Understanding Parallelism

Lecture 3
February 13, 2025

To Dos

Reading for next time

Program 1

Why Parallel Hardware?

- We no longer know how to speed up sequential processes significantly through hardware or technology improvements.
- Technology constraints (e.g., power, latency of global communication, chip verification) make it challenging to have an entire chip working in a coordinated way.
- Many tasks can be subdivided into independent pieces of work.

Throughput becomes the goal, with latency staying constant

So Why is Parallelism Considered Hard?

- If independent tasks didn't need to coordinate or share data, parallelism would be easy (both in HW and SW).

- But it's not because of
  - sharing of data
  - coordination of activities across tasks (e.g., synchronization)
  - balanced allocation of work across a parallel system

Two General Forms of Computational Parallelism

Task parallelism

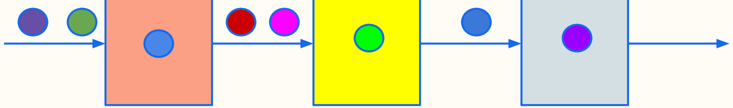
- Partition tasks among cores
- Each core may do different work
- Example:
  - Task 1: Remove capitalization
  - Task 2: Remove punctuation
  - Task 3: Search
- Example 2:
  - Task 1: Count words
  - Task 2: Sort words

Data parallelism

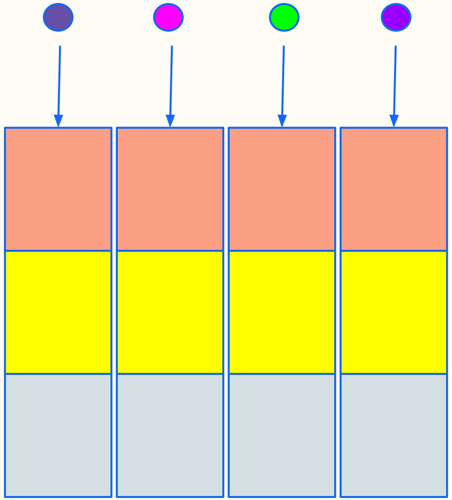
- Partition data among cores
- Each core does same work on different data
- Example:
  - Task 1: Remove capitalization, remove punctuation, and search on file 1
  - Task 2: Remove capitalization, remove punctuation, and search on file 2

Two General Forms of Computational Parallelism

Task parallelism



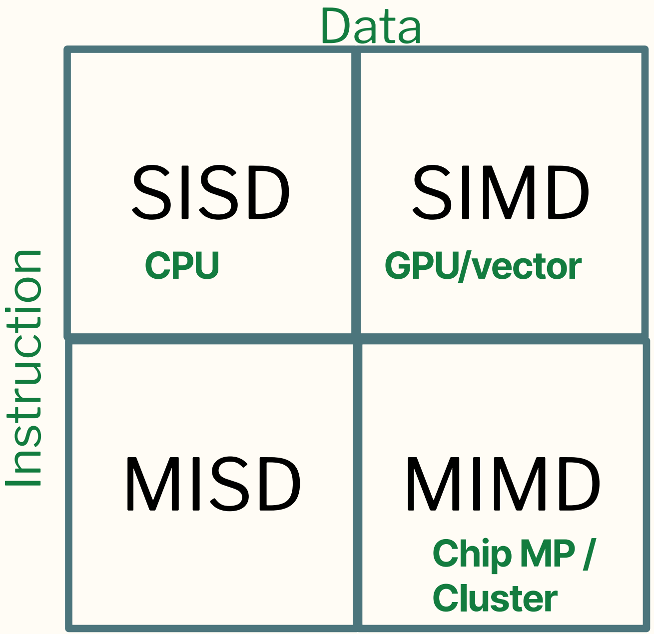
Data parallelism



Questions to Think about With Respect to Different types of Parallelism?

