Parallelism & HPDC Platforms (Section 2.1 to 2.4.5)

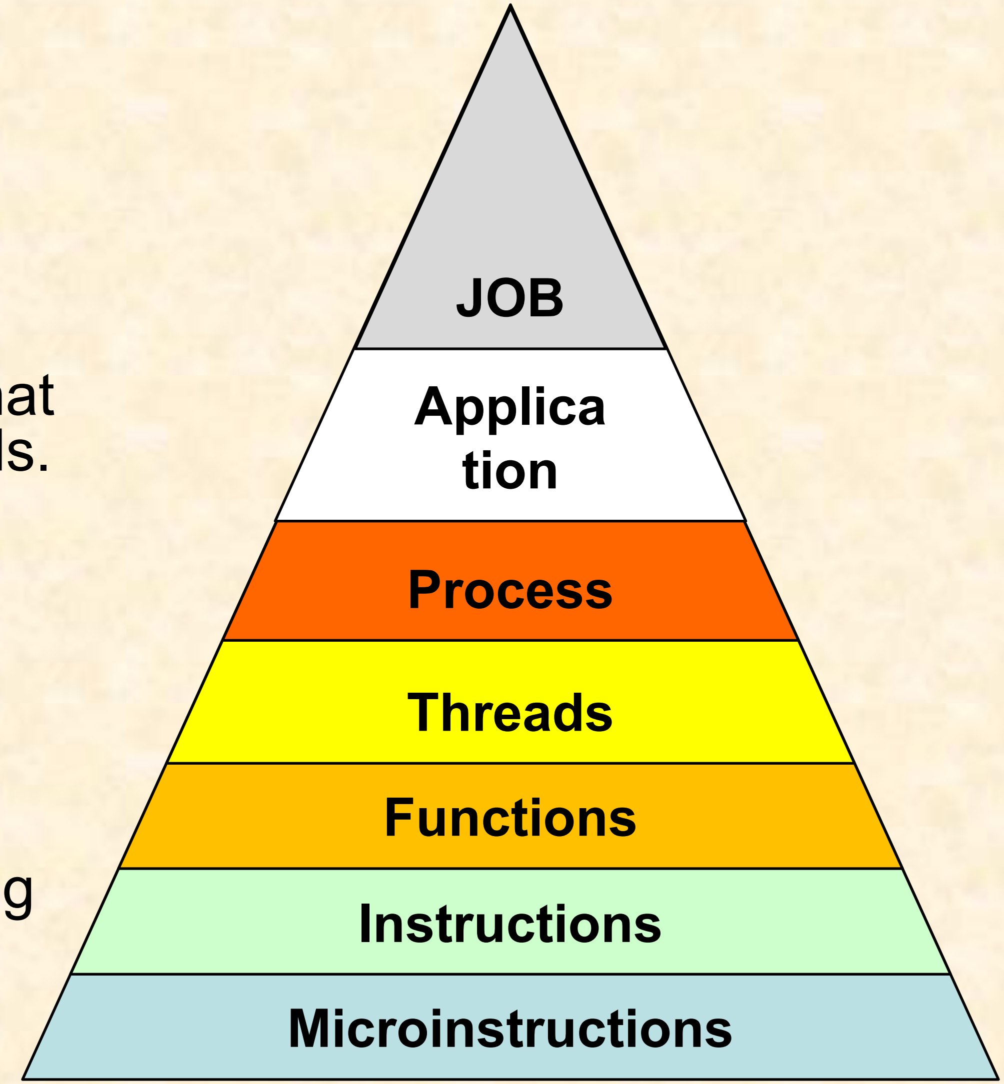
High Performance Computing
(CSE 443/543)

Parallelism

- The term parallelism is typically used to refer to the ability to perform two or more tasks at the same instant of time using independent computational devices
  - Computational devices may be
    - A separate unit (ALU or part of ALU) on a processor
    - A separate core on a microprocessor
    - A separate microprocessor or attached processor on the same computer
      - This includes video cards etc.
    - A completely different, but networked/interconnected computer.
  - Parallelism aims to effectively utilize <u>concurrency</u> in a task to reduce the overall computational time required to complete processing the task.

Tasks

- Tasks can be viewed at various levels:
  - Application level that consists of multiple programs or processes running simultaneously
  - Program/Process level that may have multiple threads.
  - Thread level that has several interoperating methods/functions
  - Method/function level consisting of several interrelated instructions
  - Instruction level consisting of several micro-instructions



Concurrency

- Concurrency
  - Refers to the lack of dependency or relationship between two tasks
    - Dependencies arise due to input-output relationships between tasks
  - Two tasks T_A and T_B are said to be concurrent if there is no dependency or relationship between T_A and T_B.
  - Identifying concurrency in tasks is challenging.
- Concurrent tasks can be executed in parallel
  - However, it is not necessary to execute concurrent tasks in parallel.
- The terms parallelism and concurrency are often interchangeably used.

