CS3210 Finals Cheatsheet v1.1 (2019-02-10) by Julius Putra Tanu Setiaji, page 1 of 2

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 comm for inter-process info exchange, Context switching between processes, must save state, overhead.
- 2 types of exec: time slicing (pseudo-parallelism), parallel exec of procs on diff 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.
- 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 shld be suitable to parallelism deg of application, available exec resources, not too large to keep the overhead small for thread creation, management, termination

1.2 Synchronisation

- **Race condition** = 2 concurrent threads accessed a shared resource without access sync
- Use **mutual exclusion** to sync access to shared resources, to protect **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 c.s., then T cannot prevent some other thread S from entering the c.s. A thread in the c.s. 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**: 1 process holding 1 resource and waiting for another resource
  3. **No preemption** – Resources cannot be preempted (c.s. 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 var, no connection b/w semaphore & data being controlled, used for both c.s. (mutex) and coordination (scheduling), hard to use, bug-prone
3. **Monitors** (the thread is "in the monitor")
   - Guarantees mutex, only 1 thread can execute any monitor procedure at any time
   - 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

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

1.3 Implementing Locks

- Implementation of locks has c.s., therefore impl of acquire/release must be atomic
- Req HW support: atomic instruction, 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 – thread spinning on a lock -> thread holding the lock cannot make progress on uniprocessor if lockholder yields/sleeps/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)

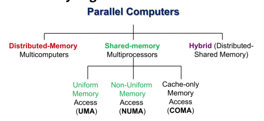
2.1 Sources of processor performance gain: (x parallelism, x is ...)

- **Bit level**: word size, larger so no need bitslicing
- **Instruction level**: **pipelining** (parallelism across time, split instr exec in multiple stages, allow multiple instr to occupy different stages in the same clock cycle given no data/control deps), **superscalar** (paralellism across space, duplicate pipelines, allow multiple instr to pass through the same stage, dispatching multiple instr to diff exec units in the processor, but scheduling is challenging (deciding which instr can be exec together))
- **Thread level**: processor provides HW support for the thread context – info specific to each thread (PC, Registers, etc) so software threads can execute in parallel. E.g. SMT: Intel HT
- **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): Single instruction stream is executed, each instruction work on single data. E.g. uniprocessor; **SIMD** 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 impl; **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


Parallel Computers
Distributed-Memory
Multicomputers
Shared-memory
Multiprocessors
Hybrid (Distributed-Shared Memory)
Uniform Memory Access (UMA)
Non-Uniform Memory Access (NUMA)
Cache-only Memory Access (COMA)

2.3.1 CC (Cache Coherence Protocol)

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

2.3.2 UMA (Uniform Memory Access)

Latency of accessing the main mem uniform for all processor, suitable for small no of processors

2.3.3 NUMA (Non-Uniform Memory Access)

Diagram just like Distributed-Memory system

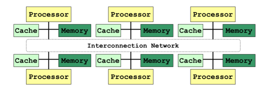
2.3.4 COMA (Cache Only Memory Architecture)

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

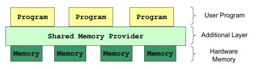
- Physically distributed memory of all processing elements are combined to form a global shared-memory address space, also called distributed shared-memory.
- Accessing local mem faster than remote mem for a processor.

2.3.5 Distributed-Memory System

- Each node is an indep unit with processor, memory and peripheral elements.
- Physically distributed memory module: mem in a node is private
- Data exchanges between nodes with message-passing



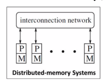
2.3.6 Shared Memory System

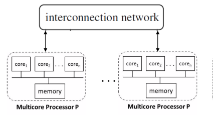


- Access memory through shared memory provider which maintain the illusion of shared memory.
- Program is unaware of the actual hardware memory architecture

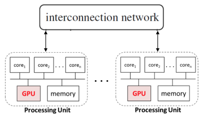
- Data exchange between nodes with shared variables
- **Advantages**: no need to partition code/data or physically move data among processors -> communication is efficient
- **Disadvantages**: Special sync constructs are required, lack of scalability due to contention

