SE-292 High Performance Computing

Intro. To Concurrent Programming & Parallel Architecture

R. Govindarajan
govind@serc

Concurrent Programming

- Read: Bryant & O'Hallaron Chapter 13
- Until now: execution involved one flow of control through program
- Concurrent programming is about programs with multiple flows of control
- For example: a program that runs as multiple processes cooperating to achieve a common goal
- To cooperate, processes must somehow communicate

Inter Process Communication (IPC)

1. Using files
   - Parent process creates 2 files before forking child process
   - Child inherits file descriptors from parent, and they share the file pointers
   - Can use one for parent to write and child to read, other for child to write and parent to read
2. OS supports something called a pipe
   - corresponds to 2 file descriptors (int fd[2])
   - Read from fd[0] accesses data written to fd[1] in FIFO order and vice versa

3

Other IPC Mechanisms

3. Processes could communicate through variables that are shared between them
   - Shared variables, shared memory; other variables are private to a process
   - Special OS support for program to specify objects that are to be in shared regions of address space
4. Processes could communicate by sending and receiving messages to each other
   - Special OS support for these messages

4

More Ideas on IPC Mechanisms

5. Sometimes processes don't need to communicate explicit values to cooperate
   - They might just have to synchronize their activities
   - Example: Process 1 reads 2 matrices, Process 2 multiplies them, Process 3 writes the result matrix
   - Process 2 should not start work until Process 1 finishes reading, etc.
   - Called process synchronization
   - Synchronization primitives
     - Examples: mutex lock, semaphore, barrier

5

Programming With Shared Variables

- Consider a 2 process program in which both processes increment a shared variable

```
shared int X = 0;
P1:                 P2:
   X++;                X++;
```

- Q: What is the value of X after this?
- Complication: Remember that X++ compiles into something like

```
LOAD    R1,    0(R2)
ADD     R1,    R1,  1
STORE   0(R2), R1
```

6

Problem with using shared variables

- Final value of X could be 1!
  P1 loads X into R1, increments R1
  P2 loads X into register before P1 stores new value into X
  Net result: P1 stores 1, P2 stores 1
- Moral of example: Necessary to synchronize processes that are interacting using shared variables
- Problem arises when 2 or more processes try to update shared variable
- Critical Section: part of program where shared variable is accessed like this

7

Critical Section Problem: Mutual Exclusion

- Must synchronize processes so that they access shared variable one at a time in critical section; called Mutual Exclusion
- Mutex Lock: a synchronization primitive
  - AcquireLock(L)
    - Done before critical section of code
    - Returns when safe for process to enter critical section
  - ReleaseLock(L)
    - Done after critical section
    - Allows another process to acquire lock

8

Implementing a Lock

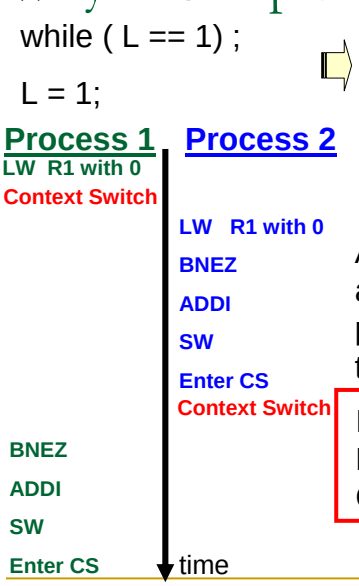
```
int L=0;                    /* 0: lock available */

AcquireLock(L):
        while (L==1);    /* `BUSY WAITING' */
        L = 1;

ReleaseLock(L):
        L = 0;
```

9

Why this implementation fails

```
while ( L == 1) ;
L = 1;
```

```
wait:   LW     R1, Addr(L)
        BNEZ   R1, wait
        ADDI   R1, R0, 1
        SW     R1, Addr(L)
```



Assume that lock L is currently available (L = 0) and that 2 processes, P1 and P2 try to acquire the lock L

IMPLEMENTATION ALLOWS PROCESSES P1 and P2 TO BE IN CRITICAL SECTION TOGETHER!

