High Performance Computing
Lecture 39

Matthew Jacob

Indian Institute of Science

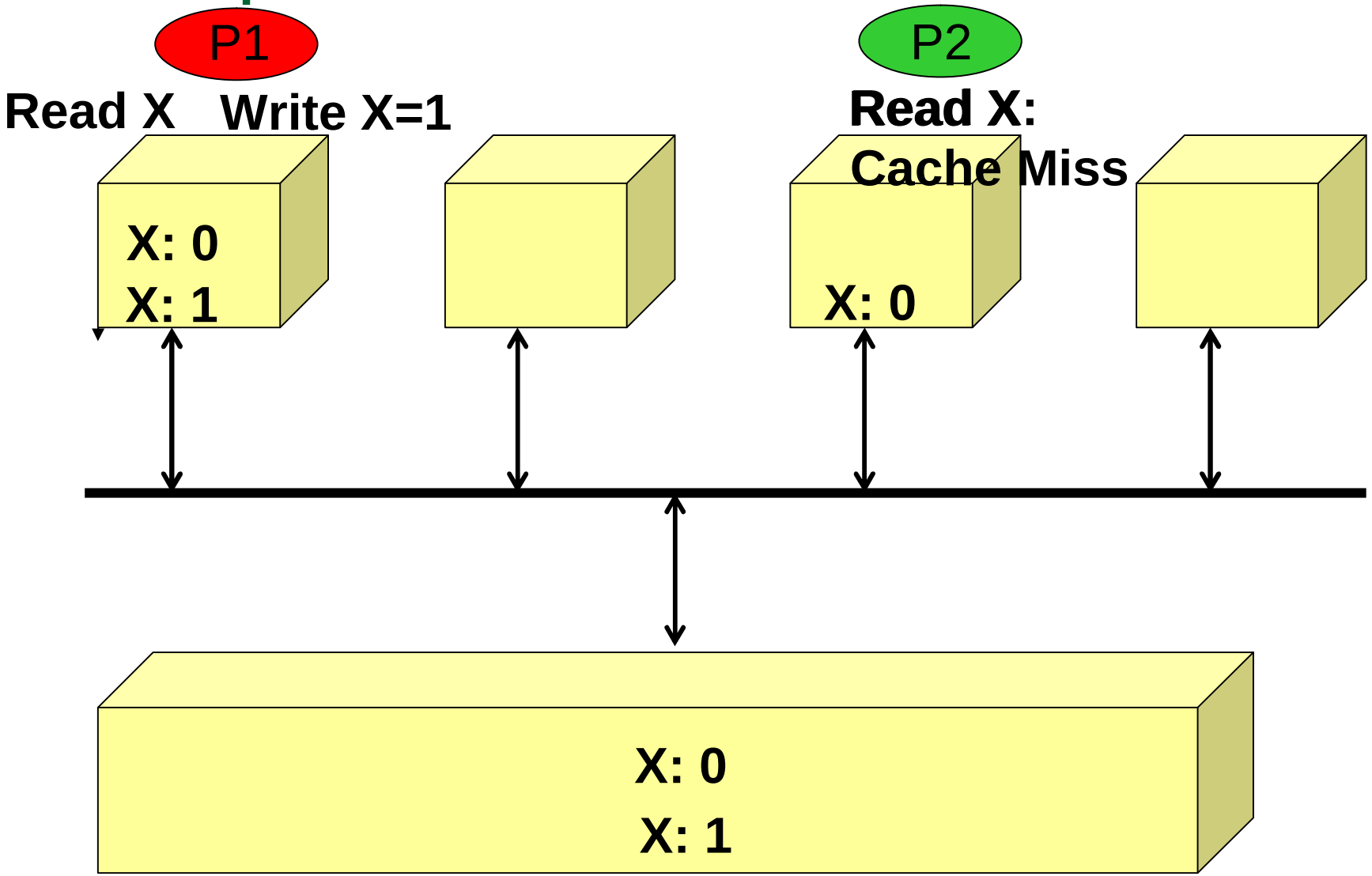
Cache Coherence Problem

- If each processor in a shared memory multiple processor machine has a data cache
  - Potential data consistency problem: the cache coherence problem
  - Shared variable modification, private cache
- Objective: processes shouldn't read `stale' data
- Solutions
  - Hardware: cache coherence mechanisms
  - Software: compiler assisted cache coherence

