Chapter 8: Parallelism

Parallelism refers to doing multiple things at the same time, in both hardware and software, by distributing work across multiple compute resources.

- Flynn's taxonomy classifies parallelism based on *how a program handles instruction streams and data streams*:
  - **Single Instruction, Single Data (SISD)**:
    - Instructions are executed sequentially and there is a single source of data. This is a normal, sequential program.
  - **Single Instruction, Multiple Data (SIMD)**:
    - A single sequence of instructions is applied to multiple pieces of data at once, using vector operations.
  - **Multiple Instruction, Multiple Data (MIMD)**:
    - Instructions are executed simultaneously and operate on multiple data at the same time.
  - **Multiple Instruction, Single Data (MISD)**:
    - Less commonly used.
- *SIMD and MIMD leverage parallel execution* through special instructions that operate on multiple data at once.

8.1 Data Level Parallelism and SIMD

Data level parallelism executes a *single operation on multiple data streams simultaneously*.

- SIMD (Single Instruction, Multiple Data) uses special registers like XMM to pack data together and perform operations on the entire register contents at once. This allows the same operation to be applied to multiple data points in parallel.

8.2 Thread Level Parallelism

A **thread** is a sequential flow of instructions performing a task.

- Each thread has its own program counter, registers, and access to shared memory.
- Cores have multiple hardware threads that can execute instructions in parallel.
- The OS multiplexes software threads onto the hardware threads.
- Thread scheduling involves:
  - Saving registers/PC of a blocked/finished thread
  - Loading registers/PC for a new software thread
  - Jumping to the new thread's PC to start execution
- Multiple threads allow parallel execution across cores by distributing work.
- The OS manages thread scheduling and execution across available hardware threads.

8.2.1 Synchronization in thread-level parallelism

Thread-level parallelism introduces **synchronization** challenges since multiple threads access shared data.

- Without coordination, threads may overwrite or interfere with each other's work.
- **Locks (semaphores)** allow only one thread to access a shared variable at a time. A thread must acquire the lock before accessing the variable.
    - Locks prevent data corruption by serializing access to shared data.
- RISC-V has atomic memory operations (AMOs) to handle synchronization in hardware.
    - AMOs perform a read-modify-write in one instruction.
- **AMOs** allow threads to coordinate access to shared data safely.
    - However, locks can lead to deadlock where threads get stuck waiting on each other in a circle.
- Careful programming is required to avoid **deadlock** when using locks for thread synchronization.
- Locks serialize access to shared data but can also limit performance if overused.

8.2.3 Cache Coherence

- In multi-core systems, *all cores access the same shared DRAM memory*; however, each core has its own private cache.
    - Caches can become incoherent if one core modifies a shared variable that other caches have cached.
- **Cache coherence** protocols coordinate the caches to ensure consistency:
    - On a cache miss or write, other caches are notified.
    - Writes invalidate copies of the data in other caches.
- Additional cache coherence bits are added:
    - *Shared*: Data is up-to-date, other caches may have a copy
    - *Modified*: Data is dirty/modified, no other cache has a copy
    - *Exclusive*: Cache has up-to-date data, no other cache has it
    - *Owner*: Cache has up-to-date data, others may have copies in Shared state. Has exclusive write access.
- The coherence protocol maintains consistency by coordinating data flows between caches.
- This ensures all cores have a coherent view of shared memory.

8.2.4 OpenMP for shared memory parallelism

OpenMP uses pragmas (compiler directives) to enable multi-threaded parallelism.

- An OpenMP program starts as a single process.
- Parallel regions fork threads to execute code in parallel.
- Key pragmas:
  - parallel for - Parallelize a for loop
  - parallel - Mark a parallel section
  - critical - Serialize access to shared data
- OpenMP handles *thread creation, synchronization, and joining transparently.*
  - It allows easy parallelization of shared memory systems.

8.2.5 Amdahl's Law

Amdahl's Law shows limits to parallel speedup:

  - Speedup = 1 / (s + (1-s)/p)
    - s = serial fraction, p = parallel speedup
  - Maximum speedup limited by 1/s, the serial portion
- In practice, parallelization is limited by the serial parts of a program.
- Amdahl's Law gives theoretical limits to parallel speedups.