

CS3210

Wang Xiyu

September 15, 2025

1 Computer architecture

1.1 Parallelism

- Concurrency:
 - Multiple tasks can start, run and complete in overlapping time period
 - may not be running or executing at the same instant
 - multiple execution flows make progress at the same time by interleaving their execution OR by the same time
- Parallelism:
 - Multiple tasks running simultaneously
 - Not only making progress, but also execute simultaneously
- Single processor:
 - Bit level parallelism:
 - * parallelism by increasing the processor word size, e.g. parallel addition of 64 bit numbers on 32 bit machine
 - Instruction level parallelism:
 - * pipelining: [time parallelism] number of pipeline stages = maximum achievable speedup
 - * superscaling: [space parallelism] Duplicate the pipeline, multiple instruction can be on the same execution stage. Scheduling is challenging. Stronger structural hazard, less cycles per instruction
 - Thread level parallelism:
 - * Simultaneous multithreading(SMT): processor provides hardware support for multiple thread level context
 - * hyper-threading: executing multiple threads per processor at the same time
- Multi-Processor:
 - Shared memory:
 - Distributed memory
- Multicore processor architecture
 - hierarchical design:
 - * multiple cores share multiple caches
 - * cache size increases from leaves to root, as more cores share the same cache
 - * external memory shared by all cores
 - pipelined design
 - * data elements are processed by multiple execution cores in a pipelined way
 - * useful for sequential data element processing
 - network-based design
 - * cores and local caches and memory are connected via interconnection network
 - Efficient on-chip interconnection: enough bandwidth, scalable

Multiprocessing vs. multithreading:

- Multiprocessing: high overhead, i.e. context switches; able to utilize multiple processing units
- Multithreading: low overhead; but effectively utilising the same processing unit

Table 1: Distinguishing “processor”, “core”, “processing unit”, and “logical core”

Term	What it is	Independence / Context	Shares With	OS Sees / Notes
Processor (CPU / package)	Physical chip/package containing compute resources (cores, caches, I/O, memory controller).	Multiple cores; each core is an independent execution engine.	All cores share off-core resources (e.g., memory controller, sometimes LLC).	OS sees one processor package with N cores; socket count in NUMA systems.
Core (physical core)	An independent execution pipeline (fetch/decode/execute).	Can run its own instruction stream; has private L1/L2 (often).	Shares last-level cache (LLC) and memory controller with other cores on the processor.	OS schedules threads to cores; true parallelism across cores.
Processing unit (execution context)	Generic term for a hardware context that can run instructions (a core <i>or</i> a hardware thread).	Independent program counter, registers, and a stack.	If it is a hardware thread, shares core pipelines with sibling threads.	Ambiguous in literature; often equals “schedulable hardware context”.
Logical core (hardware thread, SMT/HT)	A virtualized execution context exposed by SMT/Hyper-Threading.	Own PC/regs/stack, but <i>shares</i> core’s execution units and caches.	Siblings on the same physical core compete for pipelines, cache ports, bandwidth.	OS treats each logical core as a CPU; throughput gain depends on resource contention and ILP.

1.2 Memory Organization

Parallel computers:

- Distributed-memory: Multiple computers
 - each node is an independent unit, with processor, memory etc.
 - physically distributed memory modules, memory local and private to each node
- Shared-memory: multiprocessor: programs and threads access memory through shared memory provider, unaware of the actual hardware memory architecture, requires cache coherence and memory consistency.
 - cache coherence: local update by one processing unit, other PU should see the change being reflected in their copy of the same data in their cache
 - memory consistency

Shared memory model:

- uniform memory access(UMA): same latency of accessing the main memory for all processors. Contention makes this unsuitable for large number of processors.
- Non-uniform memory access(NUMA)
- Cache-coherent Non-uniform memory access(ccNUMA)
- Cache-only memory access(COMA)
- Hybrid(distributed shared memory)
- Latency: time taken for a request from the processor to be serviced by the memory
- Bandwidth: the rate at which the memory system can provide data to the processor
- stall: When the processor cannot run the next instruction in an instruction stream due to dependency on a previous instruction

1.3 Data and task parallelism

- Data parallelism: multiple processing units carry out similar task on different part of data eg SIMD, SPMD in MIMD
- Task/functional parallelism: partition the task to solve among different PUs

1.4 Task dependency and degree of concurrency

Task dependency graph is a directed acyclic graph, nodes represent tasks with its execution time as value. This represents the control dependency between the tasks.

