

# HY425 Lecture 16: Shared-Memory Multiprocessors

Dimitrios S. Nikolopoulos

University of Crete and FORTH-ICS

December 5, 2011

# Outline

**Motivation**

Multiprocessors

Challenges

Bus-based shared-memory multiprocessors

# Why do we need multiprocessors?

## Uniprocessor performance

- ▶ 25% annual improvement rate from 1978 to 1986
- ▶ 52% annual improvement rate from 1986 to 2002
  - ▶ Profound impact of RISC, x86
- ▶ 20% annual improvement rate from 2002 to present
  - ▶ Power wall: solutions for higher ILP are power-inefficient
  - ▶ ILP wall: hard to exploit more ILP
  - ▶ Memory wall: ever-increasing memory latency relative to processor speed

# Has this been attempted before?

## Flashback in the 70s

- ▶ In the 70s, many thought that uniprocessors will reach their limits, so replicating processors would be the only way to achieve higher performance
- ▶ Predictions proved wrong because of Moore's law, architecture innovation (RISC), and inability to build, program, and maintain easily scalable multiprocessors (too expensive, too hard to program, too slow to build)
- ▶ **What has changed now?**

## Parallelism at the chip-level

Vendor-Year	AMD (05)	Intel (06)	IBM (04)	Sun (05)	NVIDIA (07)
Processors/chip	2	2	2	8	128+
Threads/processor	1	2	2	4	2+
Threads/chip	2	4	4	32	768 (active) 2 billion+ (resident)

### Technology trends

- ▶ Power-capped processor design motivates use of parallelism for performance
- ▶ Design by replication: leverage one design many times

# Parallelism in applications

## Data parallelism

- ▶ Databases, file servers
- ▶ Graphics, games
- ▶ Scientific computing
- ▶ **More recently: clients, browsers**

## Request-level parallelism

- ▶ Servers, planetary-scale services

# Flynn's Taxonomy

## Classification based on data and control streams

- ▶ Jason Flynn, Very High-Speed Computers, *Proceedings of the IEEE*, Vol. 54, pp. 1900–1909, Dec. 1966
- ▶ **SISD**: Single-instruction, single-data
- ▶ **MISD**: Multiple-instruction, single-data (impractical)
- ▶ **SIMD**: Single-instruction, multiple-data
  - ▶ Data-level parallelism, vector instructions
  - ▶ Variation: **SIMT**, Single-instruction, multiple-threads enables divergence in instructions via conditionals (NVIDIA GPUs)
- ▶ **MIMD**: Multiple-instruction, multiple-data
  - ▶ Most common form of multi-processing
  - ▶ Flexible (used via multi-programming, or multi-threading)

