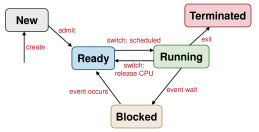
1 Processes, Threads, and Synchronisation

1.1 Processes & Threads: abstractions of flow of control

1.1.1 Processes

- Identified by PID, Own address space, exclusive access to its data
- Requires explicit communication for inter-process info exchange
- Context switching between processes, must save state, overhead



- 2 types of exec: time slicing (pseudo-parallelism), parallel execution of processes on different resources
- Create new process: `fork` or `exec(char *prog, char *argv)`

Disadvantages

- Creating a new process is costly (syscall overhead, all data structures must be allocated, initialised, copied)
- IPC is costly (must go through the OS)

1.1.2 Threads

- Share the address space of the process – all threads belonging to the same process see the same value -> **shared-memory arch**
- Thread generation is faster than process generation
- Diff threads can be assigned to run on diff cores of a multicore
- 2 types:
  - **User-level threads**: managed by a thread lib – OS unaware of user-level threads. **Advantages:** switching thread context is fast, **Disadvantages:** OS cannot map diff threads of same process to diff exec resources -> no parallelism, OS cannot switch to another thread is 1 thread execs a blocking I/O op
  - **Kernel threads**: OS is aware of existence.
  - **Mapping User to Kernel Threads**
- Global vars and all dynamically allocated data objects can be accessed by any thread
- Each thread has a private stack for function stack frames, existing iff the thread is active.
- No of threads should be suitable to parallelism degree of application, available execution resources, not too large to keep the overhead for thread creation, management, termination small

1.2 Synchronisation

- **Race condition** = 2 concurrent threads accessed a shared resource without any access synchronisation
- Use **mutual exclusion** to sync access to shared resources. Code sequence that uses mutex is called **critical section** (c.s.)
- Requirements:
  1. **Mutual exclusion (mutex)**: if one thread is in the critical section, then no other is
  2. **Progress**: If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section. A thread in the critical section will eventually leave it
  3. **Bounded waiting (no starvation)**: If some thread T is waiting on the critical section, then T will eventually enter the critical section
  4. **Performance**: The overhead of entering & exiting the critical section is small w.r.t. the work being done within it.
- **Starvation**: a process is prevented from making progressing because some other process has the resource it requires.
- **Deadlock** among a set of processes: if every process is waiting for an event that can be caused only by another process in the set. can arise when processes compete for access to limited resources, incorrectly synced.
- Condition for deadlock: if and only if these 4 conditions hold simultaneously:
  1. **Mutex**: at least 1 resource must be held in non-sharable mode
  2. **Hold and wait**: There must be 1 process hilding 1 resource and waiting for another resource
  3. **No preemption** – Resources cannot be preempted (critical sections cannot be aborted externally)
  4. **Circular wait** – There must exist a set of processes $p_1, p_2, ...p_n$ such that p_1 is waiting for p_2, p_2 for p_3, so on.
- 4 Mechanisms:

1. **Locks**
   - 2 operations: `acquire()` to enter c.s., `release()` to leave c.s.
   - Pair calls to acquire and release, can spin (**spinlock**) or block (**mutex**)
2. **Semaphores**
   - Abtract data type that provide mutex through atomic counters
   - "Integers" that support 2 operations: `wait()` – decrement, block until semaphore is open. `signal()`, increment, allow another thread to enter
   - Safety property: semaphore value ≥ 0
   - 2 Types: Mutex/binary semaphore – only 1 can enter c.s., Counting/general semaphore – multiple can enter c.s.
   - Drawback: shared global variable, no connection between the semaphore and the data being controlled, used for both c.s. (mutex) and coordination (scheduling), hard to use, prone to bugs
3. **Monitors**
   - Guarantees mutex, only 1 thread can execute any monitor procedure at any time (the thread is "in the monitor")
   - If 2nd thread invokes a monitor procedure when a 1st thread is alr executing, it blocks (monitor has to have a wait queue)
   - If a thread within a monitor blocks, another one can enter
   - Programming lang construct that controls access to shared data, added by compiler, enforced at runtime
   - Encapsulates: Shared data structure, procedures that operate on the shared data structures, sync b/w concurrent threads that invoke the procedures
4. **Condition Variables**
   - Support 3 operations: `wait()` – release monitor lock, wait for condition var to be signalled, `signal()` wake up one waiting thread, `broadcast` wake up all waiting threads
5. **Barrier**
6. **Messages**

