Introduction to Parallel Computing

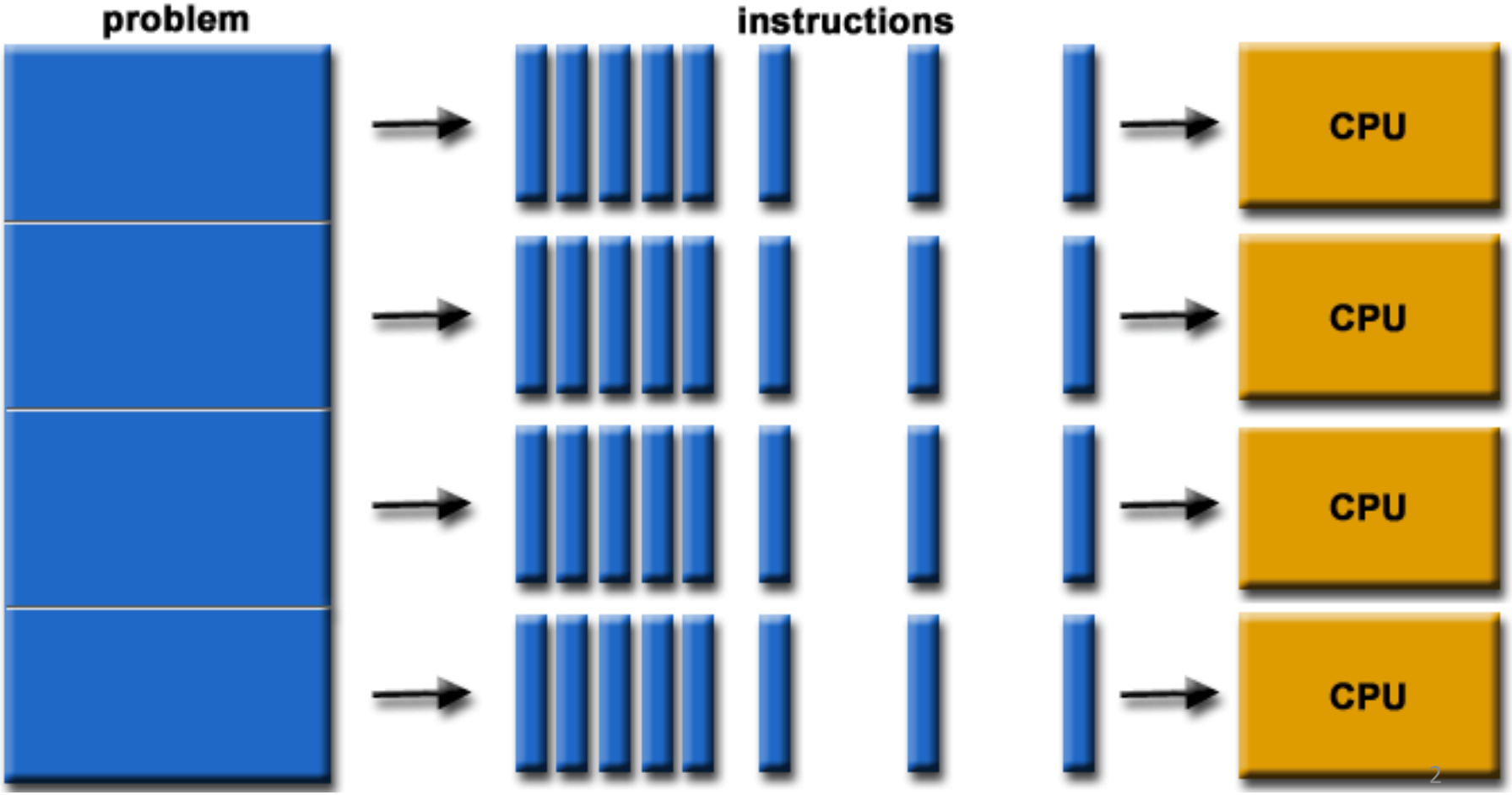
CPS 5401 Fall 2014

Shirley Moore, Instructor

October 13, 2014

Definition of Parallel Computing

- Simultaneous use of multiple compute resources to solve a computational problem



Definition (2)

- The compute resources might be
  - A single computer with multiple processors and/or cores;
  - An arbitrary number of computers connected by a network;
  - A combination of both.
- The computational problem should be amenable to
  - being broken apart into discrete pieces of work that can be solved simultaneously;
  - executing multiple program instructions at any moment in time;
  - being solved in less time with multiple compute resources than with a single compute resource.