2.3.7 Hybrid (Distributed-Shared Memory)





Hybrid with Shared-memory Multicore Processors



Hybrid with Shared-memory Multicore Processor and Graphics Processing Unit

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 processed by multiple exec cores in a **pipelined** way, from one core to 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 ≥ 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 (Static/Dynamic)

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

3.3 Mapping

- Assignment of processes/threads to execution units
- Focuses on perf: Equal utilisation of exec units, minimal comm b/w processors

3.4 Parallelism

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

3.4.1 Instruction Parallelism

Multiple instr exec may be inhibited by 3 types of data deps. 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 iterations are indep, they can be exec'd 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)

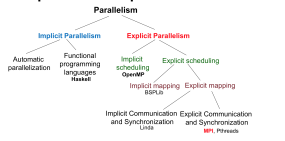
3.4.4 Task Parallelism

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

3.5 Task Dependence Graph

- DAG, **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 diff 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

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 gen requests to other tasks (client) and process requests from other tasks (server)

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 Response Time in Sequential Systems

- Wall-clock time (actual time taken) = user CPU + system CPU (OS routines) + waiting time (I/O, time sharing)
- Focus on CPU Time: depends on translation 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 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

- Collection of smaller processors will always have less perf than a single processor of same total cost
- **Rebuttal**: No longer applies to computer systems build with many inexpensive commodity processors which are more cost effective than custom design supercomputers

4.4.2 Minsky's Conjecture (1971)

Speedup achievable by parallel computer increases as log(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 algo that can't 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 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, **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

CS3210 Finals Cheatsheet v1.1 (2019-02-10) by Julius Putra Tanu Setiaji, page 2 of 2

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 comm w/ adj nodes.

5.4 Topology

5.4.1 Metric

- **Diameter**: max dist b/w any pair of nodes (small diameter, small dist for msg 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 network (network robustness)
- **Edge connectivity**: min. no of edges failing to disconnect network (no of independent paths b/w any pair of nodes)

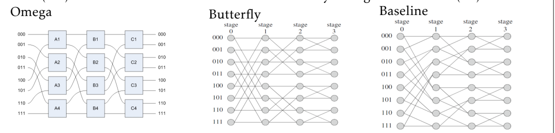
5.4.2 Direct Interconnection (Static/Point-To-Point)

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	$n-1$	$(\frac{n}{2})^2$
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 $(n=r^d)$	$2d$	$d\left\lfloor\frac{\sqrt[d]{n}}{2}\right\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
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) - examples for 8 to 8

- **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 in, m out. 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 $\beta=$ 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.



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

5.5.1 XY Routing for 2D Mesh (Deterministic)

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

5.5.2 E-Cube Routing for Hypercube (Deterministic)

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

5.5.3 XOR-Tag Routing for Omega Network (Deterministic)

T= Source Id $\oplus$ (XOR) Destination ID

A t stage-k: go straight if bit k of T is 0, crossover if bit k of T is 1

6 Parallel Programming Models

6.1 Data Distribution

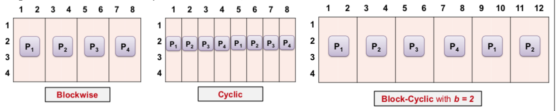
Assumptions: p identical processors, array starts from 1, Block size = B = $\left\lceil\frac{n}{p}\right\rceil$

6.1.1 1D Array

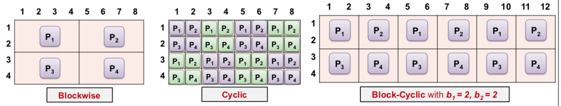
- **Blockwise**: P_i takes elements $[(j-1)\times B+1, ..., j\times B]$
- **Cyclic**: P_i takes elements $[j, j+p, ..., j+(B-1)\times p]$ if $j \leq n \mod p$, else $[j, j+p, ..., j+(B-2)\times p]$