   Simple model of communication & sync based on atomic transfer of data across a channel, Direct application to distributed systems, Messages for synchronisation are straightforward

1.2.1 Classical Synchronisation Problems

Producer-consumer (finite, infinite buffer), Readers-writers, dining philosophers, barbershop

1. **Producer-consumer**

```
mutex = Semaphore(1)
items = Semaphore(0)
```

```
// Producer
event = waitForEvent()
// finite: spaces.wait()
mutex.wait()
buffer.add(event)
mutex.signal()
items.signal()
```

```
// Consumer
items.wait()
mutex.wait()
event = buffer.get()
mutex.signal()
// finite: spaces.signal()
event.proces()
```

1.3 Implementing Locks

- Implementation of locks has critical sections, therefore implementation of acquire/release must be atomic
- Need hardware support: atomic instruction (test-and-set), disable/enable interrupts (prevents context switches)
- **Test-and-set**: record old value, set value, return old value. Hardware executes atomically

```
struct lock { int held = 0; }
void acquire(lock) { while(test_and_set(&lock->held)); }
void release(lock) { lock->held = 0; }
```

- **Problem**: spinlocks are wasteful – a thread is spinning on a lock -> thread holding the lock cannot make progress on uniprocessor if lockholder yields/sleeps or involuntary context switch.

2 Parallel Computing Platforms – Arch Memory

- Recap: CPU Time = $\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}} = \frac{\text{seconds}}{\text{program}}$
- **Instruction/Program** affected by compiler & ISA, **Cycles/Instruction** affected by processor implementation, **seconds/cycle** affected by clock speed (clock freq)

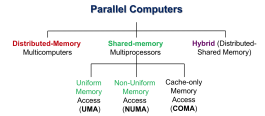
2.1 Sources of processor performance gain: (x parallelism, x is ...)

- **Bit level**: word size, larger so no need bitslicing
- **Instruction level**: **pipelining** (parallelism across time, split instruction execution in multiple stages, allow multiple instructions to occupy different stages in the same clock cycle given no data/control dependencies), **superscalar** (parallelism across space, duplicate pipelines, allow multiple instructions to pass through the same stage, dispatching multiple instructions to diff execution units in the processor, but scheduling is challenging (deciding which instructions can be executed together))
- **Thread level**: processor provides hardware support for the "thread context" – information specific to each thread (PC, Registers, etc) so software threads can execute in parallel. E.g. SMT: Intel Hyper-threading
- **Process level** – process has independent memory space, communicate with IPC through OS. can be mapped to multiple processor cores
- **Processor level**: **Shared memory**, **Distributed Memory**

2.2 Flynn's Parallel Architecture Taxonomy

SISD (Single Instruction Single Data): A single instruction stream is executed, each instruction work on single data. E.g. uniprocessor; **SIMD** A single instruction stream, working on multiple data, exploting data parallelism, known as vector processor. E.g. SSE, AVX in x86; **MISD** Multiple instruction streams, all instruction work on the same data at any time, no actual implementation; **MIMD** Each PU fetch its own instruction, each PU operates on its data. Most popular model for multiprocessor, **Variant – SIMD + MIMD** Stream processor (GPU's): a set of threads executing the same code (SIMD), multiple set of threads executing in parallel (MIMD)

