2(n)
5
6     A[1] = X
7     for i in 0 to k—1, do:
8         for j in 2^i + 1 to 2^{i+1}, in parallel do:
9             A[j] = A[j — 2^i]
```

In this method the first element of the array is set manually, then one parallel thread copies the value to the next element, then two parallel threads copy those values to the next two elements, then four parallel threads copy those values to the next four elements, etc.

Parallel Array Membership EREW

This method will use n parallel threads to compare two n-length arrays; the first is the input array, the second is the array of the element we are searching for (created with the broadcast method above). For each process, if the elements in corresponding cells match, the ID of that process is written back into the second array; if they do not match, ∞ is written.

```
fun arrMembership—PRAM—EREW:
    Input: an n—item array A,
           an n—item array B containing the search number
    Output: B[i] = process ID if A[i] = the search number,
            inf. otherwise

    for i in 1 to n, in parallel do:
        if A[i] == B[i]
            B[i] = i
        else
            B[i] = inf
```

This algorithm runs in one parallel step.

Binary Fan-in Array Minimum EREW

A binary fan-in can apply any binary associative operator to reduce an array to one element. *See more: Example: Binary Fan-In Array Sum on EREW-PRAM, page 11*. Here, we use it to find the array minimum:

```
fun arrMin—PRAM—EREW:
    Input: n numbers stored in the array A[1..n]
    Output: min of A[1..n], written into A[1]
    Note: n = 2^k; k = log_2(n)

    for i from 1 to n, in parallel do:
        B[i] = A[i]

    for h from 1 to k, do:
        for i from 1 to (n / 2^h), in parallel do:
            B[i] = min(B[2i] + B[2i — 1])

    A[1] = B[1]
```

Array Membership Algorithm EREW (Algorithm)

1. **arrBroadcast—PRAM—EREQ**
   to create the temporary n-array of the searched-for number
2. **arrMembership—PRAM—EREQ**
   to isolate the positions that contain the searched-for number
3. **arrMin—PRAM—EREQ**
   to return the lowest position from the previous step

Binary Fan-in Array Maximum EREW

This algorithm is very similar to the binary fan-in array minimum seen earlier, but it uses a double-sized array to avoid overwriting values, and **indexes 'backwards'** for no particular reason (other than to show that it can be done).

```
fun arrMax—PRAM—EREW:
    Input: n numbers stored in the array A[n+1..2n]
    Output: max of A[n+1..2n], written into A[1]
    Note: n = 2^k; k = log_2(n)

    for i from k — 1 to 0, do:
        for j from 2^i to 2^(i+1) — 2, in parallel do:
            A[j] = max(A[2j], A[2j + 1])
```

In this $\Theta(log_2(n))$ algorithm, each cell is written to only once, and no data is overwritten.

Parallel Prefix Sums EREW

Input:	1	2	3	4	5
Output:	1	3	6	10	15

For an input array of size n (where $n = 2^k$), the trivial sequential approach to this has a runtime of $\Theta(n)$. It can be parallelised to achieve $\Theta(log_2(n))$ as follows:

1. Construct a complete binary tree with n leaves (i.e. $2n - 1$ nodes in total).
2. Place the array values on the leaves.
3. Work up through the tree, putting the sum of each pair of child nodes into the parent node, ending with the full array sum in the root.
    - This can be done in $log_2(n)$ parallel steps.
4. Work down through the tree, and for each non-leaf node:
    - Set the left child to equal the parent node's value minus the right child's value.
    - Set the right child to equal the parent node's value.
    - This can be done in $log_2(n)$ parallel steps.
5. The output can now be read from the leaves.

```
fun prefixSum—PRAM—EREW:
    Input: n numbers, stored in entries n..2n—1 of array A[1..2n]
    Output: prefix sums, stored in entries n..2n—1 of array A[1..2n]
    Note: n = 2^k; k = log_2(n)

    for m from k — 1 to 0, do:
        for i from 2^m to 2^(m+1)—1, in parallel do:
            A[i] = A[2i] + A[2i + 1]

    B[1] = A[1]
    for m from 1 to k, do:
        for i from 2^m to 2^(m+1)—1, in parallel do:
            if i is odd
                B[i] = B[(i — 1) / 2]
            else
                B[i] = B[(i / 2]) — A[i + 1]
```

Array Maximum CRCW

This algorithm is very similar to the binary fan-in array minimum seen earlier, but it uses a double-sized array to avoid overwriting values, and **indexes 'backwards'** for no particular reason (other than to show that it can be done).

