PAL: Parallel Algorithms

1 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 part of how I 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**.

Contents

2 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.

2.1 Runtime of Algorithms

The runtimes and memory consumption of algorithms are 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).

3 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-core CPUs and GPUs
- Specialised hardware for processes like encryption, hashing, etc.

3.1 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).

3.2 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.

We consider the following three models:

- *See more: Multi-Threading Model, page 8*.
- *See more: PRAM Model, page 12*.
- *See more: Graph Interconnected Model (Message Passing), page 27*.

3.3 Classification of Parallel Models by Communication

All of the models we consider are **data parallel** (SIMD). Models can be classified by the methods that are used by parallel processes to communicate with each other.

- **Implicit interaction** (multi-threading) - process interaction is invisible to the programmer; the compiler and/or runtime scheduler handle parallelism.
- **Shared memory** (PRAM) - parallel processes share a global address space that they can all read/write to/from asynchronously.
- **Message passing** (graph interconnected network) - parallel processes explicitly pass messages to each other.

3.4 Sequential, Parallel, Distributed?

3.4.1 Sequential vs. Parallel

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.

3.4.2 Parallel vs. Distributed

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

4 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**.

4.1 Example: Parallelised Fibonacci Calculation

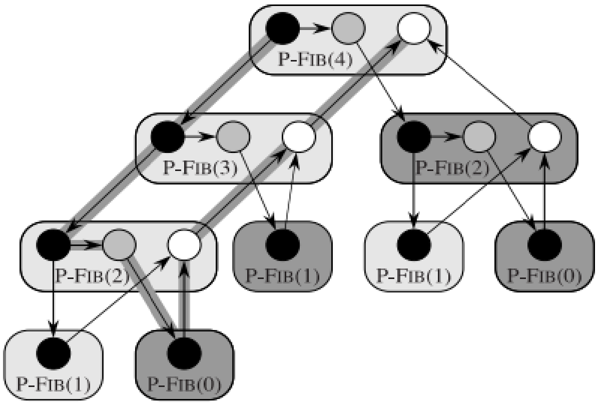
The 'dumb' method is the slow, recursive version:

```
fun recFib(n):
    if (n <= 1) return n
    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.

```
fun parFib(n):
    if (n <= 1) return n
    x = spawn parFib(n — 1)
    y = parFib(n — 2)
    sync
    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 to black edges: a spawned process
- Grey to black edges: work done within a process
- White nodes: synced results

The **critical path** is the longest path through the whole system (shaded in the diagram above) and represents the necessary amount of sequential work in the solution.

4.2 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.

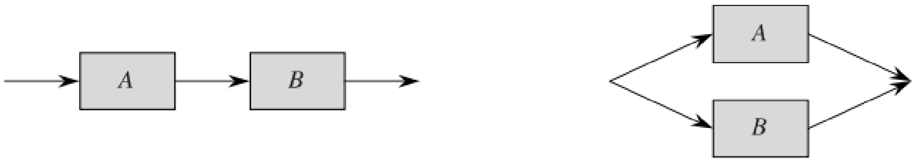
The **work** W or T_1 = the time to execute on one processor.

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 $S = max(A, B)$.

4.3 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:

```
fun recAdd(n):
    if (n <= 1) return n
    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 $n = 2^m$...

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

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

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

4.4 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).

4.5 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 and no additional overhead is incurred. Any better than this would imply 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.

4.5.1 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.**

5 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 **two options** each for an algorithm's **read/write modes**: exclusive (ER or EW) and concurrent (CR or CW). 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.

5.1 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.

5.1.1 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

5.2 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.

5.2.1 Analysis

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

5.3 Greedy Work-Time Scheduling Principle

This is the PRAM equivalent of the greedy scheduling principle defined for the multi-threading model (*see more: Greedy Scheduling Principle, page 11*).

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.

5.3.1 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.

5.3.2 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.

6 PRAM Algorithms

6.1 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, then using the parallel addition algorithm (*see more: Example: Binary Fan-In Array Sum on EREW-PRAM, page 12*) 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 $\Theta(log_2(n))$.

6.2 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, 21] + C[i, j, 21 — 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.

6.3 Binary Tree Algorithms

6.3.1 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 are 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 $2i+1$.

6.3.2 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 $\Theta(log_2(n))$.

6.3.3 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 $\Theta(log_2(n))$.

```
fun arrBroadcast—PRAM—EREW:
    Input: an n—item array A, a number X to be broadcast to the array A
    Output: A[i] = n for i = 1..n
    Note: n = 2^k; k = log_2(n)

    A[1] = X
    for i in 0 to k—1, do:
        for j in 2^i + 1 to 2^{i+1}, in parallel do:
            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.

6.3.4 Parallel Array Membership EREW

This method will use n parallel threads to compare two n-length arrays: the first is the input array and 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.

6.3.5 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 12*. 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]
```

6.3.6 Array Membership Algorithm EREW (Algorithm)

1. **`arrBroadcast—PRAM—EREW`**
   to create the temporary n-array of the searched-for number

2. **`arrMembership—PRAM—EREW`**
   to isolate the positions that contain the searched-for number

3. **`arrMin—PRAM—EREW`**
   to return the lowest position from the previous step

6.3.7 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.

6.3.8 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]
```

6.4 Array Maximum CRCW

This algorithm pairwise compares all elements and writes 1 into a secondary array `M` for all 'losing' (smaller) elements. After duplicate 0-values from `M`, the last remaining 0-value indicates the maximum element.

```
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`.

6.5 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 parallel 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`.

6.6 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]
```

6.7 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 $O(1)$, whereas EREW gives $\Theta(log_2(n))$ using binary fan-in.

- *See more: Example: Binary Fan-In Array Sum on EREW-PRAM, page 12*

6.7.1 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 17*
- 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.

7 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.

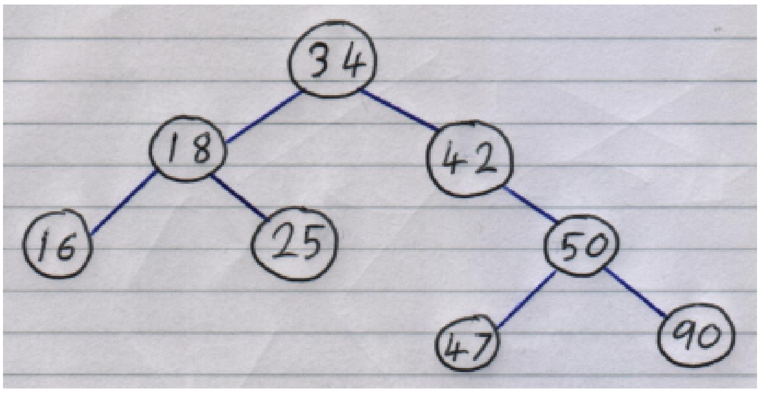
7.1 Parent Labelling

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 tree root or list head, then $P[i] = i$.

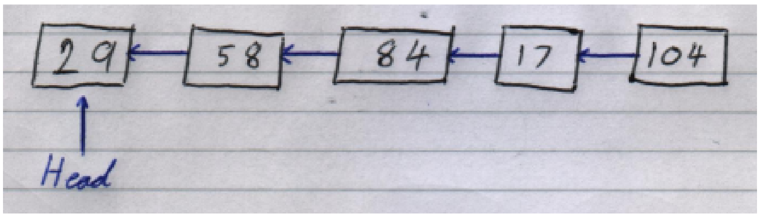
7.1.1 Example: Tree

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



7.1.2 Example: List

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



7.2 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 itself) and then repeatedly 'jumping' everything to its parent's pointer 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.