2.3 Memory Organisation



2.3.1 CC (Cache Coherence Protocol)

e.g. ccNUMA, each node has cache memory to reduce contention

2.3.2 COMA (Cache Only Memory Architecture)

Each memory block works as cache memory, data migrates dynamically and continuously according to the cache coherence scheme

2.3.3 UMA (Uniform Memory Access)

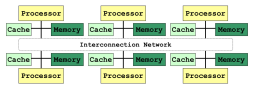
Latency of accessing the main memory is uniform for every processor, suitable for small number of processors

2.3.4 NUMA (Non-Uniform Memory Access)

Diagram just like Distributed-Memory system

- Physically distributed memory of all processing elements are combined to form a global shared-memory address space, also called distributed shared-memory.
- Accessing local mem faster than remote mem for a processor.

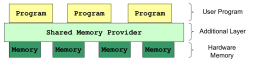
2.3.5 Distributed-Memory System



Each node is an independent unit with processor, memory and peripheral elements.

- Physically distributed memory module: mem in a node is private
- Data exchanges between nodes with message-passing

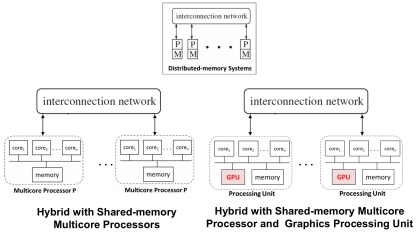
2.3.6 Shared Memory System



Access memory through shared memory provider which maintain the illusion of shared memory.

- Program is unaware of the actual hardware memory architecture
- Data exchange between nodes with shared variables
- **Advantages**: no need to partition code/data or physically move data among processors -> communication is efficient
- **Disadvantages**: Special synchronisation constructs are required, lack of scalability due to contention

2.3.7 Hybrid (Distributed-Shared Memory)



e.g. Hybrid with GPU

2.4 Multicore Architecture

2.4.1 Hierarchical Design

- Multiple cores share multiple caches, cache size increases from the leaves to the root.
- Usages: standard desktop, server processors, GPUs

2.4.2 Pipelined Design

- Data elements are processed by multiple execution cores in a **pipelined** way, from one core to the next
- Useful if same computation steps have to be applied to a long sequence of data elements, e.g. network processors used in routers and graphic processors

2.4.3 Network-based Design

- Cores and their local caches and memories are connected via an interconnection network

3 Parallel Programming Models

Common classification:

- **Level of parallelism**: granulaty from instruction -> statement -> procedural (function/methods) level or parallel loops
- **Specification of parallelism**: implicit/user-defined explicit
- **Execution**: e.g. SIMD/SPMD, sync or async
- **Communication mode**: message-passing/shared vars
- **Synchronisation (coordination mechanisms)**

3.1 Decomposition

- Generate enough tasks to keep all cores busy at all times (no of tasks $\geq$ no of cores)
- Granularity is large compared to scheduling & mapping time (size of task >> overhead of parallelism)
- **Static** at program start/compile time, **Dynamic** during runtime

3.2 Scheduling

- Find an efficient task exec order to optimise a given objective.
- Good load balancing among tasks: Computations, Memory accesses (shared), Communication ops (distributed)
- Static/Dynamic scheduling

3.3 Mapping

- Assignment of processes/threads to execution units
- Focuses on performance: Equal utilisation of execution units, minimal communication between the processors

3.4 Parallelism

- Avg no of units of work that can be performed in parallel/unit time. Work = task + dependencies.
- Types in increasing task size: instruction -> loop -> data -> task

3.4.1 Instruction Parallelism

Multiple instructions exec may be inhibited by 3 types of data deps. Can be used to create data dependency graph

- **Flow dep (RAW)**/ true dependency: (**a**) `a = b + c; e = a + d`
- **Anti-dependency (WAR)**: (**b**) `a = b + c; b = d + e;`
- **Output dep (WAW)**: (**a**) `a = b + c; a = d + e;`