6.1.2 2D Array

- Combination of blockwise/cyclic in 1/both dimension(s)
- 1D distribution: column-wise blockwise, cyclic, or block-cyclic (Form blocks of size b, then perform round-robin cyclic allocation)



- Checkerboard distribution



6.2 Information Exchange

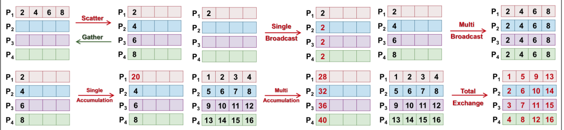
For controlling coordination of different parts of a parallel program execution

6.2.1 Shared Address Space (Shared variables)

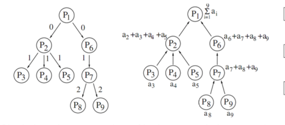
Assumes a global memory accessible by all processors, need sync ops for safe concurrent access to prevent race condition using mutex to protect critical section, e.g. `#pragma omp critical {}` or `omp_init_lock(omp_lock_t*)`, `omp_set_lock`, `omp_unset_lock`

6.2.2 Distributed Address Space (Communication operations)

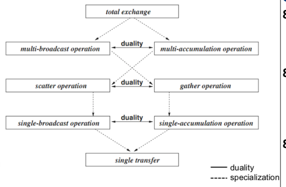
- Assumes disjoint memory space – data exchange through dedicated comm ops
- 2 main types of data exchange: point-to-point and global communication
- Examples:
  - **Single transfer**: point-to-point, `send` and `receive`
  - **Scatter/Gather**: takes elements and distributes them in order of process rank
  - **Single Broadcast**: takes a single data element at the root and copies it to all other
  - **Multi Broadcast**: (`MPI_Allgather`) "Gather + Single Broadcast"
  - **Single accumulation**: reduction (binary, associative, commutative) op applied to all elements, result stored in root (Gather + Reduce)
  - **Multi accumulation**: "Accumulation + Scatter"
  - **Total Exchange**: (`MPI_Alltoall`) "transpose"



- **Duality of comm ops**: interconnection can be represented by a spanning tree. Duality if the same spanning tree can be used for both ops



Single-broadcast operation (top-down traversal) Single-accumulation operation (bottom-up traversal)


total exchange
multi-broadcast operation
duality
multi-accumulation operation
scatter operation
gather operation
single-broadcast operation
single-accumulation operation
single transfer
duality
specialization

7 Message Passing Programming

Loosely synchronous paradigm, tasks sync to perform interactions, but otherwise exec is async: **SPMD** (Single Program Multiple Data) model

	Blocking	Non-Blocking
Buffered	Sending process returns after data has been copied into comm buffer	Sending process returns after initiating DMA transfer to buffer, may not be complete
Non-bufferred	Sending process blocks until matching receive operation has been encountered	

- **Blocking** = send op blocks until it is safe. Issue: deadlock
- **Non-blocking** = send/recv returns before it is safe – generally accompanied by a check-status op. Used correctly, can overlap comm overheads with useful computations
- **Non-buffered**, issues: idling (timing mismatch sender <-> recvr)
- **Buffered**: buffers at both ends, trades idling overhead for buffer copying overhead

Summary = **Overhead**: idling (non-buffered) vs buffer management (buffered), **Side effect**: safe and easier programming (blocking) vs hiding comm overhead (non-blocking)

7.1 Message Passing Interface (MPI)

7.1.1 Semantic, Local View

- **Blocking**: return from lib call indicates user is allowed to reuse resources specified
- **Non-blocking**: may return before op completes, before user is allowed to reuse resources

7.1.2 Semantic, Global View

- **Synchronous**: comm op doesn't complete before both processes have started their comm op
- **Asynchronous**: sender can execute its comm op without any coordination with the receiver