# Outline

Motivation

**Multiprocessors**

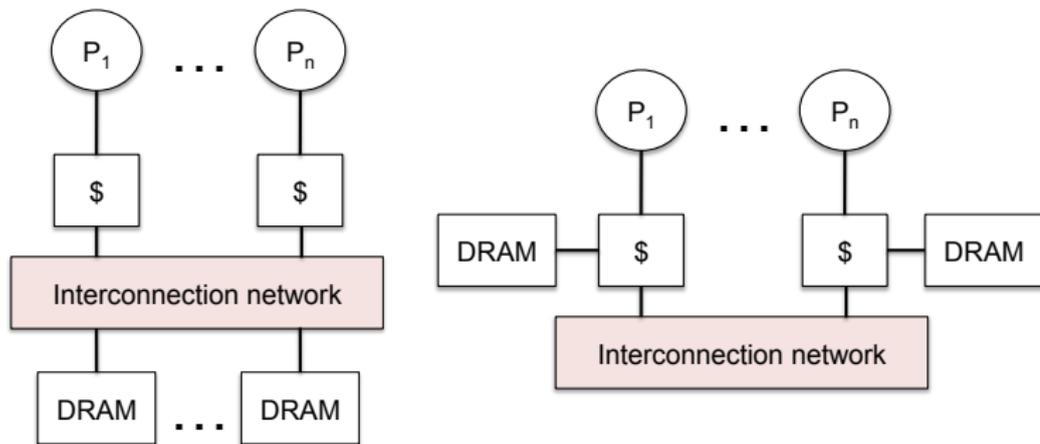
Challenges

Bus-based shared-memory multiprocessors

## Definitions

- ▶ A multiprocessor is a **collection of processing elements** that co-operate to **solve a single problem faster**.
- ▶ Multiprocessor architecture includes **processor architecture** and **communication architecture**
- ▶ Modern multiprocessors are **layered architectures** :
  - ▶ Multiple cores per chip
  - ▶ Multiple chips per board
  - ▶ Network-on-chip, network-on-board
- ▶ Scalable topologies (clusters) built with basic building blocks (cores, processors, networks).

## Memory-centric classification of multiprocessors



- ▶ Though hardware classification based on **physical** placement of memory relative to processors, multiprocessors are classified also based on the **abstraction** of memory provided to users. Abstraction of memory is typically decoupled from the actual implementation.

## Shared memory multiprocessors

- ▶ Processors share single memory address space
- ▶ Memory address space can be implemented over **single centralized memory** or **physically distributed memory**
- ▶ Centralized memory systems are usually scalable only to few (e.g. tens) processors or cores per processor
  - ▶ Shared-memory multiprocessors based on a bus interconnect are limited by bus bandwidth (late 80's, early 90's)
  - ▶ Bus-based multiprocessors are still low-cost commodity component (8-way multiprocessor available at €1,500)
  - ▶ Concept of symmetric memory still alive: modern multi-cores have large caches, each shared between few cores for fast communication

## Distributed memory multiprocessors

- ▶ Cost-effective scaling of memory bandwidth via distribution of memory between processors (or cores)
- ▶ Dependent on **effective data distribution** so that **local memory accesses are maximized**
- ▶ Local accesses cheap, **remote accesses expensive**, due to the need for **explicit communication** between processors
  - ▶ Communication delay includes delays of software (communication libraries, operating system, application overhead for preparing messages), memory, network interfaces (at communication endpoints)

## Classification based on communication medium

- ▶ **Message-passing multiprocessors:** Communication occurs by explicitly passing messages between processors
- ▶ **Remote DMA multiprocessors:** Communication occurs by explicitly reading or writing data from remote memories belonging to other processors. Data replication/migration does not imply coherence or consistency
- ▶ **Shared-memory multiprocessors:** Communication occurs through loads and stores to shared memory. Processors need to synchronize (e.g. via locks and barriers) to avoid lost updates, or violation of dependencies.
  - ▶ **UMA:** Uniform memory access time, typically through shared bus.
  - ▶ **NUMA:** Non-uniform memory access time, physically distributed coherent memories, accessed via local or remote load-stores

# Outline

Motivation

Multiprocessors

**Challenges**

Bus-based shared-memory multiprocessors

## Amdahl's Law

- ▶ What percentage of program execution time is inherently sequential?
- ▶ What is the maximum speedup if the following fractions of program execution time are sequential?
  1. 10%
  2. 5%
  3. 1%

$$Speedup_{overall} = \frac{1}{(1 - fraction_{enhanced}) + \frac{fraction_{parallel}}{Speedup_{parallel}}}$$

## Memory latency

- ▶ Processor speed improving at a rate of 50% per year (20% in last 4 years), memory latency improving at a rate of 7% per year.
- ▶ Local memory access latencies of 60 ns versus remote memory access latency of over 100 ns
- ▶ Data placement important for avoiding remote memory accesses
  - ▶ Complicates parallel programming
  - ▶ Contradicts assumption of flat shared address space

## Example

- ▶ Impact of remote memory accesses
- ▶ Assumptions:
  - ▶ 0.2% remote memory access rate
  - ▶ Base CPI = 0.5 (e.g. superscalar)
  - ▶ 200ns remote memory access latency

$$CPI = \text{Base CPI} + \text{Remote Request Rate} \times \text{Remote Request Cost}$$

$$CPI = 0.5 + 0.2\% \times 200 = 0.9$$

Remote memory accesses almost halve performance

# The role of software in parallelism

## Algorithms

- ▶ Sequential algorithms may include parallel components
- ▶ New algorithms that provide more parallelism, better scalability, lower communication/synchronization costs etc. may be needed
- ▶ Example: FFT straightforward parallelization versus Cooley-Tuckey algorithm.