```
fun arrMax—PRAM—CRCW:
    Input: n numbers stored in the array A[1..n]
    Output: max of A[1..n], written into A[1]
    Note: n = 2^k; k = log_2(n)

    // initialise n—size temp array with broadcast
    M = [0, 0, ...]

    // pairwise compare
    for all pairs (i, j), 1 <= i, j <= n, in parallel do:
        if A[i] > A[j]
            M[j] = 1 // j has "lost"

    // smallest max index
    for all pairs (i, j), 1 <= i, j <= n, i < j, in parallel do:
        if M[i] == 0 and i < j
            M[j] = 1 // higher index has "lost"

    // store max
    for i from 1 to n, in parallel do:
        if M[i] == 0
            Max = A[i]
            IndexOfMax = i
```

This algorithm runs in $\Theta(1)$, assuming infinite processors (it has $O(n^2)$ parallel steps).

The **CW model doesn't matter**, because only the value 1 is ever written into `M`.

Array Membership CRCW

```
fun arrMembership—PRAM—CRCW:
    Input: n numbers stored in the array A[1..n],
           a number X to check
    Output: true if X is in A
    Note: n = 2^k; k = log_2(n)

    Index = —1
    for i from 1 to n, in parellel do:
        if A[i] == X
            Index = i

    XIsInArray = (Index != —1)
```

This $\Theta(1)$ algorithm (with $\Theta(n)$ parallel steps) works for `priority` and `arbitrary` CW modes, but not `common`.

Is PRAM-CRCW More Powerful than PRAM-EREW?

Yes. **CRCW-`priority` is the most powerful** model; **EREW is the least powerful** model.

When computing boolean functions of size n in the form $(a_1|a_2|...|a_n)$ CRCW gives performance of $O(n)$, whereas EREW gives $\Theta(log_2(n))$ using binary fan-in.

- *See more: Example: Binary Fan-In Array Sum on EREW-PRAM, page 11*

Simulation

CRCW algorithms can be simulated on EREW with a $log_2(n)$ overhead.

- Concurrent (CRCW $O(1)$) read/write operations can be implemented on an n-processor EREW system to run in $O(log_2(n))$ time.
- CR: use broadcasting to implement on ER.
  - *See more: Broadcast (Array Filling) EREW, page 16*
- CW: the processor IDs that want to write go into an array and binary fan-in is used to find the minimum (simulating CW-`priority`), which is then allowed to write.

PRAM-CRCW Sorting

```
fun arrSort—PRAM—CRCW:
    Input: n unsorted numbers in the array A[1..n]
    Output: n sorted numbers in the array B[1..n]
    Note: n = 2^k; k = log_2(n)
          uses the work array W[1..n]

    for i from 1 to n, in parallel do
        W[i] = 1

    // check all pairs and +1 to the position of the "winner"
    for all pairs (i, j), 1 <= i, j <= n, i < j, in parallel do:
        if A[i] > A[j]
            W[i] = W[i] + 1
        else
            W[j] = W[j] + 1

    // for i from 1 to n, W[i] is the number of items that
    // are smaller than or equal to A[i]

    // copy items into position
    for i from 1 to n, in parallel do:
        B[W[i]] = A[i]
```

Modelling Other Data Structures

All data in these parallel algorithms is **represented as an array**, which is an abstraction of a block of memory, but **other data structures can be represented** within the array with the right indexing.

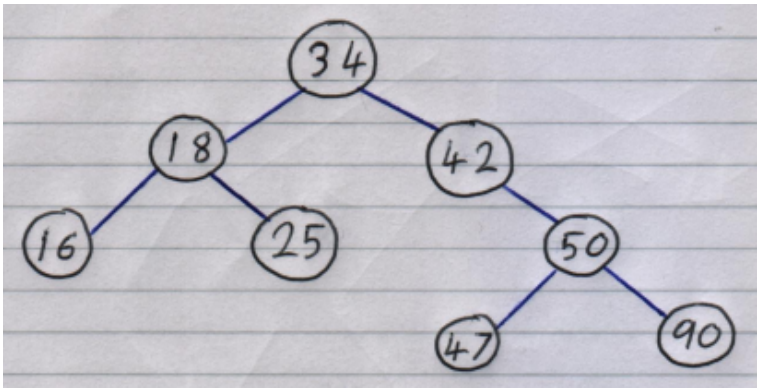
Parent Labelling

Parent labelling can be used to **represent lists or trees**. Every item in the data array is a node or item within the list/tree, each of which has a corresponding value in the same position in the **parent array** P, indicating the index of that node/item's parent.

- If i is someone's child, then $P[i]$ is the index of i's parent.
- If i is a root, then $P[i] = i$.

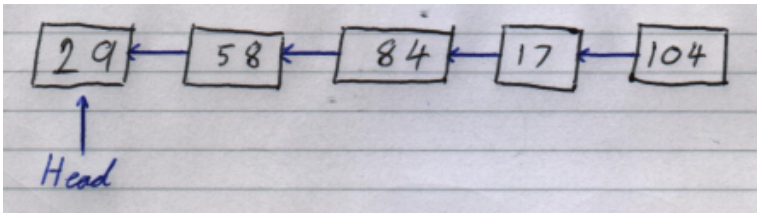
Example: Tree

Index	1	2	3	4	5	6	7	8
Data	34	18	42	50	47	16	90	25
Parent	1	1	1	3	4	2	4	2



Example: List

Index	1	2	3	4	5
Data	84	104	58	29	17
Parent	3	6	4	4	1



List Ranking

The rank of an item in a list is its **distance from the head/root of the list**. It can be determined efficiently by a parallel algorithm using **pointer jumping**.

Note: pointer jumping will **corrupt the parent array**. Make a duplicate first if needed.