Why Use Parallel Computing?

- The universe is massively parallel.
- Save time and/or money
- Solve larger problems
  - May not be able to fit data into memory of single computer
- Provide concurrency
  - Necessary to take advantage of all resources on multicore/manycore systems
- Trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing***.

Parallel Terminology

- **Node**
  - A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores. Nodes are networked together to comprise a supercomputer.
- **CPU / Socket / Processor / Core**
  - This varies, depending upon whom you talk to. In the past, a CPU (Central Processing Unit) was a singular execution component for a computer. Then, multiple CPUs were incorporated into a node. Then, individual CPUs were subdivided into multiple "cores", each being a unique execution unit. CPUs with multiple cores are sometimes called "sockets" - vendor dependent. The result is a node with multiple CPUs, each containing multiple cores. The nomenclature is confusing at times.

Terminology (2)

- Task
  - A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.
- Pipelining
  - Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

Terminology (3)

- Shared memory
  - From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.
- Symmetric Multi-Processor (SMP)
  - Hardware architecture where multiple processors share a single address space and access to all resources.
- Distributed memory
  - In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communication to access memory on other machines where other tasks are executing.

Terminology (4)

- Communication
  - Exchange data between parallel tasks. There are several ways this can be accomplished, such as through a shared memory or over a network.
- Synchronization
  - The coordination of parallel tasks in real time, very often associated with communication. Often implemented by establishing a synchronization point within an application where a task may not proceed further until other tasks reaches the same or logically equivalent point.
- Granularity
  - A qualitative measure of the ratio of computation to communication.
    - ***Coarse:*** relatively large amounts of computational work are done between communication events
    - ***Fine:*** relatively small amounts of computational work are done between communication events

Speedup and Overhead

- Speedup = $\frac{\text{Wallclock time of serial execution}}{\text{Wallcock time of parallel execution}}$

- Parallel overhead
  - The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
    - Task start-up time
    - Synchronizations
    - Data communications
    - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
    - Task termination time

Scalability

- Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors.
- Factors that contribute to scalability include:
  - Hardware
    - Especially memory-CPU and network bandwidths
  - Application algorithm
  - Parallel overhead
  - Characteristics of your specific implementation

Types of Scalability

- Strong scaling
  - Fixed overall problem size
  - Goal is to reduce time to solve the problem
  - Ideal is perfect linear speedup (e.g., with twice as many processors, the problem is solved in half the time)
- Weak scaling
  - Fixed problem size per processor
  - Scale up overall problem size with number of processors (e.g., with twice as many processors, double the overall problem size)
  - Ideal is to solve the larger problem in the same amount of time
- Iso-efficiency scaling
  - Increase problem size along with increase in number of processors so as to maintain constant efficiency in use of resources
  - Often requires increases problem size per processor

Parallel Programming Models

- There are several parallel programming models in common use:
  - Shared Memory
    - Threads
  - Distributed Memory / Message Passing
  - Data Parallel
  - Hybrid
  - Single Program Multiple Data (SPMD)
  - Multiple Program Multiple Data (MPMD)
- Parallel programming models exist as an abstraction above hardware and memory architectures.

Shared Memory

- Tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks, simplifying program development.
- A disadvantage in terms of performance is that it becomes more difficult to understand and manage data locality. Keeping data local to the processor that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processors use the same data.

Threads

- A type of shared memory programming in which each process can have multiple execution paths
- From a programming perspective, threads implementations commonly comprise
  - A library of subroutines that are called from within parallel source code – e.g., Pthreads, Java threads
  - A set of compiler directives embedded in either serial or parallel source code – e.g., OpenMP

POSIX Threads (aka Pthreads)

- Library based; requires parallel coding
- Specified by the IEEE POSIX 1003.1c standard (1995).
- C Language only
- Commonly referred to as Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
- Very explicit parallelism; requires significant programmer attention to detail.

OpenMP

- Compiler directive based
- Jointly defined and endorsed by a group of major computer hardware and software vendors
  - www.openmp.org
- Portable / multi-platform
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism“
- Until OpenMP 4.0, didn’t have support for data locality => can perform poorly

Distributed Memory/Message Passing Model

- A set of tasks that use their own local memory during computation.
  - Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process.
  - For example, a send operation must have a matching receive operation.

Message Passing Model Implementation

- From a programming perspective, message passing implementations usually comprise a library of subroutines.
  - Calls to these subroutines are embedded in source code.
  - The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
  - www.mpi-forum.org
- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms. Not all implementations include everything in the standard.