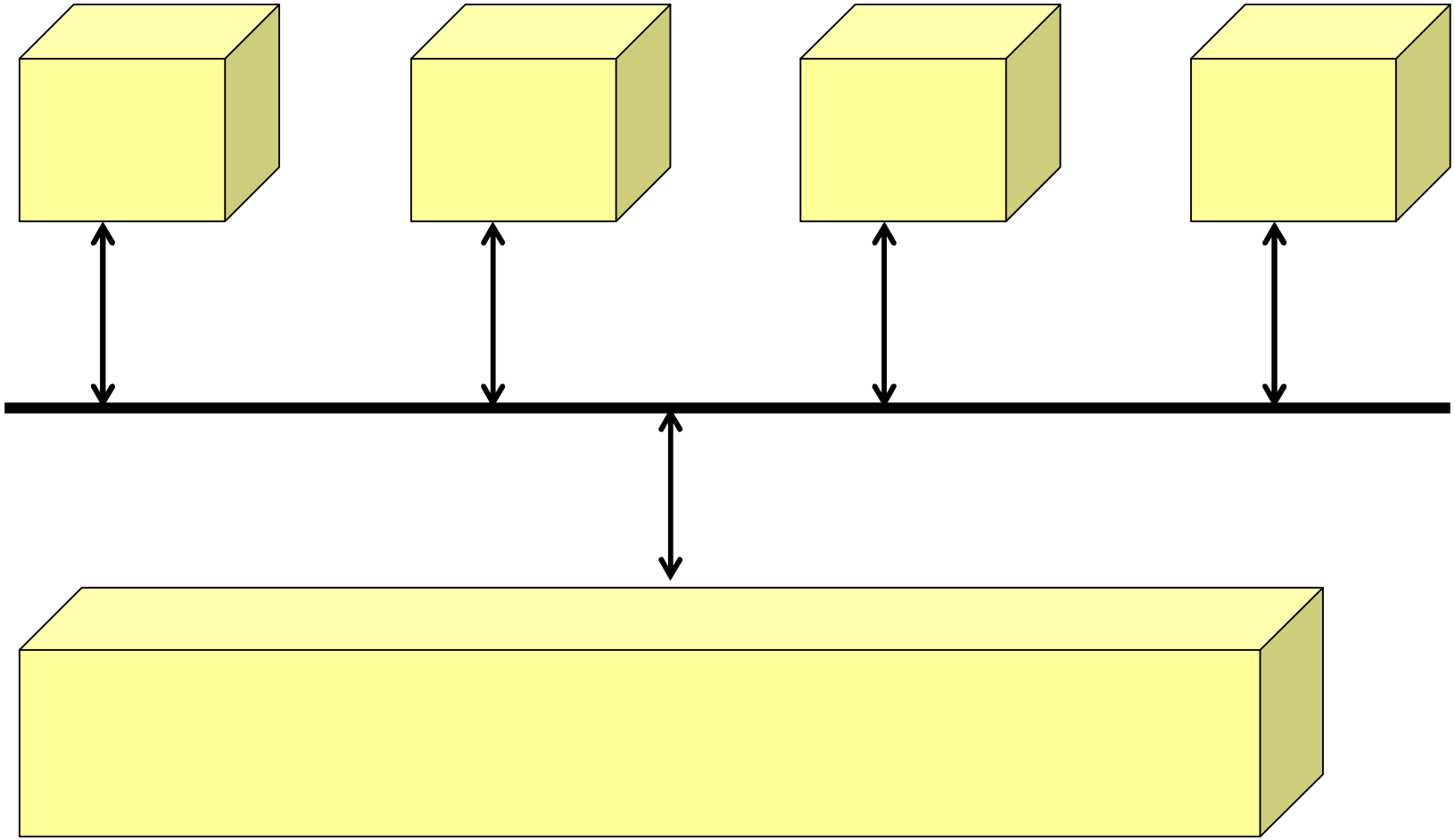
Example: Write Once Protocol

- Assumption: shared bus interconnect where all cache controllers monitor all bus activity
  - Called snooping
- There is only one operation through bus at a time; cache controllers can be built to take corrective action and enforce coherence in caches
  - Corrective action could involve updating or invalidating a cache block

Example: Write Once Protocol.



Snoopy Cache Coherence and Locks



Lock Implementation

```
while ( Test&Set (L) );
```

- With snoopy invalidate cache coherence protocol, spinning on Test&Set leads to lock pingponging
- High bus utilization slows down memory accesses

```
repeat
        while (L);
until ( ! Test&Set (L) );
```

- Reads of L will be cache hits – no bus traffic
- When lock is available, many spinners may find that L=0. First one to get Test&Set on bus wins and causes invalidation of other cache copies

Lock Implementation …

- But, many processes finding L=0 will all try and do Test&Set(L) causing a burst of bus traffic
- Could try and prevent all of these processes from attempting Test&Set at about the same time

```
repeat
   while (L);
   wait ( different time for each process );
until ( ! Test&Set (L) );
```

PARALLEL PROGRAMMING

- Recall: Flynn's SIMD, MIMD
  - How would you describe a program meant to run on each of these parallel computer classes?
    - SIMD: Single Program Multiple Data (SPMD)
    - MIMD: Multiple Program Multiple Data (MPMD)
  - The programs could involve cooperating activities using either
    - Shared memory: threads with shared variables
    - Message passing: communication using messages
- Speedup = $\frac{T_{sequential}}{T_{parallel}}$

