CS3210

Tut & Lab

Concurrency

- Two or more tasks can start, run, and complete in overlapping time periods
- They might not be running (executing on CPU) at the same instant
- Two or more execution flows make **progress** at the same time by interleaving their executions or by executing instructions (on CPU) at exactly the same time

Parallelism

- Two or more tasks can run (execute) **simultaneously** at the exact same time.

Processor Arch

Unicore Arch

Levels of parallelism achievable with a single processor:

1. Bit-level parallelism → executing on a word
2. Instruction-level parallelism
    1. pipelining (across time): multiple instruction in different stages in the same clock cycle. Danger: data & control-flow hazards, out-of-order execution → maximum speedup = number of stages
    2. superscalar (across space): multiple instructions to pass through the same stage. Danger: structural hazard (cannot access memory at the same time)
3. Thread-level parallelism: multithreading.
    1. Fine-grained MT : switch after each instruction
    2. Coarse-grained MT : switch on stalls
        1. Timeslice MT: switch after a predefined timeslice
        2. Switch-on-event MT: switch if a processor is waiting for an event
        3. Simultaneous MT: schedule instructions from different threads in the same cycle. Processor provide hardware support for thread context (PC, stack, registers) by duplicating registers
4. Process-level parallelism: multiprocessing. Each process needs independent memory space → use IPC provided by OS.

Flynn's Parallel Arch Taxonomy:

Taxonomy of parallel archs based on the parallelism of instructions and data streams in the most constrained component of the processor.

1. SISD (Single Instruction Single Data): A single stream of instruction. Each instruction work on single data (uniprocessor)

2. SIMD: A single stream of instruction. Each instruction works on multiple data. (SSE, AVX instruction in x86, vector processor) → lock step.
3. MISD: multiple instruction stream on same data. (only theoretical)
4. MIMD: each Processing Unit (PU) fetch its own instruction and operates on its data. Most common now.
5. Variant (SIMD + MIMD): Stream processor (NVidia GPUs) → a set of threads executing the same code (SIMD). Multiple set of threads executing in parallel (MIMD).

Multicore arch

1. Hierarchical design
   - Multiple cores share multiple caches, that increase in size from the leaves (separate L1 cache) to the root (shared L2/L3 cache)
   - All cores share the same common external memory
   - Examples: standard desktop, server processors, GPU
2. Pipelined design: data elements are processed by multiple execution cores in a pipelined way. (specialized cores, roughly equal process time). E.g. routers / graphics processors
3. Networked-based design: cores and their local caches and memories are connected via an interconnection network

Memory Arch

Memory Latency: amount of time for a memory request to be serviced
Memory Bandwidth: rate at which the memory system can provide data.

Distributed Memory (multicomputers)

- Each node is an independent unit (has its own processor, memory, etc)
- Physically distributed memory module (memory in a node is private)
- Data exchanges using message passing through **interconnection network**

Shared Memory (multiprocessors)

- Access memory through **shared memory provider** (SMP) which maintains the illusion of shared memory, and transparent to the program
- Data exchanges via shared variables
- Differentiation based on
  1. processor-to-memory delay:
     1. Uniform Memory Acess (UMA): more suitable for small number of processors due to memory contention.
     2. Non-Uniform Memory Access (NUMA) or distributed shared memory
  2. presence of a local cache with cache coherence control (CC / NCC)

Problems with shared memory:

1. Cache coherence: All processes must agree on the order of reads/writes to the *same* memory location X. a.k.a. writes to X must eventually propagate to other processors.
2. Memory consistency: concerns with the behaviour of reads/writes to *different* locations. There is no guarantee that the order of memory change is the same as the instruction

order

Advantages	Disadvantages
No need to partition code / data	Need sync constructs
Efficient communication (no need to move data)	Lack of scalability due to memory contention

Hybrid: Distributed + Shared Memory

Parallel Programming Models

Parallelism

Parallelism → average number of units of work that can be performed in parallel per unit time (MIPS / MFLOPS / avg number of threads per second)

It is limited by program dependencies and runtime (memory contention, overheads, synchronization)

Types of Parallelism

1. Data parallelism: **same ops** on **different elements** of the data
   - **all data needs to be available** at the start of the processing.
   - e.g. independent loop executions
   - common model: Single Program Multiple Data
2. Task parallelism: independent program parts **tasks** / functions distributed among the PU
   - Task Dependence Graph : a DAG where edges represent control dependency between tasks.
   - Critical Path Length: slowest completion time
   - Degree of concurrency = total work / critical path length.

