Jin Wei CS3210

Introduction

Concurrency vs Parallelism

- Concurrency: Two or more tasks are **processed** together, i.e. make progress together by interleaving their executions or executing at the same time.
- Two or more tasks **executing** simultaneously (on separate cores).

Program Parallelisation

1. Decomposition of computations
2. Scheduling (assignment of tasks to processes or threads)
3. Mapping of processes / threads to physical processors / cores

Mutexes can only be released by the task that acquires it, unlike semaphores which have no connection to the data being controlled.

Deadlock can happen iff **ALL** these conditions hold:

1. mutual exclusion: at least one resource must be held in a non-shareable mode (X-lock)
2. hold and wait: there must be one process holding one resource, and waiting for another resource
3. no pre-emption: resources cannot be pre-empted (critical sections cannot be aborted externally)
4. circular wait: there must exist a set of processes waiting in a circle

Semaphores

- named semaphores (`sem_open`) are shared between processes because memory spaces are duplicated on `fork`
- semaphores are saved as virtual file

Parallel Computing Architectures

Granularity of Parallelism

- Bit-Level
- Instruction-Level
- Thread-Level
- Processor-Level

Bit-Level Parallelism

Word size may mean

- unit of transfer between processor & memory
- memory address space capacity
- integer size
- single precision floating point number size

Word size is typically > 1 bit, thus there is multiple data.

Instruction-Level Parallelism (most common)

Fairly limited due to data / control dependencies.

Pipelining (across time)

- Split instruction execution into multiple stages: Fetch (IF), Decode (ID), Execute (EX), Write-Back (WB)
- Allow multiple instructions to occupy different stages in the same clock cycle
  - provided there is no data / control dependencies
- Number of pipeline stages == maximum number of instructions in parallel = maximum achievable speedup

Disadvantages:

- Conditional branching requires speculation (branch prediction)
- Need to find independent instructions (often not the case)
- Frequently has "bubbles" of wasted cycles

Superscalar (across space)

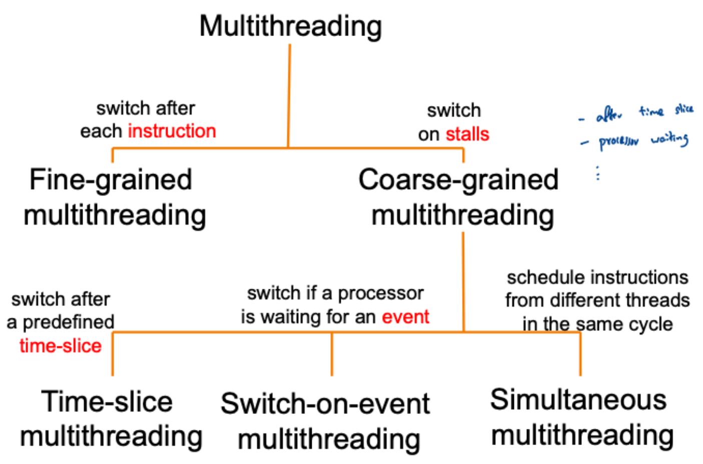
- Duplicate the pipelines by allowing multiple instructions to pass through the same stage.
- More challenging to schedule
- Instructions must be independent and come from the same thread! (hard to find)

Most modern processors are superscalar

Thread-Level Parallelism

Processor provide hardware support through *Simultaneous Multithreading* (SMT) aka hyper-threading.

- Each SMT has its own set of *processor stuff* (program counter, registers, ...)
- No need to context switch
- However, this increases contention, so typically each core only has 2 SMTs


Multithreading
switch after each instruction
switch on stalls
- after time slice
- processor waiting
⋮
Fine-grained multithreading
Coarse-grained multithreading
switch after a predefined time-slice
switch if a processor is waiting for an event
schedule instructions from different threads in the same cycle
Time-slice multithreading
Switch-on-event multithreading
Simultaneous multithreading

Processor-Level Parallelism

- Each process has an independent context, mapped to multiple processor cores.

Flynn's Parallel Architecture Taxonomy

Instruction stream: a single **execution flow**, i.e. single Program Counter.

Data stream: data being manipulated by instruction stream.

- SISD
  - single instruction stream is executed
    - ∗ each instruction works on single data
  - most uniprocessors (e.g. personal computers from 1980s) are SISD
    - ∗ even if processor has pipelining, it's still SISD
- SIMD
  - single instruction stream is executed
  - each instruction works on multiple data
  - same instruction is broadcasted to all physical ALUs
    - ∗ think: applying function to entire vector
  - not great for divergent executions
  - e.g. SSE, AVX instructions in Intel x86, GPGPUs
- MISD
  - multiple instruction streams are executed
  - all instructions work on the **same data**
  - e.g. (arguably) students in exam hall (each student is an instruction; same exam paper is same data)
  - e.g. space shuttle CPUs (compute same thing on more than one CPU for redundancy)
  - not practical!
- MIMD
  - each PU fetch its own instruction
  - each PU operates on its own data
  - e.g. most modern multi-core laptops / computers

Stream Processors (nvidia GPU) use SIMD + MIMD

- set of threads executing same code (SIMD)
- multiple set of threads executing in parallel (MIMD)

Multicore Architecture

- Hierarchical
  - Cache size increases from leaves (L1) to root (L3)
- Pipelined
  - Data elements are processed in a pipeline
  - useful if same computation steps are applied to a long sequence of data elements
    - ∗ used commonly in network routers and graphics processors
- Network-based
  - Cores and local caches / memories are connected via an interconnection network, i.e. many-to-many instead of one-to-one

Memory Organisation

Shared Memory Systems

- Uniform Memory Access (UMA) vs Non-Uniform Memory Access (NUMA)
  - whether delay to memory is uniform
  - UMA: suited for small number of processors - due to contention
  - NUMA: typically, local memory is faster than remote memory
- Presence of local cache with cache coherence protocol (CC / NCC)
  - cache coherence ensures that same shared variable in different caches are kept correct
- ccNUMA = CC + NUMA
  - each node has cache memory to reduce contention
- Cache-only Memory Access (COMA): ccNUMA without separate memory for each core

In reality, most systems are a *hybrid* (Distributed-Shared memory).

Performance Optimisation

Performance Aspects

Performance Goals

- Response time
- Speedup

- Throughput
- Utilisation

Evaluation approach should be consistent with the goals

Performance Quality Aspects

- Quality attributes
  - Scalability
  - Reliability
    - Correctness in high workload or failures
  - Resource usage
- Different workloads
  - normal load or overload
    - input size, #connections, weight of queries (heavy or light)
    - historical data / experience is needed to know baseline / expected metrics under heavy load (observability)
- Timeline
  - Not time sensitive: testing before release
  - Time sensitive: incident performance response
- Terminology
  - Latency = waiting time or response time (waiting + actual work)
  - Throughput = rate of work done (e.g. bytes per second OR transactions per second)
  - Utilisation = CPU utilisation, or memory utilisation
  - Saturation = degree to which a resource has queued work it cannot service
  - Bottleneck = resource limiting performance of the system (in systems programming)

Performance Perspectives

- System administrators: maximise utilisation
- Application developers: maximise performance / output

Memory is the most expensive but cloud providers sell by CPU cores, not memory. So, they divide memory to CPUs and bundle with CPUs. However, leftover memory is often wasted!

Methodologies

- Anti-methods (bad!)
  - lack of a deliberate methodology
    - tuning things too early
  - random change anti-method
  - streetlight anti-method
    - look for only obvious issues (familiar things, found on Internet, found at random)
  - blame-someone-else anti-method
  - drunk man anti-method
    - tune things randomly until problem goes away
    - tune wrong software (e.g. OS instead of application)

Problem Statement Method

1. What makes you think there is a performance problem?
2. Has this system ever performed well?
3. What has changed recently? (Software? Hardware? Load?)
4. Can the performance degradation be expressed in terms of latency or run time?
5. Does the problem affect other people or applications (or is it just you)?
6. What is the environment? Software, hardware, instance types? Versions? Configuration?