3.4.2 Loop Parallelism

If the iterations are independent, they can be executed in arbitrary order & in parallel on different processors

3.4.3 Data Parallelism

- **Partition the data** used in solving the problem among the processing units; each processing unit carries out similar operations on its part of the data
- Same op is applied to diff elements of a data set, if ops are independent, elements can be distributed among processors for parallel execution, or using SIMD computers/instructions
- e.g. `for (i = 0; i < N; i++) a[i] = b[i - 1] + c[i];`
- On MIMD, common model: SPMD (Single Program Multiple Data) – 1 parallel program is executed by all processors in parallel (both shared & distributed address space)

3.4.4 Task Parallelism

- **Partition tasks** in solving the problem among processing units

3.5 Task Dependence Graph

- A directed acyclic graph, **node**: task, value = expected exec time; **edge**: control dep b/w task
- **Properties**:
  - **Critical Path Length**: min (fastest) completion time
  - **Degree of concurrency** = total work / critical path length (indication of amount of work that can be done concurrently)

3.6 Representation of parallelism

3.6.1 Automatic Parallelisation

- Parallelising compilers perform decomposition & scheduling
- Drawbacks: Dep analysis is difficult for pointer-based computations/indirect addressing; Exec time of function calls/loops with unknown bounds is difficult to predict at compile time

3.6.2 FP languages

- Describe computation as eval of maths functions w/o side effects.
- **Advantage**: new lang constructs are not necessary to enable parallel exec, **Challenge**: extract parallelism at right level of recursion

3.7 Parallel Programming Patterns

Provides a coordination structure for tasks

3.7.1 Fork-Join

Task T creates a number of child tasks with `fork()`, then waits for termination using `join` call

3.7.2 Parbegin-Parend

When an executing thread reaches parbegin-parend, a set of threads is created and statements of the construct are assigned to these threads for execution. Statements following the construct are only executed after all these threads have finished their work.

3.7.3 SIMD

Single instructions are executed synchronously by the different threads on different data

3.7.4 SPMD

Same program executed on different processors but operate on different data, No implicit synchronisation – must use explicit sync ops

3.7.5 Master-Slave (or Master-Worker)

Master coord, init, timings, output, assigns work to slaves

3.7.6 Client-Server

- MPMD (multiple program multiple data) model
- Useful in heterogeneous systems
- Server compute requests from multiple client tasks concurrently, can use multiple threads to compute a single request
- A task can generate request to other tasks (client role) and process requests from other tasks (server role)

3.7.7 Pipelining

Data in the application is partitioned into a stream of data elements that flows through the each of the pipeline tasks one after the other to perform different processing steps.

3.7.8 Task (Work) Pools

- Common data structure from which threads can access to retrieve tasks for execution. During processing of a task, a thread can generate new tasks and insert them into the task pool.
- No of threads is fixed, threads created statically by main thread
- Access to the task pool must be synced to avoid race conditions
- Exec is completed when Task pool is empty, Each thread has terminated the processing of its last task
- **Advantages**: Useful for adaptive & irregular applications (tasks can be generated dynamically), Overhead for thread creation is independent of problem size & no of tasks
- **Disadvantages**: For fine-grained tasks, the overhead of retrieval & insertion of tasks becomes important

3.7.9 Producer-Consumer

- Prod threads produce data which are used as input by con threads
- Sync must be used to ensure correct coordination b/w prod & con

4 Performance of Parallel Systems

4.1 Performance factors

Down the list, Higher level of abstraction, higher loss in performance

- **Machine Model**: provide a description of hardware & OS
- **Architectural Model**: includes interconnection network, memory org, sync/async processing, exec mode
- **Computational Model**: provide an analytical method for designing and evaluating algo on a given arch model
- **Programming Model**: define how programmer can code an algo