# The role of software in parallelism

## Languages, compiler, runtime systems

- ▶ Language constructs and runtime libraries are used for communication, synchronization, data distribution, in parallel programs.
- ▶ Performance and scalability depend on the efficiency of the language/library mechanisms that implement parallelism
  - ▶ How fast can processor A provide processor B with work to do?
  - ▶ How fast can I send data from the memory of processor A to the memory of processor B?
  - ▶ How fast can I coordinate processors to provide mutual exclusion or implement a global barrier?

# Outline

Motivation

Multiprocessors

Challenges

**Bus-based shared-memory multiprocessors**

# Shared-memory multiprocessor architecture

## History

- ▶ Multiple processor on a single board, communicating over a shared bus, using loads/stores and a cache coherence protocol (80's–90's)
- ▶ Multiple processors on multiple boards in a single cabinet, communicating over a shared bus (on-board) and a scalable switch-based interconnection network (late 90's)
- ▶ Multiple processors on a single chip, communicating over a shared bus (2004–onwards) or a scalable switch-based interconnection network (2008–onwards)

# Shared-memory multiprocessor architecture

## Technology trends

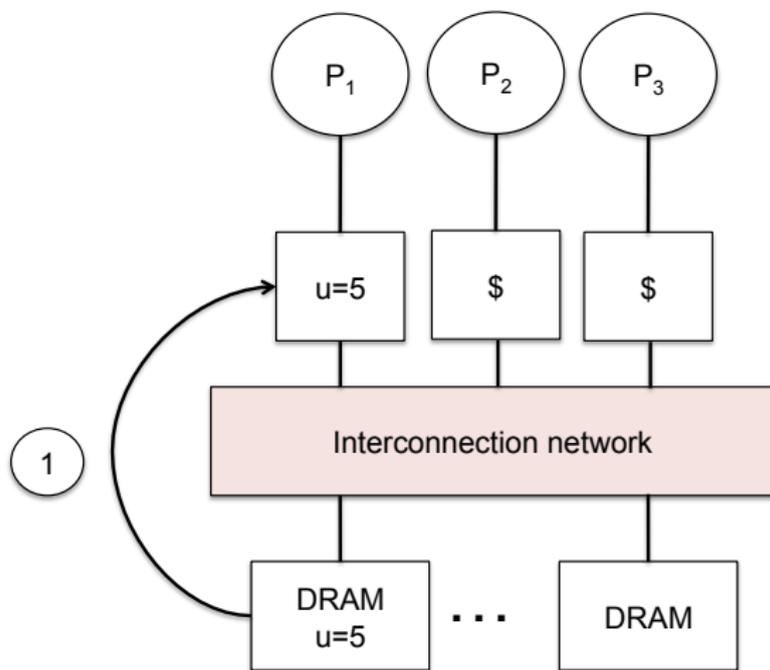
- ▶ Bus is a centralized bottleneck and its BW is not adequate to support more than a few (e.g. 10) processors).  
**Replaced by switch-based interconnect**
- ▶ Cache coherence desirable due to **programmability**
  - ▶ Processors communicate through loads and stores, with a model which is more familiar to sequential programmers.
  - ▶ Communication between producers and consumers done by producer storing data in local cache and consumer requesting data from remote cache
  - ▶ Coherence protocol maintains consistency between replicas of data potentially written by one or more processors.