10

Busy Wait Lock Implementation

- Hardware support will be useful to implement a lock
- Example: Test&Set instruction

Test&Set Lock:

```
tmp = Lock
Lock = 1
Return tmp
```

Where these 3 steps happen atomically or indivisibly.
i.e., all 3 happen as one operation (with nothing happening in between)

Atomic Read-Modify-Write (RMW) instruction

11

Busy Wait Lock with Test&Set

```
AcquireLock(L)
    while (Test&Set(L)) ;
ReleaseLock(L)
    L = 0;
```

- Consider the case where P1 is currently in a critical section, P2-P10 are executing AcquireLock: all are executing the while loop
- When P1 releases the lock, by the definition of Test&Set exactly one of P2-P10 will read the new lock value of 0 and set L back to 1

12

More on Locks

- Other names for this kind of lock
  - Mutex
  - Spin wait lock
  - Busy wait lock
- Can have locks where instead of busy waiting, an unsuccessful process gets blocked by the operating system

13

Semaphore

- A more general synchronization mechanism
- Operations: ***P*** (wait) and ***V*** (signal)
- ***P***(S)
  - if S is nonzero, decrements S and returns
  - Else, suspends the process until S becomes nonzero, when the process is restarted
  - After restarting, decrements S and returns
- ***V***(S)
  - Increments S by 1
  - If there are processes blocked for S, restarts exactly one of them

14

Critical Section Problem & Semaphore

- Semaphore S = 1;
- Before critical section: ***P***(S)
- After critical section: ***V***(S)
- Semaphores can do more than mutex locks
  - Initialize S to 10 and 10 processes will be allowed to proceed
  - P1:read matrices; P2: multiply; P3: write product
  - Semaphores S1=S2=0;
  - End of P1: ***V***(S1), beginning of P1: ***P***(S1) etc

15

Deadlock

Consider the following process:

P1: lock (L); lock(L);

- P1 is waiting for something (release of lock that it is holding) that will never happen
- Simple case of a general problem called deadlock
- Cycle of processes waiting for resources held by others while holding resources needed by others

16

Classical Problems

Producers-Consumers Problem

- Bounded buffer problem
- Producer process makes things and puts them into a fixed size shared buffer
- Consumer process takes things out of shared buffer and uses them
- Must ensure that producer doesn't put into full buffer or consumer take out of empty buffer
- While treating buffer accesses as critical section

17

Producers-Consumers Problem

shared Buffer[0 .. N-1]

Producer: repeatedly

Produce x; if (buffer is full) wait for consumption

Buffer[i++] = x; signal consumer

Consumer: repeatedly

If (buffer is empty) wait for production

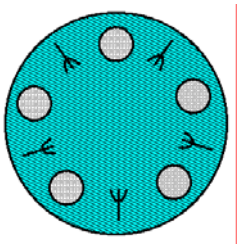
y = Buffer[- - i]

Consume y; signal producer

18

Dining Philosophers Problem

- N philosophers sitting around a circular table with a plate of food in front of each and a fork between each 2 philosophers
- Philosopher does: repeatedly
  - Eat (using 2 forks)
  - Think
- Problem: Avoid deadock; be fair



19

THREADS

Thread

- Thread of control in a process
- `Light weight process'

- Weight related to
  - Time for creation
  - Time for context switch
  - Size of context
- Recall context of process

20

Threads and Processes

- Thread context
  - Thread id
  - Stack
  - Stack pointer, PC, GPR values
- So, thread context switching can be fast
- Many threads in same process that share parts of process context
  - Virtual address space (other than stack)
- So, threads in the same process share variables that are not stack allocated

21

Thread Implementation

- Could either be supported in the operating system or by a library
- Pthreads: POSIX thread library
  - int pthread_create
    - pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine), void *arg
  - pthread_attr
  - pthread_join
  - pthread_exit
  - pthread_detach

22

Synchronization Primitives

Mutex locks

int pthread_mutex_lock(pthread_mutex_t *mutex)

If the mutex is already locked, the calling thread blocks until the mutex becomes available. Returns with the mutex object referenced by mutex in the locked state with the calling thread as its owner.

pthread_mutex_unlock

Semaphores

sem_init

sem_wait

sem_post

23

PARALLEL ARCHITECTURE

Parallel Machine: a computer system with more than one processor

Question: What about a network of machines?

