Parallel Architecture 1

Module 3.1

Peter J. Hawrylak

Module Learning Objectives

- Classify a system according to Flynn's taxonomy (SISD, SIMD, MISD, MIMD).
- Explain how memory hierarchy is used to maintaining efficiency in computing.
- Describe Foster's methodology (see: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/) for designing parallel programs.

Flynn's Taxonomy (1

	Single instruction	Multiple instruction
Single data	SISD	MISD (?)
Multiple data	SIMD	MIMD

- SISD: Single instruction operates on single data element
  - One program with one data set
- SIMD: Single instruction operates on multiple data elements
  - One operation on several data items
  - Intel MMX instruction extensions
    - Perform arithmetic operations using the ALU on multiple sets of data. ALU is designed to do this.
      - 16-bit ALU can process 2, 8-bit values for sound (left and right) using 1 addition.
      - 32-bit ALU can process 4, 8-bit values for video using 1 operation
  - Array processor – work on data that is smaller than the **width** of the ALU in the processor (e.g., 32-bit, 64-bit, 128-bit)

Flynn's Taxonomy (2

	Single instruction	Multiple instruction
Single data	SISD	MISD (?)
Multiple data	SIMD	MIMD

- MISD : Multiple instructions operate on single data element
    - Closest form: systolic array processor, streaming processor
- MIMD : Multiple instructions operate on multiple data elements (multiple instruction streams)
    - A collection of **independent** processing elements work on **different** data streams
    - Example: Multiprocessor or Multithreaded processor

Memory (Storage) is SLOW

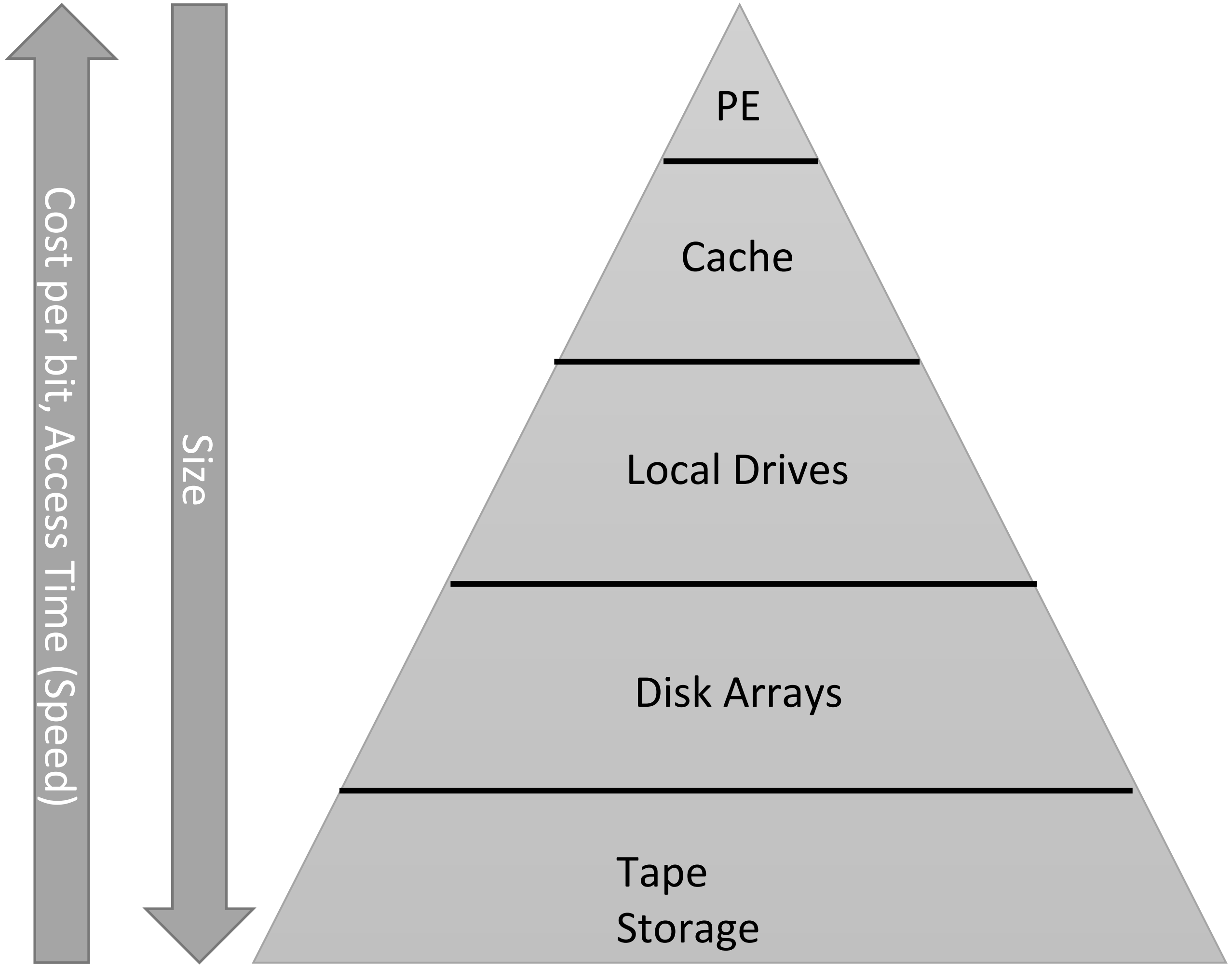
- Objective 1: Run processing element as fast as possible. Maximize throughput (work per unit time) and/or clock frequency (GHz).
- Objective 2: Never run out of input data.
- Objective 3: Never pause or wait to write output data.
- **These objectives conflict because memory is slow.**
- Memory inside the processing element is fast, often equal in speed to the processing element, but it is very scarce
  - This memory is very costly – space on the integrated circuit (chip)
  - This memory is very small – cannot hold the entire problem data set or even a large piece of it