- critical path: the longest path
- degree of concurrency: $\frac{\sum \text{work}}{\text{critical path length}}$, an indication of the amount of work that can be done in concurrence

Table 2: Flynn's taxonomy: instruction/data streams, examples, and caveats

Class	Instr. Streams	Data Streams	Typical Examples	Notes / Caveats
SISD	1	1	Classic single-core CPUs; pipelined or superscalar scalar execution.	Pipelining/superscalar improve throughput but pipelining does not add new data streams, superscalar does not add new instruction streams.
SIMD	1	Many	Vector/SIMD ISAs (SSE/AVX/-NEON), GPU warp/warp-lane execution, vector processors.	Same instruction applied to multiple data elements in lock-step; divergence harms efficiency (masks).
MISD	Many	1	Rare/mostly theoretical; certain fault-tolerant or systolic designs sometimes cited.	Not common in general-purpose computing; examples are niche/controversial.
MIMD	Many	Many	Multicore/multiprocessor systems, clusters; CPUs with SMT (each HW thread its own stream).	Threads/processes can execute different code on different data; includes shared-memory and distributed models.

Hybrids: Modern systems often combine MIMD (many cores/threads) with SIMD (per-core vectors). A single program may be MIMD + SIMD (e.g., OpenMP across cores + AVX within each core; GPUs: MIMD across warps/SMs, SIMD within a warp).

2 Performance

Goal: reduce response time: wall clock time

2.1 sequential

- User CPU time: for user prog
- System CPU time: OS routines: depends on the OS implementation
- waiting time: I/O and other programs (due to time sharing, other programs get higher priority etc): depends on the load on the computer system (for seq program this shouldn't be high)

$$Time_{user}(A) = (N_{cycle}(A) + N_{mm_cycle}(A)) \times Time_{cycle}$$

User CPU time of prog A = number of CPU cycles needed for all instr times cycle time of CPU $\frac{1}{\text{clock rate}}$
instructions have different exe time:

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

$$N_{mm_cycle}(A) = N_{read}(A) + N_{write}(A) =$$

$$N_{read_cycle}(A) \times R_{read_miss}(A) \times N_{read_miss_cycle}(A) + N_{write_cycle}(A) \times R_{write_miss}(A) \times N_{write_miss_cycle}(A)$$

$$Time_{user}(A) = N_{instr}(A) \times CPI(A) \times Time_{cycle}$$

CPI depends on the internal organization of the CPU, memory system and compiler; N_{instr} depends on the compiler and ISA

Memory access

$$MFLOPS = \frac{N_{fi_ops}(A)}{Time_{user}(A)^6}$$

2.2 parallel

$$T_{processor}(n)$$

- time for local computations
- time for exchange of data between PUs
- time for synchronization between PUs
- waiting time: unbalanced load; memory contention; synchronization eg waiting to access shared data structure

Speed up

$$S_p(n) = \frac{T_{best_seq}(n)}{T_p(n)}$$

Theoretically $S_p(n) \leq p$, but in practice, superlinear speedup is possible, eg: problem working task fits in the cache, and one thread pulls data into L3 and other threads can use

Cost

$$C_p(n) = p \times T_p(n)$$

A parallel program is cost-optimal if it executes the same number of instructions as the best sequential program

Efficiency

$$E_p(n) = \frac{T_{best_seq}(n)}{C_p(n)} = \frac{S_p(n)}{p} = \frac{T_{best_seq}(n)}{p \times T_p(n)}$$

Ideally, $S_p(n) = p \rightarrow E_p(n) = 1$

2.3 Scalability

Amdahl's law:

speed up is limited by the fraction of the algo that cannot be parallelized

$$T = fT_*(n) + (1-f)T_*(n)$$
$$S_p = \frac{T_*(n)}{fT_*(n) + \frac{(1-f)}{p} \times T_*(n)} = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

implication: need better compilers to reduce f

drawback: f is not constant wrt n

Gustafson's law:

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

-
-

The interaction between size of the problem and the size of the parallel machine, depends on application,

Contention

many requests to a resource are made within a small window of time, resource hotspot. Use tree structure communication to reduce contention, at the cost of high latency

Locality

Exploit sharing: co-locate tasks that operate on the same data,