

Portland State University
ECE 588/688

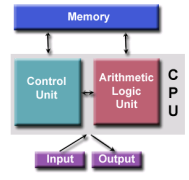
Introduction to Parallel Computing

Reference: Lawrence Livermore National Lab Tutorial
https://computing.llnl.gov/tutorials/parallel_comp/

© Copyright by Alaa Alameldeen 2009

von Neumann Architecture

- Uses the stored-program concept
 - ◆ The CPU executes a stored program that specifies a sequence of memory read and write operations



- Basic design:
 - ◆ Memory stores both program and data
 - ◆ Program instructions are coded data which instructs the computer on what to do
 - ◆ Data: Information to be used by the program
 - ◆ CPU gets instructions & data from memory, decodes instructions and then performs them sequentially

Portland State University – ECE 588/688 – Fall 2009 2

Multiprocessor Taxonomy (Flynn)

- Instructions and Data streams can be either single or multiple
- **Single Instruction, Single Data (SISD)**
 - ◆ Serial, non-parallel computer – e.g., single CPU PCs, workstations and mainframes
- **Single Instruction, Multiple Data (SIMD)**
 - ◆ All processor units execute same instruction on different data
 - ◆ Example: Vector processors such as IBM 9000, Cray C90
- **Multiple Instruction, Single Data (MISD)**
 - ◆ Rare (e.g., C.mmp), data operated on by multiple instructions
- **Multiple Instruction, Multiple Data (MIMD)**
 - ◆ Most modern parallel computers
 - ◆ Processors may be executing different instructions on different data streams

> Review Terminology from tutorial

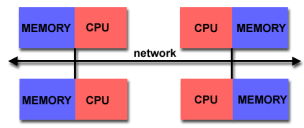
Portland State University – ECE 588/688 – Fall 2009 3

Shared Memory Architecture

- All processors can access all memory
- Processors share memory resources, but can operate independently
- One processor's memory changes are seen by all other processors
- **Uniform Memory Access (UMA) Architecture**
 - ◆ Example: Symmetric Multiprocessor (SMP) machines
 - ◆ Identical processors with equal access and equal access time to memory
 - ◆ Also called CC-UMA - Cache Coherent UMA. Cache coherent means that if one processor updates a location in shared memory, all the other processors know about the update
- **Non-Uniform Memory Access (NUMA) Architecture**
 - ◆ Often made by physically linking two or more SMPs
 - ◆ One processor can directly access memory of another processor
 - ◆ Not all processors have equal access time to all memories
 - ◆ Memory access across link is slower
 - ◆ Called CC-NUMA - Cache Coherent NUMA - if CC is maintained
- Pros and Cons?

Portland State University – ECE 588/688 – Fall 2009 4

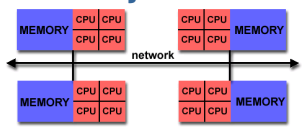
Distributed Memory Architecture



- Require a communication network to connect inter-processor memory
- Processors have their own local memory
 - ◆ Memory addresses in one processor do not map to another processor
 - ◆ No concept of global address space across all processors.
- Each processor operates independently, changes to its local memory have no effect on the memory of other processors
 - ◆ Cache coherence does not apply
- Programs explicitly define how and when data is communicated and how tasks are synchronized
- Pros and cons?

Portland State University – ECE 588/688 – Fall 2009 5

Hybrid Distributed-Shared Memory Architecture



- Shared memory component is usually a CC SMP
 - ◆ Processors on a given SMP can address that machine's memory as global memory
- Distributed memory component is the networking of multiple SMPs
 - ◆ SMPs know only about their own memory
 - ◆ Network communications needed to move data between SMPs
- Largest, fastest computers share this architecture

