MULTIPROCESSOR (II)

Lecturer: Hui Annie Guo
h.guo@unsw.edu.au
K17-501F

Lecture overview

- **Topics**
  - **Shared memory multiprocessors**
    - **Memory consistency**
  - **Program model and parallel programming**

- **Suggested reading**
  - **H&P Chapter 5.10**
  - **“Memory Consistency Models for Shared-Memory Multiprocessors”, Kourosh Gharachorloo.**

Memory consistency

Motivational example

- **What value is in register r2 after the execution?**

Core C1	Core C2	Comments
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

Memory consistency

- The order between accesses to different memory locations is very important
- Some rules on the order of memory assesses are required
- Memory consistency model
  - Provides such rules
  - Affects programmers and system designers

Memory consistency model generation

- Many consistency models exist
- One simple consideration is easy to use
  - keep it similar to serial semantics for uniprocessor
  - Benefit to concurrent programming
    - Executions managed by OS
- A typical design
  - Sequential consistency model

Sequential consistency (SC)

- **Lamport (1979) definition:**
  - **The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.**

- Overall memory accesses are serialized
- Memory accesses from each processor follow the same order specified by its program
- Writes should be atomic

Sequential consistency (SC)

- **How to realize SC?**
  - **Consider the typical multiprocessor designs given in the next slides.**

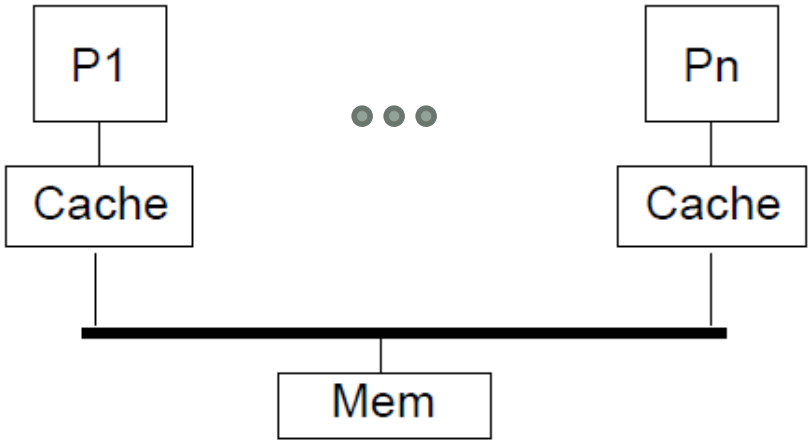
- Overall memory accesses are serialized
- Memory accesses from each processors follow the same order specified by its program
- Writes should be atomic

Design 1

Bus-based multiprocessor with 1-level cache

- Each processor has a local cache
  - Write back caches are used
- Accesses to memory through a single bus
- Cache coherence is implemented

Does this design conform to the SC model?



UMA multiprocessor

Design 1 (cont.)

- Memory accesses is serialized
  - Bus
    - One memory access at a time
  - Cache coherence
    - Write to the same location serialized and observed in the same order by all
- Writes are atomic
  - all processors observe the write at the same time
- Accesses from a single processor complete in program order
  - With cache hits: no reorder
  - With cache misses: Cache is busy while serving a miss, effectively delaying later access

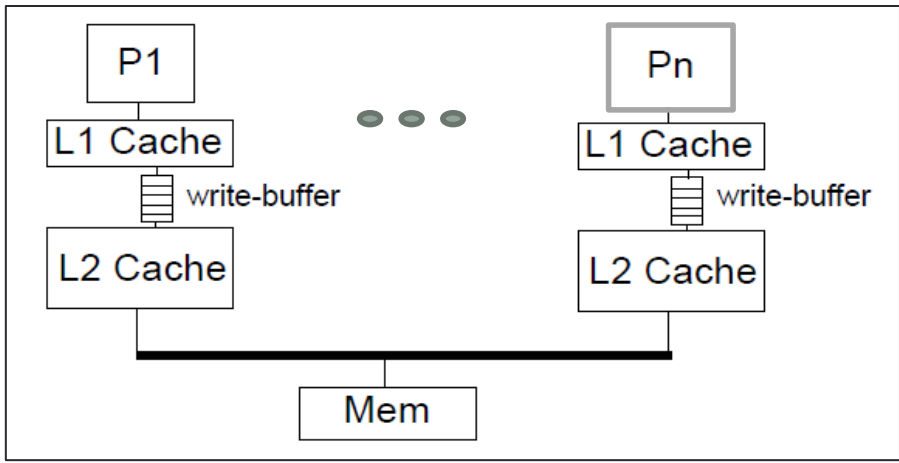
SC is guaranteed without any extra mechanism.