Implicit vs. Explicit Parallelism

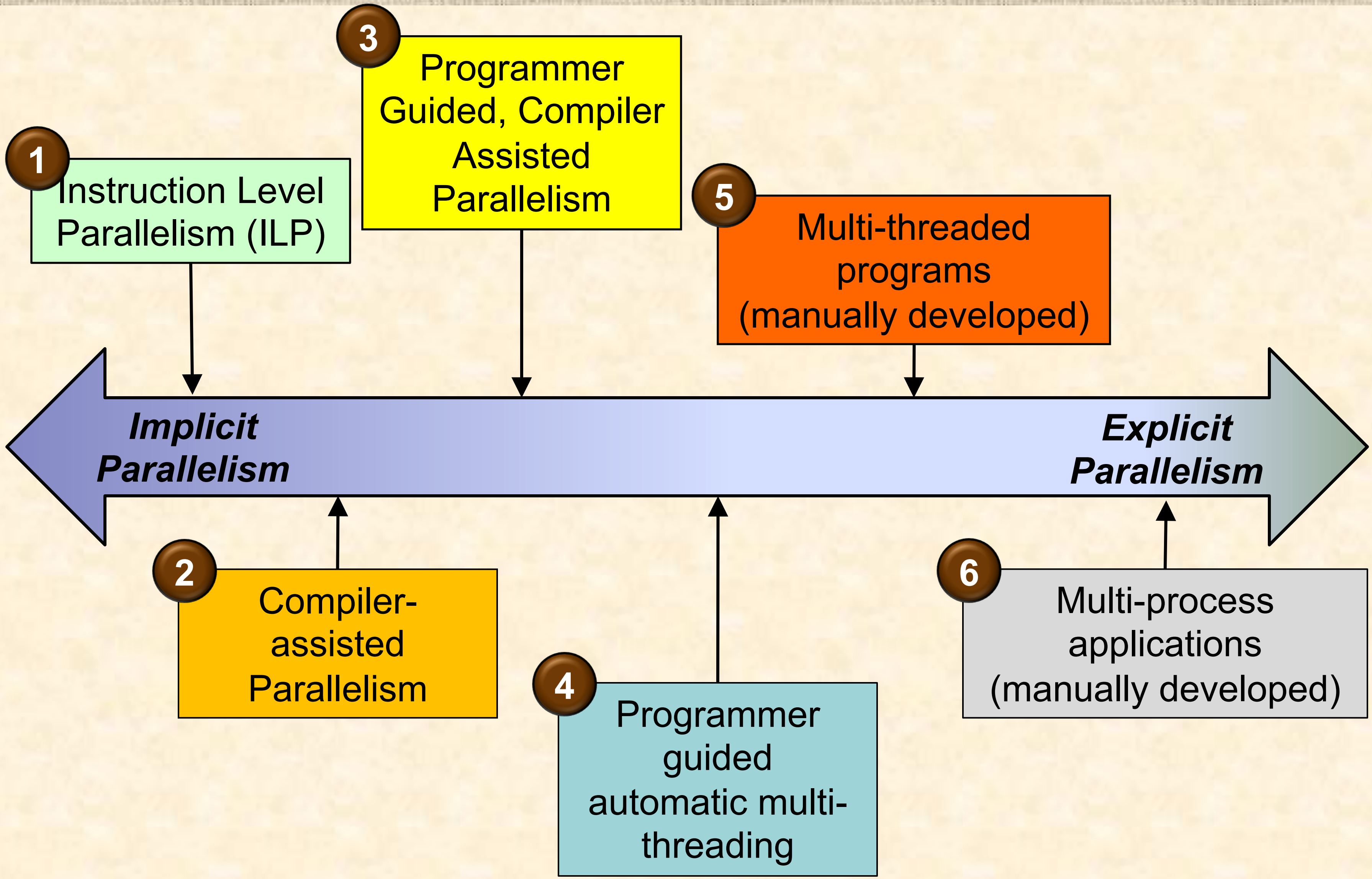
Implicit Parallelism

- Parallelism is automatically or semi-automatically realized
- Microprocessor and compiler orchestrate the task of extracting parallelism
- Typically operates at multi-instruction or multi-thread level
- Easily realizable for any program.
- Scope and gains are limited to a single processor or single machine

Explicit Parallelism

- Parallelism is manually (or programmatically) realized
- The programmer is responsible for developing the program to run in parallel
- Typically operates at multi-threaded or multi-processes level
- Requires considerable programming effort
- Can be applied to multiple processors, machines, or supercomputing clusters.