USE Method

- Utilisation
- Saturation
- Errors

Good to have system monitoring tools

Monitoring Tools Up to 20% overhead for cloud monitoring (not insignificant!)

Useful for:

- capacity planning
- quantifying growth
- showing peak usage

Use historical values (time series data) as a way to compare against current metrics

Performance Analysis in Linux

- `uptime`
  - check load average changes
- `dmesg | tail`
  - check performance issues
- `vmstat 1`
  - check CPU and memory utilisation (take note of user mode)
- `mpstat -P ALL 1`
  - check CPU balance
- `pidstat 1`
  - check usages by process
- `iostat -xz 1`
  - check disk I/O
- `free -m`
  - look at memory usage
  - note buffers and cache line
- `sar -n DEV 1`
  - check network throughput
- `sar -n TCP,ETCP 1`
  - check TCP / ETCP
  - check re-transmissions
  - NOTE: ETCP is External Transmission Control Protocol (not a very common thing?)
    - used in fast LAN when bits are cheap, latency is expensive
    - https://github.com/mgrosvenor/etcp
- `top`
- `strace <cmd>`
  - trace syscalls

Types of Tools

- Observability (real / active workload)
  - watch under actual workload
  - usually safe, depending on overhead
  - do: insert timing statements, checking performance counters
- Static (no workload)
  - at rest rather than active workload
  - generally safe
- Benchmarking (synthetic workload)
  - load testing
  - production tests can cause issues due to contention with real system
  - not very accurate (hard to create real workload), but useful for comparing systems (e.g. Top500)
- Tuning
  - change default settings
  - changes could hurt performance now or even later with load

Tools can be system-wide or per-process

- Fixed counters
  - Kernels maintain and expose various **counters** for system statistics
- Event-based counters (enabled as needed)
  - **Profiling**: collects a set of samples / snapshots
  - **Tracing**: instruments every occurrence of an event, stored for later analysis / summary
    - a lot of overhead (~ 3x slower)

Valgrind shadows all program values (registers & memory)

- 4x overhead
- requires serialising threads
- Valgrind memcheck has 10 to 20x overhead (`valgrind --tool=memcheck <prog>`)

Alternative: **sanitisers** - compilation-based approach (`-fsanitize=address`)

- ~2x overhead
- ThreadSanitizer (data races), MemorySanitizer (uninitialised reads), UndefinedBehaviorSanitizer (integer overflow, null pointer), LeakSanitizer (memory leaks)

Benchmarking is often wrong, and results are usually misleading…

Memory

- Fetch data from *memory* less often!
  - Reuse data previously loaded by the same thread (temporal locality)
  - Share data across threads (inter-thread cooperation)
- Favour performing math compared to storing / loading values

Tips & Tricks

- Start with simplest solution and **measure**
- Determine bottlenecks
  - Instruction-rate limited: add "math" (non-memory instructions)
  - Memory bottleneck: remove "math", load same data
  - Locality of data accesses: change all array accesses to `A[0]`
  - Synchronisation overhead: remove all synchronisation

Parallelism Programming Models

average number of units of work

- MIPS = million instructions per second
  - easily manipulated by having small fast instructions (but need more instructions to do one operation)
- MFLOPS = million floating point operations per second
  - no differentiation between different floating point operations
  - manipulated by optimising for floating point operations
- average number of threads / processes per second

Work = tasks + dependencies

Data Parallelism

- **Partition the data** among the PUs.
- Each PU carries out *similar operations* on its subset of the data.

Same operation applied to **different elements** of a dataset. Operations should be independent, i.e. independent iterations of a `for` loop.

e.g. Loop Parallelism (subtype of Data Parallelism)

```
omp_set_num_threads(5);

int i, j, k;

#pragma omp parallel for \
    shared(a, b, result) private(i, j, k)
for (i = 0; i < size; i++)
```

```
{
  // work done here in parallel
}

#pragma omp barrier

#pragma omp atomic
x += 1;

// #pragma omp single // only one thread (may be non-master)
#pragma omp master
{
    // only run on master thread
}

// dynamically scheduled (can use `static` too)
// each thread takes 4 contiguous iterations
#pragma omp parallel for schedule(dynamic, 4)
for ()
{
    // work done
}

#pragma omp parallel sections
{
    #pragma omp section
    {
        // do work in one section
    }

    #pragma omp section
    {
        // do work in another section
    }
}
```

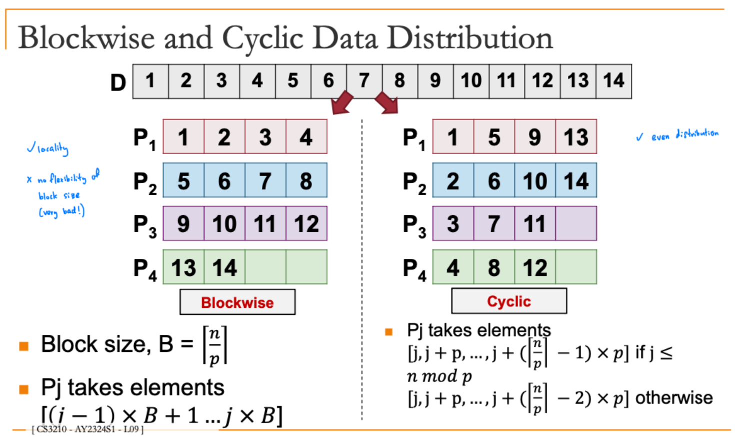
Problem: Let each thread work on subarray.

```
int array[] = {};
int array_len = 12;

#pragma omp parallel num_threads(threads)
{
    int subarray_length = array_len / threads;
    int thread_id = omp_get_thread_num();

    #pragma omp for
    for (int i = subarray_length * thread_id; i < subarray_length *
    {
        // do work
    }
}
```

Common model on MIMD and SPMD (MIMD, but allows conditional branches based on data).

1D Data Distribution

- Blockwise
  - + locality
  - - no flexibility of block size
  - - uneven distribution
- Cyclic
  - + even distribution
  - - no locality



Blockwise and Cyclic Data Distribution

D	1	2	3	4	5	6	7	8	9	10	11	12	13	14

✓ locality

x no flexibility of block size (very bad!)

P_1	1	2	3	4
P_2	5	6	7	8
P_3	9	10	11	12
P_4	13	14		

Blockwise

✓ even distribution

P_1	1	5	9	13
P_2	2	6	10	14
P_3	3	7	11	
P_4	4	8	12	

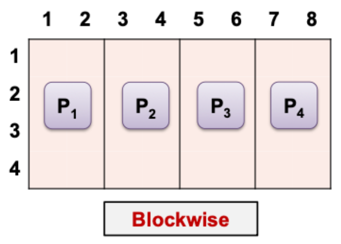
Cyclic

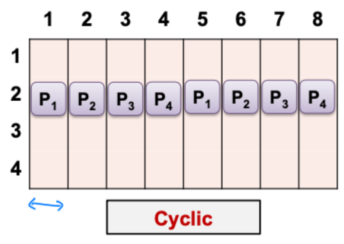
- Block size, B = $\lceil \frac{n}{p} \rceil$
- Pj takes elements $[(j-1) \times B + 1 \dots j \times B]$
- Pj takes elements $[j, j+p, \dots, j + (\lceil \frac{n}{p} \rceil - 1) \times p]$ if j ≤ $n \bmod p$; $[j, j+p, \dots, j + (\lceil \frac{n}{p} \rceil - 2) \times p]$ otherwise

[CS3210 - AY2324S1 - L09]

2D Data Distribution

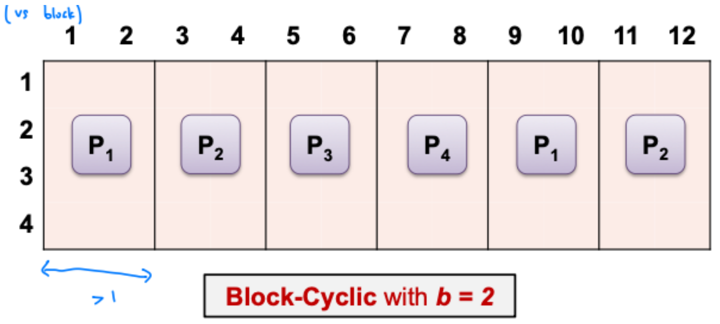
(examples are using *column dimension* -> processor gets entire column)