4.1.1 Reponse Time in Sequential Systems

- Wall-clock time (actual time taken) = user CPU + system CPU (exec OS routines) + waiting time (I/O, time sharing)
- Focus on CPU Time: depends on Translation of program statements by compiler, exec time of each instruction
- $Time_{user}(A) = N_{cycle}(A) \times Time_{cycle}$, $N_{cycle}(A) = \sum_{i=1}^{n} n_i(A) \times CPI_i$
- Thus, $Time_{user}(A) = N_{instr}(A) \times CPI(A) \times Time_{cycle}$
- Refinement with memory access time: $Time_{user}(A) = (N_{cycle}(A) + N_{mm\_cycle}(A)) \times Time_{cycle}$
- One-level cache: $N_{mm\_cycle}(A) = N_{read\_cycle}(A) + N_{write\_cycle}(A)$, $N_{read\_cycle}(A) = N_{read\_op}(A) \times R_{read\_miss}(A) \times N_{miss\_cycles}(A)$
- $Time_{user}(A) = (N_{instr}(A) \times CPI(A)) + N_{rw\_op}(A) \times R_{miss}(A) \times N_{miss\_cycles} \times Time_{cycle}$ ($R_{miss}(A)$ r/w miss rate)
- **Average memory access time**: $T_{read\_access}(A) = T_{read\_hit} + R_{read\_miss}(A) \times T_{read\_miss}$
- **Two-level cache**:
  $T_{read\_access}(A) = T^{L1}_{read\_hit} + R^{L1}_{read\_miss}(A) \times T^{L1}_{read\_miss}$
  $T^{L1}_{read\_miss}(A) = T^{L2}_{read\_hit}(A) + R^{L2}_{read\_miss}(A) \times T^{L2}_{read\_miss}$
  Global miss rate $= R^{L1}_{read\_miss}(A) \times R^{L2}_{read\_miss}(A)$

4.1.2 Throughput: MIPS (Million-Instruction-Per-Second)

- $MIPS(A) = \frac{N_{instr}(A)}{Time_{user}(A) \times 10^6} = \frac{clockfreq}{CPI(A) \times 10^6}$
- **Drawbacks**: Consider only the no of instr, easily manipulated

4.1.3 Million-FLoating-point-Operations-Per-Second

- $MFLOPS(A) = \frac{N_{flops}(A)}{Time_{user}(A) \times 10^6}$
- **Drawbacks**: No differentiation between diff types of flops

4.1.4 Benchmarks

Industry Standards: SPEC benchmark suites, EEMBC benchmark suites, Numerical Aerodynamic Simulation (NAS) from NASA (massively parallel benchmark for computer cluster), **Simple Benchmark**: Linpack, Dhrystone/Whetstone, Tak function

4.2 Parallel Execution Time

- $T_p(n)$, problem of size n, p processors
- Consists of Time for executing local computations, exchange of data and sync b/w processors, waiting time (unequal load distribution, wait to access shared data structure)

4.2.1 Cost

- $C_p(n) = p \times T_p(n)$ = total amount of work performed by all processors (processor-runtime product)
- **Cost optimal** = execs same total no of ops as fastest seq program

4.2.2 Speedup

- $S_p(n) = \frac{T_{best\_seq}(n)}{T_p(n)}$
- Theoretically, $S_p(n) \leq p$ always holds, but in practice $S_p(n) > p$ (superlinear speedup) can occur (due to better cache locality, etc.)

4.2.3 Best Sequential Algo: Difficulties

- Best sequential algo may not be known OR there exists an algo with optimum asymptotic exec time, but other algo lead to lower exec time in practice

4.3 Parallel Program Efficiency

$E_p(n) = \frac{T_{best\_seq}}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T_{best\_seq}}{p \times T_p(n)}$, Ideal $S_p(n) = p \rightarrow E_p(n) = 1$

4.4 Scalability

4.4.1 Grosch's Law (1953)

The speed of a computer is proportional to the square of its cost

- A collection of smaller processors will always have less performance than a single processor of the same total cost
- Rebuttal: No longer applies to computer systems build with many inexpensive commodity processors, Commodity processors are more cost effective than custom design supercomputers