Hybrid Model

- A hybrid model combines more than one of the previously described programming models.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI
- This hybrid model lends itself well to the increasingly common hardware environment of clustered multi/many-core machines.
- Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming.
  - GPUs perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI

Automatic vs. Manual Parallelism (or something in between)

- Designing and developing parallel programs has characteristically been a very manual process.
- Manually developing parallel codes is a time consuming, complex, error-prone and ***iterative*** process.
- The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.
- A parallelizing compiler generally works in two different ways:
  - Fully Automatic
    - The compiler analyzes the source code and identifies opportunities for parallelism.
    - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
    - Loops (do, for) loops are the most frequent target for automatic parallelization.
  - Programmer Directed
    - Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
    - May be able to be used in conjunction with some degree of automatic parallelization also.
- If you are beginning with an existing serial code and have time or budget constraints, then automatic parallelization may be the answer. However, there are several important caveats that apply to automatic parallelization:
  - Wrong results may be produced
  - Performance may actually degrade
  - Much less flexible than manual parallelization
  - Limited to a subset (mostly loops) of code
  - May actually not parallelize code if the analysis suggests there are inhibitors or the code is too complex

Designing Parallel Programs

- Understand the problem
- Decompose the problem
- Design inter-task communication and synchronization (requires understanding dependencies)

Cost of Communication

- Inter-task communication virtually always implies overhead.
- Machine cycles and resources that could be used for computation are instead used to package and transmit data.
- Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

Synchronous vs. Asynchronous Communication

- Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
- Synchronous communications are often referred to as ***blocking*** communications since other work must wait until the communications have completed.
- Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
- Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place.
- Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

Scope of Communication

- Knowing which tasks must communicate with each other is critical during the design stage of a parallel code.
- Both of the two scopings described below can be implemented synchronously or asynchronously.
  - ***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - ***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective.

Types of Synchronization

- Barrier synchronization
- Lock/semaphore
- Synchronous communication

Barrier Synchronization

- Usually implies that all tasks are involved
- Each task performs its work until it reaches the barrier. It then stops, or "blocks".
- When the last task reaches the barrier, all tasks are synchronized.
- What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

Lock/semaphore

- Can involve any number of tasks
- Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
- The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
- Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
- Can be blocking or non-blocking

Synchronous Communication

- Involves only those tasks executing a communication operation
- When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication.
- For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

Latency vs. Bandwidth

- ***Latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- ***Bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
- Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

Data Dependencies

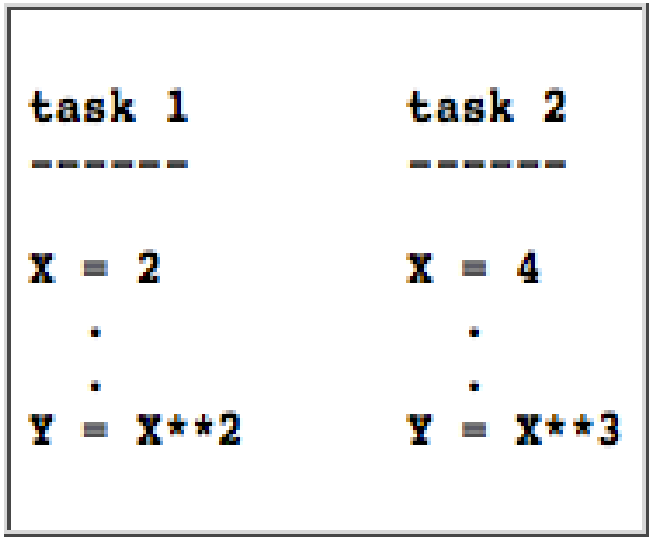
- A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.
- A ***data dependence*** results from multiple use of the same memory location by different tasks.
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.
- How to Handle Data Dependencies:
  - Distributed memory architectures - communicate required data at synchronization points.
  - Shared memory architectures - synchronize read/write operations between tasks.

Example: Loop-carried data dependence

```
DO 500 J = MYSTART,MYEND
      A(J) = A(J-1) * 2.0
500 CONTINUE
```

- The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.
- If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:
  - Distributed memory architecture - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
  - Shared memory architecture - task 2 must read A(J-1) after task 1 updates it

Example: Loop independent data dependence



The value of Y is dependent on:

- Distributed memory architecture - if or when the value of X is communicated between the tasks.
- Shared memory architecture - which task last stores the value of X.