Representation of Parallelism

OpenMP: Explicit parallelism, implicit scheduling
MPI: Explicit parallelism, explicit scheduling, explicit mapping

> scheduling → assign tasks to processes/threads
> mapping → map process/threads to physical cores

Models of Coordination

1. Shared Address Space: very little structure
   - All threads can read / write from / to all **shared variables**
   - Use locks for mutual exclusion
   - Drawbacks: requires hardware support (e.g. system-wide load and store), costly to scale, memory contention
   - Examples: shared memory systems → UMA / NUMA

2. Data Parallel: very rigid computation structure
    - Basic structure: map a (side-effect free) function onto a large collection of data, no communication between distinct function invocations
    - Examples: CUDA, OpenCL
3. Message Passing: highly structured communication
    - Tasks operate within their own private address spaces
    - Tasks communicate by **explicitly sending/receiving messages**
    - Examples: MPI, distributed memory systems

Note:

It is possible to implement shared address space abstraction on distributed memory machines, and to implement message passing on a shared memory machines.

Parallel Programming Patterns

Shared Memory:

1. Fork-Join. data / task parallelism.
2. Parbegin-Parend : similar to fork-join, more equal-sized task. typically used for data parallelism (OpenMP).
3. Pipelining: good for high number of **even**, repeated operations. Each task (stage) has about the same computation time.
4. SIMD : **single instruction** executed **synchronously** (lock-step execution). Used for data parallelism.
5. SPMD: **same program** executed on different processors and operating on different data.
    - no implicit synchronization and threads work **asynchronously**
    - different threads may execute different parts of the parallel program (non lock-step execution). Generally used for data parallelism.

Shared / Distributed Memory

1. Master-Slave: similar to parbegin-parend but suitable for distributed memory. Used where some form of **centralized coordination** is needed. Both *data parallelism* and task parallelism.
2. Client-Server: MPMD model. Used in cases where there are multiple incoming requests to process. Typically used for *task parallelism*.
3. Task Pool: good when tasks vary in execution time / arrival time and doesn't require centralized coordination.
    - number of threads is fixed.
    - each thread can create / process tasks. *task parallelism*
    - not very good for fine-grained tasks.
4. Producer-Consumer: similar to task pool but used in cases where the problem can be divided into **source and sink**. producer create tasks, consumer process tasks. number of producer and consumer might be different

Data Distribution

a.k.a. Partitioning: useful for problems exhibiting data parallelism

1-Dimension distribution:

- Blockwise: Form blocks of size B and assign each block to a different core.
- Cyclic: Perform round robin allocation
- Block-cyclic: Form blocks of size B and then perform cyclic allocation

We could also split across 2 (or more) dimensions. This is called checkerboard distribution. The types are similar to 1-Dimension distribution but the blocks are now of size $B_1 \times B_2$.

Factors to think:

1. Work granularity + locality
2. Communication overhead: how many neighbors to communicate with?

Information Exchange

Shared Variables (for Shared Address Space)

- Need synchronization ops (e.g. mutex) to avoid data race.
- Example: OpenMP (`omp_set_lock`, `omp_unset_lock`)

Dedicated Communication Ops (for Distributed Address Space)

- Explicit send and receive in message passing model
- Loosely synchronous: interactions might need synchronization, but between these interactions, tasks execute completely independent.

Communication Protocols

Local view (only the sender / receiver)

- **Blocking**: Send operations blocks until it is **safe** to reuse the input buffer.
  - for **buffered**: until data is copied to communication buffer. Need to deal with buffer capacity management.
  - for **non-buffered**: until matching receive has been performed by receiver. Need to deal with idling.
- **Non-blocking**:
  - needs to poll to ensure completion of the operation, e.g. `check-status`
  - needs to make sure that the input is not modified before the operation completes.