Design 2

Bus-based multiprocessor with 2-level cache

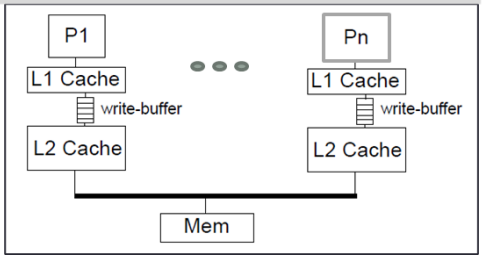
- Each processor has two-level caches
- L1 cache: write-through
  - Write buffer with no forwarding
    - Reads to L2 are delayed until buffer empty
- L2 cache: write back
- Accesses to memory through a single bus
- Cache coherence is implemented

Does this design conform to the SC model?



Design 2 (cont.)

- Overall memory accesses are serialized
- Memory accesses from each processors follow the same order specified by its program
- Writes should be atomic

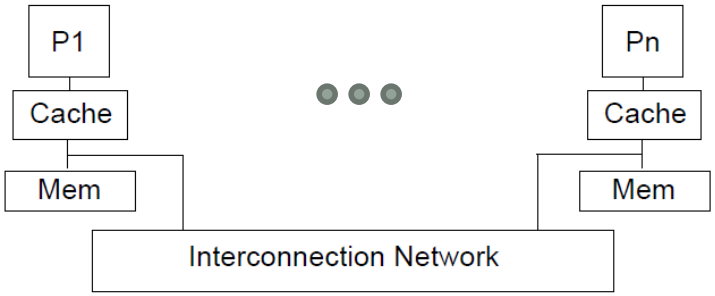


- **If accesses never hit L1 cache**
  - **the system behaves the same as Design 1**
- **If a read hits L1 cache**
  - **Completion of accesses can be out of order**
    - **E.g. write-then-read operations requested by the program can be completed in the order of read-then-write if write is buffered and read hits L1**
- **To maintain SC**
  - **access to L1 cache should be delayed until there are no writes pending in write buffer**
    - **Performance offered by the write buffer is nullified.**

Design 3

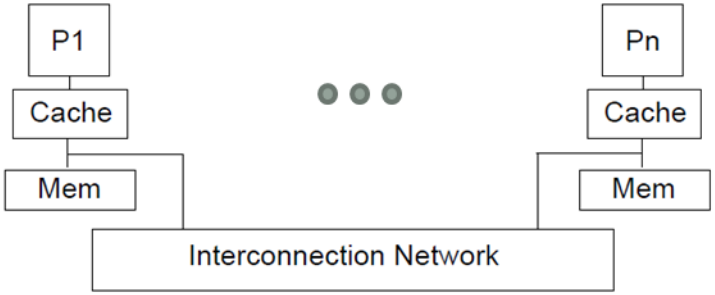
Scalable Shared-memory multiprocessor

- Each processor has a local cache and a shared memory component
- Access to remote memory through the network
  - Point-to-point communication
    - Multiple parallel data accesses are therefore possible
- General cache coherence:
  - Serialize at memory location; point-to-point order is required



Design 3 (cont.)

- **Accesses issued in order do not necessarily completed in order:**
  - **Due to distribution of memory and varied-length paths in network**
- **Write are inherently non-atomic:**
  - **New value is visible to some while others can still see old value**
  - **No one point in the system where a write is completed**



Design 3 (cont.)

- To maintain SC
  - Need to know when a write completes
    - For providing atomicity
    - For delaying an access until previous one completes
  - Require acknowledgement messages
    - Write is complete when all invalidations are acknowledged
    - Use a counter to count the number of acknowledgments
  - Ensure atomicity for writes
    - Delay access to new value until all acknowledgements are back
    - Can be one for invalidation-based schemes; unnatural for updates
  - Ensure order from a processor:
    - Delay each access until the previous one completes
  - etc

Problem with SC

- **Severely restricts common hardware and compiler optimizations**
- **Does not fully guarantee the single execution result**

More-software-oriented solution

- **Programmer model**
  - **Is a contract between programmer and system:**
    - **Programmer provides synchronized programs**
    - **System provides sequential consistency at a higher performance**
  - **Allows portability over a wide range of implementations**
  - **Provides programmer the methodology for writing programs**
  - **Enables system designer safe optimization for such programs**

More-software-oriented solution

- **Programmer model**
  - **Provides programmer the methodology for writing programs**
  - **Enables (hardware) system designer safe optimization for such programs**

Exercise

[P & H] Consider the following portions of two different programs running in parallel on two processors in a UMA multiprocessor. Assume that before execution, both x and y are 0.

Processor 1: **... ; x:=x+1; y:=x+y; …**
Processor 2: **… ; y:=x+1; …**

What are the possible resulting values of x and y assuming the code is implemented using a load-store architecture? For each possible outcome, explain how x and y might obtain those values. (Hint: You must examine all of the possible interleaving cases of the assembly language instructions,)