	Synchronous	Asynchronous
Blocking	`MPI_SSend`, `MPI_SRecv`	`MPI_Send`, `MPI_Recv`
Non-blocking	`MPI_ISSend`, `MPI_ISRecv`	`MPI_ISend`, `MPI_IRecv`

7.1.3 Initialisation, Finalisation, Abort

- `int MPI_Init(int* argc, char** argv[])`, initialise MPI program, called only once
- `int MPI_Finalize(void)`, terminate all MPI processing, last MPI call
- `int MPI_Abort(MPI_Comm comm, int errorCode)`, force all processes to terminate, return errorCode to `mpirun`
- Other functions: `MPI_Comm_size`, `MPI_Comm_rank`

7.1.4 MPI Messages Format

- **Message** = data + envelope (how to route the data)
- **Data** = start-buffer (pointer to data) + count + datatype
- **Envelope** = destination/source (using rank in a communicator) + tag (arbitrary number to distinguish messages) + communicator

7.1.5 Process Group

- An ordered set or processes where each process has a unique rank
- A process may be member of multiple groups (and may have diff ranks in each of these groups) – handled by MPI

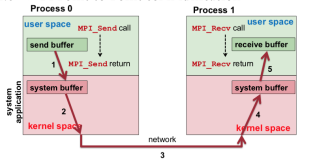
7.1.6 Communicator

- Communication domain for a group of processes
- 2 types: **Intra-communicators** (supports the execution of arbitrary collective comm op on a single group, default: `MPI_COMM_WORLD`), **Inter-communicators** (supports the point-to-point comm op b/w 2 process groups)
- Allows to organise tasks based on func into task groups, Enable collective communication operations across a subset of related tasks, Provide basis for user-defined virtual topologies

7.1.7 Process Virtual Topologies

- A communicator with a Cartesian style of addressing the ranks of the processes
- `MPI_Cartdim_get` and `MPI_Cart` prefix: `create`, `get`, `coords`, `rank`, `shift`

7.1.8 MPI Point-to-Point communication



Messages to one particular destination are delivered in the order they are sent.

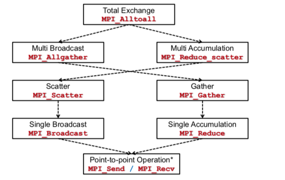
No guarantee for multiple messages to multiple destinations

7.1.9 Deadlocks

```
MPI_Recv(recvbuf, ...);          MPI_Recv(recvbuf, ...);
MPI_Send(sendbuf, ...);          MPI_Send(sendbuf, ...);
```

- For `MPI_Recv`, as above
- For `MPI_Send`, occurs when runtime doesn't use system buffers/system buffers are too small.
- **Deadlock-free logical ring**: procs w/ even rank: send -> receive; odd rank: receive -> send

7.1.10 Collective Communication



- Operations involving all procs in a communicator, blocking by default
- **Scatter**: one-to-many
- **Gather**: many-to-one (with/without accumulation)
- `int MPI_Barrier(MPI_Comm);` the only collective sync op – all procs must execute, blocks until all procs reach the barrier

7.1.11 Measuring Program Timing

- `double MPI_Wtime(void)` Time in seconds since an arbitrary time in the past.
- `double MPI_Wtick(void)` Time in seconds of resolution of `MPI_Wtime`

8 CUDA Programming

8.1 GPU Architecture

- Multiple **Streaming Multiprocessors** (SMs) – memory, cache, connecting interface
- SM consists of multiple compute cores, memories (registers, L1 cache, texture memory, shared memory), logic for thread & instruction management

8.2 CUDA Kernel and Threads

- **Device** = GPU, **Host** = CPU, **Kernel** = function that runs on device
- Parallel portions execute on device as kernels, one kernel at a time, 1 kernel = many threads
- CUDA threads are extremely lightweight, very little creation overhead, instant switching
- All threads run the same code – SPMD (Single Program Multiple Data)