# Cache coherence

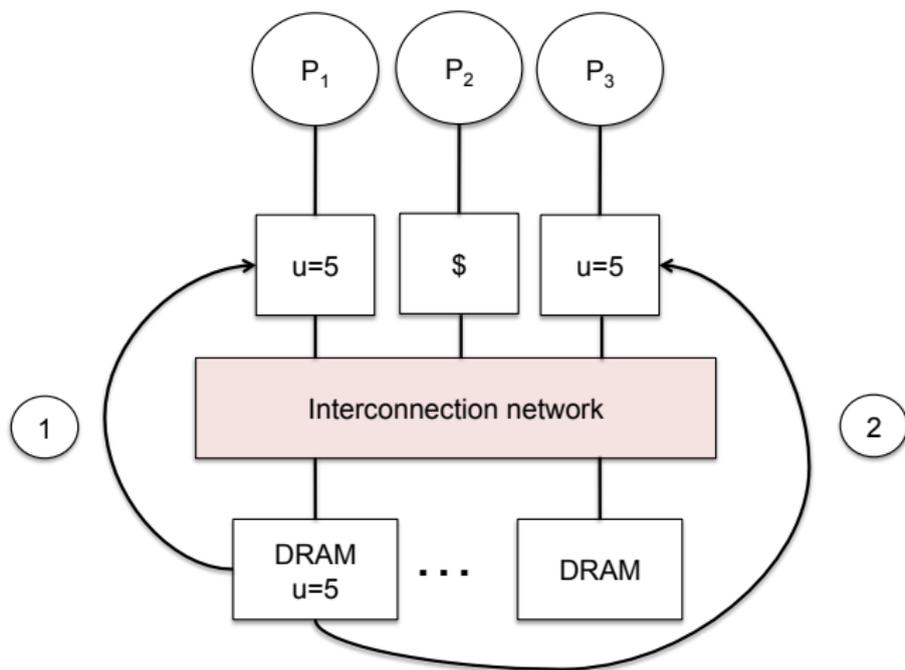
## Shared data

- ▶ Private cache per processor (or processor core) on SMPs
- ▶ Cache stores both
  - ▶ **Private data** used only by **owner** processor
  - ▶ **Shared data** accessed by multiple processors
- ▶ Caches reduce latency to access shared data, memory bandwidth consumption, interconnect bandwidth consumption
- ▶ **Cache coherence problem**

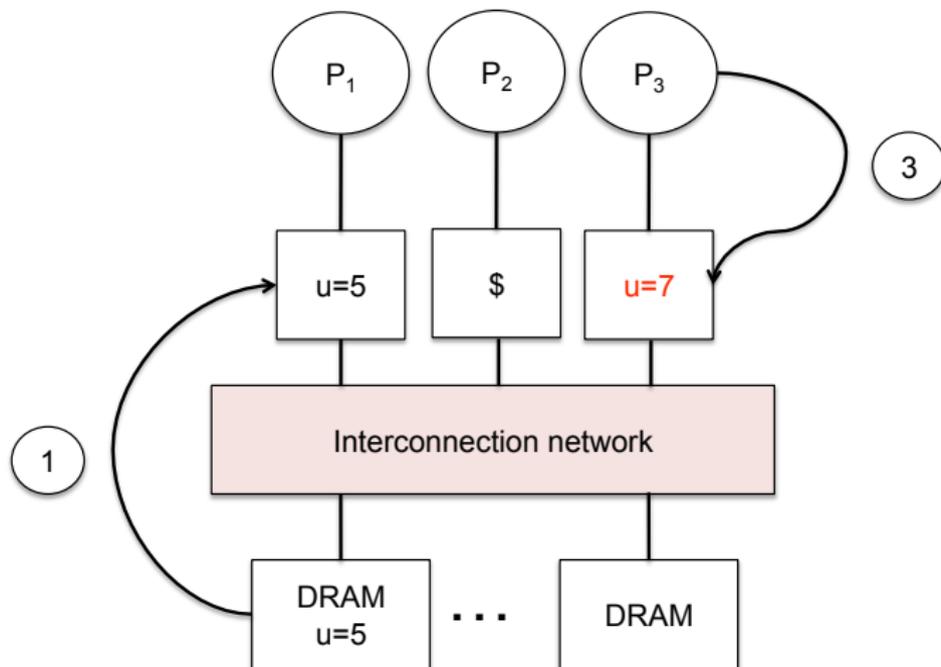
# Cache coherence problem



## Cache coherence problem



## Cache coherence problem



Processor 3 writes new value of  $u$ . Processor 1 and processor

## Cache coherence problem

 $P_1$  $P_2$ 

---

Assume initial values  $A=0$ ,  $flag=0$

```
A = 1;      while (flag==0); /* busy-wait */  
flag = 1;  print A;
```

$P_1$  expects that  $A=1$  after exiting the while. **Intuition not guaranteed by coherence.** If memory writes from  $P_0$  commit in order then intuition is verified. If not, then  $P_1$  may see  $A = 0$ ! The memory system is typically expected to preserve ordering of memory accesses by a **single processor** but **not across processors**.