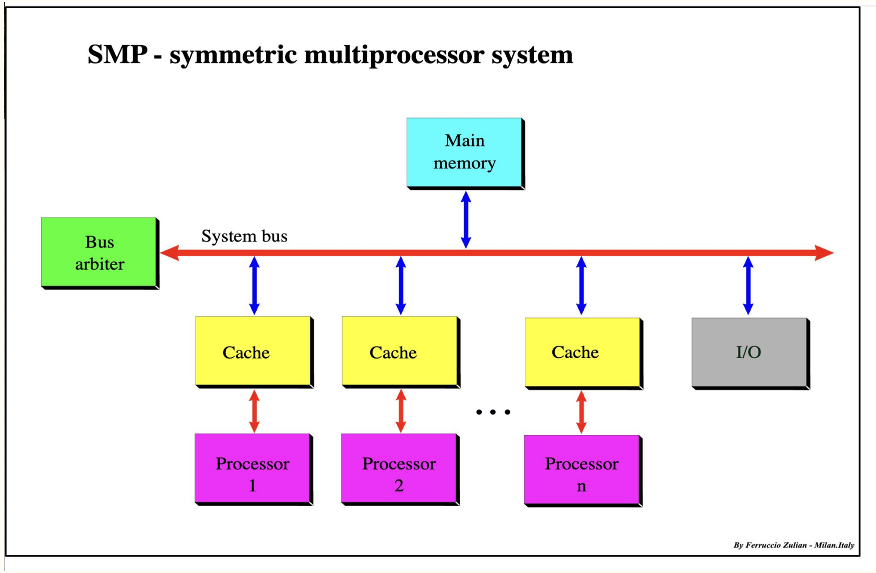
- How much data needs to be processed?
- How much variability in computation per data item?
- How much computation per data item?