Global View (both sender and receiver)

- **Synchronous**: Communication operation doesn't complete before both processes started their communication operation
- **Asynchronous**: Sender can execute its communication operation without any coordination with the receiver

GPGPU

Architecture

- Multiple Streaming Multiprocessors (SMs) with memory, cache and connecting interface (usually PCI Express)
- SM consists of multiple SPs, each containing memories (registers, L1 cache, texture memory, shared memory) and logic for thread and instruction management.

CUDA

Definitions: Device = GPU, Host = CPU, Kernel = function that runs on device

Parallel portions execute on device as **kernels**, multiple kernels can execute in parallel in newer CUDA hardware.

A kernel → executed by an **array of parallel (CUDA) threads** in SPMD fashion.

- Although the most efficient execution happens in SIMD fashion, but threads can have different execution flows (e.g. in branching)

Thread Blocks (TB): The monolithic thread array are partitioned into blocks of CUDA threads.

- In each block, threads can cooperate by sharing computation results, sharing memory accesses, atomic operations and synchronize execution
- However, threads in different blocks cannot cooperate.
- TB executes on one SM until it completes (no migration)
- Multiple TB can reside concurrently on one SM
  - this number is limited by SM resources (i.e. register file, shared memory) as these resources are partitioned among all SM residents
  - enables other TB to execute when the currently executing TB stalls
- This enables programs to transparently scale (i.e. hardware is free to schedule TB to any SM) to any number of processors (limited sync overheads)

Kernel is executed by a **grid** of TB.

Block IDs and Thread IDs

- Blocks in a grid can be laid out in 1/2/3 dimensions. Each block has a blockID
- Threads in a block can be laid out in 1/2/3 dimensions. Each thread has a threadID.
- IDs are simply virtual arrangement to simplify memory addressing by the programmer
- It is used to compute memory address and make control decisions

Thread Execution

- SM will break down TB into groups of 32 threads (the division is consistent across runs, based on threadIDs), called **warps**.
- Warps execute in a SIMT (single instruction, multiple thread) model, similar to the SIMD, i.e. lock-step execution.

CUDA Memory Model

Type	Scope	Access Type	Speed	Explicit sync	Declaration

Type	Scope	Access Type	Speed	Explicit sync	Declaration
Register	thread	RW	fastest	no	-
Local	thread	RW	depends*	no	-
Shared	block	RW	fastest	yes	__shared__
Global	program	RW	slow	yes	__device__
Constant	program	R (cached)	slow	yes	__constant__
Texture	program	R (cached)	slow	yes	__texture__

Note:

- Local memory is actually an abstraction of global memory that is private to the thread, that's why access speed is slower than a shared memory.

Description:

- Local Memory: useful for automatic array variables (>4 elements) allocated by compiler
- Constant Memory: useful for uniformly-accessed data
- Texture Memory: useful for spatially coherent random access data
- Shared Memory
  - Divided into equally sized memory modules called **banks**.
  - Memory accesses to the same bank (bank conflict) needs to be serialized.
- Global Memory:
  - Simultaneous access to global memory by threads in a half-warp can be *coalesced* into memory transactions of 32, 64 or 128 bytes.

Cache Coherence and Memory Consistency

System Model: Shared Address Space Model (shared variables, locks, any processor can load and store from any address)

Note: Shared Address Space Model can also be implemented on distributed memory systems.

Cache

Cache Properties

- Cache size: larger cache, larger access time, smaller cache misses
- Block size: larger blocks (tradeoff between spatial locality vs number of blocks stored in cache)

Write Policy

1. Write-through: write access is immediately transferred to main memory
2. Write-back: write op only performed in cache (uses dirty bit)

Cache Coherence

Problem: multiple copies of the same data exists in multiple cache. We want each processor to have a consistent view of memory through its local cache.

