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 simultaneosuly
  - Not only making progress, but also execute simultaneously
- Single processer:
  - Bit level paralleism:
    - 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 excution 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 archiecture
  - 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 piplined 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
  - Multthreading: 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 *or* 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 *shares* 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:

- Distibuted-memory: Multiple computers
  - each node is an indenpendent 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 acual 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 instrucion

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).

1.5 IPC models

- Shared address space:
  - communication abstraction:
    - Tasks communicate by reading and writing to shared variables
    - use locks to ensure mutua exclusion
    - logical extension of single processor programming
    - require hardware support: able to share address space, ie shared memory systems
    - hard to scale due to memroy contention when multiple PUs accessing the same shared address space
    - can be mimiced on systems without required hardware support, implemented by message passing:
      - write to shared variables: send message to invalidate all pages containing shared variables
      - read a shared variable: page fault handler issues appropriate network requests
      - pure software solution, but inefficient
    - very little structure
    - all threads can read and write to all shared variables
    - Not all reads and writes have the same cost (NUMA), and the cost is not apparent in program text
  - Data parallel
    - SIMD, vector processors
    - Basic structure: map a function onto a collection of data
    - side effect free
    - no communication among distinct function invocations
    - Stream programming model
    - very rigid computation structure
- Message passing
  - tasks communicate via explicit sending and receiving messages
  - no need for system wide load and store implementation supported by hardwares
  - matches distributed memory system where theres no physically possible shared address space
  - more costly than shared address space model
  - Possible and common to implement message passing abstraction on shared memory machines:
    - sending message: copy date into message library buffer
    - receiving message: copy data from message library buffer
  - higly structured, all communication occurs in the form of messages

1.6 Program Parallelization

The transformation of sequential program into parallel programs

- Fine-grain: a sequence of instructions
- in between: a sequecne of statements
- coarse-grain: a function

Steps to parallelize a program: (Foster's methodology)

1. Partitioning: break down problem into many smaller pieces
   - Data centric: domain decomposition: divide data into pieces of similar size, and determine how to associate computatons with the data (Data parallelism)
   - Computation centric: Functional decompositon: determine how to associate data with the conputations (Task parallelism)
   - Rules of thumbs:
     - At least 10x more primitive tasks than cores
     - minimize redundant computation and data storage
     - primitive tasks should have roughly the same size
     - number of tasks is an increasing function of the problem size
2. Communicaton: provide data required by the partitioned tasks
   - Tasks are intended to be executed in parallel, but may not be indenpently executed, thus need to determine data passed among tasks
     - Local communicaton
       - Task needs data from a small number of other tasks (neighbours)
       - create channels illustrating data flow
     - Global communication
       - significant number of tasks contribute data to perform a computation
       - no need to create channels for communication early in design
     - Rules of thumb:
       - Communication operations are balanced aamong tasks
       - tasks commnunication with only a small group of neighbours
       - commincation can be performed in parallel
       - Ideally, distribute and overlap computation and commincation
3. Agglomeration: decrease communication and develpent cost, while maintaining flexibility
   - eliminate communication between primitive tasks by agglomerating into consolidated tasks
   - Ideally combine groups of sending and receiving tasks
     - increase locality of parallel program
     - ensure the number of tasks increase with problem size
     - ensure the number of consoliated tasks is suitable for the target system
     - balance agglomeration and code modification cost
4. Mapping: make tasks to processors, goal: minimize execution time
   - Maximize processor utilization
   - minimize inter-processor communication
   - Rules of thumb:
     - NP-hard in general, but can rely on heuristics
     - evaluate static and dynamic task allocation
       - If use dynamic tasks allocation, the task allocator should not be a bottleneck
       - if use static task allocation, the ratio of tasks to cores is at least 10:1

1.7 Parallel programming patterns

Fork-join

Parbegin-Parend

Suitable for openMP implementation

SPMD and SIMD

Master-worker

Task pool

Suitable for heterogeneous jobs, eg when the size of incoming problems is unpredicatbale

Producer-consumer

Pipelining

Suitable for task parallelism, when different partition of the task are roughly the same size. Uneven partition size renders pipelining or task parallelism inefficient uuu

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 implementatuon
- waiting time: I/O and other porgrames (due to time sharing, other programs get hihger piroirty etc): depends on the load o the coputer system (for seq program this shouldnt 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 fo rall instr times cycle time of CPU $\frac{1}{\text{clock rate}}$
instructions have different exe time:

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

Memory access

$$N_{mm\_cycle}(A) = N_{read}(A) + N_{write}(A) =$$
$$N_{read\_cycle}(A) \times R_{read_miss}(A) \times N_{read\_miss\_cycle}(A + N_{write\_cycle}(A) \times R_{write_miss}(A) \times N_{write\_miss\_cycle}(A))$$
$$Time_{user}(A) = (N_{instr}(A) \times CPI(A) + N_{rw\_op}(A) \times R_{miss}(A) \times N_{miss\_cycles}) \times Time_{cycle}$$

k level cache mis:

$$R_{miss\_global} = \prod_{i=1}^{k} R^{Lk}_{read\_miss}(A)$$

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

Throughput: Million Instruction Per Second (MIPS) Not a good indicator

$$MIPS(A) = \frac{N_{instr}(A)}{Time_{user}(A) \times 10^6} = \frac{clock\_frequency}{CPI(A) \times 10^6}$$

Throughput: Million Floating Operations Per Second

$$MFLOPS = \frac{N_{fl\_op}(A)}{Time_{user}(A) \times 10^6}$$

issues:

- no differentisation between different types of fp op
- specfic goal: do fp op

2.2 parallel

$T_{processor}(n)$ depends on:

- time for local computatio s
- time for exchanged of data between PUs
- time for synchronizaiton between PUs
- waititng time: unbalanced load; memory contention; synchronization eg waiting to access shared data structure

Speed up

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

Theoratically $S_p(n) \leq p$, but in practice, superlinear speedup is possile, eg: problem working task fits in the cache, and one threads 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 exe the same numbr of insructions as the best sequntial program
Effiiency

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

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

- Best sequential algorithm may not be known
- asymptotically optimal algorithm may not be the fastest in practice

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:

$$\lim_{n\to\infty} f(n) = 0 \rightarrow \lim_{n\to\infty} S_p(n) = \frac{p}{1 + (p-1)f(n)} = p$$

Implication: Amdahl's law can be circumvented for large size problems

-
-

The interaction between size of th eproble and the size of the parallel machine, dependedt on application,

Arithmetic Intensity

$$\text{Arithmetic intensity} = \frac{\text{the amount of computation}}{\text{the amount of communication}}$$

Contentiion

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

Locality

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

3 GPGPU