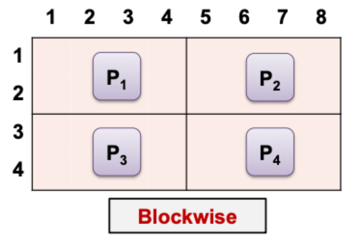
- Block-Cycle (form blocks of size b)
  - maintains locality
  - maintains even distribution
  - granularity of task sizes are smaller (compared to Block)
- Cyclic = Block-Cycle, but $b = 1$

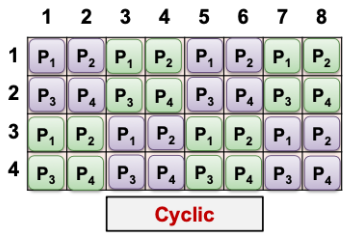
(vs block)

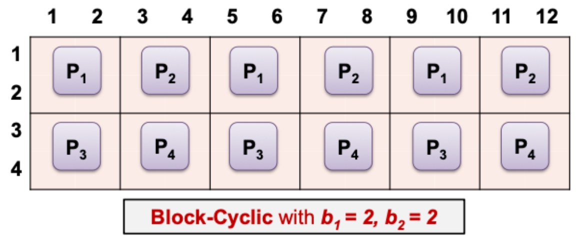


Checkerboard Distribution

Similar to actual mesh organisation of processors







Layout matters if you need data from neighbouring processors!

Simple optimisations:

- aggregate transfers to transfer only once (reduce overhead)
- include border cells (overlapping regions of data)

If block size is smaller, more communication.

If block size is larger, less parallelisation, and might run into uneven workloads.

Task Parallelism

- **Partition the tasks** among the PUs.

Various granularities: single statement, series of statements, loops, or function calls.

Tasks should be as independent as possible (reduce dependencies)

Task Dependency Graph

Use a DAG where:

- node: a task, value = *expected* execution time
- edge: control dependency between tasks

Properties:

- **critical path length**: longest path (slowest completion time)
  - minimise this as much as possible
- degree of concurrency / average concurrency / speedup: (total work) / (critical path length)
  - *up to* `degree` tasks executing concurrently
  - higher is better
- maximum concurrency: maximum number of concurrent tasks at any point in time

* need to consider number of PUs! No general way to allocate tasks, try out different possibilities.

- lay out critical path first
  - try to reduce blocking of critical path!

Automatic Parallelisation

Using compilers to perform decomposition and scheduling.

- Dependence analysis is difficult for pointer-based and indirect addressing

- Execution time of function calls and loops with unknown bounds is hard to predict

Functional Programming Languages

- Easier to enable parallel execution, don't need to add new language constructs
- However, hard to choose right level of recursion

Parallel Programming Patterns

Just design patterns.

- Fork-Join
- Parbegin-Parend
- SIMD Pattern
- SPMD Pattern
- Master-Worker
  - more for homogeneous tasks
- Task Pool
  - more for heterogeneous tasks
- Producer-Consumer
- Pipelining
  - for task parallelism

Fork-Join

- No / low communication across tasks needed
- Tasks execute same or different program parts
- Tasks can join parent at different times

Parbegin-Parend

Fork-Join BUT

- usually executes same code (but might have conditional statements)
- all forks done at **same time**, all joins done at **same time**

e.g. OpenMP or compiler directives

SIMD Pattern

- single **instructions** executed synchronously by different threads on different data
- similar to Parbegin-Parend but all threads execute **same instruction** at **same time**

e.g. AVX / SSE instructions on Intel (operates on 4x 32-bit values)

SPMD Pattern

Parbegin-Parend BUT **same program** on different cores on different data

However, different threads can execute different parts of the program because of:

- different clock speeds (not lockstep)
  - no implicit synchronisation between threads
- conditional `ifs`

e.g. GPGPUs

Master-Worker

- Master: responsible for coordination, initialisation, timings, outputs
- Worker: waits for instruction from master

Task Pool (Work Pool)

- new tasks are explicitly retrieved by worker threads
- threads can generate new tasks and insert into the pool
- access to pool must be synchronised

Pros / Cons:

- + useful for adaptive and irregular applications
- + overhead of thread creation is independent of program (statically generated at the start)
- - for fine grained tasks, overhead of task retrieval / insertion is expensive

Producer-Consumer

Producer produces, Consumer consumes

- Need to synchronise

Pipelining (Stream Parallelism)

- Data stream flows through pipeline stages
- Common in network routers / processors

Models of Coordination / Communication

Shared Address Space (matches Shared Memory Systems - UMA, NUMA)

- communication abstraction
  - read / write from shared variables
    - ∗ cost might be uneven (and might not be clear to programmer)
  - mutual exclusion via locks (to avoid race conditions)
- requires **hardware** support to be efficient
  - difficult to scale! (memory contention)

Data Parallel

- Maps a function (no side-effects) onto a dataset
- No communication among distinct function calls (allows for easy parallelism)
- Very rigid computation structure (but modern languages are looser)

but modern data-parallel languages do not enforce this structure anymore

Message Passing (matches Distributed Memory Systems)

- Tasks have separate address spaces
  - Communicate through **messages** (MPI, Go channels)

Foster's Design Methodology

1. Partitioning
   - Divide computation and data into independent pieces
     - Data parallelism: divide data into equally sized pieces
       - ∗ need to associate computations with data
     - Task parallelism: divide computation into pieces
       - ∗ need to associate data with computations
   - 1D, 2D, 3D...
   - at least 10x more primitive tasks than cores
   - minimise redundant computations an data storage
   - number of tasks should be an increasing function of problem size
2. Communication: tasks might need to communicate
   - Local Communication
     - tasks need data from small number of neighbours (minimise this)
   - Global Communication
     - most tasks contribute data OR tasks talks to many other tasks
   - Ideally, have enough work to do while communicating (overlap work and communication)
3. Agglomeration: combine tasks into larger tasks (#tasks > #cores)
   - Reduce cost of task creation
   - Reduce number of sends and receives (communication)
     - Improve locality of algorithm
   - Maintain scalability of program, simplify programming
4. Mapping: assign tasks to PUs
   - Maximise processor utilisation: place tasks on different PUs
   - Minimise inter-processor communication: place tasks that need to communicate on same PUs
   - May be performed by OS or user
   - Optimal mapping is NP-hard, use heuristics to improve it

Granularity Tradeoffs

- Smaller granularity: introduces parallelism overhead (from creating / merging / maintaining threads + communication overhead)
- Larger granularity: hard to exploit parallelism (uneven tasks + not enough tasks for all cores)

In context of OpenMP, task where a single element is operated on is probably too small. Task should ideally operate on entire (or a chunk of) row / column.