7.3 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]]
        R[i] = P[i]
```

8 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**).

8.1 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.

8.2 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.

8.3 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.

8.3.1 Speed-Up

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

For **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.

9 Graph Interconnected Model (Message Passing)

A typical message passing architecture includes many isolated units, each with their own **processor and memory**, which are connected together and allowed to communicate. We consider communication to be possible **only between adjacent nodes** (i.e. directly connected). This is because the network topology can have a significant impact as the size grows, both in terms of execution speed and cost to create.

Formally, a network can be viewed as a **graph**, where each node represents a processor with its own memory. We consider **synchronous, static (point-to-point) communication**.

9.1 Common Structures

- **Complete graph** - all nodes are connected to all other nodes.
  - Full connectivity, but very expensive to create.
- **Meshes**
  - **1D mesh** is a path or cycle.
  - **2D mesh** is a grid.
  - **3D mesh** is a cube.
  - The ends of the grid are sometimes wrapped around to the other side, creating a **torus** or **wrapped grid**.
- **Hypercubes**
  - Each node in a d-dimension hypercube stores its ID as a binary vector, $(b_1, b_2, ..., b_d)$
  - A node's neighbours are the nodes for which the ID differs by one bit
  - Example: the node $(0, 1, 0)$ has neighbours at $(\mathbf{1}, 1, 0)$, $(0, \mathbf{0}, 0)$ and $(0, 1, \mathbf{1})$.
  - 0D, 1D, 2D and 3D hypercubes are points, lines, squares and cubes; 4D and upwards get more interesting.

Symmetric layouts are preferred, as they make nodes easier to connect.

This course will consider 1D meshes as **paths**, 1D meshes as **cycles**, and 2D meshes (**grids**).

9.2 Message Passing Costs

Precise modelling of costs is impossible in practical terms, so standard approximations respect two factors for messages passed between **directly-connected processors**:

- Every message incurs some fixed start-up costs.
- The larger the message, the longer it takes.

For messages that are passed along several 'hops' there are two standard models:

- **Store-and-forward (SF)** routing (**This is the model we consider in this course**).
- **Cut-through (CT)** routing.

9.2.1 Store-and-Forward (SF) Routing

We assume that the message is **passed in its entirety between each pair** of processors on the route before continuing.

The connectivity of the interconnection network influences algorithm design and performance. We used **graph-based measures of network quality**, which are equally valid for parallel and distributed algorithms.

9.3 Graph-Based Network Quality Measures

- **Diameter** - maximum shortest distance between any pair of nodes.
  - Very important for runtime.
- **Node or Arc Connectivity** - minimum number of links that must be cut to split the network into two disjoint parts.
- **Bisection Width** - minimum number of links that must be cut to split the network into two *equal* parts.

For networks with p nodes:

Network	**Diameter**	**# Links**
Complete	1	$p(p-1)/2$
Star	2	$p-1$
Binary tree	$2log_2((p+1)/2)$	$p-1$
Linear	$p-1$	$p-1$
Ring	$\lfloor p/2 \rfloor$	p
2D mesh	$2(\sqrt{p}-1)$	$2(p-\sqrt{p})$
Mesh (wrapped)	$\lfloor \sqrt{p}/2 \rfloor$	$2p$
Hypercube	$log_2(p)$	$(plog_2(p))/2$

10 Graph Interconnected Algorithms

10.1 Notation

- $P(i)$ is the processor at position i in a 1D mesh.
- $P(r, c)$ is the processor at row r and column c in a 2D mesh.
- $P(...).Name$ is the value labelled as $Name$ in $P(...)$'s memory.

10.2 Indexing

The mesh $M(h, w)$ has $h * w$ processors. Processor indexing is **0-based**.

To use the array $A[0..n-1]$, the index k corresponds to processor $P(r, c)$ where $k = r * w + c$.

10.3 Array Minimum, 2D Mesh (Pseudocode)

```
Model: Mesh M(q, q) with n = q*q processors
Input: array X[0..n—1]
Output: min(X)

1. Compute column minimums in parallel.
     Pairwise compare rows starting at the bottom.
     Write the minimum into the higher node.
     At the end, the minimum of each column will be in the top row.

2. Compute the minimum of the top row.
     Pairwise compare nodes, starting at the end.

3. Min(X) is in the node (0, 0) (top—left).
```

10.4 Array Minimum, 2D Mesh

```
Model: Mesh M(q, q) with n = q*q processors
Input: array X[0..n−1]
Output: min(X)

for c from 0 to q−1, in parallel do:
    for r from q−2 to 0, do:
        P(r, c).Temp = P(r + 1, c).X
        X = min(X, Temp)

for c from q−2 to 0, do:
    P(0, c).Temp = P(0, c + 1).X
    X = min(X, Temp)

return P(0, 0).X
```

Running time is $\Theta(n + m)$ for an $n * m$ grid.

10.5 Broadcast, 2D Mesh

```
Model: Mesh M(q, q) with n = q*q processors
Input: P(0, 0).X with data to broadcast
Output: All processors have the data X

// propagate across the top row
for c from 0 to q−2, do:
    P(0, c + 1).X = P(0, c).X

// propagate down the columns in parallel
for c from 0 to q−1, in parallel do:
    for r from 0 to q−2, do:
        P(r + 1, c).X = P(r, c).X
```

Running time is $\Theta(n + m)$ for an $n * m$ grid.

10.6 Membership Query, 2D Mesh

```
Model: Mesh M(q, q) with n = q*q processors
Input: Array A[0..n—1] held in the mesh, value Y to search for
Output: Minimum index of processor containing Y

// share the value Y
2d—mesh—broadcast(M, Y)

// check membership and store matching indexes
for each r, c where 0 <= r, c < q, in parallel do:
    if P(r, c).L == P(r, c).Y
        P(r, c).S = r * q + c
    else
        P(r, c).S = infinity

// determine minimum index
2d—mesh—minimum(M) on each processor's S—value
```

Running time is $\Theta(n+m)$ for an $n*m$ grid.

10.7 Array Prefix, 2D Mesh

```
Model: Mesh M(q, q) with n = q*q processors
Input: Array X[0..n—1] held in the mesh
Output:

// initialise rows in parallel
for r from 0 to q—1, in parallel do:
    P(r, 0).S = P(r, 0).X
    for c from 1 to q—1, do:
        P(r, c).S = P(r, c — 1).S + P(r, c).X

// add up last column
for r from 1 to q—1, do:
    P(r, q — 1).S = P(r, q — 1).S + P(r — 1, q — 1).S