Property

1. **Program Order**: P reads its own write.
2. **Write Propagation**: Writes become visible to other processors.
3. **Write Serialization**: All writes to a location (by same or different processors) are seen in the same order by all processors.

How to maintain coherence?

- Software based solution: OS + compiler + hardware-aided (e.g. page fault to propagate writes)
- Hardware based solution: cache coherence protocols

Major tasks

1. Track the sharing status of a cache line
2. Handle updates to a shared cache line

Based on how cache coherence protocols track sharing status:

- Snooping Based
  - No centralized directory
  - Each cache monitors / snoops on the bus and takes action for relevant bus transaction (See bus-based cache coherence protocol)
- Directory Based:
  - store sharing status in a centralized location.
  - Commonly used in NUMA

Bus-based cache coherence

1. All processors can observe every transactions on the bus → satisfy Write Propagation
2. Bus transactions are visible to the processors in the same order → satisfy Write Serialization

Implication
Overhead in shared address space:

1. Increased memory latency: CC needs some time to run
2. Lower cache hit rate: cache line might be invalidated due to a write by another processor to the same address or to a different address sharing a cache line (a.k.a. **false sharing**).

Memory Consistency

Concerns with the order in which memory operations (to different data) appear to execute to other processors.

Consistency Models

Consistency models is used to decide the possible reordering by hardware and compiler. Reordering independent memory access is often done to hide write latencies.

We can try to relax the ordering *(of the results being seen)* of memory ops if data dependencies allow, i.e. no RAW, WAW, WAR dependencies on the same memory location.

Define:

1. Write Propagation: If 1 processor sees a mem op's results, all processors see it.
2. Write Serialization: All processors see the same order of mem ops.

Model	Relaxation	Effect seen
Sequential Consistency	-	Propagation, Serialization
Total Store Order	W→R	Propagation
Processor Consistency	W→R	-
Partial Store Order	W→R + W→W	Propagation

Interconnection Networks

Big Questions:

1. Topology: What is the geometrical shape of the connection?
2. Routing: What is the path for a message to follow?
3. Switching: How to transfer a message along a path?
4. Flow Control: How to handle concurrent messaging?

Topology

What is the geometrical shape of the connection?

Direct Interconnection

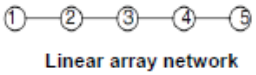
(a.k.a. Static / Point to Point): endpoints of same type

Metrics

- Diameter: small diameter = small distance between any pair of nodes
- Degree: small degree = reduce hardware overhead
- Bisection width: measure capacity of a network when transmitting msg simultaneously
- Node (Edge) Connectivity: number of nodes (edges) that must fail to disconnect the network (robustness, edge measure number of independent paths between any pair)