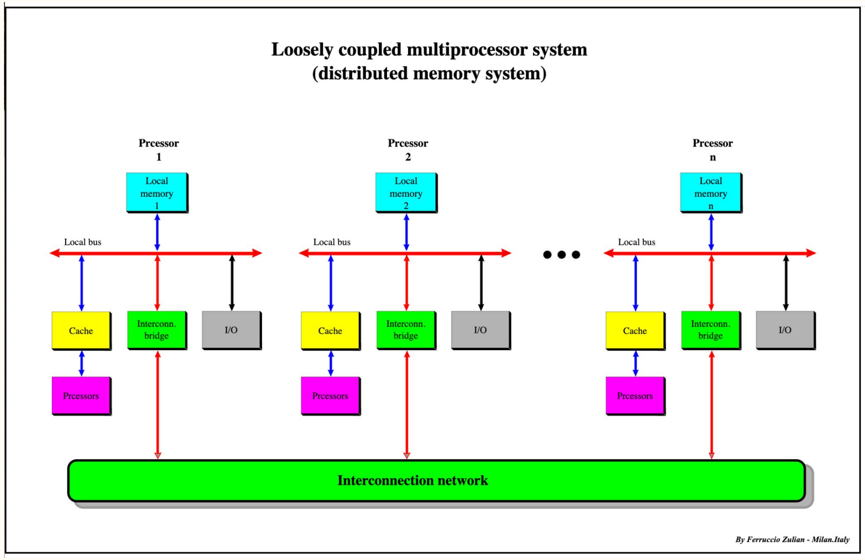
Flynn's Taxonomy



SPMD is a special case of MIMD

Example MIMD

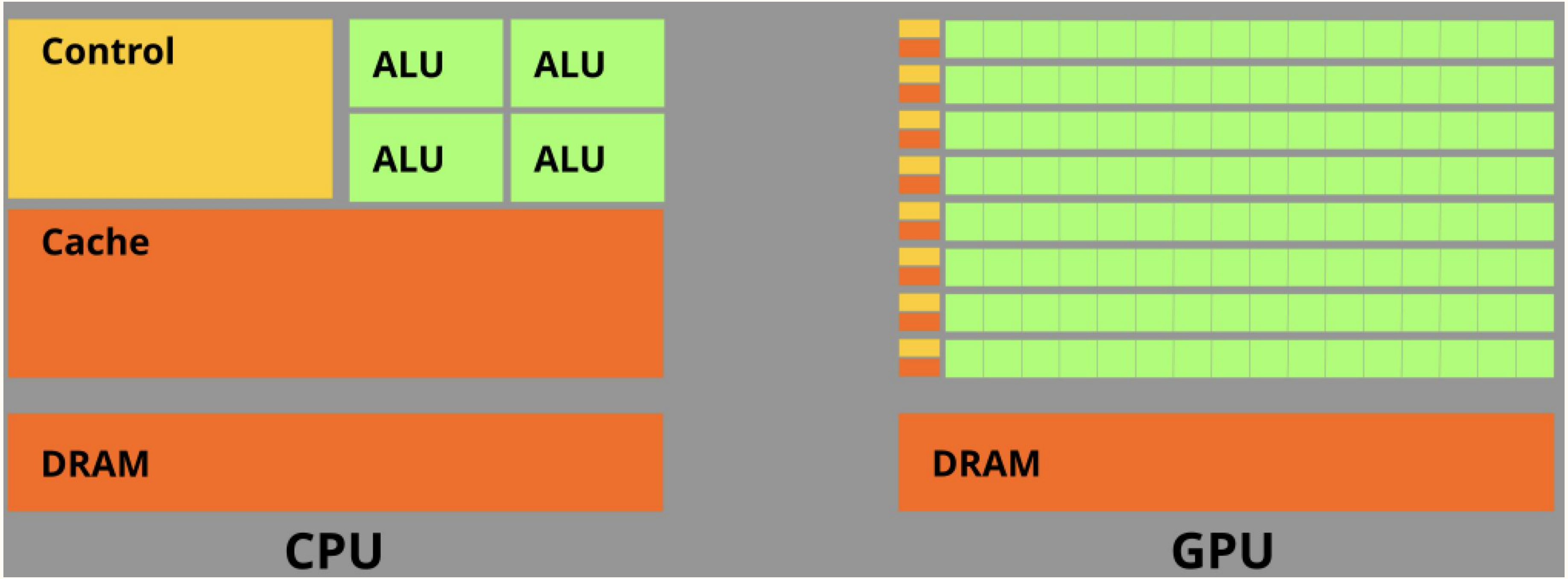




https://commons.wikimedia.org/wiki/File:SMP_-_Symmetric_Multiprocessor_System.svg

https://commons.wikimedia.org/wiki/File:Loosely_Coupled_Multiprocessor_System.svg

Example SIMD



https://commons.wikimedia.org/wiki/File:Cpu-gpu.svg

Another Way to Think of Parallelism

Shared-memory

- Can tasks directly access and share data?
- One large pool of memory shared across tasks

Distributed-Memory

- Do tasks have to explicitly send messages to share data?
- Many private pools of memory

Other HW Parallelism

Instruction Level Parallelism

Multiple Memory Banks

Co-processors (e.g., DMA, accelerators)

Approach for Writing Parallel Programs

1. Divide work among processes/threads such that
    a. each process/thread gets roughly the same amount of work
    b. amount of communication is minimized
2. Arrange for synchronization among processes/threads when needed
3. Arrange for communication among processes/threads

Shared Memory Systems

- Processes contain threads
- Threads
  - Have their own stack and registers
  - Share data in the address space
    - Communicate through shared variables implicitly

Shared Memory System Challenges

- Defn: nondeterminism - a given input can result in different outputs
- Defn: race condition - when threads try to access shared resource and the outcome depends on the order of the threads' execution
  - e.g., two threads want to do x++ to shared variable x

Shared Memory Solution for Nondeterminism - Synchronization

- Need to make access to shared variables atomic
- Do this by creating a critical section of code that only one thread can execute at a time (i.e., it will always be run serially)
- We must provide mutual exclusion to the critical section
  - How?
    - Hardware primitives allow us to create
      - Locks
      - Semaphores
      - Monitors
    - Busy waiting
      - Enter a loop where you test a condition to see if thread can enter code exclusively

Distributed Memory Systems

- Processes have private memory spaces
- Processes are numbered
- Processes do different work based on their number
- Processes communicate through explicit send and receive messages
  - Sending messages to process i
  - Sending collective messages to multiple/all other processes

Distributed Memory Communication

- Send/receive messages are often blocking
  - Sender has to wait until receiver has started receiving
  - Receiver has to wait until sender has started sending
- Collective communication
  - Broadcast sends message to all other processes
  - Reduction collects results computed by all other processes into a single result

How do we use duplicate resources to speed up our programs?

- Identify portions of our programs that take up the most time
- Determine if there are independent computations in those sections
- Map those independent computations onto different computing resources

Leaving out lots of important details that impact mapping and performance (e.g., communication)

Performance

- Sequential execution time : T_serial
- Number of parallel processes : p
- Parallel execution time : T_parallel

- Ideal parallel execution time : T_ideal = T_serial / p
    - Linear speedup

- Actual parallel execution time: T_parallel = T_serial / p + T_overhead

- Overheads
    - Shared memory: critical sections serialize portions of code
    - Distributed memory: communicate between processes
    - Overheads increase with number of processors

Speedup and Amdahl's Law

- Speedup: T_serial/T_parallel
- Amdhahl's Law: Performance improvement you get from an enhancement is dependent on:
  - The size of the enhancement
  - How frequently the enhancement is used

$$T_{enhanced} = (1 - fraction_{enhanced}) \times T_{unenhanced} + (\frac{fraction_{enhanced}}{Speedup_{ehnhancement}} \times T_{unenhanced})$$

Limitations to Performance Improvements

$$T_{enhanced} = (1 - fraction_{enhanced}) \times T_{unenhanced} + (\frac{fraction_{enhanced}}{Speedup_{ehnhancement}} \times T_{unenhanced})$$

- Sequential parts of code
- Parallel parts of code
  - Communication of data between processes
  - Load balancing
  - Synchronization

Let's Talk Through Performance Ideas

- First let's talk about sequential performance on a CPU
- Then let's talk about adding in communication / synchronization