// back—factor last column across next row in parallel
for r from 1 to q — 1, in parallel do:
    for c from 0 to q — 2, do:
        P(r, c).S = P(r, c).S + P(r — 1, q — 1).S
```

Running time is $\Theta(n+m)$ for an $n*m$ grid.

10.8 Matrix Multiplication, 1D Path, Pipelining

The columns of a matrix A are fed synchronously into the processor of the path in a skewed fashion, as shown below. At each step, the value held by processor $P(i)$ is the value held by $P(i-1)$ in the previous step, plus the i^{th} item from the current input matrix row multiplied by the i^{th} item in the vector. Example:

Matrix:	1	2	3	4	Vector:	3
	0	2	2	0		3
	5	3	6	4		4
	1	1	1	0		2

This is computed as follows:

				4	Step 7
			3	0	Step 6
		2	2	4	Step 5
Matrix:	1	2	6	0	Step 4
	0	3	1		Step 3
	5	1			Step 2
	1				Step 1
Vector:	3	3	4	2	
Step 1:	3	0	0	0	
Step 2:	15	3 + 3	0	0	
=	15	6	0	0	
Step 3:	0	15 + 9	6 + 4	0	
=	0	24	10	0	
Step 4:	3	0 + 6	24 + 24	10 + 0	
=	3	6	48	**10**	
Step 5:	0	3 + 6	6 + 8	48 + 8	
=	0	9	14	**56**	
Step 6:	0	0	9 + 12	14 + 0	
=	0	24	21	**14**	
Step 7:	0	0	0	21 + 8	
=	0	0	0	**29**	

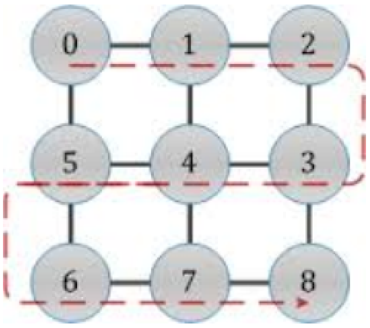
10.9 Even/Odd Transposition Sort, 1D Mesh

```
Model: Mesh M(n) with n processors
Input: Array X[0..n—1] held in the mesh
Output: Sorted array X[0..n—1] held in the mesh

for s from 0 to n — 1, do:
    if s is even
        for all even processor labels i, in parallel do:
            compareExchange(P(i).X, P(i + 1).X)
    else
        for all odd processor labels i, in parallel do:
            compareExchange(P(i).X, P(i + 1).X)
```

Coloured pairs show the **result** of `compareExchange` operations:

Processor	P_0	P_1	P_2	P_3	P_4	P_5	P_6
Input	4	2	8	5	1	3	0
Step 0 (E)	2	4	5	8	1	3	0
1 (O)	2	4	5	1	8	0	3
2 (E)	2	4	1	5	0	8	3
3 (O)	2	1	4	0	5	3	8
4 (E)	1	2	0	4	3	5	8
5 (O)	1	0	2	3	4	5	8
6 (E)	0	1	2	3	4	5	8
Output	0	1	2	3	4	5	8

10.10 Snake Order Sort, 2D Mesh

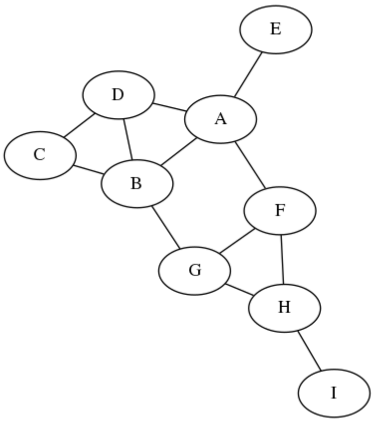


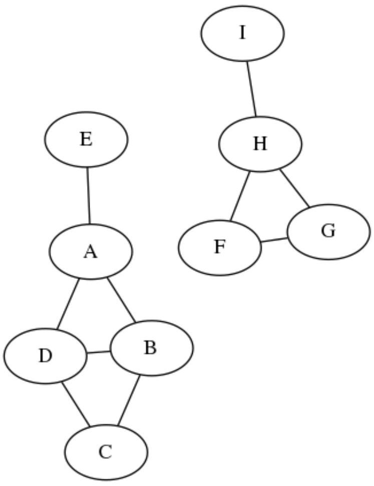
```
Model: Mesh M(q, q) with n = q * q processors
Input: Array X[0..n-1] held in the mesh
Output: Snake-order sorted array X[0..n-1] held in the mesh

for s from 0 to ceil(log_2(n)), do:
    if s is even
        in parallel, sort all rows with 1d-mesh-sort
    else
        in parallel, sort all columns with 1d-mesh-sort
```

11 Parallel Graph Algorithms

11.1 Graph Components

A graph is **connected** if there is a path from every node to every other node. A **connected component** of a graph is a subset of its nodes that are **connected within the set**, but not within the rest of the graph. For example, the graph on the left is connected; the graph on the right has two connected components:





11.1.1 Sequential Solution

This solution uses breadth-first search to find all of the neighbours of each given node:

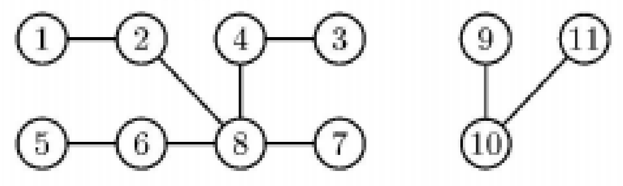
```
Input:  list of vertices V, with neighbours
Output: list of sets S of vertices, each being a component

while (V is not empty):
    // select a node, create a set for it, and remove it from V
    c = lowest—index node in V
    S[c] = { c }
    V.remove(c)

    // find neighbours with BFS
    until (no more nodes are discovered):
        fan out from c using BFS
        add discovered nodes to S[c]
        delete discovered nodes from V

return S
```

The runtime of this algorithm is $\Theta(|Vertices|+|Edges|)$, because BFS will inspect every vertex and every edge once. In a very densely connected graph, the high number of edges can push this towards $O(|Vertices|^2)$.

11.1.2 Parallel Solution (Approach)

How do we test connectivity in parallel? We can do it by assigning **parents** (*see more: Parent Labelling, page 23*). We assume that every node knows its neighbours and their labels, so we instruct each node to **point to it's lowest-label neighbour**, $C(v)$.

Given this graph:

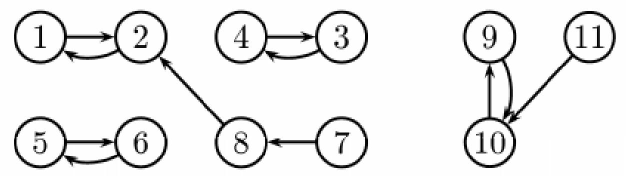


We declare a parent relationship as:

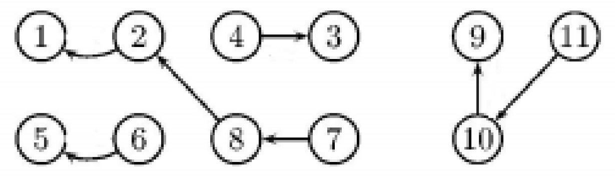
$$C(v) = min\{u \in N(v)\}$$

v	1	2	3	4	5	6	7	8	9	10	11
$C(v)$	2	1	4	3	6	5	8	2	10	9	10

Which yields this:



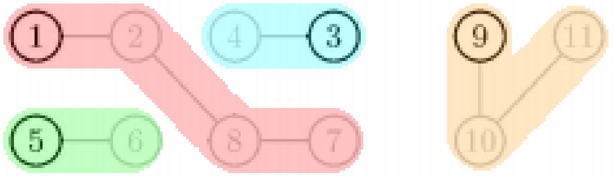
To remove the loops in the graph, we additionally declare that **if a node's parent has a higher label than it, it should become a root** (i.e. point to itself). This yields the following:



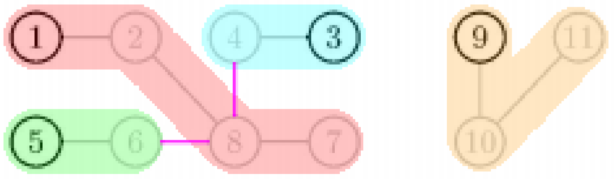
We can now apply the **roots of the forest** algorithm (*see more: Forest Root Finding, page 24*) to yield the following:

v	1	2	3	4	5	6	7	8	9	10	11
$Root(v)$	1	1	3	3	5	5	1	1	9	9	9

Each rooted tree in the forest is now represented by its root, creating **super-vertices labelled by their root**:



By restoring any edges that were 'lost' in the first round, we can then **repeat the process and consider only the super-vertices**:



$C(v) = min\{u \in N(v)\}$ (or a self-loop if lowest)

v	1	3	5	9
$C(v)$	1	1	1	9

The process is **repeated until a cycle makes no changes**, after which the components have been found, and every node is labelled with the lowest-index label within it's component.

11.1.3 Parallel Solution (Pseudo-pseudocode)

Note: a node's 's-vertex' is it's **super-vertex** label (eventually the label of it's component, which is the lowest-label within the component).