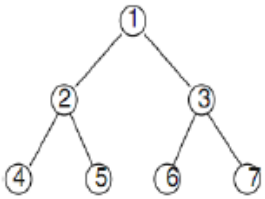
Examples

Complete network

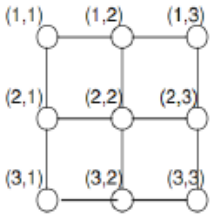


Linear array network

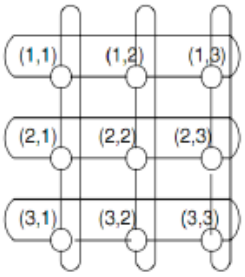
Ring network



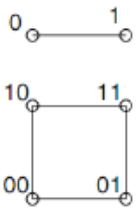
complete binary tree

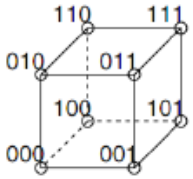


2-dimensional mesh

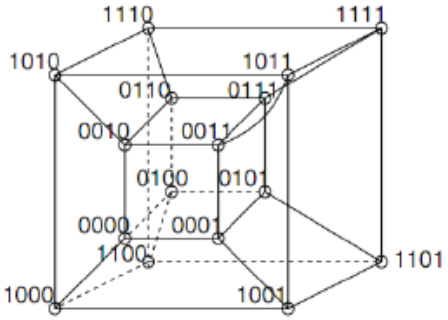


2-dimensional torus





Hypercube



Summary of Metrics

network G with n nodes	degree $g(G)$	diameter $\delta(G)$	edge-connectivity $ec(G)$	bisection bandwidth $B(G)$
complete graph	$n-1$	1	$n-1$	$\left(\frac{n}{2}\right)^2$
linear array	2	$n-1$	1	1
ring	2	$\lfloor \frac{n}{2} \rfloor$	2	2
d-dimensional mesh ($n = r^d$)	$2d$	$d(\sqrt[d]{n}-1)$	d	$n^{\frac{d-1}{d}}$
d-dimensional torus ($n = r^d$)	$2d$	$d\lfloor \frac{\sqrt[d]{n}}{2} \rfloor$	$2d$	$2n^{\frac{d-1}{d}}$
k-dimensional 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 Interconnection

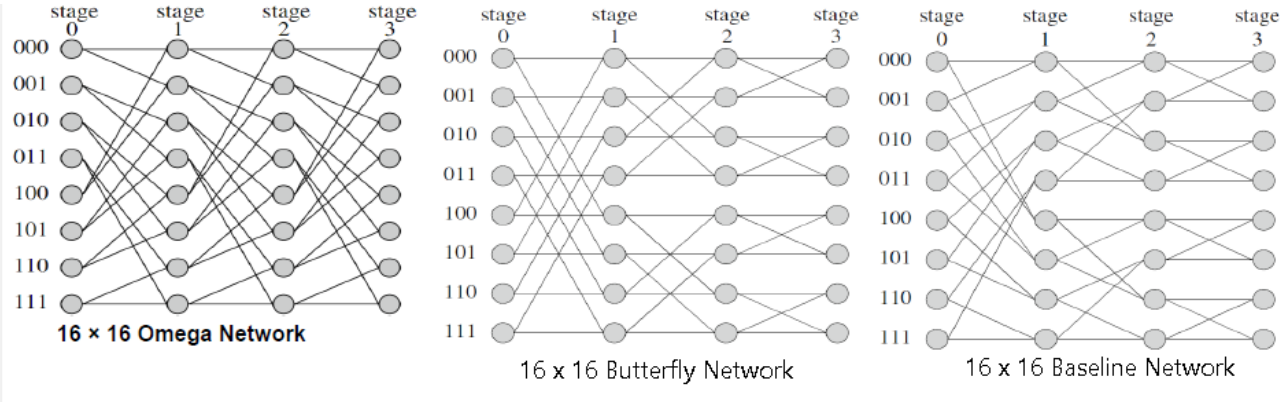
- switches provide indirect connection between nodes
- switches can be configured dynamically.
- sharing switches and links → reduce hardware costs

Metrics

- Cost: number of switches / links
- Concurrent connections

Examples

1. Bus Network: only 1 concurrent connections. Good for small number of processors
2. $n \times m$ Crossbar Network:
   - n inputs and m outputs
   - each switch can have two states: straight / direction.
   - Good for small number of processors as hardware is costly
3. Multistage switching network: obtain small distance between arbitrary pairs of input and output devices.
4. $n \times n$ Omega Network:
   - $\log n$ stages @ $n/2$ switches
   - A switch (α, i) will connect to $(\alpha \ll 1, i+1)$ and $(\alpha \ll 1+1, i+1)$
5. Butterfly network
6. Baseline network



Routing

What is the path for a message to follow?