4.4.2 Minsky's Conjecture (1971)

The speedup achievable by a parallel computer increases as the logarithm of the no of processing elements

- Implication: large-scale parallelism unproductive
- Rebuttal: Experimental results prove that speedup depends strongly on particular algo & computer arch, and many algo exhibit linear speedup for over 100 processors

4.4.3 Amdahl's Law (1967)

Speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelised

- $0 \leq f \leq 1$ = sequential fraction (fixed-workload performance)
- $S_p(n) = \frac{T_{best\_seq}(n)}{f \times T_{best\_seq}(n) + \frac{1-f}{p} T_{best\_seq}(n)}$
- Implication: manufacturers are discouraged from making large parallel computers, more research attn shifted towards developing parallelising compilers that reduces f
- Rebuttal: in many computing problems, f is not constant, dependent on problem size n. An effective parallel algorithm is $\lim_{n\to\infty} f(n) = 0$, thus speedup $\lim_{n\to\infty} S_p(n) = \frac{p}{1+(p-1)f(n)} = p$, thus Amdahl's law can be circumvented for large problem size

4.4.4 Gustafson's Law (1988)

- Main constraint is exec time, then higher computing power is used to improve accuracy/better result
- If f decreases when n increases, then $S_p(n) \leq p$, $\lim_{n\to\infty} S_p(n) = p$

5 Coherence, Consistency, Interconnections

5.1 Cache

5.1.1 Write Policy

- **Write-through** - write immediately transferred to main memory, Advantage: no stale value, disadv: slow (soln: use a write buffer)
- **Write-back** - write op performed only in the cache, write performed only when cache block is replaced, tracked with a dirty bit. Adv: less write ops, disadv: mem may contain invalid entries

5.1.2 Cache Coherence Problem

3 properties of coherent memory system:

1. P write to X, no write to X, P read from X, should be same value
2. P_1 write to X, no write ro X, P_2 read from X, should be same value (**Write Propagation**)
3. Write v_1 to X, write v_2 to X, processors read in same order (v_1 then v_2) (**Write Serialisation**)

5.1.3 Maintaining cache coherence

- Software-based soln (compiler+hw aided soln)
- Hardware-based soln (most common on multiprocessor, known as **cache coherence protocols**)

Major Tasks:

- **Track cache line sharing status**, 2 major categories:
  - **Snooping-based**: no centralised directory, each cache keeps track of the sharing status, cache **monitors/snoops** on the bus to update the status of cache line & take appropriate action. Most common in arch with a bus. Granularity is cache block.
    - All the processors on the bus can observe every bus transactions (**write propagation**), Bus transactions visible to the processors in the same order (**write serialisation**)
  - **Directory based**: sharing status kept in a centralised location. Most common in NUMA
- Handle the update to a shared cache line (maintain coherence)

5.2 Memory Consistency Models

4 types of memory op orderings: RAW, RAR, WAR, WAW

5.2.1 Sequential Consistency

Maintains all four orderings

- Every processor issues its mem ops in program order
- Mem accesses are atomic (effect of each mem op must be visible to all before next mem op)

5.2.2 Total Store Ordering (TSO)

- Relaxing RAW (WAW still exists, writes by same thread in-order)
- Processor P can read B before its write to A is seen by all (processor can move its own reads in front of its own writes)
- Read by other processors cannot return new value of A until write to A is observed by all processors

5.2.3 Processor Consistency

- Relaxing RAW (WAW still exists, writes by same thread in-order)
- Any processor can read new value of A before the write is observed by all processors

5.2.4 Partial Store Ordering

- Relaxing RAW & WAW, but still guarantees write atomicity

5.3 Interconnection Networks

5.3.1 Shearsort

1. Row sorting, odd rows sort asc, even rows sort desc
2. Column sorting, all columns sort asc
3. Repeat until sorted

For n numbers, $log(n)$ phases, $O(\sqrt{n}\log n)$. Good for 2D mesh network, only communication with adjacent nodes.