Spectrum of Parallelism



Instruction Level Parallelism (ILP)

- Instruction Level Parallelism (ILP)
    - This is classical implicit parallelism
  - Parallelism that is automatically extracted from a serial program by a microprocessor
    - This is a pure hardware solution
    - CPU aims to run concurrent instructions in parallel
  - Microprocessor has special architectural designs to enable ILP
- ILP is typically achieved via
  - Pipelining
    - Various architectural enhancements are used to minimize hazards and maximize pipelining
  - Hyper-threading
    - CPU uses complicated pipelines to run instructions from concurrent/independent threads in parallel
  - Superscalar architectures
    - CPU has multiple ALUs to run several concurrent operations in parallel
  - Single Instruction Multiple Data (SIMD)
    - A single instruction performs the same operation on multiple data elements in parallel
    - Modern x68 processors support SSE (Streaming SIMD Extensions) to perform limited number of algebraic operations (such as addition, subtraction, bitwise operations etc.)

Implicit ILP

- Modern microprocessors already execute several instructions in a program concurrently
  - There are two primary architectural features that are heavily used for this purpose
    - Pipelining
      - Processing of an instruction is divided into several independent stages
      - Various stages of different instructions are overlapped thereby enabling processing of multiple instructions to proceed simultaneously
    - Superscalar Execution
      - Multiple instructions are processed in parallel by introducing additional hardware components in each core/CPU
      - Instructions may even be executed out-of-order in a program!
        - » That is an instruction occurring later in a program may be executed before the preceding instruction(s).

Compiler-Assisted Parallelism

- Compiler-Assisted Parallelism
    - This falls under the category of implicit parallelism
  - This is a symbiosis between CPU and compiler
  - The compiler identifies concurrent instructions as it compiles high level source code
    - Compiler tags concurrent instructions
      - With special OP codes or
      - Packs instructions into Very Long Instruction World (VLIW) on VLIW architectures (such as Itanium)
  - Microprocessor executes concurrent instructions (as tagged by compiler) in parallel
  - This is a tradeoff between amount of hardware (transistors on silicon) and software overheads
    - Compiler needs to be highly targeted and optimized for the hardware

Programmer Assisted & Compiler Guided Parallelism

- Parallelism, including ILP, is extracted by compiler/CPU with some help from the programmer
  - This approach may appear as explicit parallelism -- but it is implicit parallelism
- Programmer Assistance includes:
  - Reorganizing parts of code to aid compiler to identify parallelism
    - Typically done through a process called Profile Guided Optimization (PGO) where sample execution of the program is analyzed in detail to identify concurrency in a program.
  - Tagging source codes with pragmas/compiler directives providing optimization tips to the compiler

Programmer Guided automatic Multi-threading

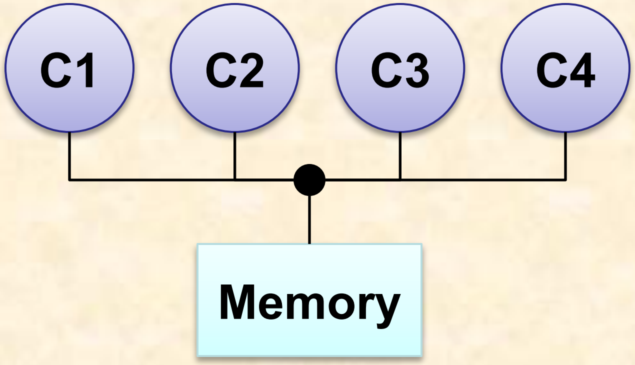
- Programming Language/compiler is extended/enhanced with special constructs that can be used by the programmer to identify macro regions with concurrency
  - Threads are sub-processes or Light Weight Processes (LWP)
    - Each thread runs different instructions on different data
  - Intermediate between implicit and explicit parallelism – but is classified as explicit parallelism
    - Example: OpenMP
    - Example: GNU C++ parallel libraries (uses OpenMP)
  - Provides special constructs to identify parts of the program to be run as multiple threads.
  - Compiler handles the task of creating, synchronizing, and destroying threads
    - The number of threads can vary depending on hardware
- This scheme is limited to shared-memory architectures

Shared memory architecture

- Shared memory: All CPUs (and cores) share and access the same main memory
  - Two types of shared memory architectures:

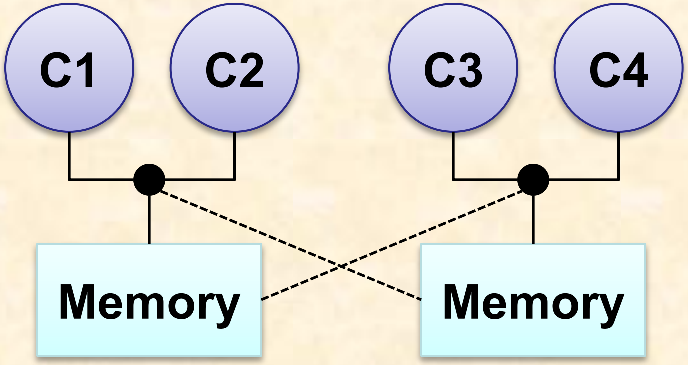
Uniform memory access (UMA)