Memory Hierarchy Objectives

- Objective 1: Have a large memory available that can hold all of the needed data.
  - Cost must be low – cheap because a lot of memory is needed
  - Size must be large – hold all the data
- Objective 2: We need the memory to be fast, ideally as fast as the processing element
  - Processing element must "see" the memory as fast
- Solution: Arrange memory in a hierarchy to support fast access times while providing the needed storage space.
  - Many levels – cache (L1, L2, L3,…), solid-state disk, spinning disk, tape storage

Memory Hierarchy - Implementation

- Processing element (PE)
  - Registers/on-board memory
- Cache
  - Small & Fast
  - Provide data to PE in 1 clock cycle
  - Many layers – higher number layers are slower
- Large Storage – Local drives
  - Solid State Drive – fast, higher cost
  - Spinning Media – slow, lower cost
- Disk Arrays
  - Very large and slow
- Tape Storage System
  - Very Very large and very slow



Foster’s Methodology (1)

- Methodology to design parallel programs
  - 4 step process
  - Identifies key issues and helps identify performance bottlenecks
  - Helps give a better estimate of speedup – use WITH Amdahl’s Law
- Step 1: Partitioning
- Step 2: Communication
- Step 3: Agglomeration
- Step 4: Mapping

See: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/

Foster's Methodology (2)

- Step 1: Partitioning
  - Break program into very small computational and data tasks – fine-grained decomposition
  - Domain Decomposition – Tasks based on the data they process, focus is on breaking up the data
    - Focus on large data structures and those most frequently accessed
  - Functional Decomposition – Tasks based on the calculations they perform, focus is on breaking up the calculations
    - Focus on disjoint calculations and use one task for each one
    - Communication of data between calculations (tasks) may be required
      - Increases overhead
      - If overhead increase from communication is too great look at Domain Decomposition
  - Use BOTH Domain and Functional Decomposition in this step.