# Coherence versus consistency

## What is the value of a memory location?

- ▶ Every **read** of a memory location should return **the last value written** to the memory location
  - ▶ In uniprocessors this is easy to implement and guaranteed except from when the processor performs I/O (DMAs)
- ▶ **Coherence** defines the values returned by a read
- ▶ **Consistency** defines when a write from a processor becomes visible to other processors
- ▶ Coherence defines the behavior of **a single processor** while consistency defines the behavior **across processors**

# Coherent memory system

## Preserving program order

A read by processor P to location X that follows a write by P to X, with no writes to X made by other processors between the write and the read by P will always return the value written by P.

## Coherent view of memory

A read by a processor to location X that follows a write by **another processor** to X, returns the value of the write of **the two accesses are separated sufficiently apart in time and no other writes to X occur between the two accesses.**

# Coherent memory system

## Write serialization

2 writes to the same location by any 2 processors are **seen in the same order by all processors**

- ▶ Assume that writes are not serialized: Two processors may proceed assuming **different last values of the same location**

## Write consistency

A write does not complete and does not allow the next write to occur until all processors have “seen” the effect of that write. The processor **does not change the order of any write with respect to other reads or writes**

- ▶ If a processor writes location A then location B, any processor that sees the new value of B must also see the new value of A
- ▶ **Reads can be reordered (module dependencies) but writes must happen in program order**

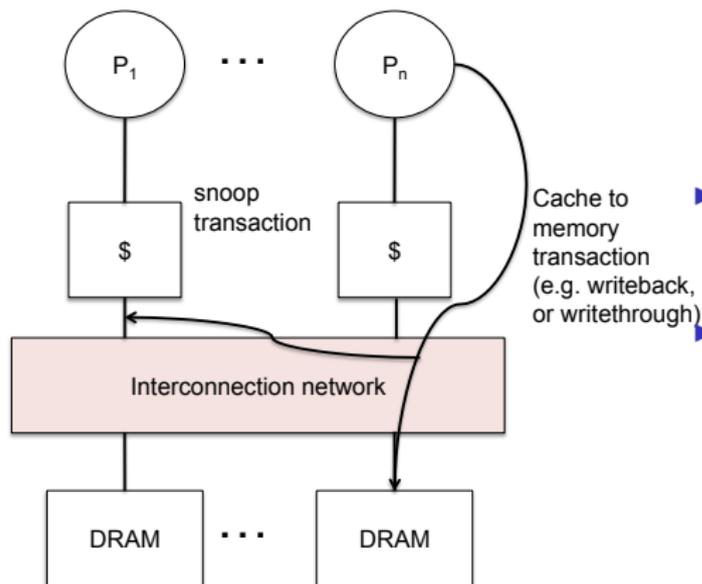
## Schemes for enforcing coherence

- ▶ Multiple processors may have copies of same data (common in parallel programs)
- ▶ SMPs typically use a cache coherence protocol implemented in hardware, although slower software solutions are also available
- ▶ Key operations: **replication** and **migration** of data:
  - ▶ **Migration**: data can be moved to the cache of a single processor and used for **reading or writing transparently**. Reduces latency and demand for bandwidth.
  - ▶ **Replication**: Data can be simultaneously read by multiple processors, by having processors make copies of data in their local caches. Reduces latency, demand for bandwidth and contention for accessing shared data.