- Yes, but time involved in interaction (communication) might be high, as the system is designed assuming that the machines are more or less independent
- Special parallel machines might be designed to make this interaction overhead less

24

Classification of Parallel Machines

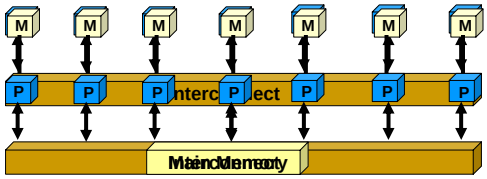
Flynn's Classification

- In terms of number of Instruction streams and Data streams
- Instruction stream: path to instruction memory (PC)
- Data stream: path to data memory
- SISD: single instruction stream single data stream
- SIMD
- MIMD

25

Classification 2: Shared Memory vs Message Passing

- Shared memory machine: The n processors share physical address space
  - Communication can be done through this shared memory



- The alternative is sometimes referred to as a message passing machine or a distributed memory machine

26

Shared Memory Machines

The shared memory could itself be distributed among the processor nodes

- Each processor might have some portion of the shared physical address space that is physically close to it and therefore accessible in less time
- Terms: Shared vs Private
- Terms: Local vs Remote
- Terms: Centralized vs Distributed Shared
- Terms: NUMA vs UMA architecture
  - Non-Uniform Memory Access

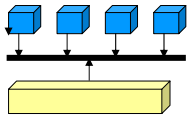
27

Parallel Architecture: Interconnections

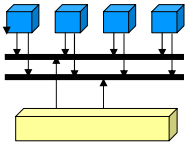
- Indirect interconnects: nodes are connected to interconnection medium, not directly to each other
  - Shared bus, multiple bus, crossbar, MIN
- Direct interconnects: nodes are connected directly to each other
  - Topology: linear, ring, star, mesh, torus, hypercube
  - Routing techniques: how the route taken by the message from source to destination is decided

28

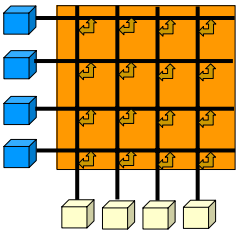
Indirect Interconnects



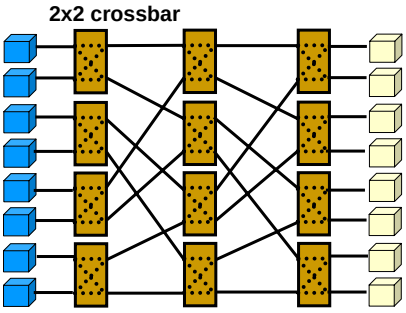
Shared bus



Multiple bus



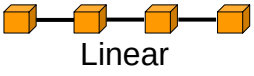
Crossbar switch



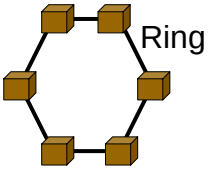
Multistage Interconnection Network

29

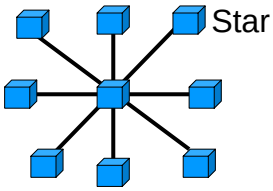
Direct Interconnect Topologies



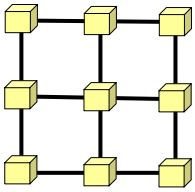
Linear



Ring

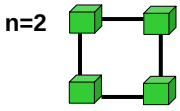


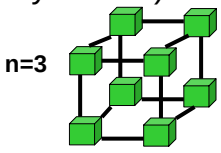
Star

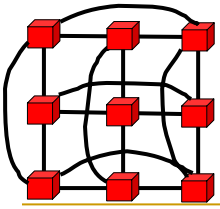


2D Mesh

Hypercube (binary n-cube)