Portland State University – ECE 588/688 – Fall 2009 6

Parallel Programming Models

- Parallel programming models:
 - ◆ Shared Memory
 - ◆ Threads
 - ◆ Message Passing
 - ◆ Data Parallel
 - ◆ Hybrid
- Parallel programming models exist as an abstraction above hardware and memory architectures
- Models not specific to certain types of memory architecture or machine
 - ◆ Shared memory can be implemented on a distributed memory architecture
 - ◆ Message passing can be implemented on a shared memory architecture (e.g., SGI Origin)

Portland State University – ECE 588/688 – Fall 2009 7

Shared Memory Model

- Tasks share a common address space, which they read and write asynchronously
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory
- Relatively simple programming model
 - ◆ No notion of data "ownership"
 - ◆ No need to specify explicitly communication of data between tasks
- More difficult to understand and manage data locality
- Implementations:
 - ◆ On shared memory platforms, compilers translate user program variables into actual memory addresses, which are global
 - ◆ KSR ALLCACHE approach provide a shared memory view of data even though the physical memory of the machine was distributed

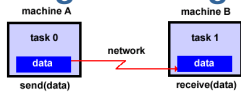
Portland State University – ECE 588/688 – Fall 2009 8

Threads Model

- Single process can have multiple, concurrent execution paths
- Analogy: a single program that includes a number of subroutines
 - ◆ Program creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently
- Each thread has local data, but also shares the entire resources of the program (e.g., memory space)
- Threads communicate with each other through global memory by updating address locations
 - ◆ This requires synchronization constructs to insure that more than one thread is not updating the same global address at the same time
- Threads are commonly associated with shared memory architectures and operating systems
- Implementation standards
 - ◆ POSIX threads (Pthreads): Library-based, available in C
 - ◆ OpenMP: Compiler-directive-based

Portland State University – ECE 588/688 – Fall 2009 9

Message Passing Model



- Tasks use their own local memory during computation
- Multiple tasks can run on the same physical machine OR across an arbitrary number of machines
- Tasks exchange data through communications by sending and receiving messages
 - ◆ Requires cooperative operations by each process, e.g., a send operation must have a matching receive operation
- Implementations
 - ◆ Library of subroutines that are imbedded in source code. The programmer is responsible for determining all parallelism.
 - ◆ Message Passing Interface (MPI) is the current industry standard for message passing
 - ◆ For shared memory architectures, MPI implementations use shared memory (memory copies) for task communications

Portland State University – ECE 588/688 – Fall 2009 10

Data Parallel Model

- Most parallel work focuses on performing operations on a data set
 - ◆ Data set is typically organized into a common structure, such as an array or cube
- Each task works on a different partition of the same data structure, all tasks work collectively on the structure
- Tasks perform the same operation on their partition of work, e.g., "add 4 to every array element"
- On shared memory architectures, all tasks may have access to the data structure through global memory
- On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task

➤ Other programming models (see tutorial)

Portland State University – ECE 588/688 – Fall 2009 11

Designing Parallel Programs: Concepts

- Parallelization: Automatic vs. Manual
 - ◆ Manual parallelization is more common, but automatic compilers exist
- Understand the program and problem
 - ◆ Need to decide whether program can be parallelized, investigate algorithms, identify hotspots (where most work is done) and bottlenecks
- Partitioning
- Communications
- Synchronization
- Data dependencies
- Load balancing
- Granularity
- I/O
- Costs and limits
- Performance analysis & tuning: Harder than serial programs

Portland State University – ECE 588/688 – Fall 2009 12

Partitioning

- Breaking the problem into discrete "chunks" of work that can be distributed to multiple tasks
 - ◆ Two main types: domain decomposition and functional decomposition
- Domain Decomposition
 - ◆ Data is decomposed, each task works on portion of data
 - ◆ Different ways to partition data (see figures in tutorial)
- Functional Decomposition
 - ◆ The problem is decomposed according to the work that must be done, each task performs a portion of the overall work
 - ◆ See examples in tutorial
- Programmer can combine both types