Performance

- Response time (user): duration of program execution is shorter
- Throughput (computer manager): more work can be done in same duration

Response Time (Wall Clock Time)

- user CPU time: time CPU spent on executing program
- system CPU time: time CPU spent on OS routines
- waiting time: I/O waiting time and other programs (because of time sharing)

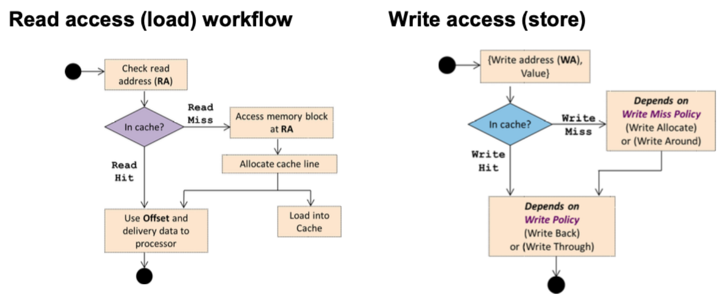
For program A:

$$\mathrm{Time}_{\mathrm{user}}(A) = N_{\mathrm{cycle}}(A) * \mathrm{Time}_{\mathrm{cycle}}$$

$$N_{\mathrm{cycle}}(A) = \sum_{i=1}^{n} n_i(A) * \mathrm{CPI}_i$$

CPI_i = average number of CPU cycles needed for instructions of type I_i

$$\text{Simplified: } \mathrm{Time}_{\mathrm{user}}(A) = N_{\mathrm{instr}}(A) * \mathrm{CPI}(A) * \mathrm{Time}_{\mathrm{cycle}}$$


Read access (load) workflow
Check read address (RA)
In cache?
Read Miss
Access memory block at RA
Allocate cache line
Read Hit
Use Offset and delivery data to processor
Load into Cache
Write access (store)
{Write address (WA), Value}
In cache?
Write Miss
Depends on Write Miss Policy (Write Allocate) or (Write Around)
Write Hit
Depends on Write Policy (Write Back) or (Write Through)

$$\text{with memory access: } \mathrm{Time}_{\mathrm{user}}(A) = (N_{\mathrm{cycle}}(A) + N_{\mathrm{mm\_cycle}}(A)) * \mathrm{Time}_{\mathrm{cycle}}$$

$N_{\mathrm{mm\_cycle}}(A)$ = number of additional clock cycles due to memory accesses

For a one-level cache:

$$N_{\mathrm{mm\_cycle}}(A) = N_{\mathrm{read\_cycle}}(A) + N_{\mathrm{write\_cycle}}(A)$$

$$N_{\mathrm{read\_cycle}}(A) = N_{\mathrm{read\_op}}(A) * R_{\mathrm{read\_miss}}(A) * N_{\mathrm{miss\_cycles}}(A)$$

$R_{\mathrm{read\_miss}}(A)$ = read miss rate

NOTE: typically, write is slower than read, so number of cycles differ, but generally assume same for estimates

$$T_{\text{read\_access}}(A) = T_{\text{read\_hit}}(A) + R_{\text{read\_miss}}(A) * T_{\text{read\_miss}}(A)$$

NOTE: for two level cache: just expand $T_{\text{read\_miss}}$ recursively

Parallel Execution Time

Includes:

- executing local computations
- exchange of data between PUs
- synchronisation between PUs
- waiting time (due to unequal load)

Parallel Program Cost

$$C_p(n) = p * T_p(n)$$

A parallel program is **cost-optimal** if it executes the same total number of operations as the **fastest** sequential program.

Parallel Program Speedup

$$S_p(n) = \frac{T_{\text{best\_seq}}(n)}{T_p(n)}$$

Theoretically, $S_p(n) <= p$, but superlinear speedup can occur due to caching! (each smaller task's data fits in cache)

Parallel Program Efficiency

$$E_p(n) = \frac{T_{\text{best\_seq}}(n)}{p * T_p(n)}$$

Ideal: $E_p(n) = 1$

Scalability

Size of problem

- Impacts
  - load balancing
  - overhead
  - arithmetic intensity
  - locality of data access
- Application dependent!

Small problem size:

- parallelism overheads > benefits
- problem size may be appropriate for small machine, but too small for large machines

Large problem size:

- key working set does not fit in cache / memory (causes thrashing) / disk (GG)

Constraints

Resource-oriented properties

- problem constrained scaling (PC): solve **same problem** faster
- time constrained scaling (TC): complete **more work** in fixed amount of time
- memory constrained scaling (MC): run the **largest problem** without overflowing main memory

Amdahl's Law (1967)

(for fixed-size problems OR constant f with increasing problem size, i.e. sequential section increases with problem size)

Speedup of parallel execution is limited by the fraction (f) of the algorithm that cannot be parallelised.

f: sequential fraction, or fixed-workload performance

$$S_p(n) = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

No point in making large parallel computers. Instead, try to reduce f.

However, f is not a constant, but is commonly a function of problem size n.

For an effective parallel algorithm (Amdahl's Law is circumvented for large problem sizes):

$$\lim_{n\to\infty} f(n) = 0 \implies \lim_{n\to\infty} S_p(n) = p$$

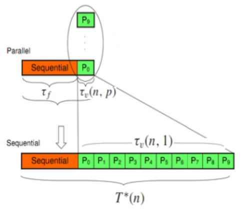
Gustafson's Law (1988)

(for varied problem size OR decreasing f with increasing problem size, i.e. sequential section is fixed)

If f decreases when problem size increases, then $S_p(n) \leq p$.

Gustafson's Law

- τ_f = constant execution time for sequential part
- $\tau_v(n, p)$ = execution time of the parallelizable part for a problem of size **n** and **p** processors



$$S_p(n) = \frac{\tau_f + \tau_v(n, 1)}{\tau_f + \tau_v(n, p)}$$

- Assume parallel program is perfectly parallelizable (without overheads), then

$$\tau_v(n, 1) = T^*(n) - \tau_f \text{ and } \tau_v(n, p) = (T^*(n) - \tau_f)/p$$

$$S_p(n) = \frac{\tau_f + T^*(n) - \tau_f}{\tau_f + (T^*(n) - \tau_f)/p} = \frac{\frac{\tau_f}{T^*(n)-\tau_f} + 1}{\frac{\tau_f}{T^*(n)-\tau_f} + \frac{1}{p}}$$

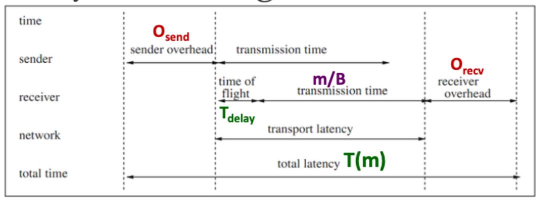
- If T*(n) increases strongly monotonically with n, then $\boxed{\lim_{n\to\infty} S_p(n) = p}$

Gustafson's Law matters when the workload is increased. In contrast, Amdahl's Law is for fixed-workload performance.

Gustafson's Law: How much you need to increase the problem size by -> to reach some speedup

Communication Time

Total Latency of a Message of Size m



$T(m) = O_{send} + T_{delay} + m/B + O_{recv} = T_{overhead} + m/B = T_{overhead} + t_B * m$

where B is network bandwidth,
T_{delay} = time first bit to arrive at receiver
no checksum error and network contention and congestion,
$T_{overhead}$ (= $O_{send} + T_{delay} + O_{recv}$) is independent of the message size;
t_B (=1/B) is the byte transfer time

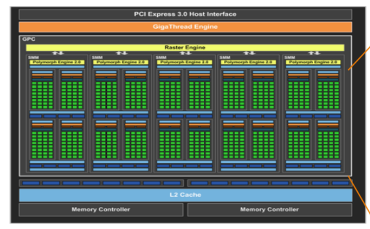
GPGPU (General Purpose GPU)

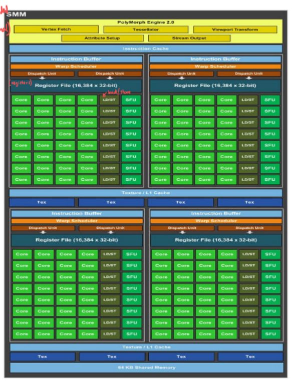
* try to avoid double precision unless required, because it needs 2 cores to work together (approximately halves performance)

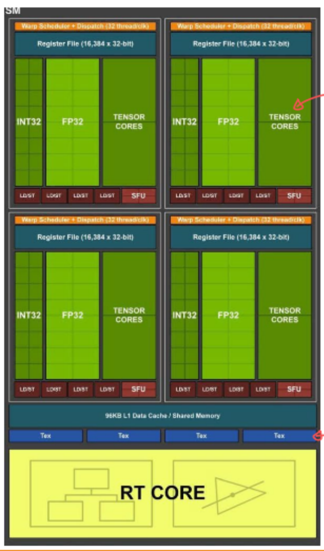
Architecture

① just ONE example, variations in generation
② use case (i.e. embedded vs personal)

- GPU
- SMs







SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Tensor cores (pretty much)
only used for ML computation
→ ML libraries optimise to use these

→ fairly new (~ 2 years)

← Texture memory

CUDA (Compute Unified Device Architecture)

- simple extension to C and reduced C++
- compiled, but still requires a CUDA runtime API
- NVCC outputs:
  - C code (host CPU code)
  - PTX (object code / PTX source interpreted at runtime)
- Definitions
  - device = GPU
  - host = CPU
  - kernel = function that runs on device
  - SM = streaming multiprocessor
- CUDA threads are extremely lightweight (very little creation overhead + instant switching)
- one or more kernels are executed together
- SPMD model
  - threads execute in lockstep
  - threads can share results / memory accesses within **same block**

Thread Blocks (logical grouping of threads)

- In a block: shared memory, atomic operations, barrier synchronisation
- In different blocks: 0 communication

Hardware can schedule thread blocks to any processor. Each block executes on **one** SM. Multiple blocks can reside on same SM concurrently.