See: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/

Foster’s Methodology (3)

- Step 1 Goals
- Have many more tasks than processing elements
- Eliminate redundant operations and memory accesses/requirements
- Try to give each task equal work □ load balance the tasks
- Double check scalability □ larger problem should require more tasks
  - Use this to find out early on if algorithm is or is not scalable
- Try to develop 1 or 2 other partitions □ other options if needed down the road

See: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/

Foster’s Methodology (4)

- Step 2: Communication
  - Determine messages that must be sent between tasks
  - Often complex for Domain Decomposition partitions
    - Communication is costly so it must be efficient
    - What data need to be shared? Share these data and no more.
  - Easier for Functional Decomposition partitions – just pass data from one computation to the next (assembly line type process)
- Communication Types
  - Local – Immediate and close neighbors
  - Global – ALL tasks take part
    - Identify centralized (one task does something for everything) and sequential (series of steps) issues Parallel algorithm may need to be reworked if these are found – look for concurrent operations
  - Static, Structured – Communication pattern is the same throughout program and well defined
  - Asynchronous – Producer-consumer approach consume data once it is produced

See: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/

Foster’s Methodology (5)

- Step 2 Goals
- Load balance communication
  - Unbalanced communication is a sign that algorithm may not scale well
- Look at communication patterns identified
  - Are there better ones for some of those?
  - Consider global communication for data needed by many processes.
- Try to make communications and calculations are concurrent (not in lock-step)
  - If not algorithm may not scale well
  - Redesign to improve scalability

See: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/

Foster’s Methodology (6)

- Step 3: Agglomeration
  - Move from abstract algorithm to implementation of a parallel program
  - Implement on a particular computing resource □ take hardware and architecture into account
  - Identify related tasks to group into a single process
    - Perform more computation per task to reduce communication
  - Identify what data can and should be replicated in multiple processes
    - Replicate calculations on several tasks rather than just one
  - Communication and memory accesses are slow
    - Reduce communication message count and size
    - Group related work together in a single task to reduce communication □ surface-to-volume effect
  - Preserve concurrent behavior □ this is a key
  - Code must support multiple numbers of processing elements □ code must adapt to new configurations (upgrades)

See: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/

Foster’s Methodology (7)

- Step 3 Goals
- Reduce communication costs
  - Increase locality of calculations □ more work per task
  - Verify that any data or computation replication cost is less than the benefit
- Maintain flexibility
  - Keep several options for scalability and mapping of tasks to processing elements on the table
  - Code must support working on a changing number of processing elements □ support upgrades or new systems with the same code
- Reduce software development and maintenance costs
  - Apply good software engineering practices □ make sure code is reusable and easily adapted to new applications
- Are the new tasks still load-balanced?
  - Can the number be reduced further and still be load-balanced? □ If yes, do this.
- Recheck scalability at start and end of this step □ use your time wisely

See: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/

Foster's Methodology (8)

- Step 4: Mapping
  - NP-Complete problem no known efficient algorithm to determine the OPTIMAL mapping
    - Heuristics are used to get a "good" solution that may be the optimal solution
    - Load-balancing is a good approach for data decomposition problems
    - Task-scheduling is a good approach for function decomposition problems
  - Assign tasks to processing elements
  - Maintain concurrent execution
    - Put these tasks on DIFFERENT processing elements
  - Minimize communication
    - Group tasks that communicate frequently on the SAME processing element
  - Determine the overhead costs associated with any manager components
    - Do these costs outweigh the benefits? Does the program still scale?

See: Designing and Building Parallel Programs, by Ian Foster, available at: https://www.mcs.anl.gov/~itf/dbpp/

Summary

- Classify a system according to Flynn's taxonomy (SISD, SIMD, MISD, MIMD).
  - SISD - CPU
  - SIMD - GPU

	Single instruction	Multiple instruction
Single data	SISD	MISD (?)
Multiple data	SIMD	MIMD

- Memory hierarchy provides fast access to a small amount of data.
  - Quick access to the data needed but not much more
  - Move data between fast and slow memories in advance of it being needed
- Foster's methodology to design parallel programs
  - Step 1: Partitioning, Step 2: Communication,
Step 3: Agglomeration, Step 4: Mapping