```
Input: Graph structure (vertices V and edges E)
Output: List of vertices in each component
Notes: S(v) = v's s—vertex
       N(v) = graph neighbours of vertex v

for each v in V, in parallel do:
    S(v) = v

repeat log_2(|V|) times:
    for each v in V, in parallel do:
        point edge from S(v) to its lowest—labelled s—vertex in N(v),
        or from S(v) to S(v) if s—vertex label is smaller

        find root vertex of S(v)
        S(v) = Root(S(v))
```

11.1.4 Parallel Solution (Analysis)

Time $T_\infty(n)$

- Line 9 repeats the `parallel for` statement $log_2(n)$ times sequentially.
- Each iteration of line 9 reduces the number of s-vertices for each final component by at least half.
- Line 11-12 can use binary fan-in on the v^{th} row of the adjacency matrix, which is a $\Theta(log_2(n))$ procedure.
- Line 14 uses parallel forest root finding for vertex v by pointer jumping, which is an $O(log_2(n))$ procedure.
- Therefore, time $T_\infty(n) = \Theta(log_2(n)^2)$.

Work $T_1(n)$

- Line 10 has work $\Theta(n * log_2(n))$ for each of n vertices.
  - This is because line 11 uses n processors for $log_2(n)$ sequential rounds of work.
- Line 9 repeats this sequentially $log_2(n)$ times.
- Therefore, work $T_1(n) = \Theta(n^2 * log_2(n)^2)$.

11.1.5 Parallel Solution (Pseudocode)

```
Input: Graph structure (vertices V and edges E)
Output: List of vertices in each component
Notes: S(v) = v's s—vertex
       N(v) = graph neighbours of vertex v

for each v in V, in parallel do:
    S(v) = v

repeat log_2(|V|) times:
    // pick lowest labelled s—vertex neighbour, or self
    for each v in V, in parallel do:
        T(v) = min_j{ S(j) in N(v), S(v) != S(j) }
        if no such j, T(v) = S(v)

    // point all parts of super—vertex to lowest—label
    for each v in V, in parallel do:
        T(v) = min_j{ T(j) where s(j) = v, T(j) != v }
        if no such j, T(v) = S(v)

    // temporarily save what we have so far
    for each v in V, in parallel do:
        B(v) = T(v)

    // find forest roots
    repeat log_2(|V|) times:
        for each v in V, in parallel do:
            T(v) = T(T(v))

    // keep the minimum of the new root and the previously saved
    for each v in V, in parallel do:
        S(v) = min{ T(v), B(T(v)) }
```

11.2 Minimum-Weight Spanning Trees

The basic approach to forming a minimum-weight spanning tree (MST) of a connected graph is to remove all edges and then add the **cheapest edge between two components** until a tree is formed. This has $|V| - 1$ steps, because every step 'absorbs' one more vertex into a component; this is also evident because each step adds one edge, and a tree must have $|V| - 1$ edges.

Formally:

- A graph $G = (V, E)$ with positive edge weights.
- The minimum weight edge between set U and $U - V$ is part of the MST.

- For all $u \in V$, the edge $\mu(u)$ of minimum weight incident with vertex u is in the MST.

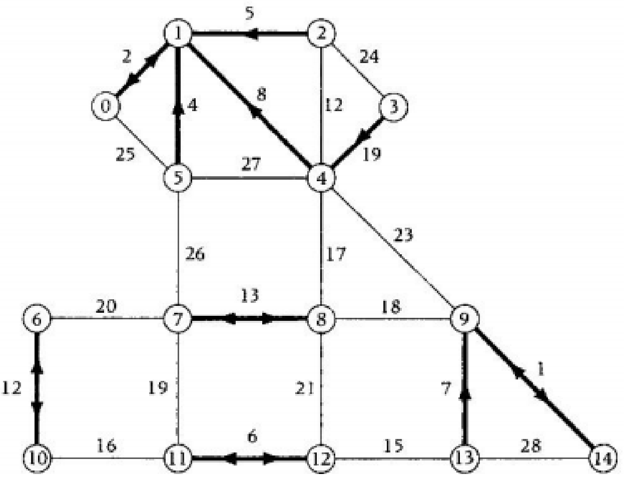
The parallel algorithm requires **distinct edge weights**, but the sequential algorithm does not.

11.2.1 Parallel Solution (Boruvka's Algorithm)

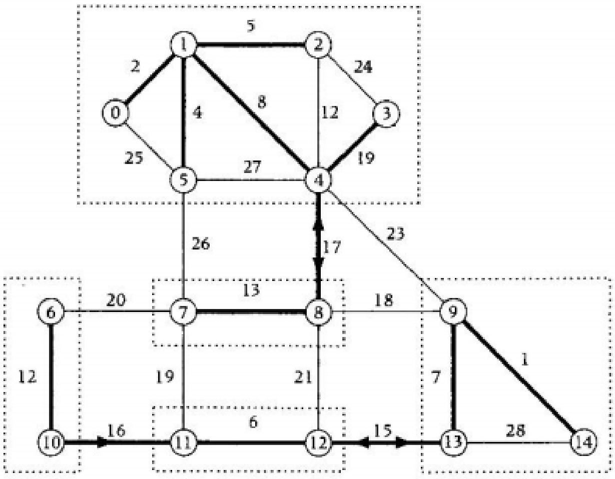
Input: connected graph with distinct positive edge weights.

- While there are edges remaining:
  - Select the minimum weight edge out of each super-vertex.
  - Contract each connected component defined by these edges into a super-vertex.
  - Remove self-loops.
  - If there are multiple edges between components, keep the edge with the minimum weight (this is where the uniqueness of edges is important).
  - Add all selected edges to the MST.

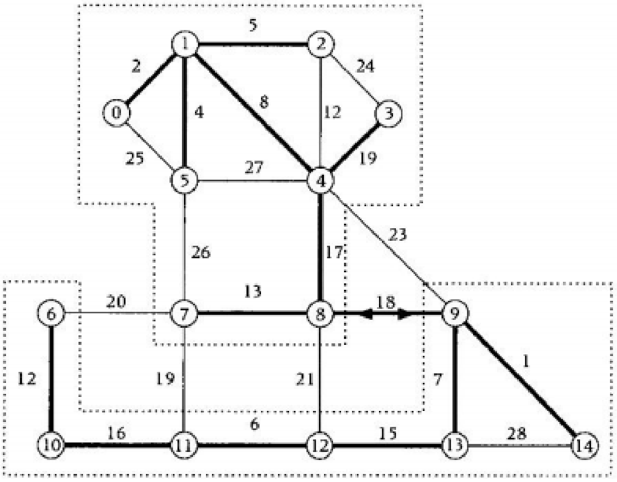
In the following example, the first iteration has already been done: the bold edges indicate those that were selected between the super-vertices (initially, every vertex is its own super-vertex).



These edges form five super-vertices, and the following edges between them are now selected:



With two super-vertices remaining, a final edge is selected to complete the MST:



12 Divide and Conquer (D+C) Algorithms

D+C is an algorithm design paradigm based on **multi-branch recursion**. It works by recursively breaking a problem down into two or more **disjoint sub-problems** of the same type, until they become trivial enough to be solved directly (i.e. base cases). The general strategy is as follows:

- Divide the problem into roughly equal-sized sub-problems.
- Recursively solve sub-problems in the same manner.
- Combine the solutions to sub-problems into a single answer and pass it back to the recursion call.

It must be possible to **perform steps 1 and 3 efficiently** for the D+C approach to work effectively. **Sub-problems must be independent** for step 2 to work. Sorting arrays or adding their contents are two examples of problems that are suitable for this pattern.

D+C can be implemented in the multi-threading model using **`spawn`** and **`sync`** operators. It is also possible in the PRAM model if we have an operator `in parallel, do...`.

12.1 D+C General Statement