- limited by SM resources
- register file is partitioned amongst resident threads
- shared memory is partitioned amongst resident thread blocks

Sizes

- Should be multiple of warp size (otherwise, there will be threads doing 0 work)
- minimum of 64 threads (for A100): more might be better
- common sizes: 128, 256, 512
  - 128 is common
  - 16*16=256 is also common
- for modern GPUs, up to 1024 threads

Grids

Sizes

- Should be greater than number of SMs (`prop.multiProcessorCount`) - 14 on A100

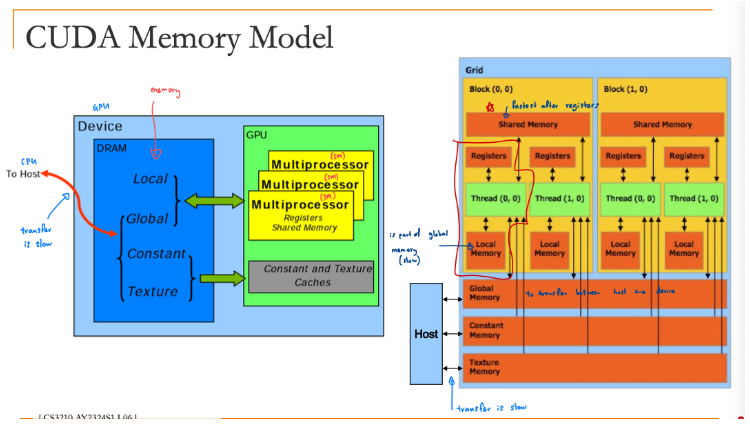
Warps (subset of blocks)

Warp = group of 32 parallel threads

Blocks are always split into warps in the same way, i.e. first 32 in first warp, next 32 in second warp, . . .

Each warp executes one **common instruction** at a time (lockstep). Diverges with `if / else`, execution will alternate between different conditional branches. If one thread isn't ready, the entire warp blocks.

Number of warps should be larger than number of SMs. So that all SMs have at least one warp to execute. (improve occupancy)



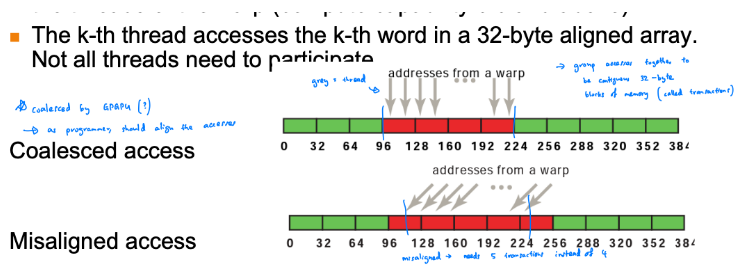
CUDA Memory Spaces

- Data is explicitly transferred from host <-> device
- Global memory is cached
- Shared memory is not cached (because it is the cache. Shared Memory Unit –(partitioned into)-> L1 Cache + Shared Memory)
- Local memory: automatic array variables allocated by compiler (cached)
- Constant memory: useful for uniformly-accessed read-only data (cached)
- Texture memory: useful for spatially coherent random-access read-only data (cached, provides filtering, address clamping, wrapping)

Prefer register > local > shared > global.

Coalesced Access to Global Memory

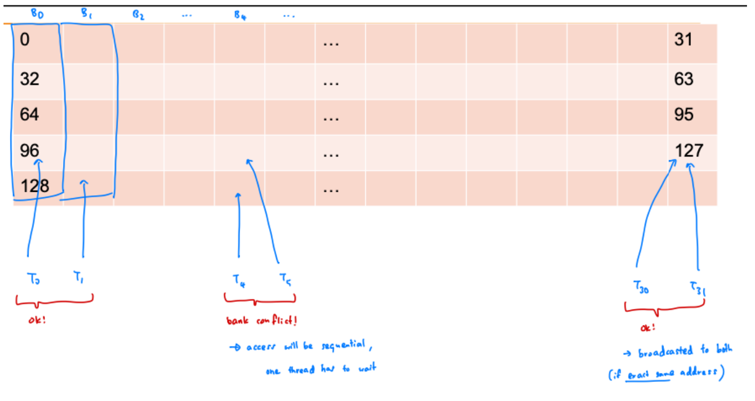
Simultaneous accesses to **global memory** by threads in a **warp** are **coalesced**.



- The k-th thread accesses the k-th word in a 32-byte aligned array. Not all threads need to participate

Shared Memory

- higher bandwidth and lower latency than local and global
- divided into equally sized **banks**
- addresses from different banks can be accessed together
- bank conflict: two addresses of a memory request falls into same bank (will be serialised)
- bank has bandwidth of 32 bits per clock cycle, and successive 32-bit words are assigned to successive banks
  - warp size = 32 threads, number of banks = 32



Strided Accesses

Stride = "gap" between consecutive elements that are accessed. Stride of 1 = consecutive, stride of 2 = skip one element.

- threads within a warp access words in memory with a stride of 2
- stride of 2 results in 50% load / store efficiency

CUDA Optimisation

- optimise memory usage to achieve maximum memory bandwidth
- maximise parallel execution: **data parallelism** by restructuring algorithm
- optimise instruction usage to maximise **instruction throughput**
  - use arithmetic instructions with high throughput - prefer single-precision, avoid integer division and modulo (use bitwise operations instead) - use signed loop counters
  - avoid divergent branches in same warp
  - reduce number of instructions

CUDA Memory Optimisation

- minimise data transfer between host and device
  - even for kernels that have no speedup over CPU
  - batch small transfers together
  - use page-locked (or pinned) memory transfer
    * pinned memory is not cached
    * use zero-copy feature to allow threads to directly access host memory
- ensure global memory accesses are coalesced if possible
- minimise global memory accesses by using shared memory
- minimise bank conflicts in shared memory accesses
- use `cudaMemcpyAsync` instead of `cudaMemcpy` if possible
- avoid multiple contexts per GPU within same CUDA application
  - don't let multiple CUDA application processes access GPU concurrently! (think: Slurm –exclusive)

CUDA Syntax

```
// qualifiers
__host__
__device__ // lifetime: application, stored in global memory
__global__
__managed__ // does NOT require explicit data transfer

// device code qualifiers
__constant__ // lifetime: application, same as __device__
             // but cached and read-only
__shared__ // lifetime: block, stored in on-chip shared memory,
           // read / write by all threads in block

// device code unqualified variables, lifetime: thread
// scalars, built-in vector types are stored in registers
// arrays of more than 4 elements,
  // or runtime indices stored in local memory

__global__ void kernel(float *a)
```

```
{
    const unsigned long long id =
        threadIdx.x + blockIdx.x * blockDim.x;
}

int grid = 32; // allowed in place of dim3
dim3 block { 16, 16 };

// smem = bytes per block (shared memory)
kernel<<<dim3 grid, dim3 block>>>();
kernel<<<dim3 grid, dim3 block, int smem, int stream_id>>>();

cudaMalloc(void **pointer, size_t nbytes);
cudaMemset(void *pointer, int value, size_t count);
cudaFree(void *pointer);

cudaMemcpy(void *dst, void *src, size_t nbytes,
    enum cudaMemcpyKind direction);
cudaMemcpy(device_mem, start, num_elements * sizeof(int),
    cudaMemcpyHostToDevice);
cudaMemcpy(start, device_mem, num_elements * sizeof(int),
    cudaMemcpyDeviceToHost);

// barrier: synchronises all threads in a block (run on device)
__syncthreads();

// synchronises device & host,
// can be used for flushing print statements (run on host)
cudaDeviceSynchronize();
```