5.4 Topology

5.4.1 Metric

- **Diameter**: max. distance b/w any pair of nodes (small diameter, small distances for message transmission)
- **Degree**: no of direct neighbour nodes (small node degree reduces node h/w overhead)
- **Bisection width**: min. no. of edges to be removed to divide the network into 2 equal halves. (measure for capacity of network when transmitting messages simulatenously)
- **Node connectivity**: min. no of nodes failing to disconnect the network (robustness of the network)
- **Edge connectivity**: min. no of edges failing to disconnect the network (no of independent paths b/w any pair of nodes)

5.4.2 Direct Interconnection (Static/Point-To-Point)

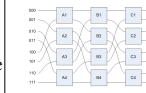
Torus = 2 links for every dim, Mesh = torus w/ no wraparound, CCC = k-dim hypercube but each node replaced with cycle of k-nodes

network G with n nodes	degree $g(G)$	diameter $\delta(G)$	edge-connectivity $ec(G)$	bisection bandwidth $B(G)$
complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
linear array	2	$n-1$	1	1
ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d-dimensional mesh ($n = r^d$)	$2d$	$d(\sqrt[d]{n}-1)$	d	$n^{\frac{d-1}{d}}$
d-dimensional torus ($n = r^d$)	$2d$	$d\lfloor \frac{\sqrt[d]{n}}{2} \rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dimensional hyper-cube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
k-dimensional CCC-network ($n = k2^k$ for $k \geq 3$)	3	$2k - 1 + \lfloor k/2 \rfloor$	3	$\frac{n}{2k}$
complete binary tree ($n = 2^k - 1$)	3	$2\log \frac{n+1}{2}$	1	1
k-ary d-cube ($n = k^d$)	$2d$	$d\lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

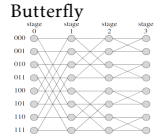
5.4.3 Indirect Interconnection (using switches)

- **Bus Network** – a set of wires, only one pair of devices can communicate at a time, bus arbiter for coordination
- **Crossbar Network** has $n \times m$ switches for n inputs, m outputs. Switch either straight/change dir
- **Omega Network** – 1 unique path for every input to output, Uses $\log n$ stages, $\frac{n}{2}$ switches/stage. Edge from node (α, i) to two nodes $(\beta, i+1)$ where $\beta = \alpha$ by a cyclic left shit (+ inversion of the LSBit)
- **Butterfly Network** – Node (α, i) connects to $(\alpha, i+1)$ and $(\alpha', i+1)$, α and α' differ in the (i+1)th bit from the left (cross edge)
- **Baseline Network** – Node (α, i) connects to 2 nodes $(\beta, i+1)$ where β = cyclic right shift of last (k-1) bits of α OR inversion of the LSBit of α + cyclic right shift of last (k-i) bits.

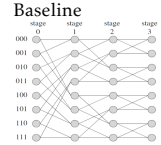
Omega



Butterfly



Baseline



5.5 Routing

Based on:

- Path length (minimal/non-minimal) whether shortest path
- Adaptivity (deterministic/adaptive) whether always the same path for same pair of nodes/take into account network status

3 deterministic examples

5.5.1 XY Routing for 2D Mesh

(X_{src}, Y_{src}) to (X_{dst}, Y_{dst}): move in x-direction until $X_{src} = X_{dst}$, then move in y-direction similarly

5.5.2 E-Cube Routing for Hypercube

No of bits difference b/w 2 pairs of nodes = number of hops (hamming distance). Start from MSB to LSB (or vice versa), find first different bit, go to the neighbouring node with the bit corrected.

5.5.3 XOR-Tag Routing for Omega Network

T = Source Id $\oplus$ (XOR) Destination ID, at stage-k: go straight if bit k of T is 0, crossover if bit k of T is 1