```
Input: I (an input to the given problem)
Output: J (a solution for the problem input I)

DivideAndConquer(I):
    if (i is a base case):
        return J // solution for I
    else:
        divide I into M >= 2 sub—problems
        for k from 1 to m, do:
            sol[k] = DivideAndConquer(sub[k])
        combine sol[1..m] into solution J
        return J
```

12.2 Parallel In-Order Tree Traversal (Multi-Threading Model)

```
Input: some tree node T
Output: in—order traversal of nodes

fun ParaInOrder(T):
    if (T == null):
        return
    else:
        l = spawn ParaInOrder(T.leftChild)
        m = T.data
        r = ParaInOrder(T.rightChild)
        sync
        return [l, m, r]
```

12.3 Parallel Loops (Multi-Threading Model)

The idea here is to create a parallel `for i from 1 to n, do S(i)` system. We can use D+C to execute the first and last half of each range in parallel:

```
Input: start of range, i
       end of range, j
       statement to execute, S

fun ParallelFor(i, j, S):
    if (i == j):
        S(i) // base case
    else:
        m = floor((i + j) / 2)
        spawn ParallelFor(i, m, S)
        ParallelFor(m + 1, j, S)
        sync
```

Assuming the range n is some 2^k, the span of this algorithm is $\Theta(log_2(n))$.

12.4 Horner's Method

Very old, very efficient method algorithm for **calculating polynomials**. It can be used for evaluating polynomials (e.g. 'what is $y = 3x^3 + 7x^8$ when $x = 5$?'), finding roots of polynomials (e.g. 'when is $x^7 + 2x^4 + 1 = 0$?') and converting number bases, amongst other things.

12.4.1 Representing Polynomials

The polynomial $y = 1 + x^2 + 4x^3 + 7x^7$ can be efficiently represented in an array, as follows:

```
Y = [1, 0, 1, 4, 0, 0, 0, 7]
```

In this array, the value at `Y[i]` corresponds to the coefficient of x^i. For example, $4x^3$ is stored as the value 4 at `Y[3]`.

12.4.2 The Method (Sequential)

So how do we compute the value of $Y(x)$ at $x = 42$, for example?

```
Input: polynomial stored as an array, Y
       highest index of x, n
       value for x, x

fun SequentialHorner(Y, n, x):
    y = 0  // subtotal
    i = n  // descending index
    while (i >= 0), do:
        y = Y[i] + (x * y)
        i = i − 1
    return y
```

This method is $\Theta(n)$ for an array of size n. This cannot be improved on (i.e. $T^*(n) = \Theta(n)$) because we need to inspect $n + 1$ entries to evaluate the value of x at all indexes $0..n$.

12.4.3 The Method (Parallel)

Most D+C methods work by splitting something in the middle (at a median data point). We can split the array (and hence the polynomial) at a median $s = n/2$.

If x^s is factored from `A[s..t]` then the resulting polynomial is

$$Q(x) = A[s] + A[s+1]x + A[s+2]x^2 + ... + A[t]x^{t-s}$$

Array indexes and coefficient powers get out of step by s, so an additional multiplication is required when combining results:

$$Y(x) = A[0] + A[1]x + A[2]x^2 + ... + A[s-1]x^{s-1} + Q(x)x^s$$

```
Input: polynomial stored as an array, Y
       start index, p
       highest index of x, r
       value for x, x

fun ParallelHorner(Y, p, r, x):
    if (p == r):
        return A[p]  // base case
    else:
        q = floor((p + r) / 2)
        in parallel, do:
            L = ParallelHorner(A, p, q, x)
            M = ParallelHorner(A, q + 1, r, x)
        return L + (M * x^(q — p + 1))

Note: first call should have p = 0
```

We pass in the array indexes `p` and `r` to the method to compute the polynomial between those points in the array. The return statement shifts the 'top half' of the sub-problem result upwards by a factor of x^{q-p+1}.

This method runs in $O(log_2(n)^2)$ because the recursion depth of the method is $O(log_2(n))$ and computing x^n also takes $O(log_2(n))$ (as shown below).

12.4.4 Side Note: Parallel Powers

```
Input: number to raise to power, x
       power, p

fun ParallelPower(x, j):
    if (j == 1):
        return x
    else:
        k = floor(j / 2)
        in parallel, do:
            a = ParallelPower(x, k)
            b = ParallelPower(x, j — k)
        return a * b
```

12.5 Merge Sort

12.5.1 Sequential Merge Sort

```
Input: array, A
       start index, s
       end index, e

fun SeqMergeSort(A, s, e):
    if (s == e):
        do nothing
    else if (s < e):
        m = floor((s + e) / 2)
        SeqMergeSort(A, s, m)
        SeqMergeSort(A, m + 1, e)
        SeqMerge(A, s, m, e)
```

We assume that the function `SeqMerge(A, s, m, e)` merges the sorted sub-arrays in `A[s..m]` and `A[m+1..e]` into `A[s..e]`.

The runtime of this method is $\Theta(n * log_2(n))$, because the depth of an n-item tree is $log_2(n)$, and at each level up to n items may have to be sorted.

12.5.2 Slow Parallel Merge Sort

```
Input: array, A
       start index, s
       end index, e

fun ParaMergeSort(A, s, e):
    if (s == e):
        do nothing
    else if (s < e):
        m = floor((s + e) / 2)
        spawn ParaMergeSort(A, s, m)
        ParaMergeSort(A, m + 1, e)
        sync
        SeqMerge(A, s, m, e)
```

The span of this method is $\Theta(n)$ and the speed-up is $O(log_2(n))$. The span is high and the speed-up is poor because of the $\Theta(n)$ operation in sequential merging.

12.5.3 Side Note: Recursive Binary Search

This D+C search algorithm operates on a sorted array, but never explores more than one sub-problem.

