

## Topic 10: Parallel and distributed algorithms

1. Concurrent computing
2. Modeling concurrency
3. Parallel programming
4. Parallel algorithms
5. Distributed algorithms
6. Parallel and distributed complexity

*Reading: Ch. 12*

### 1. Concurrent computing

- A processor may run two or more programs at the same time (multitasking)
- To do so, it saves its current state in one program's fetch/execute cycle, and loads the state of another program's cycle, to run a time slice of that program, numerous times per second
- A *process* is a programming abstraction that simulates ownership of all computer resources

## Some parallelizable problems

*Challenges:*

*Given as many processors as you need, and shared memory:*

- Search an unsorted array in constant time
- Find maximum of a list in log time
- Add an array in logarithmic time

## Why parallel computing?

- Networked PCs can cooperate today to perform gigantic tasks in parallel cheaply
- Multi-CPU servers are common today to meet the needs of client/server computing
- “Moore’s Law”, which predicts computer speed doubling every 1.5 years, is losing steam
- In a 1-gigahertz CPU, electricity travels only 1 inch per clock cycle, close to size of chip
- The brain’s billions of neurons work in parallel

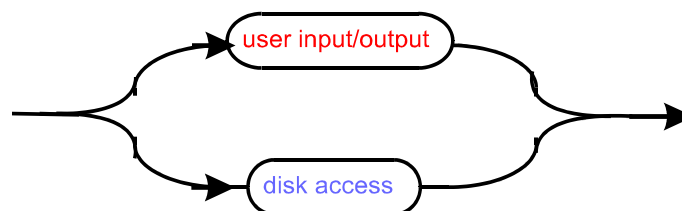
## Serial computing

- The 50-year-old *Von Neumann architecture* defines most computing today:
  - one processor
  - data is in memory
  - program is in memory
  - enables general-purpose computing
- The microprocessor carries out a *fetch/execute cycle*, retrieving and executing machine instructions one at a time
- *Multi-core* machines break the one-processor assumption



## A Java thread simulates ownership of a CPU

- One program may run multiple threads, e.g., for disk access and user I/O
- The processor or processors execute multiple threads concurrently



## 2. Modeling concurrency

- *Circuit model*: Processing is hard-wired into a combinational circuit that executes an algorithm
- *PRAM* (Parallel Random-Access Machine): an extension of the Von Neumann architecture to multiple CPUs sharing memory
- *Process algebras, e.g.,  $\pi$  Calculus*: Assume communication by message passing

## PRAM model

- Assume all processors have access to *shared* memory in  $O(1)$  time
- Each processor also has private memory
- Shared memory is globally addressable
- PRAMs are {concurrent, exclusive} read, {concurrent, exclusive} write – CRCW, CREW, ERCW, EREW
- PRAM is the standard assumption for theoretical work in parallel computation

## Single and multiple instructions and data

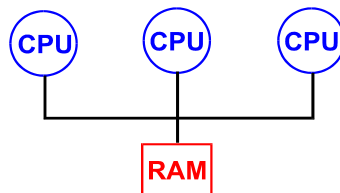
### MIMD

- Most common
- Every processor may execute a different instruction stream on a different data stream
- May be synchronous or asynchronous, deterministic or nondeterministic

### SIMD

- All CPUs synchronously execute the same instruction, but on different data items
- *Varieties*: processor arrays, vector pipelines
- Deterministic

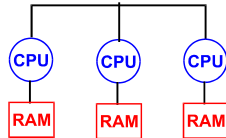
## Shared-memory architectures



- All CPUs access all memory as global address space
- Memory access may be uniform (UMA) or non-uniform (NUMA)
- Where processors use cache, *cache coherency* may be an issue

## Distributed-memory and hybrid architectures

- Memory addresses in *distributed-memory systems* do not map between CPUs; no global address space exists



- Cache coherency does not apply
- Scalability is in thousands of CPUs
- *Hybrid* distributed-shared memory systems network multiple symmetric multiprocessors

## Great speed gains are possible with parallelism

- *The folklore speedup theorem:*  
with  $p$  processors, the maximum speedup, versus one processor, is  $p$  -- *false!*
- *Counter-examples:*
  - if RAM access is slower than network communication
  - if doubling data doubles cache size
  - if intermediate calculations occupy great time resources

### 3. Parallel programming

- *Programming model* is an abstraction lying above hardware and memory architecture and is not architecture specific
- *Shared memory*
  - May be implemented using distributed physical memory as virtual memory
  - Programming interface is simple, using global addresses
- *Threads*
  - Like subroutines, called in parallel, sharing some memory
  - UNIX (POSIX) and Java support threads

### Cache coherence

- To make slow RAM access appear fast, a *cache* is used on CPUs today, parallel and sequential
- Using a cache is somewhat like keeping a phone list on your desk to avoid paging through a large phone book
- CPUs that share memory need to keep a common view of RAM



## Other parallel programming models

- *Message-passing interface (MPI)*
  - Tasks exchange data via messages
  - MPI (1994) is de facto industry standard
  - MPI may be used in shared-memory architectures
- *Data parallel*: parallel tasks work on different parts of a common data structure
- *Hybrid*: combines the four above programming models, e.g., MPI w/ threads or shared memory

## Designing parallel programs

- Problem is *parallelizable* if it may be partitioned into *independent* sub-problems, e.g., uniformly changing pixel values
- Communication between tasks is necessary if tasks share data; e.g., heat diffusion problem requires knowledge of neighboring data
- Types of synchronization
  - *Barrier*: all tasks halt at barrier until last task reaches it
  - *Lock/semaphore*: serializes access to global data or to critical section of code
  - Synchronization of (message) *communication*

## Concerns in parallel programming

- *Data dependencies*: Tasks cannot be parallel if one task uses data that is dependent on another
- *Load balancing*: Distributing work among tasks to minimize idle time
- *Granularity*: Measure of ratio of computation to communication
- *Amdahl's Law*: Maximum speedup is limited by the fraction of code that is parallelizable

## 4. Parallel algorithms

### Addition

Add  $(n/2)$  adjacent pairs, then add sums pairwise, like tournament, until done

|   |    |    |    |   |    |   |   |
|---|----|----|----|---|----|---|---|
| 5 | 2  | 3  | 8  | 1 | 6  | 4 | 5 |
| 7 |    | 11 |    | 7 |    | 9 |   |
|   | 18 |    |    |   | 16 |   |   |
|   |    |    | 34 |   |    |   |   |

Parallel time =  $\log_2(n)$