- path length: shortest path
- adaptivity
  - deterministic:
    - XY-routing for 2D mesh: move in X dir before Y dir
    - E-cube routing for Hypercube: start from MSB to LSB (or vice versa), find first different bit and go to the neighboring node with bit corrected. at most n hops
    - XOR-tag routing for Omega Network: Let T be the XOR of source id and dest id. Repeat $\lg n$ times: cyclic left shift + flip last bit k-th bit of T is 1.
  - adaptive: take into account network status

Performance

Performance metrics: response time vs throughput
Performance depends on the programming, computational and architectural model

Possible Bottlenecks

1. Instruction-rate limited
2. Memory bottleneck
3. Locality of data access
4. Synchronization overhead

Execution Time

We focus on **user CPU time** = $(N_{\text{instr}}(A) \times CPI(A) + N_{\text{rw\_op}}(A) \times R_{\text{miss}}(A) \times N_{\text{miss\_cycles}}) \times T_{\text{cycle}}$

Note: memory hits (cache-access) has been included in the calculation of $CPI(A)$

CPI depends on the internal organization of the CPU, memory system and compiler
N_Instr depends on the computer architecture and the compiler.

Reads → read hit/miss → load into cache → deliver data
Writes → write hit/miss → Write Miss Policy (Write Allocate or Write Around) → Write Policy (Write Back or Write Through)

Average Memory Access Time $T_{\text{access}}(A) = T_{\text{hit}} + R_{\text{miss}}(A) \times T_{\text{miss}}$. Note that T_{hit} here refers to cache access regardless of a hit / miss.

MIPS (Million Instruction / Second) and **MFIOPS**

Note: Higher MIPS doesn't necessarily correspond to faster execution time

Speedup and Efficiency

- **Cost** $C_p(n) = p \times T_p(n)$ where p is the number of processor. A parallel program is *cost-optimal* if it executes the same total number of ops as the fastest sequential program.
- **Speedup** $S_p(n) = T_{\text{best\_seq}}(n)/T_p(n)$. Theoretically $S_p(n) \leq p$, but in reality the alternative (superlinear speedup) can occur due to higher cache hits.
- **Efficiency** $E_p(n) = S_p(n)/p$ where the theoretical maximum is 1.

Scalability

Amdahl's Law → **fixed workload**, how fast can we go?

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

where f is the sequential fraction. This law is applicable at all levels of parallelism.

Gustafson's Law → **fixed time**, how big of a problem can we solve?

In many computing problems, the sequential fraction f decreases with increasing problem size n.

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

where τ_f (τ_v) is the constant execution time for sequential (parallel) part and $T^*(n)$ is the execution time for the best sequential algorithm.

Hence $\lim_{n\to\infty} S_p(n) = p$, i.e. the speedup increases as n increases.

Instrumentation

How-to look inside processing tasks and analyze their timing and energy usage.

Approach: Analyze performance bottlenecks and fix.
Based on timeline: testing before release / incident performance response

Methodologies

- Performance analysis: `uptime, dmesg, vmstat 1, mpstat -P ALL 1, pidstat 1, iostat -xz 1, free -m, sar -n DEV 1, sar -n TCP, ETCP 1, top`
- USE (Utilization: busy time, Saturation: queue length / queued time, Errors) method
- Instrumentation tools: modify the source code, executable / runtime env to understand performance
  - Types: manual, automatic source level, intermediate lang/compiler assisted (add code to assembly / decompiled bytecode), binary translation (add code to executable), runtime instrumentation (run in fully supervised env), runtime injection (code modified at runtime)

Types of tools: observability, benchmarking, tuning, static

Observability

Watch activity, e.g. insert timing statements, check perf counters

Profiling

- Profile CPU usage by stack sampling / Generate CPU flame graphs
- Use `perf` / `perf_counters`
- Understand CPU consumers, initialization of I/O, locks, etc.

Debugging

- Valgrind: Dynamic Binary Instrumentation
  - Memcheck, a memory checker tool: use **shadow memory**, tools that shadow every byte of memory used by a program with another value in software
    - A ("Addresability") bits: 1 indicates if an addressable memory byte. detect heap buffer overflows, wild reads and writes.
    - V ("Validity") bits: 0 indicates a defined bit. detect dangerous use of undefined values
    - Heap blocks: records the location of every live heap block. detect repeated frees, memory leaks
  - Helgrind detects
    - misuses of POSIX pthreads API: intercepts calls to functions and instruments them.
    - Potential deadlocks arising from lock ordering problems: build a directed graph indicating order in which locks have been acquired.
    - Data races: Build a DAG representing the collective happens-before dependencies, and monitors all memory accesses for illegal order using a set of rules.
- Sanitizer: compilation-based approach
  - e.g. `-fsanitize=address`, ThreadSanitizer (e.g. data races), MemorySanitizer (e.g. uninitialized reads), UndefinedBehaviorSanitizer (e.g. Integer Overflow, Null Pointer), Leak Sanitizer (e.g. memory leaks)