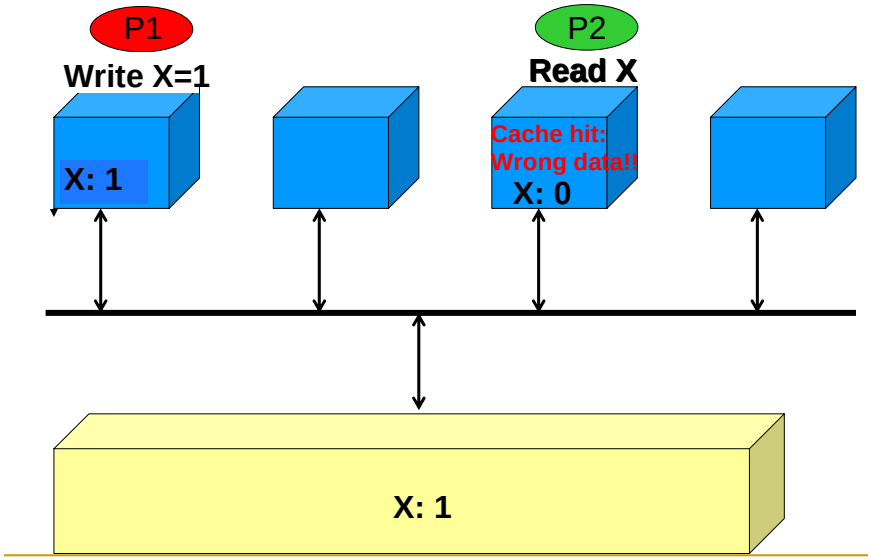




Torus

30

Shared Memory Architecture: Caches



31

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

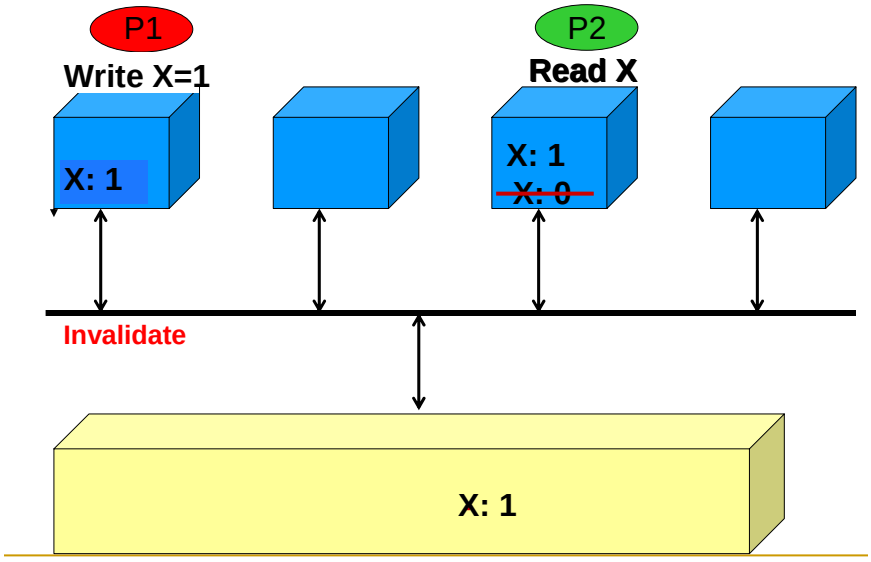
32

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

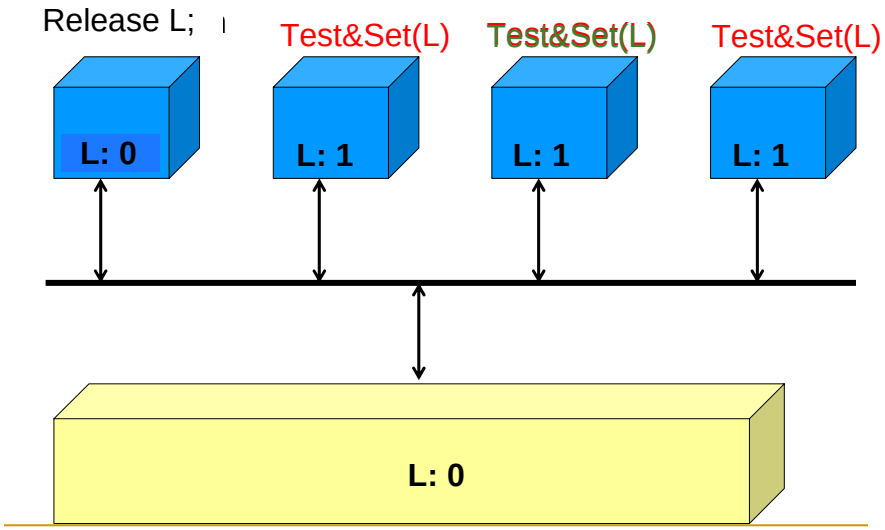
33

Invalidation Based Cache Coherence



34

Snoopy Cache Coherence and Locks



35

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

36

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) );
```

37