8.3 Thread Cooperation

- Share results to save computation, share memory accesses to reduce bandwidth
- Divide monolithic thread array into multiple blocks
- In a block: **shared memory, atomic ops, barrier sync**. Threads in diff blocks can't cooperate
- HW is free to schedule thread blocks to any processor at any time

8.4 Thread Hierarchy

- A kernel is executed by a grid of thread blocks. A block executes on 1 SM, does not migrate.
- Several blocks can reside concurrently on 1 SM, limited by control limitations (32 blocks/SM, 1024 threads/block) and resources (register file and shared memory are partitioned among all resident threads)

8.5 Thread Execution Mapping to Architecture

- SIMT (Single Instruction Multiple Thread) execution model
- SM creates, manages, schedules, executes threads in SIMT warps (group of 32 parallel threads). Threads in a warp start together at the same program address. A block is always split into warps the same way
- Warp executes 1 common instruction at a time. Scheduler optimises by grouping threads with the same exec paths in the same SIMT unit

8.6 CUDA Memory Model

Memory	Location	Cached	Access	Scope	Lifetime
Register	on chip	n/a	R/W	1 thread	thread
Local	off chip	No	R/W	1 thread	thread
Shared	on chip	n/a	R/W	All threads in block	block
Global	off chip	No	R/W	All threads + host	host alloc
Constant	off chip	Yes	R	All threads + host	host alloc
Texture	off chip	Yes	R	All threads + host	host alloc

- **Shared** = higher bandwidth and lower latency than local/global. Divided into equal-sized mem mods called **banks**. Access to diff banks can be simultaneous. **Bank conflict**: mem request in the same bank, has to be serialised
- **Local**: automatic array vars alloc by compiler, **Texture**: for spatially coherent random-access r/o data (provides filtering, address clamping, wrapping)

8.7 Programming in CUDA

- **Func qual**: `__host__`, `__device__`, `__global__`, **Launching kernel:** `kernel<<<80, 64>>>`, **Var qual:** `__device__`, `__constant__`, `__shared__`, **unqualified**: scalar, built-in vector stored in registers, arrays of more than 4 elements orruntime indices stored in local mem
- **Thread synchronisation**: `void __syncthreads()`, generates barrier sync instruction

8.8 Memory Optimisations

Minimise data transfer between host and device mem, use page-locked/pinned mem transfer. Overlapping async transfers with GPU computations

8.8.1 Coalesced access to Global Mem

Simultaneous accesses to global mem by threads in a half-warp can be coalesced into as few as a single mem txn of 32, 64, or 128 bytes

8.8.2 Overall Memory Optimisations

Minimize data transfer between host and device, Ensure global memory accesses are coalesced whenever possible, Minimize global memory accesses by using shared memory, Minimize bank conflicts in shared memory accesses

8.9 Overall Optimisation

- **Maximise parallel exec**: restructure algo to expose as much data parallelism as possible, map to HW carefully choosing exec config of each kernel invocation
- **Optimizing memory usage** to achieve maximum memory bandwidth: diff mem spaces and access patterns
- **Optimizing instruction usage** to achieve maximum instruction throughput: Use high throughput arithmetic instructions, avoid diff exec paths within the same warp

9 Parallel Algorithm Design

9.1 Overheads of Parallelism

Tradeoff: task granularity – small to reduce overhead, large to still have enough parallel work

9.2 General Design Approach

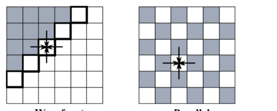
- Consider machine independent issues before machine specific aspects of design
- Task/Channel model: **Tasks** interact by sending messages through **channels**

9.3 Foster's Design Methodology

9.3.1 Partitioning

- Divide **computation** & **data** into independent pieces to discover max parallelism.
- Data centric – **Domain decomposition**: divide data into pieces of approx. equal size, determine how to associate computations with the data
- Computation centric – **Functional decomposition**: divide computation into pieces, determine how to associate data with the computations.
- **Rules of thumb**: at least 10x more primitive tasks than processors, minimise redundant computations and data storage, primitive tasks roughly same size, no of tasks an increasing function of problem size