- Address Sanitizer: make use of **shadow byte**. Each aligned 8 bytes can have exactly 9 states → state stored in shadow byte.
- Thread Sanitizer: runtime library
  - malloc replacement, intercepts all synchronization, reads, writes
  - **Shadow cell**: an 8-byte to represent 1 memory access (16bits: threadId, 42bits: epoch, 5 bits: position/size in 8-byte word, 1 bit: isWrite)
  - Store stack trace for previous access:
    - each thread has a cyclic buffer containing 64-bit (type + PC) events (memory access, function entry/exit)
    - Events can be replayed ot be shown on report
  - Detects normal data races; use-after-free; races on mutexes, file descriptors, barrier; leaked threads; destruction of locked mutex; potential deadlocks, etc.
- `perf c2c`: debug false sharing

Benchmarking

Load test. Results are usually misleading, since target might be wrong (e.g. FS cache instead of disk)

Active benchmarking method (synthetic performance testing)

1. Run the benchmark for hours
2. While running, analyze and confirm the performance limiter using observability tools

Energy-Efficient Computing

Mobile Computing

Trends

- Decrease power consumption in the hardware components
- Increase performance (closing in with x64 systems)
- e.g. ARM Cortex-A53 or ARM Cortex-A series

ARM Cortex-A family

Similarities to general purpose Intel/AMD servers:

1. Processors cores + RAM + I/O interfaces + peripherals
2. Cores use similar execution model (pipelines)
3. Memory hierarchy : L1 + L2 + RAM
4. Uses VM, commodity Linux, hardware virtualization + hardware-lvl security

Differences:

1. Cores: RISC ISA + heterogeneous cores (a.k.a. big.LITTLE)
2. Lower instruction level parallelism exploitation, smaller caches, less RAM (typically non-upgradable), lower main-memory bandwidth, simpler I/O interfaces

ARM big.LITTLE:

- big CPU : high performance for compute intensive applications
- little CPU: low power execution for majority workloads, switch to big CPU after a certain limit

How to reduce energy consumption:

- Move less data: reduce data transfers to/from memory, exploit locality, use compression
- Use specialized processing: avoid parallelization if unnecessary, combine cpu-like + gpu-like (throughput optimized) cores, programmable hardware (FPGA)

Challenges:

- how to reduce power consumption while maintaining good performance
- usage of low-power (energy efficient) nodes: scheduling problem, energy-efficient configuration for parallel application
- effective usage of resources is left to the programmer: write efficient, portable code for heterogeneous architectures.

Enterprise Computing

Data Centers

Problem: more power → more heat → needs cooling → more heat (for cooling)

Efforts: use renewable energy, measure Power Use Effectiveness (PUE): total amount of energy used / amount needed to run only the processors.

Case study: Energy Efficiency at Google

1. Continuously measure efficiency
2. Build custom highly-efficient servers
    1. minimize power loss in AC/DC conversions
    2. remove unnecessary parts, e.g. peripheral connectors and video cards
    3. decrease fan speed to optimize cooling
    4. strategic positioning on racks
    5. keep high performance computers always on
3. Extend equipment lifecycle: reuse/resell components
4. Control equipment temperature (26 C) : use thermal modeling, manage airflow
5. Cooling with water instead of chillers.

Cloud Computing

Cloud computing: abstraction of underlying applications so that resources can be provided and consumed in a more elastic manner and on demand.

Virtualization: Create a virtual version of something, e.g. OS, server, storage device, network device, that can be accessed without being concerned where or how the resource is physically located / managed.