How Much Speedup is Possible?

Let s be the fraction of sequential execution time of a given program that can not be parallelized

Assume that program is parallelized so that the remaining part $(1 - s)$ is perfectly divided to run in parallel across n processors

$$\lim_{n \to \infty} \text{Speedup} = \frac{1}{s + \frac{1-s}{n}} \qquad \frac{1}{s}$$

i.e., the maximum speedup achievable is limited by the sequential fraction of the sequential program

Amdahl's Law

Programming with Message Passing

- Need
  1. Mechanism to create processes to execute on different processors
  2. Mechanism to send/receive message
- Must specify the identity of the communicating process
- Example

P1: send(&x, P2) P2: receive(&y, P1)

- Message passing libraries
  - PVM (1980s)
  - MPI (1990s)

MPI References

1. Using MPI

   Gropp, Lusk, Skjellum

   www.mcs.anl.gov/mpi/usingmpi

2. MPI: The Complete Reference

   Snir, Otto, Huss-Lederman, Walker, Dongarra

   www.netlib.org/utk/papers/mpi-book/mpi-book.html

Message Passing Interface (MPI)

Standard API

- Hides software/hardware details
- Portable, flexible

Implemented as a library

<table>
<tr><td colspan="2">Your program</td></tr>
<tr><td colspan="2">MPI Library</td></tr>
<tr><td>Custom software</td><td>Standard TCP/IP</td></tr>
<tr><td>Custom hardware</td><td>Standard network HW</td></tr>
</table>

Key MPI Functions and Constants

- MPI_Init (int *argc, char ***argv)
- MPI_Finalize (void)
- MPI_Comm_rank (MPI_COMM comm, int *rank)
- MPI_Comm_size (MPI_COMM comm, int *size)
- MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)
- MPI_CHAR, MPI_INT, MPI_LONG, MPI_BYTE
- MPI_ANY_SOURCE, MPI_ANY_TAG