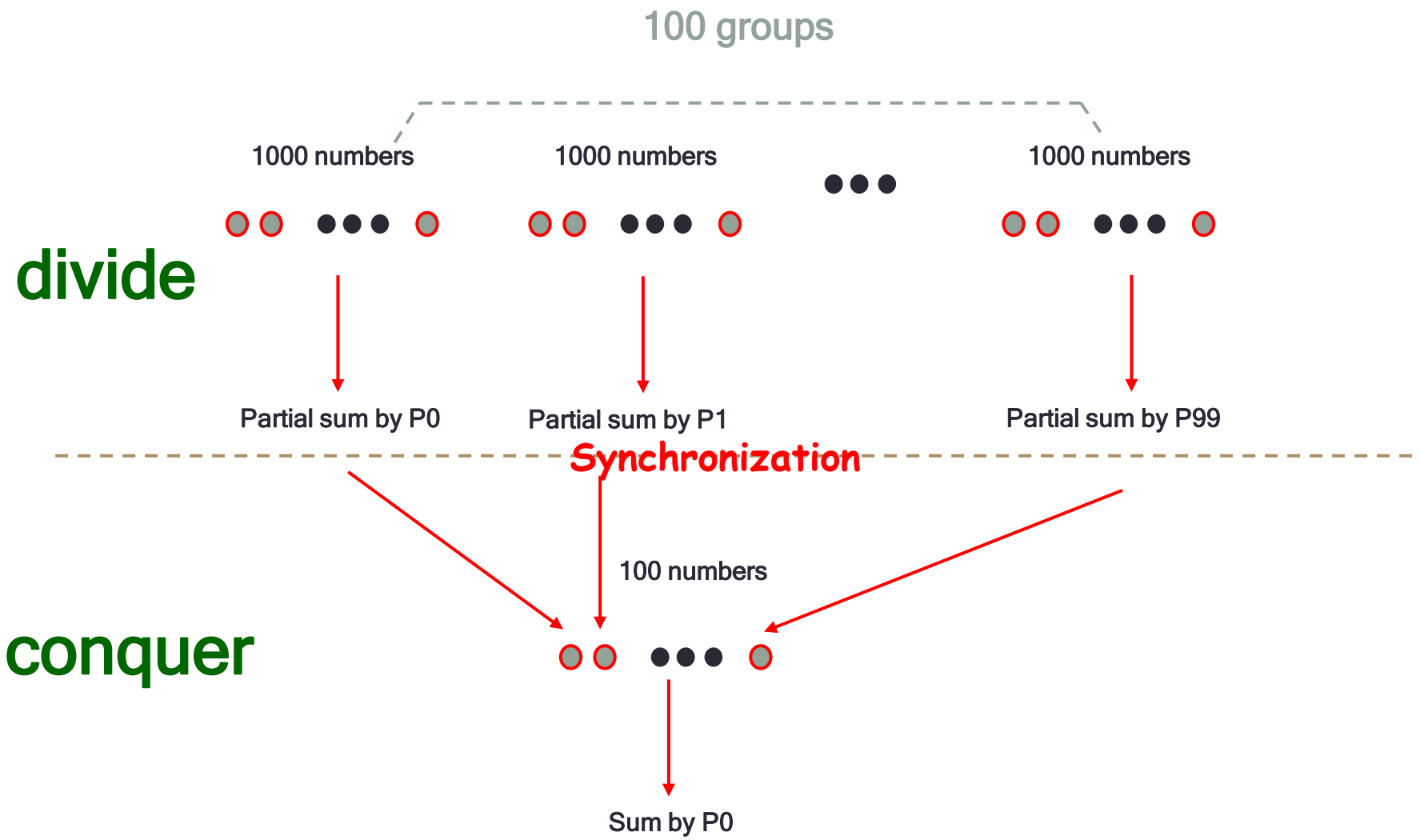
Extended question*

- **How to sync for y=2?**

Parallel programming – example

- **Suppose we have a single-bus multiprocessor of 100 processor cores. Write a parallel processing program to calculate sum of 100,000 numbers.**

- **General design idea:**
  - **Use the 100 processors to finish the sum function as quick as possible**
- **Approach: divide and conquer**
  - **Divided the 100,000 numbers into 100 subsets, each of 1000 numbers, and each subset is summed by an individual processor core.**
  - **Then add the partial sums together with $\log_2 100$ steps**


100 groups
1000 numbers
1000 numbers
1000 numbers
divide
Partial sum by P0
Partial sum by P1
Partial sum by P99
Synchronization
100 numbers
conquer
Sum by P0

Example (cont.)

- **Partial sum for each processor**
  - **All processors share the same memory space**
  - **Each processor has a slightly different program**
    - **E.g. program for processor n, n=0,1,…99**

```
sum[Pn] = 0;
for (i = 1000*Pn; i < 1000*(Pn+1); i = i + 1)
        sum[Pn] = sum[Pn] + A[i]; /* sum the assigned areas*/
```

Example (cont.)

- **Sum of partial-sums**
  - **Coordination between processors through synch() function**

```
half = 100; /* 100 processors in multiprocessor*/
repeat
        synch(); /* wait for partial sum completion*/
        if (half%2 != 0 && Pn == 0)
                sum[0] = sum[0] + sum[half-1];
                /* Conditional sum needed when half is
                odd; Processor0 gets missing element */
        half = half/2; /* dividing line on who sums */
        if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1); /* exit with final sum in Sum[0] */
```