```
Input: element to find, x
       sorted array, A
       start index, s
       end index, e

fun BinarySearch(x, A, s, e):
    if (s == e):  // end of search
        if (A[s] == x):
            return s
        else:
            return FAIL
    else:
        m = floor((s + e) / 2)  // mid point
        if (x <= A[m]):
            return BinarySearch(x, A, s, q)  // search bottom half
        else:
            return BinarySearch(x, A, q + 1, r)  // search top half
```

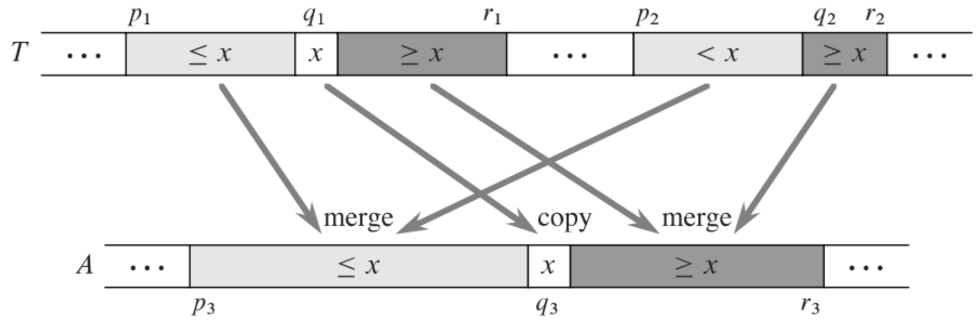
The search space is halved with each iteration, so the runtime for this approach is $\Theta(log_2(n))$ for an n-item array.

12.5.4 Fast Parallel Merge

The sequential merge stage is the bottleneck of the parallel merge algorithm. The sequential merge can be replaced with a faster parallel version that works as follows:

- Take two sorted arrays, A and B.
- Pick the median x of A.
- Split A around x giving $A(\leq x)$, x, $A(\geq x)$.
- Use binary search to find the first occurrence of x in B.
- Split B below x, giving $B(< x)$ and $B(\geq x)$.
- Merge the parts $A(\leq x)$, $B(< x)$ and $A(\geq x)$, $B(\geq x)$ around x.

This process is shown in the following diagram:



Informally, the algorithm can be written as follows:

```
Input: sorted array A[1..n]
       sorted array B[1..m]

fun ParaMerge(A, B):
    q = floor((n + 1) / 2)  // median of A
    x = A[q]
    split A into A[1..q—1], x, A[q+1..n]
    using binary search:
        split B into B[< x] and B[>= x]
    return ParaMerge(A[1..q—1], B[< x]), x,
               ParaMerge(A[q+1..n], B[>= x])
```

12.5.5 Fast Parallel Merge Sort

Now, our merge sort algorithm uses the parallel merge method:

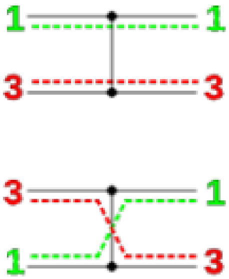
```
Input: array, A
       start index, s
       end index, e

fun ParaMergeSort(A, s, e):
    if (s == e):
        do nothing
    else if (s < e):
        m = floor((s + e) / 2)
        spawn ParaMergeSort(A, s, m)
        ParaMergeSort(A, m + 1, e)
        sync
        ParaMerge(A[s..m], A[m+1..e])
```

This has a span of $\Theta(log_2(n)^3)$: `ParaMergeSort` has $\Theta(log_2(n))$ levels of recursion; at each level, parallel merge has a recursion distance of $\Theta(log_2(n))$, each of which uses a $\Theta(log_2(n))$ binary search.

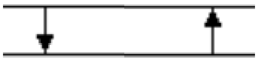
13 Sorting Networks

A sorting network is a special kind of sorting algorithm, where the sequence of operations is not data-dependent. This makes it useful for implementation in hardware or parallel processor arrays.

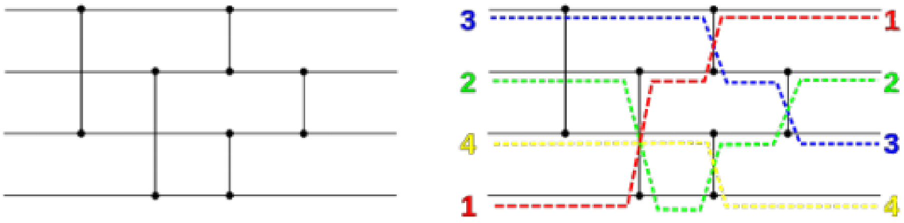
Sorting networks are composed of **wires** and **comparators**, with wires running from left to right and comparators connecting two wires. A set of values are passed through the network, **one value per wire**, all at the **same time**. When a comparator is encountered, the values on the two wires are **swapped** if the top wire's value is greater than the bottom wire's value.



Diagrams typically assume an **ascending comparator** (smaller values move upwards) as shown on the left, but sometimes use a **descending comparator** as shown on the right:



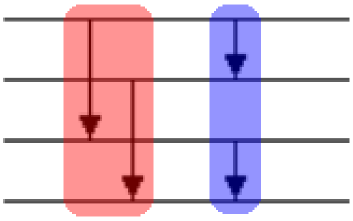
The following network would correctly sort any 4 input values. The first 4 comparators sink larger values to the bottom and float smaller values to the top, and the final comparator sorts the two middle wires:



13.1 Parallel Sorting: Comparator Stages

We label n wires with $0, 1, ..., n-1$. A comparator $[i : j]$ sorts the i^{th} and j^{th} elements of the input into non-decreasing order by **swapping** them if necessary.

A **comparator stage** is a sequential set of comparators $S = [i_1 : j_1], ..., [i_k : j_k]$ such that all i and j values are distinct (i.e. all start and end wires are different). For example, the following network has two stages:



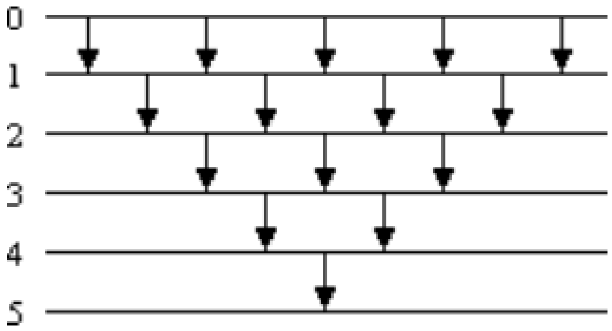
All comparators within a stage are **independent of each other** and can be executed **in parallel**.

13.2 Comparator Networks vs. Sorting Networks

A **comparator network** is a composition of comparator stages.

A **sorting network** is a comparator network that accurately sorts any possible input sequence.

13.3 Bubble Sort



The bubble sort network has $n-1$ **diagonal rows** of comparators of decreasing length. The first diagonal row moves the greatest element to the last wire, the second diagonal row to the second-from-last wire, etc.

This approach uses $(n(n-1))/2$ comparators, arranged in $2n-3$ stages. There are $n-1$ diagonals.

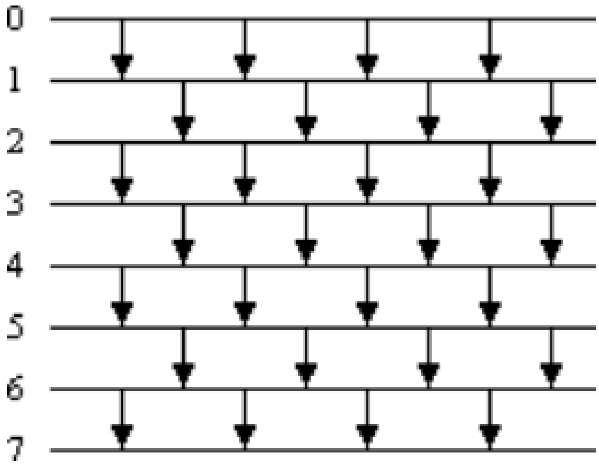
This can be coded as follows:

```
Input: array, A

fun BubbleSort(A):
    for i from 1 to n−1, do:
        for j from 1 to n−1, do:
            if (A[j] > A[j+1]):
                swap A[j], A[j+1]
```

Note: the algorithm above uses n^2 ‘comparators’ to keep the code simple.

13.4 Even/Odd Transposition Sort

Similar to the approach seen earlier (*see more: Even/Odd Transposition Sort, 1D Mesh, page 33*), this network sorts an input by alternately checking and swapping elements at even and odd indexes with their neighbours. This can be achieved with n parallel stages:



13.5 The 0-1 Principle

The validity of sorting networks can be **proved** without checking all possible input permutations by using the 0-1 principle, which is based on the fact that the validity of a sorting network depends on its **structure** and not the input values.

The 0-1 Principle: a comparator network with n inputs is valid if it correctly sorts all 2^n sequences of zeros and ones.

This is good news, as it vastly reduces the number of checks that must be made, and can inform some decisions when designing the network.

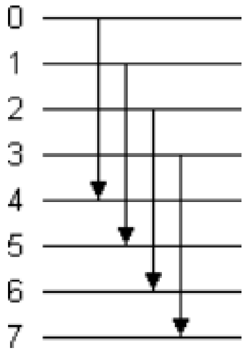
13.6 Bitonic Sort

Bitonic sort is one of the fastest sorting networks, which requires $n(log_2(n)^2)$ comparators and sorts in $log_2(n)^2$ parallel time (span). The term **bitonic** means that the data is **two-fold monotonic** (i.e. the data can have two maxima and one minima, or two minima and one maxima; the values either go down then up, or up then down).

A 0-1 sequence is bitonic if it has **at most two changes** between contiguous zeros and ones. For example, 001100, 000111 and 110011 are valid, but 0101 is not.

13.6.1 The B_n Network (Half Cleaner)

The B_n network compares and swaps the first $n/2$ values with their counterparts in the second half of the array, as in the following diagram:

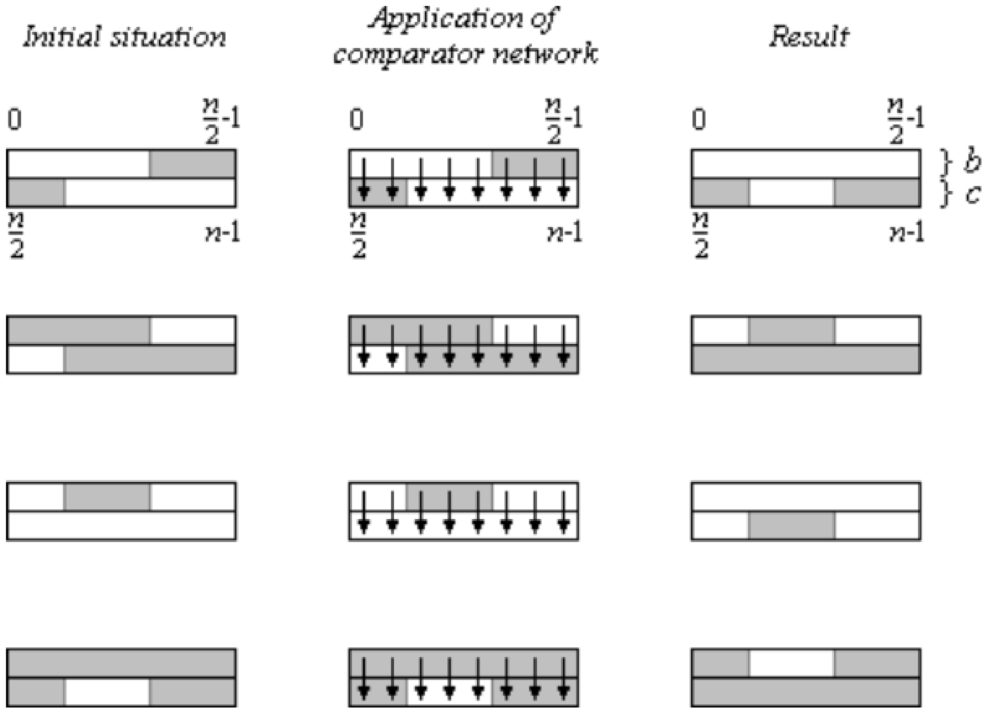


The bitonic network B_2 is a single, normal comparator.

Formally, $B_n = [0 : n/2], [1 : n/2 + 1], ..., [n/2 - 1 : n - 1]$.

After applying this network to a **bitonic** sequence, we know that everything in the **first half is less than or equal to the second half**, and that **both halves are bitonic**. We also know that at least **one half is fully sorted**, and that we can sort further by applying $B_{n/2}$ to both remaining bitonic halves.

The following diagram shows the application of the B_n network to a bitonic input, skipping the trivial all-zero and all-one cases. Zeros are drawn as white and ones are drawn as grey; the top and bottom halves of each block represent the first and last half of the input respectively.



The B_n network is sometimes called a **half-cleaner** because after it is applied, half of the sequence is clean.

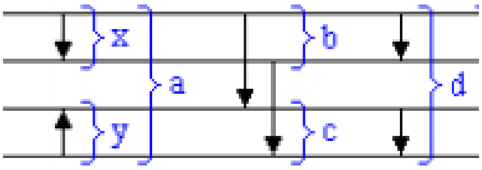
13.6.2 Bitonic Build

Before half cleaners are useful for fast sorting, a **bitonic build** phase is needed to get the input into bitonic order. The **bitonic build** phase can also be built with half cleaners in various configurations.

The following notation is used:

- $\oplus B_2$ is the B_2 network 'pointing down' (i.e. as normal).
- $\ominus B_2$ is the B_2 network 'pointing up' (i.e. upside-down).

The following diagram shown a bitonic sort network for $n = 4$:



This can also be represented as follows:

$\oplus B_2$ $\ominus B_2$	$\oplus B_4$	$\oplus B_2$ $\oplus B_2$
Build	Merge	

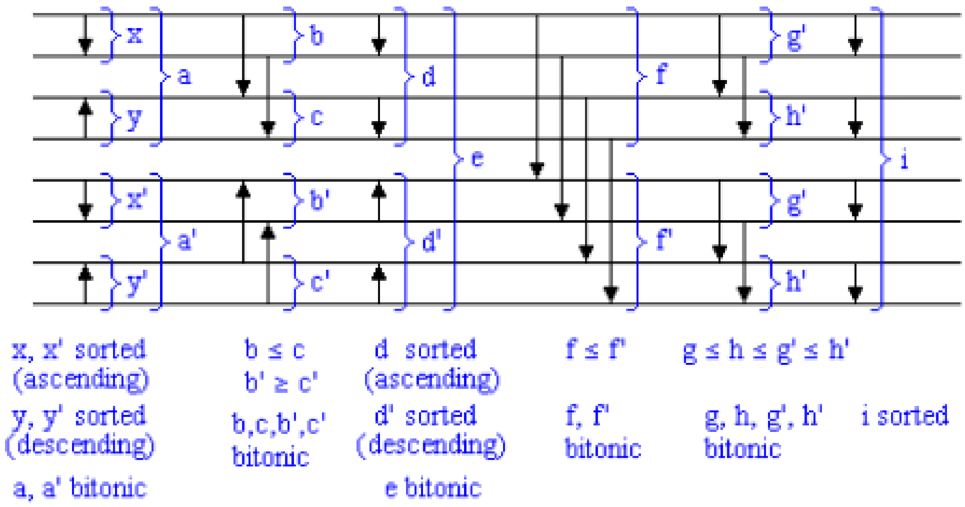
Giving the following aliases to different sections allows for a more concise notations:

$BM[2] =$	B_2

$BM[4] =$	B_4	B_2 B_2

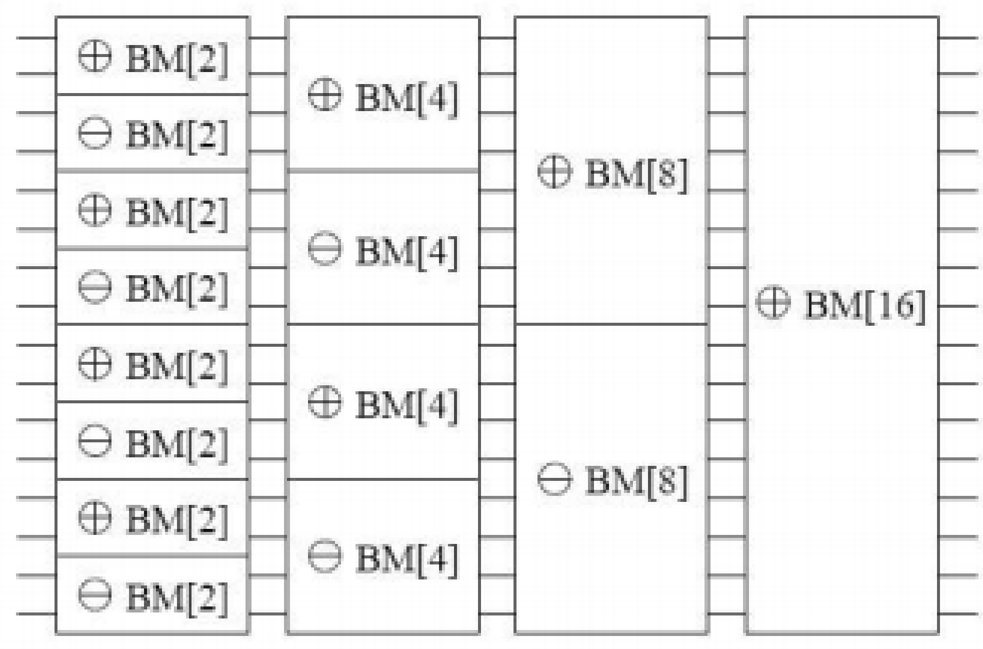
$\oplus BM[2]$ $\ominus BM[2]$	$\oplus BM[4]$
Build	Merge

The following diagram shows a bitonic sort network for $n = 8$, where the first 3 stages make up the bitonic build stage and the last 3 stages make up the bitonic sort (also called the bitonic merge) stage.



$\oplus BM[2]$	$\oplus BM[4]$	$\oplus BM[8]$
$\ominus BM[2]$		
$\oplus BM[2]$	$\ominus BM[4]$	
$\ominus BM[2]$		
Build		Merge

A bitonic sort for $n = 16$ would look like this:



13.6.3 Conclusion

Bitonic sort with n values is sequentially formed from alternating $\oplus$ and $\ominus$ variants of $BM[2]$, $BM[4]$, $BM[8]$, ..., $BM[n]$.

$BM[n]$ has $log_2(n)$ comparator stages.

The number of comparator stages $T(n)$ in the entire network is found as follows:

$$\begin{aligned} T(n) &= log_2(n) + log_2(n) - 1 + log_2(n) - 2 + ... + 1 \\ &= log_2(n)(log_2(n) + 1)/2 \end{aligned}$$

Each stage consists of $n/2$ comparators, giving $\Theta(n(log_2(n)^2))$ comparators.

14 Distributed Networks

14.1 Distributed Network Model

- Networks are modelled as **graphs**.
- Each vertex v has a unique label v, plus its own processor, memory and copy of all programs.
- $v : X$ is the value of X in v's local memory.
- Each vertex v knows its neighbours, $N(v)$.
- A vertex can pass messages to its neighbours, and all vertices cooperate in program running and message passing.
- Some networks have directed edges.
- The vertices know nothing about the global structure, outside of their neighbourhood.

We assume that all vertices are operating synchronously and that the current time step (or some global clock T) is known to all vertices.

We deal with two forms of complexity:

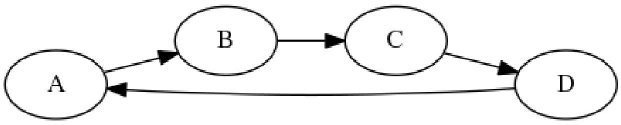
- **Time complexity**: worst case number of steps of execution of program at any vertex until successful completion.
- **Communication complexity**: total number of messages passed (usually the more important metric).

We consider the 'local program' view of a vertex.

14.2 Leader Election Algorithm (Directed Ring Network)

Leaders can be used for some level of coordination, or as a 'contact point' for the outside world. Leader election algorithms are needed so that every vertex can decide whether it is a leader or not, with only one leader being elected.

In this example, we consider a network modelled as a directed ring:



```
algo ElectRingLeader // viewed at any given vertex, v
    status = "unknown"
    send my label v to out—neighbour
    while not received "leader" message, do:
        on receipt of message from in—neighbour:
            if (message is a vertex label u):
                if (u > v):
                    send u to out—neighbour
                if (v > u):
                    send v to out—neighbour
                if (u == v):
                    status = "leader"
                    send "leader" message to out—neighbour
            if (status == "unknown" and message == "leader"):
                status = "not leader"
                send "leader" to out—neighbour
```

This algorithm elects the leader with the greatest-value label. It does this by passing labels around the ring and discarding smaller labels until a vertex receives its own label (and elects itself as leader, because no other node on the ring had a greater value). At this point it sends a `leader` message once around the ring, so that all other nodes can move from `unknown` to **`not`** `leader`.

This algorithm runs in $O(n)$, requiring at most 3 loops around the ring.

14.3 Leader Election Algorithm (General Network)

In this example, we consider any generic undirected network. Each vertex knows the network diameter[1], d.

```
algo ElectLeader // viewed at any given vertex, v
    v:maxId = v
    for round from 1 to d, do:
        send v:maxId to all neighbours u in N(v)
        for each u:maxId received from neighbour u, do:
            if (v:maxId < u:maxId):
                v:maxId = u:maxId

    if (v:maxId == v):
        status = "leader"
    else:
        status = "not leader"
```

The loop is executed d times, because that is the longest time it would take a message to pass from one node to another.

[1] The graph diameter is the longest of the shortest paths between all pairs of vertices.

14.4 Breadth First Search Tree (General Network)

This algorithm starts from a given vertex, which could be the leader. By the end of this, every vertex will know its parent.

```
algo InitBfsTree  // viewed at start node, s
    // nothing is in the tree to start with
    for each node, node:status = "not in tree"

    // s is its own parent
    s:parent(s) = s
    s:status = "in tree"

    send "join tree" to all neighbours in N(s)
```

```
algo DistributedBfsTree  // viewed at any given vertex, v
    if (v:status == "not in tree" and v receives "join tree"):
        amongst neighbours who sent "join tree",
        select the vertex u with the lowest—value label

        v:parent = u
        send "your child" to u
        send "not your child" to (N(v) — u)

        send "join tree" to all neighbours in N(v)

    when v has received child—status messages from all (N(v) — u):
        if (v has no children):
            send "terminated" to v:parent
        else:
            when v has received "terminated" from all children:
                send "terminated" to v:parent
```

This approach can be used for many things:

- Testing connectivity.
- Finding single-source shortest paths.
- Finding upper-bound network diameter.
- Efficient broadcasting.
    - Send messages to children, rather than ‘flooding’ the network by sending to all neighbours..
- Routing pathways.