Access to all memory addresses takes the same time from all cores



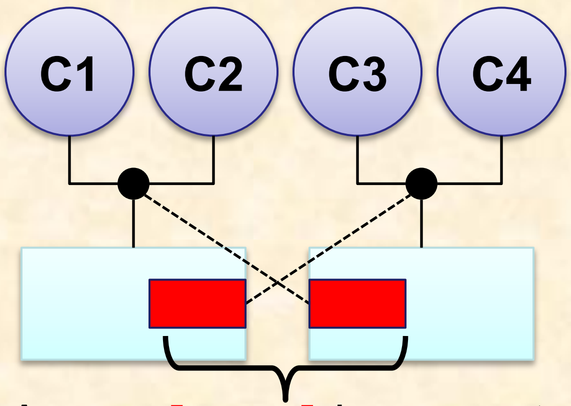
Non-Uniform memory access (NUMA)

Access to different memory addresses takes different times from different cores



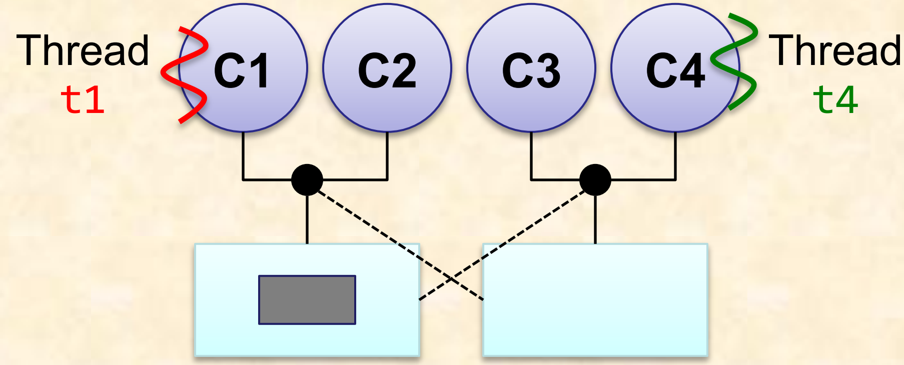
Impact of NUMA in programs

Split arrays/data structures



Array `a[0..n]` happens to get split between memory modules. So access to `a[0]` takes different time than to access a[n] depending on the core – *i.e.*, c1↦`a[0]` is faster but c1↦`a[n]` is slower.

Threads on different cores



Array `a[0..n]` happens to fit in 1 memory module. So access from thread `t1` is faster than access from thread `t4`.

Programmer Guided automatic Multi-threading (Contd.)

- Many C++ algorithms provide parallel implementations
  - See: https://gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html
  - Libraries use explicit parallelism via OpenMP
- Usage:
  1. Write standard C++ program using algorithms
     - Such as: `std::sort`, `std::transform`, etc.
  2. Test and ensure program work
  3. Specify compiler flags to enable multithreading
     - Just Add 2 flags: `-D_GLIBCXX_PARALLEL -fopenmp`

Multi-threaded Programs

- These are standard multi-threaded programs
  - Clearly falls under the category of explicit parallelism
  - Many languages provide libraries or language constructs to create and run multiple-threads
    - Most common solution for parallelism
    - With multi-core technology it is a mandatory design approach these days
  - It is the responsibility of the programmer to identify concurrency in a program and suitably develop the program as multiple threads
    - The programmer is responsible for arbitrating resource contentions, critical sections, and synchronization overheads
- It is a double edged sword
  - It is a very effective solution for parallelizing applications
  - However, programming overheads are much higher
  - It is hard to develop good multi-threaded programs

Multi-process Programming

- Here the program consists of multiple processes
  - Falls under the category of explicit parallelism
  - Ideal strategy for distributed memory and not just shared memory
    - Currently this is the only solution for most supercomputing clusters
  - Each process run on a different, but interconnected, computer
    - Each process may consists of multiple threads
  - It is the programmers responsibility to create, synchronize, and manage parallel processes
    - Makes program development a challenge
    - Many libraries are available to streamline programming
    - Can be classified into two categories
      - Single Program Multiple Data (**SPMD**) where multiple copies of the same executable are run on different machines. This is the most common one and the type we will be using in this course
      - Multiple Program Multiple Data (MPMD) where each process may run a different executable on different machines. This model is seldom used because of the complexity of programming