Cache Coherence

Caches

- larger cache increases access times (because of increased addressing complexity), but reduces cache misses
- cache line (block size): fixed length of data transferred between main memory and cache
  - larger blocks reduces number of blocks (less to manage), but replacing blocks take longer => block size should be small
  - larger blocks increase chance of spatial locality cache hits => block size should be large

Typical sizes for L1 cache lines are 4 or 8 memory words (one word = 4 bytes)

Cache: Write Policy

- Write-through: write access is immediately transferred to main memory
  - always newest value, but slows down program (can use a write buffer to mitigate)
- Write-back: write operation performed only in cache
  - Write is performed to main memory when cache block is replaced
  - Use dirty bit to track
  - less write operations, but memory may contain outdated / dirty entries

Cache Coherence Problem: multiple copies of same data exists on different caches. For any update, other processors may still see outdated data!

Memory Coherence

Coherence ensures each PU has consistent view of each memory location through its local cache

- All PUs must agree on order of reads / writes to **same** memory location

Three properties:

1. Program Order
   1. PU_1 writes to X
   2. No write to X from other PUs
   3. PU_1 reads X (should be value from step 1)
2. Write Propagation (writes should be eventually visible)
   1. PU_1 writes to X
   2. No write to X from other PUs
   3. PU_2 reads X (should *eventually* be value from step 1)
3. Transaction Serialisation (all writes to a location are seen in the same order by all PUs)
   1. Write v_1 to X
   2. Write v_2 to X
   3. No PU can read X as v_2 first, then v_1

Mostly maintained by HW now (with some OS + compiler help).

However, programmer is still responsible for program synchronisation.

Cache Line Sharing Status

- Snooping based (most common for architectures with bus)
  - No centralised directory, each cache tracks locally
  - Caches snoops on the bus for operations done by other caches
    - ∗ to update its own cache
    - ∗ ensures Write Propagation and Transaction Serialisation
  - Granularity: cache block
- Directory based (common for NUMA architectures)
  - sharing status is kept in centralised location

Cache Coherence Implications

- CC leads to increased memory latency
- CC lowers hit rate in cache (because of updates from other processes)
- Cache ping-pong can happen (multiple PUs read and modify same global variable)
- False sharing
  - two PUs write to different memory addresses on the same cache line!
  - hard to detect (but very common for structs / arrays)
  - can be identified using `perf c2c`

Memory Consistency

Constraints the order in which memory operations performed by one thread become visible to other threads for **different** memory locations.

- Reorder instructions for better performance (hide write latencies)
  - Consistency model dictates what is allowed
- Consistency model is used by:
  - programmers to reason about correctness and program behaviour
  - system / compiler designers to decide reordering of memory operations possible by hardware and compiler
- Extra notes
  - GPGPUs have weak consistency
  - Intel CPUs have strong consistency
  - C++ >= 11 has strong memory model, but might have overhead if architecture doesn't natively support

Sequential Consistency (SC) Model - Lamport 1976

- TL;DR: acts like all PUs access ONE memory space
  - so, all PUs see same ordering of updates! (even for different memory locations)
- Global result of all memory accesses of all PUs appears to all PUs in **same sequential order** irrespective of arbitrary interleaving by different PUs
  - effect of each operation must be visible to all PUs before next memory operation on any PU
- All four memory orderings are preserved

Relaxed Consistency Model

- R -> W (WAR): anti-dependency
- W -> W (WAW): output dependency
- W -> R (RAW): flow dependency

Used to hide memory latency by overlapping memory access operations with other independent operations (and buffering writes)

- Data dependencies must be preserved
  - two operations are accessing the **same memory location**
    - ∗ Write X, Read X (cannot re-order!)
  - if different location, NOT dependencies
    - ∗ Write X, Read Y (can re-order!)
- Cache coherence must be preserved

* cannot swap for control instructions (IF, JUMP) - (if, for, while)

NOTE: all models provide overriding mechanisms for programmers

Total Store Ordering (TSO)

- Remove W -> R
- Reads by other PUs cannot return new value of A until write to A is **observed** by all PUs (write atomicity)
  - aka if some PU x sees the update, ALL PUs see it

Processor Consistency (PC)

- TSO, but no write atomicity!
- Return the value of any write (even from other PUs) before the write is observed by all PUs

In reality, put writes in a *write buffer*, and flush later

Partial Store Ordering (PSO)

- Remove W -> R (same as TSO)
- Remove W -> W

In reality, writes can bypass earlier writes (to different locations) in write buffer

NOTE: CANNOT reorder `A = B, C = D` because `A = B` is a read (of B), then write (of A).

- read B, write A, read D, write C
- under PSO, can become:
  - read B, read D, write A, write C
  - read B, read D, write C, write A

The reads must stay in the same order!

Interconnections

- Mesh = grid of PUs, only connected to adjacent PUs
- Torus = Mesh with wrap-around

Parallel Sorting Algorithms

Odd-Even Transposition Sort Algorithm

1. Sort even pairs (i, i+1, where i is even)
2. Sort odd pairs (i, i+1, where i is odd)

N rounds needed, time complexity = $O(N)$

Shear Sort Algorithm

- Order N PUs into a 2D mesh, final sorting position is a "snake" (start top left, go right to end, go down by 1, go left to end, go down by 1, ...)

1. Row sorting
   - odd rows sort in ascending order
   - even rows sort in descending order
2. Column sorting
   - all columns sort in ascending order (top to bottom)
3. Repeat until sorted

For N numbers, $\log_2 N + 1$ phases

Using Odd-Even Transposition Sort as subroutine, time complexity is $O(\sqrt(N) * (\log_2 N + 1))$

Interconnections Topology

- Direct interconnection (PUs are connected by direct edges)
  - aka Static or Point-to-Point
  - usually, endpoints are of the same type (e.g. both CPU, or both memory)
- Indirect Interconnection (PUs are connected through a network of switches)

Treat topology as CS graphs

- Diameter $\delta(G)$: maximum distance between any two nodes
  - Usefulness: small diameter ensures smaller distances for message transmission
- Degree $g(v)$ = number of *direct neighbours* of some node v
  - Degree $g(G)$ = maximum degree of nodes in network G
  - Usefulness: small node degree reduces node hardware overhead (makes it easier to scale -> due to heat and physical space constraints)
- Bisection width $B(G)$ = minimum number of *edges* that must be removed to divide network G into **two equal halves**
  - Bisection bandwidth $BW(G)$ = total bandwidth available between the two bisected portions of the network
  - Usefulness: measure of capacity of a network
- Node connectivity $nc(G)$ = minimum number of nodes that must fail to disconnect the network
  - Usefulness: determine robustness / resilience of network to failure
- Edge connectivity $ec(G)$ = minimum number of edges that must fail to disconnect the network
  - Usefulness: determine number of independent paths between any two nodes

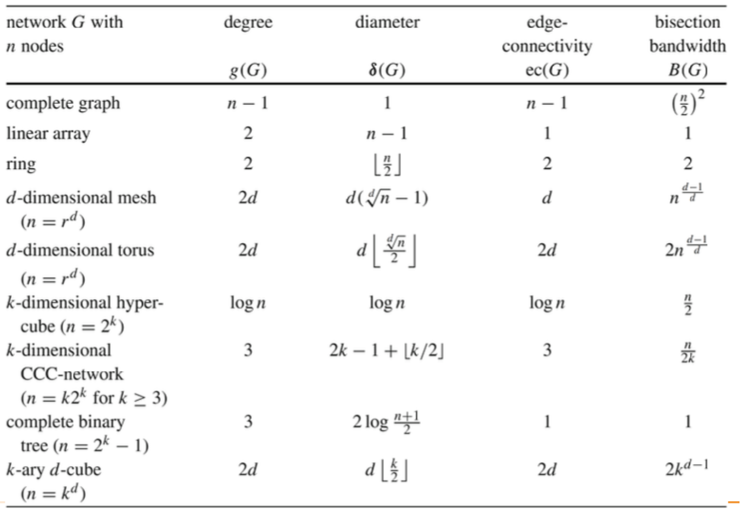
Hypercubes

Hypercubes can embed binary trees, aka it is efficient to use hypercube interconnect for tree-style communication.