9.3.2 Communication



- **Local communication**: task needs data from a small no of other tasks ("neighbours"), create channels illustrating data flow
- **Global communication**: significant no of tasks contribute data to perform a computation, don't create channels for them early in design
- Ideally, distribute and overlap computation and communication

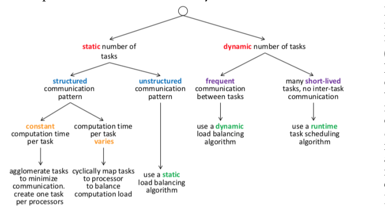
- Local comm: 2 finite diff update strategies. Parallel: time t unshaded, time (t+1) shaded
- Global comm: optimisation through pipeline/divide-and-conquer

9.3.3 Agglomeration

- Combine tasks into larger tasks to improve perf (reduce task creation and comm overhead), maintain scalability, simplify programming
- In MPI, goal: create 1 agglomerated task/processor
- E.g. reduce dim of decomposition from 3 to 2, **3D decomposition**: combine adjacent tasks, **divide-and-conquer**: coalesce sub-trees, **tree algo** – nodes are combined
- Rules of thumb: increase locality of parallel algo, no of tasks increases with problem size, no of tasks suitable for target, reasonable trade-off b/w agglomeration and code modification

9.3.4 Mapping

- Assigning tasks to processors, performed by OS (centralised multiprocessor), User (distributed mem systems)
- Conflicting goals: **Maximise processor util** – place tasks on diff processors to increase parallelism, **Minimise inter-processor comm** – place tasks that communicate frequently on the same processor to increase locality


static number of tasks
dynamic number of tasks
structured communication pattern
unstructured communication pattern
frequent communication between tasks
many short-lived tasks, no inter-task communication
constant computation time per task
computation time per task varies
use a dynamic load balancing algorithm
use a runtime task scheduling algorithm
agglomerate tasks to minimize communication. create one task per processors
cyclically map tasks to processor to balance computation load
use a static load balancing algorithm

Rules of thumb: Finding optimal mapping in general is NP-hard (thus, must rely on heuristic), consider designs based on 1 task/processor and multiple tasks/processor, evaluate static & dynamic alloc (dynamic -> ensure task allocator not a bottleneck, static -> ratio of tasks:processors at least 10:1)

10 Energy-efficient Computing and Data Centers

10.0.1 ARM Cortex-A family Vs general purpose Intel/AMD x86 servers

Similarities	Differences
• Processors cores + RAM + I/O interfaces + misc. peripherals • Cores use similar exec model (pipelines) • Memory hierarchy (L1 + L2 + RAM) • Uses Virtual Memory • Uses commodity Linux – Supports all programming models available on Linux, most server SW is available, Porting is trivial • Hardware virtualization, hardware-level security	• RISC Instruction-Set Architecture • Heterogeneous cores (big.LITTLE) – Fast cores / slow cores • Lower instruction-level parallelism exploitation • Smaller L1/L2 caches • Less RAM (0.5 GB – 4GB), typically non-upgradable, depends on config • Lower main-memory bandwidth • Simpler I/O interfaces (USB2.0/USB 3.0/SATA2), next gen: PCI-Express, SATA 3

10.1 Cloud Computing

- Abstraction of underlying applications, information, content and resources, allows resources to be provided and consumed in a more elastic manner and on demand.
- Models: Software/Platform/Infrastructure as a Service (SaaS e.g. Google Apps; PaaS, e.g. Google App Engine; IaaS, e.g. Amazon EC2)
- Deployment models: Private, Community, Public, Hybrid Cloud
- Characteristics: On-demand self-service, Broad network access, Resource pooling, Rapid elasticity, Measured service

10.2 Virtualisation

- Creation of a virtual version of something. Access resource without being concerned with where and how the resource is physically located/managed.
- Types: **Services**, **Server**, **Storage**, **Network**