## Classes of cache coherence protocols

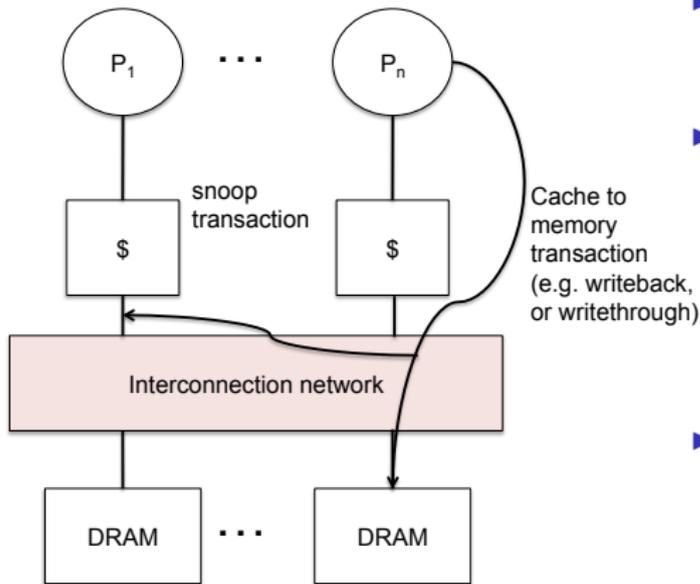
- ▶ **Directory based:** Sharing status of a cache block (i.e. what processors have a copy of the block in the cache and whether this copy has been updated) is kept in one location (in memory, or on-chip in recent multi-core processors) called **the directory**
- ▶ **Snooping:** Every cache with a copy of a block also has information on the sharing status of the block, but no centralized state is kept.
  - ▶ All caches are accessible via a centralized **broadcasting mechanism** (typically a bus, nowadays a switch).
  - ▶ All cache controllers **monitor (or snoop)** the centralized medium to determine whether they have or not a copy of the block requested by another processor, and update sharing state.

# Snoopy cache coherence



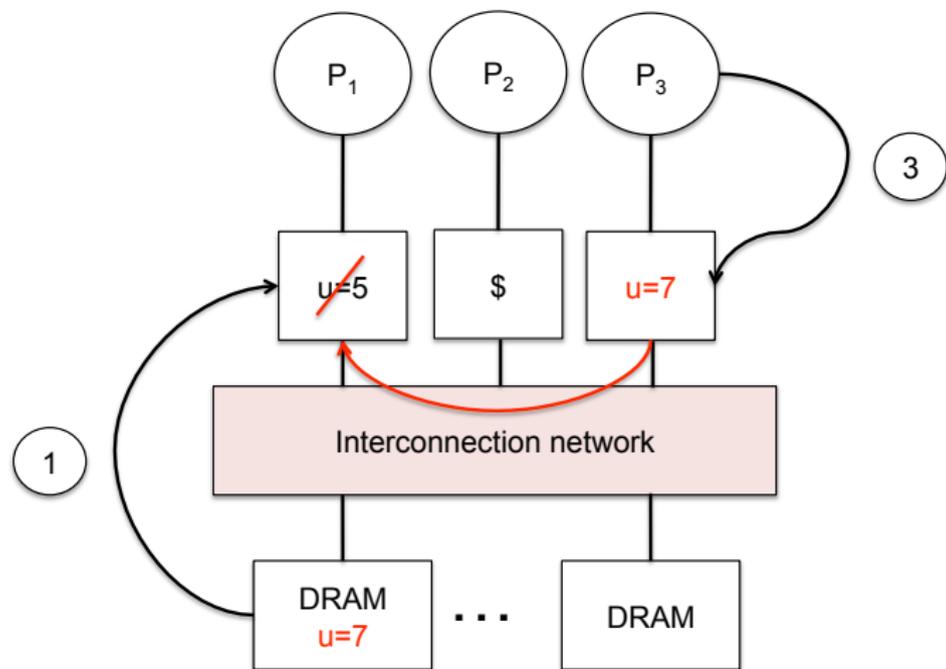
- ▶ Cache controller snoops all transactions on the shared interconnect.
- ▶ A transaction is **relevant** if it involves a block **stored in the cache of the snooping processor**.

# Snoopy cache coherence



- ▶ If transaction is on relevant block, controller takes action to ensure coherence.
- ▶ Action may be **invalidate** (block written by other processor), **update** (block written by another processor and new value stored in the cache of the snooping processor), or **supply new value** (requested by other processor).
- ▶ Processor that needs to write either gets **exclusive** access to block by **invalidating other copies**, or **writes and updates other copies**.

## Example: write-through, write-invalidate



## Example: write-through, write-update