```
fun listRank—PRAM—CREW:
    Input: parent array P[1..n]
    Output: distance array D[1..n] giving the distance of each
            node i from the head/root of the list
    Note: n = 2^k; k = log_2(n)

    // initialise distance
    for i from 1 to n, in parallel do
        if P[i] == i
            D[i] = 0 // for the head/root
        else
            D[i] = 1 // for everything else

    for i from 1 to log_2(n), do:
        for j from 1 to n, in parallel do:
            if P[j] != P[P[j]]
                D[j] = D[j] + D[P[j]]
                P[j] = P[P[j]]
```

This algorithm determines the distance/rank value for n items in $\Theta(log_2(n))$.

It works by initialising everything at 1 away from the root (except for the root iteself) and then repeatedly 'jumping' everything to its parent's pointed until the root is reached. At each jump, it counts the distance it already was from the root, plus the distance of the parent it just 'jumped' past.

Forest Root Finding

```
fun forestRoots—PRAM—CREW:
    Input: parent array P[1..n]
    Output: root array R[1..n] giving the root of each node i in the forest
    Note: n = 2^k; k = log_2(n)

    for i from 1 to n, in parallel do:
        for j from i to log_2(n), do:
            P[i] = P[P[i]]
        P[i] = P[i]
```

Evaluating Parallel Algorithms

There are many things to consider when understanding parallel algorithms:

- Pseudocode with some parallelisation syntax.
- Algorithm design ('why is something done a certain way?') - often linked to *where* the data needs to go.
- Flow charts ('what is being done?').
- The number of sequential steps (i.e. the **span**).
- The best sequential algorithm, for comparison (this might not be known).
- Analysis of work (multi-threading)/time (PRAM).
- Efficiency metrics (like **speed-up** and **parallelism**).

Analysis of Algorithms

We apply **'Big-O thinking'** and a **uniform cost model** to evaluate algorithms, wherein we charge 1 unit for each **basic operation**. What we charge for depends on the computation model:

- Multi-Threading: we charge for recursive calls, including base cases, spawn, sync, etc.
- PRAM: we charge for every basic operation.
- Graph Interconnected Network: we charge for communication/messaging.

Work/Time and Work/Span

In the multi-threading and PRAM models, work W is the number of basic operations required in the sequential execution.

- **Span** $S = T_\infty$ refers to the multi-threading model.
- **Time** $T = T_\infty$ refers to the PRAM model, assuming we have as many processors as we need (i.e. $p = \infty$).

W, S and T are all functions of n, which we can assume is a power of two (i.e. $n = 2^k$).

In both models, **parallelism** is a measurement of the maximum number of processors p that can be efficiently used.

Other Efficiency Metrics

Cost and **efficiency** can be used to compare algorithms, based on the number of processors available.

- **Cost** $C_p(n) = p \times T_p(n)$ (this may be much higher than $W(n)$).
- **Efficiency** $E_p(n) = \dfrac{W(n)}{C_p(n)}$.
- $T_p(n)$ is normally obtained experimentally.

Speed-Up

What should speed-up measure, exactly? This is tricky.

In **parallelism** we compared against T_1 to see how well the algorithm scaled beyond a single processor. For **speed-up** though, comparing to T_1 may be misleading because there **may be a sequential solution that is faster** than forcing a parallel solution to run sequentially.

In PRAM we can use T^* to denote the **best known sequential algorithm runtime**; this may be very different from the T_1 runtime. In this case, speed-up is computed as $S_p(n) = \dfrac{T^*(n)}{T_p(n)}$.

For a lot of problems, T^* is not known, so we use T_1 instead.