Cube-Connected Cycles (CCC)

From a k-dimensional hypercube ($k \geq 3$)

- substitute *each node* with a cycle of k-nodes
- each of the k-nodes take one of the original k links
  - total nodes = $k2^k$
- each node (X, Y) is connected to: where X = original index, Y = position in cycle (from 0 to k-1)
  - (X, (Y+1) mod k) -> forming cycle
  - (X, (Y-1) mod k) -> forming cycle
  - (X XOR 2^Y, Y) -> original link



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 ($n = r^d$)	$2d$	$d \left\lfloor \frac{\sqrt[d]{n}}{2} \right\rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dimensional hypercube ($n = 2^k$)	$\log n$	$\log n$	$\log n$	$\frac{n}{2}$
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 ($n = k^d$)	$2d$	$d \lfloor \frac{k}{2} \rfloor$	$2d$	$2k^{d-1}$

Indirect Interconnections

- reduce hardware costs by sharing switches and links, metrics:
  - cost (number of switches / links)
  - concurrent connections

Bus Network

Everything communicates through a single wire

- Only one pair of devices can communicate
  - coordinated by bus arbiter
- Only used for small number of PUs

Crossbar Network

$n * m$ crossbar network has n inputs, m outputs

Requires $n * m$ switches laid out in a mesh. Each switch has two states: *straight* or *direction change*

Multistage Switching Network

Crossbar, but greatly reduces the number of switches required

Omega Network (aka $\log n - 1$-dimensional Omega Network) $n * n$ Omega network has $\log n$ stages; each stage needs $n/2$ switches

A switch (a, i): a = index in stage, i = stage number

- edge to (a « 1, i+1)
- edge to (a « 1 but invert LSB, i+1)

e.g. (010, 1) has edges to (100, 2) and (101, 2)

Butterfly Network A switch (a, i): a = index in stage, i = stage number

- edge to (a, i+1)
- edge to (a but invert (i+1)-th bit from the left, i+1)
  - 1-indexed bit

e.g. (000, 0) has edges to (000, 1) and (100, 1)

Baseline Network A switch (a, i): a = index in stage, i = stage number

- edge to (a but cyclic right shift of (k - i) bits, i+1)
- edge to (a but invert LSB + cyclic right shift of (k - i) bits, i+1)
  - 1-indexed bit

e.g. (001, 1) has edges to (000, 2) and (010, 2)

Indirect Interconnections Routing

- Based on path length
  - Minimal (or non-minimal) = shortest path is always chosen
- Based on adaptivity
  - Deterministic: always same path for same (source, destination)
  - Adaptive: may take into account network status (avoid congestion or dead nodes)
    * has computation overhead!

XY-Routing (for 2D Mesh)

1. Move in X direction until X_source == X_destination
2. Move in Y direction until Y_source == Y_destination

(YX-Routing does step 2 then step 1)

Surprisingly low contention with simple rule: one clock (per hop) in Y direction, two clocks (per hop) in X direction

E-Cube Routing (for Hypercube)

Start from MSB -> LSB (or LSB -> MSB):

1. find first different bit
2. Hop to neighbouring node with the bit corrected

Takes at most n hops (because n bits) aka hamming distance

XOR-Tag Routing (for Omega Network)

let T = Source_ID XOR Destination_ID

At stage k:

- Go straight if $T[k] == 0$
- Crossover if $T[k] == 1$

Message Passing

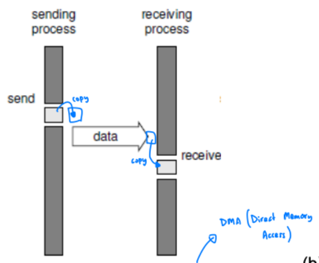
- Explicit send, explicit receives
- Loosely synchronous (only synchronised at messages)
- Should assume buffers don't exist

Assumes independent, distributed address space.

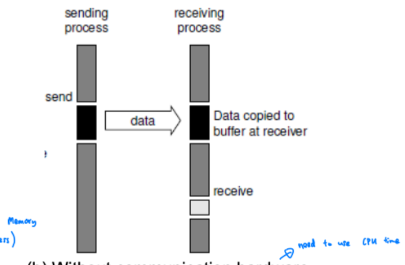
Can run on separate nodes / machines

Operations

- Buffered + Blocking
  - Sending process returns after data has been copied into buffer
    * Sender copies data into designated buffer in OS (or library), then returns
    * Receiver OS (or library) receives in designated buffer, then copy to program
  - (safe to reuse variable)
  - Possible to cause idling & deadlocks if buffer is full
- Buffered + Non-Blocking
  - Sending process returns after *initialising* transfer. Operation might not have completed!
  - (NOT safe to reuse variable)
  - programmer must explicitly ensure if message is sent by polling
  - won't run into deadlocks! (if no other issues)
- Non-Buffered + Blocking
  - Sending process blocks until *matching receive* operation has been called
  - (someone must be receiving)
  - Often causes idling & deadlocks!
- Non-Buffered + Non-Blocking
  - Similar to Non-Buffered + Blocking, but program is allowed to do other stuff while waiting for matching receive
    * if no hardware support: original program will be blocked
    * if there's hardware support: original program will not need to be blocked

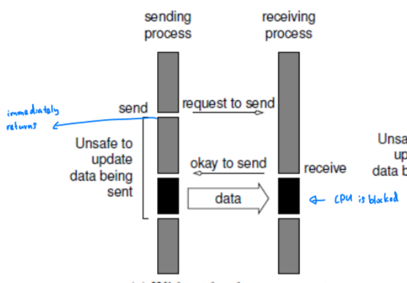


(a) Using communication hardware with buffers at send and receive ends

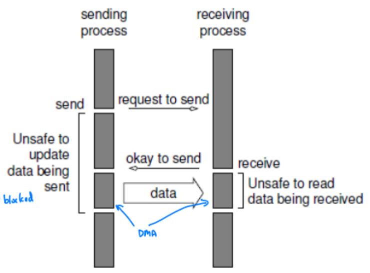


(b) Without communication hardware, sender interrupts receiver and deposits data in buffer at receiver end

(Buffered + Blocking Operations)



(a) Without hardware support

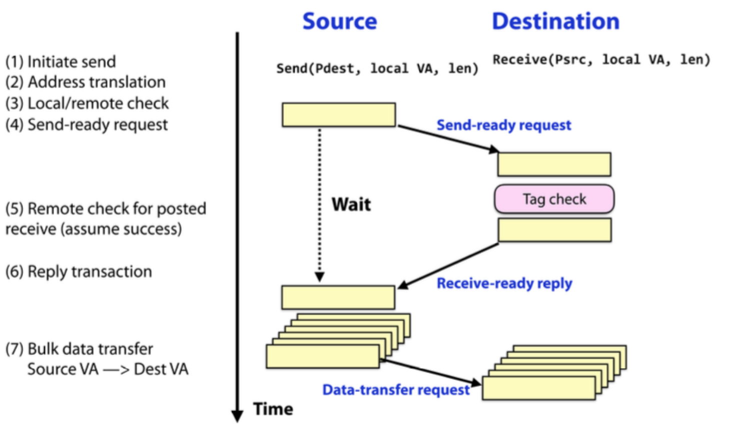


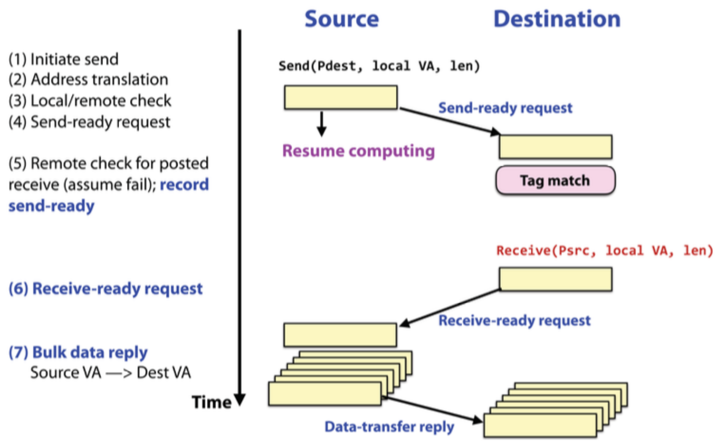
(b) With hardware support

(Non-Buffered + Non-Blocking Operations)