Portland State University – ECE 588/688 – Fall 2009

13

Communications

- Some problems can be decomposed and executed in parallel with virtually no need data sharing, called *embarrassingly parallel* problems
 - ◆ Example: image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work
- Most parallel applications are not quite so simple, and do require data sharing between tasks
- Issues to consider when writing a program that needs communications
 - ◆ Cost of communications
 - ◆ Latency vs. bandwidth
 - ◆ Visibility of communications
 - ◆ Synchronous vs. asynchronous communications
 - ◆ Scope of communications
 - ◆ Efficiency of communications
 - ◆ Overhead and complexity

Portland State University – ECE 588/688 – Fall 2009

14

Synchronization

- **Barrier** (all tasks)
 - ◆ Each task performs its work until it reaches the barrier then stops (blocks)
 - ◆ When the last task reaches the barrier, all tasks are synchronized
 - ◆ Tasks can then continue or serial work is done
- **Lock / semaphore** (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 own the lock / semaphore / flag.
 - ◆ The first task to acquire the lock "sets" it and then it 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 operations** (tasks performing communication operation)
 - ◆ Coordination is required with the other task(s) participating in the communication, e.g., before a task can perform a "send" operation, it must receive an acknowledgment from the receiving task that it is OK to send.

Portland State University – ECE 588/688 – Fall 2009

15

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
- Loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts (see code example in tutorial)
- How to handle data dependencies
 - ◆ Distributed memory architectures: communicate required data at synchronization points
 - ◆ Shared memory architectures: synchronize read/write operations between tasks

Portland State University – ECE 588/688 – Fall 2009

16

Load Balancing

- Refers to distributing work among tasks so that all tasks are kept busy all the time (i.e., minimize idle time)
- Important for performance reasons
 - ◆ If tasks are synchronized by a barrier, slowest task determines the overall performance
 - ◆ See figure in tutorial
- How to achieve load balancing
 - ◆ Equally partition the work each task receives
 - ◆ Dynamic work assignment

Portland State University – ECE 588/688 – Fall 2009

17

Granularity

- Granularity is a qualitative measure of the ratio of computation to communication
- Periods of computation are typically separated from periods of communication by synchronization events.
- **Fine-grain Parallelism**
 - ◆ Relatively small amounts of computational work are done between communication events, low computation to communication ratio
 - ◆ Facilitates load balancing
 - ◆ High communication overhead, less opportunity to improve performance
 - ◆ Overhead required for communications and synchronization between tasks can be longer than the computation
- **Coarse-grain Parallelism**
 - ◆ Relatively large amounts of computational work are done between communication/synchronization events
 - ◆ High computation to communication ratio
 - ◆ Implies more opportunity for performance increase
 - ◆ Harder to load balance efficiently

Portland State University – ECE 588/688 – Fall 2009

18

I/O

- I/O operations are inhibitors to parallelism
- Parallel I/O systems are immature or not available for all platforms
- In an environment where all tasks see the same filespace, write operations will result in file overwriting
- Read operations will be affected by the fileserver's ability to handle multiple read requests at the same time
- I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks

Limits and Costs

- Amdahl's Law:
 - ◆ P: Parallel portion, S: Serial portion = 1-P, N: Number of processors

$$Speedup = \frac{1}{S + P/N} = \frac{1}{1 - P + P/N}$$

- ◆ Speedup is also the ratio of serial execution time to parallel execution time
- Complexity
 - ◆ Parallel applications more complex than serial applications (longer time to design, code, debug, tune and maintain)
- Scalability
 - ◆ A scalable problem has proportionally higher speedup when more processors are added
 - ◆ A scalable system provides proportionally higher speedups for a scalable problem when more processors are added
 - ◆ Limited by algorithm, hardware, and parallelization overhead

Reading Assignment

- Monday: Kunle Olukotun et al., "The Case for a Single-Chip Multiprocessor," ASPLOS 1996 (Review due in my inbox before class)
- HW1 due Monday – turn in paper copy in class (not by email)
- Wednesday: Deborah T. Marr et al., "Hyper-Threading Technology Architecture and Microarchitecture," Intel Technology Journal, 2001 (Review)