Cloud Services Models

- SaaS: Provider's application on Provider's servers
- PaaS: Customer's application on Provider's OS and servers
- IaaS: Customer-managed application + OS on Provider's servers

Challenges:

- Technical: tricky programming, evolving tools, moving large data is costly, security, Quality of Service, green computing, internet dependence
- Non-Technical: Vendor lock-in, non-standardized, Privacy, Legal, SLA

Appendix

Odd-Even Sort (on linear array): $O(n)$ for $P = n$
Shear Sort (Row (alternate order) + Column phases on 2D mesh): $O(\sqrt{n}\lg n)$ for $P = \sqrt{n}$
Bitonic (up-down or down-up order) Sort: $O(\lg^2 n)$

OpenMP

- loops: `#pragma omp parallel for [shared(vars), private(vars), reduction(op:vars), if(expr)]`, `#pragma omp ordered` (ordered execution between loop iterations)
- sections: `#pragma omp sections [shared(vars), private(vars), reduction(op:vars), if(expr)]` and `#pragma omp section`
- sync constructs: `#pragma omp barrier` (sync all threads), `#pragma omp master` (only master), `#pragma omp single` (single thread), `#pragma omp critical`, `#pragma omp atomic` (mini-critical section)
- lock: `omp_init_lock(omp_lock_t*), omp_set_lock, omp_unset_lock, omp_destroy_lock, omp_test_lock`

CUDA

- function specifiers: `__device__`, `__global__`, `__host__` (same as no specifier)
- thread/block organization: `dim3 myVar(4,4,4)`, `gridDim`, `blockIdx.{x,y,z}` (within the grid), `blockDim`, `threadIdx.{x,y,z}` (within its block)
- CUDA kernel function invocation: `kernel_name<<<gridDim, blockDim>>>(args)`. args are < 4KiB and passed using constant memory.
- Memory type: no specifier (local memory), `__shared__`, `__device__` (global memory, compile-time known size), `__constant__`, `__texture__`
- `cudaError_t cudaMalloc(void** devPtr, size_t size)`. size in bytes
- `cudaFree(void** devPtr)`
- `__managed__`, `cudaMallocManaged`
- `cudaGetLastError()`, `cudaDeviceSynchronize()`
- `atomicAdd` (and, xor, or, min, max, sub). default to device-wide atomics. Other variants of atomics scope: use `_block` or `_system` suffix.
- `__syncthreads`: sync threads in a block.

MPI

Note: bufcnttype = `void *buf, int count, MPI_Datatype datatype`

- `MPI_Init(&argc, &argv)`,, `MPI_Finalize()`
- `MPI_Send`, `MPI_Recv` (`bufcnttype, desttag, Comm` + `Status*` for `MPI_Recv` + `Request*` for non-blocking)
- `int MPI_Sendrecv(send-bufcnttype, desttag, recv-bufcnttype, srctag, Comm,Status*)`
- `MPI_Wait(Request*, Status*), MPI_Waitall, MPI_Testall`
- Process groups: `MPI_Comm_group(MPI_Comm, MPI_Group*)`, `MPI_Group_incl(old_group, count, ranks, MPI_Group* new_group)`, `MPI_Group_rank`, `MPI_Group_size`
- Communicators: `MPI_Comm_size(MPI_COMM_WORLD, &size)`, `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`, `MPI_Comm_create(MPI_Comm, new_group, MPI_Comm* new_comm)`
- Cartesian topology: `MPI_Cart_create(Comm_old, ndims, dims[], periods[], reorder_rank?, cart_comm), MPI_Cart_coords(comm, rank, maxdims, coords[]), MPI_Cart_shift(Comm, direction: 0 is y, 1 is x, displacements, rank_src*, rank_dest*)`
- Communication: `MPI_Bcast(bufcnttype, root, Comm)`, `MPI_Scatter/Gather(send-bufcnttype, recv-bufcnttype, root, Comm)`, `MPI_Alltoall(send-bufcnttype, recv-bufcnttype, Comm)`, `MPI_[All]Reduce(send-bufcnttype, recvbuf, op, root, Comm)`, `MPI_Accumulate, MPI_Reduce_scatter`