NE 155/255, Fall 2019
Parallel Computing Overview
December 6, 2019

Lecture notes adapted from Blaise Barney's "Introduction to Parallel Computing" tutorial at `https://computing.llnl.gov/tutorials/parallel_comp/`.

Serial Computing

Traditionally, we tend to think about algorithms in a serial way, and it has followed that software has been historically written serially:

- Problem broken into discrete series of instructions
- Instructions executed sequentially, one after another
- Single processor used for execution
- Only one instruction may execute at any point in time

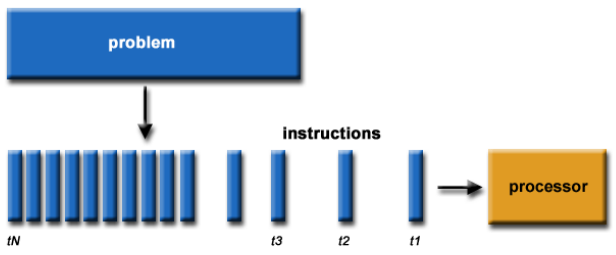


Figure 1: Serial computing.

Parallel Computing

On the other hand, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- Problem broken into discrete parts that can be solved concurrently
- Each part further broken down into series of instructions
- Instructions from each part execute simultaneously on different processors
- Some overall control/coordination mechanism is used to collect results

In order to use parallel computing techniques, we impose requirements on the computational problem that we're trying to solve.. The problem should be able to:

- Break down into discrete work pieces that can be solved simultaneously;
- Execute multiple program instructions at any point in time;
- Be solved in less time with multiple compute resources than with a single compute resource.

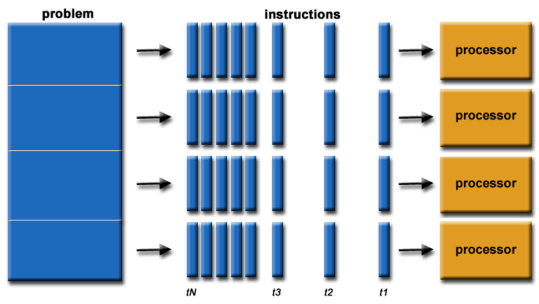


Figure 2: Parallel computing.

Parallel Computers

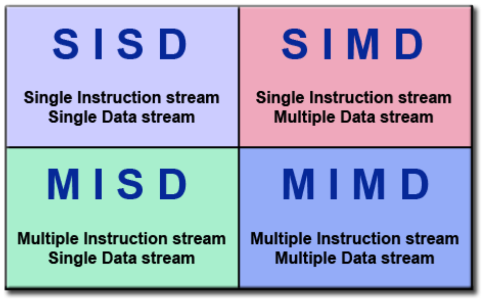
Parallel compute resources can be a single computer with multiple processors or a number of computers connected by a network. Today, any given standalone computer is typically parallel from a hardware perspective; it has multiple functional units, multiple execution units (processors), and mulitple hardware threads.

Networks connect multiple standalone computers ("nodes") to make larger parallel computer "clusters". So, each compute node is a multiprocessor parallel computer by itself, and then multiple compute nodes are networked together for even more parallelism.

Flynn's Taxonomy

There are different ways to classify parallel computers; one of the more widely-used classifications is called Flynn's Taxonomy. Flynn's Taxonomy defines the two independent dimensions of "instruction stream" and "data stream" and holds that aech dimension can have only one of two possible states "single" or "multiple". This gives us 4 possible classifications:

- **SISD**: single instruction, single data
- **SIMD**: single instruction, multiple data
- **MISD**: multiple instruction, single data
- **MIMD**: multiple instruction, multiple data


S I S D
Single Instruction stream
Single Data stream
S I M D
Single Instruction stream
Multiple Data stream
M I S D
Multiple Instruction stream
Single Data stream
M I M D
Multiple Instruction stream
Multiple Data stream

SISD

- Serial computer
- Only one instruction stream acted on by the CPU during any one clock cycle
- Only one data stream used as input during any one clock cycle
- Deterministic execution

SIMD

- Parallel computer
- All processing units execute the same instruction at a given clock cycle
- Each processing unit can operate on a different data unit
- Synchronous (lockstep) and deterministic execution
- Best suited for problems with a high degree of regularity (e.g., image processing)
- Most modern computers use SIMD instructions and execution units

MISD

- Parallel computer
- Each processor operates on data independently via separate instruction streams
- A single data stream is fed into multiple processors
- Few actual examples of this have ever existed

MIMD

- Parallel computer
- Every processor may be executing a different instruction stream
- Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Many MIMD architectures include SIMD execution sub-components
- Most current supercomputers fall into this category

Parallel Programming Models

Shared Memory (without Threads)

- Tasks share a common address space, which they read and write to asynchronously
- Mechanisms like as locks are used to control access to the shared memory, resolve contentions, and to prevent race conditions and deadlocks
- No need to specify explicitly the communication of data between tasks
- All processes see and have equal access to shared memory
- In terms of performance, it becomes more difficult to understand and manage data locality

Threading

- Threading is a type of shared memory programming
- A single "heavy weight" process can have multiple "light weight" concurrent execution paths
- Threads communicate with each other through global memory (updating address locations)
- Synchronization constructs are needed to ensure that more than one thread is not updating the same global address at any time
- Commonly-used implementations of threads are **POSIX threads** and **OpenMP**

Distributed Memory

- Also called the "message-passing" model
- This model uses a set of tasks that use their own local memory during computation
- Multiple tasks can reside on the same standalone computer and/or across a network of computers
- Tasks exchange data through communications by sending and receiving messages
- Data transfer usually requires cooperative operations to be performed by each process (e.g., a "send" operation must have a matching "receive" operation)
- **Message Passing Interface (MPI)** is the de facto standard for distributed computing

Designing Parallel Programs

Understand the problem and the program. The first step in developing parallel software is to understand the problem that your wish to solve in parallel. If you're starting with serial code, you must also understand the existing code. Before spending any time attempting to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

- Partitioning
  - Break the problem into discrete "chunks" of work that can be distributed to multiple tasks
  - Domain decomposition and functional decomposition
- Communications
  - The need for communications between tasks depends upon your problem
  - "Embarrassingly parallel" problems need little to no communication
  - Communication always incurs some overhead
  - Latency versus bandwidth
  - Synchronous (blocking) versus asynchronous (non-blocking)
- Synchronization
  - Managing the sequence of work and the tasks performing it is a critical design consideration
  - Can be a significant factor in program performance
  - Often requires "serialization" of segments of the program
  - Barrier, lock/semaphore, synchronous communications
- Data dependencies
  - A dependence exists between program statements when the order of statement execution affects the results of the program
  - A data dependence results from multiple use of the same location(s) in storage by different tasks

  - Dependencies are one of the primary inhibitors to parallelism

- Load balancing
  - Distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time
  - Minimization of task idle time
  - Important for performance reasons; the slowest task will determine overall performance

- Granularity
  - Qualitative measure of the ratio of computation to communication
  - Fine-grained versus coarse-grained parallelism
  - Most efficient granularity is dependent on the algorithm and the hardware environment in which it runs

- I/O
  - Generally regarded as inhibitor to parallelism
  - I/O operations require orders of magnitude more time than memory operations

- Debugging
  - Can be incredibly difficult, especially as codes scale

- Performance Analysis and Tuning
  - Much more challenging for parallel software than serial code