Semantics of Operations

- Local View
  - Blocking: return from a library call => user can reuse variables
  - Non-blocking: library call can return before operation is done, and *before* user is allowed to reuse variable
- Global View
  - Synchronous: communication operation does not complete before both processes have **started** their communication operation
    - Send implementation: send completes after matching receive and source data is sent (data might still be "travelling" and unavailable to the receiver)
    - Receive implementation: receive completes after data transfer completed from matching send
  - Asynchronous: sender can execute its communication operation without coordination with receiver
    - Send implementation: send completes after input buffer may be reused





MPI Syntax

```c
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size, tag, i;

    // called before ANY OTHER MPI routines
    MPI_Init(*argc, *argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (false)
    {
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    // do stuff here

    // LAST MPI call
    MPI_Finalize();
}

// async + blocking (may be buffered)
MPI_Send(void* buf, int count, MPI_Datatype dt,
    int dest, int tag, MPI_Comm c);
// async + blocking (buffered)
MPI_Bsend()
// async + non-blocking
MPI_Isend(const void *buf, int count, MPI_Datatype dt,
    int dest, int tag, MPI_Comm c, MPI_Request *req);
// sync + blocking
MPI_SSend()

// src = MPI_ANY_SOURCE to receive from any process
// tag = MPI_ANY_TAG to receive from any tag
MPI_Recv(void* buf, int count, MPI_Datatype dt, int src,
    int tag, MPI_Comm c, MPI_Status *status);
MPI_Irecv(void *buf, int count, MPI_Datatype dt, int src,
    int tag, MPI_Comm c, MPI_Request *req)

// test if MPI_Request is finished, stores inside flag pointer
MPI_Test(MPI_Request *req, int *flag, MPI_Status *status);

// waits for MPI_Request
MPI_Wait(MPI_Request *req, MPI_Status *status);
```

MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED_LONG, MPI_UNSIGNE, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED

MPI Guarantees

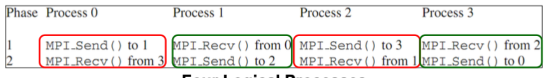
- If one sender + one receiver:
  - sender sends two or more messages to receiver
  - messages will be delivered **in order**
- If more than two processes:
  - no guarantee of order

MPI Deadlock-Free

MPI program is **secure** if correctness of program does not depend on assumptions (including existence of buffers) about specific properties of the MPI runtime system

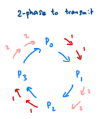
Deadlock-Free Logical Ring

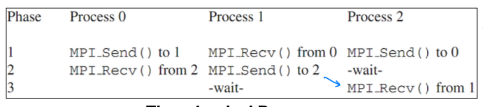
- processes with *even rank*: send, then receive
- processes with *odd rank*: receive, then send



Phase	Process 0	Process 1	Process 2	Process 3
1	MPI_Send() to 1	MPI_Recv() from 0	MPI_Send() to 3	MPI_Recv() from 2
2	MPI_Recv() from 3	MPI_Send() to 2	MPI_Recv() from 1	MPI_Send() to 0

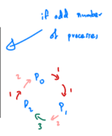
Four Logical Processes





Phase	Process 0	Process 1	Process 2
1	MPI_Send() to 1	MPI_Recv() from 0	MPI_Send() to 0
2	MPI_Recv() from 2	MPI_Send() to 2	-wait-
3		-wait-	MPI_Recv() from 1

Three Logical Processes



MPI Process Groups & Communicators

- Each process has unique rank *within group*
- Process can be part of multiple groups
- Communicator is communication domain for a group of processes (created from process groups)
  - Intra-communicators
    - Arbitrary collective communication within a *single process group*
  - Inter-communicators
    - Point-to-point operations between *two process groups*

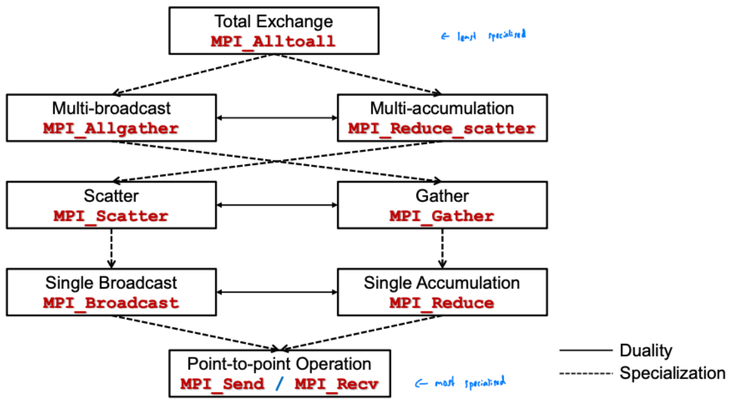
MPI Collective Communication

Operation involve **all** processes, will deadlock and wait for **all** processes.

```c
// no data movement
MPI_Barrier(MPI_COMM_WORLD);
```

```
// Scatters data evenly
MPI_Scatter(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm c);
MPI_Gather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPIDatatype recvtype,
    int root, MPI_Comm c);

// Root processor sends **same** data block to all other processors
MPI_Bcast(void *buf, int count, MPI_Datatype dt,
    int root, MPI_Comm c);

// Every processor sends **same** data block to all other processors
// Collected in rank order
MPI_Multibroadcast()

// Gather, with accumulation of result into root processor
// must be binary, associative and commutative
// MPI single accumulation
// MPI_Op = { MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD... }
MPI_Reduce(const void *sendbuf, void *recvbuf,
    int count, MPI_Datatype dt,
    MPI_Op op, int root, MPI_Comm c);

// Every processor sends a (potentially) different data block
// to all other processors
MPI multi accumulation

// Each processor provides for every other processor
// a (potentially) different data block
// e.g. transpose
// MPI Total Exchange
MPI_Alltoall(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf,
    int recvcount, MPI_Datatype recvtype,
    MPI_Comm c);
```

Two communication operations are a **duality** if same spanning tree can be used to represent both operations



Energy Efficiency

at Google

1. Continuously measure efficiency
   - energy for computation
   - energy for cooling / other stuff (overhead)
2. Building custom, highly-efficient servers
   - minimise power loss in AC-DC conversions
   - remove unnecessary parts
3. Extending equipment lifecycle
   - reuse components, resell components
4. Controlling temperature of equipment
   - at about 26 degree celsius (much higher than other data centres)
   - using thermal modelling: with hot-cold aisles
5. Cooling with water instead of chillers (air)

Reducing Energy Consumption

- reduce data transfers
  - exploiting locality
  - use data compression

Costs of Computing

Higher performance -> more / faster computers -> power -> heat -> cooling -> space -> money / environmental costs

cooling -> power -> money / environmental costs

Cloud Computing vs Data Centers

Cloud computing = data center with a layer of software

- Cloud typically offers services at different levels: platform, software, infrastructure; Data centers typically only offer infrastructure
- Cloud services are pay-per-use; Data centers are a huge upfront investment
  - Cloud is easier to scale as a result
- Cloud typically refers to using a 3rd party (or another department); Data centers can be privately owned

Homogeneous Computing

Use the ratio of performance on single-core to compare across different core types.

$$E_{\text{par}} = T_{\text{par}} * (N_{\text{fast}} * Power_{\text{fast}} + N_{\text{slow}} * Power_{\text{slow}})$$