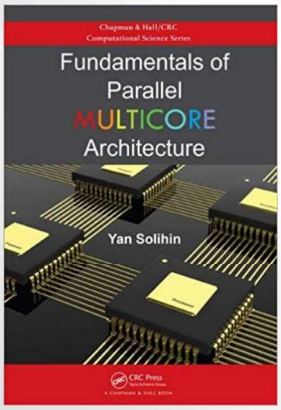
Course Overview

Lecture 1
(Chapter 1)
http://go.ncsu.edu/ece506

Learning Objectives

1. Understand the problem of race conditions in concurrent systems,
2. Learn how to decompose a program for parallel execution,
3. Be able to write simple parallel programs in the important programming models,
4. Understand the operation of common cache-coherence protocols, both bus-based and network-based, and
5. Learn about common memory-consistency models, and appreciate the advantages and disadvantages of each.

Textbook

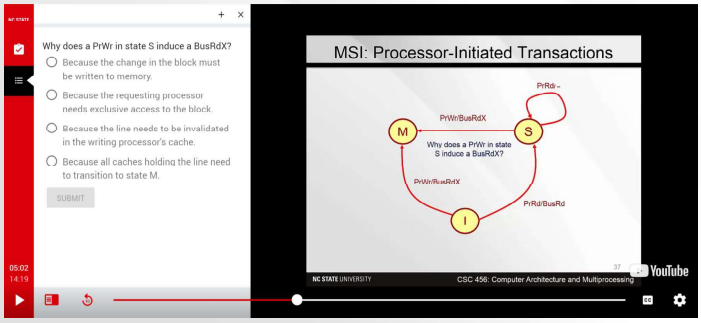


"Attendance" requirement

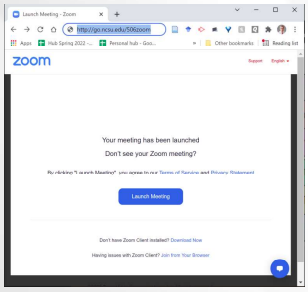
- You are required to "attend" 20 of the 26 classes.
  - **16** of these must be in the classroom.
  - "Attend" → Respond intelligently to ≥ ½ of Google forms
  - Each one not passed ⇒ –0.5% on semester average.
- You are required to pass 24 of 25 daily quizzes, plus the Syllabus Quiz. First one due Wednesday!
  - "Passed" ⇒ score of ≥ 80%
  - Each one not passed ⇒ –0.5% on semester average.
- You are required to team with 3 students.
  - Each teammate you are lacking ⇒ –0.5% on semester average

Playposit quizzes

- 3 lectures will be videos to watch.
- They have embedded quizzes.
  - Do the quizzes to get attendance credit.



Zoom session



http://go.ncsu.edu/506zoom

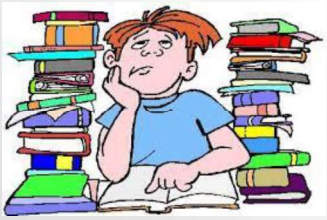
If you join the Zoom session from the classroom, be sure to let me know.

Grading

Homework 50%	4 programs: 24%
	3 problem sets: 18%
	1 peer-reviewed exercise: 8%
Tests 50%	Test 1: 10%
	Test 2: 15%
	Final exam: 25%

Homework

- 4 programs
- 3 problem sets*
- 1 peer-reviewed madeup problem



Tests

- Two 120-minute midterm tests (10%, 15% of grade)
- 150-minute final (24% of grade)
- Open book, open notes
- No computers or communication devices

Extra Credit

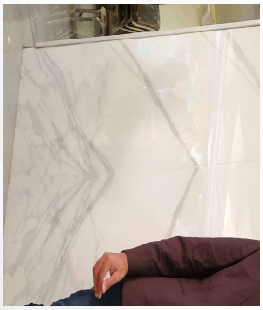
- All activities for which extra credit is given must *help other students* to learn the course material.
- Examples
  - Making outstanding contributions to answering other students' questions on **Piazza**
  - Contributing useful practice problems via **Peerwise**
  - Doing extra peer reviews of madeup problems submitted to Expertiza
  - Suggesting Web or print resources that will help other write useful madeup problems

The Staff

- Instructor

TAs

Jianxun “George” Wang



Sharan Jilla

Outline for Lecture 1

- Architectural trends
- Types of parallelism
- Flynn taxonomy
- Scope of CSC/ECE 506

Key Points

- More and more components can be integrated on a single chip
- Speed of integration tracks Moore’s law, doubling every 18–24 months.
- *Exercise:* Look up how the number of transistors per chip has changed, esp. since 2006. Submit here.
- Until recently, performance tracked speed of integration
- At the architectural level, two techniques facilitated this:
  - Cache memory
  - Instruction-level parallelism
- Performance gain from uniprocessor system was high enough that multiprocessor systems were not viable for most uses.

Illustration

- 100-processor system with perfect speedup
- Compared to a single processor system
  - Year 1: 100x faster
  - Year 2: 62.5x faster
  - Year 3: 39x faster
  - …
  - Year 10: 0.9x faster
- Single-processor performance catches up in just a few years!
- Even worse
  - It takes longer to develop a multiprocessor system
  - Low volume means prices must be very high
  - High prices delay adoption
  - Perfect speedup is unattainable

How did uniprocessor performance grow so fast?

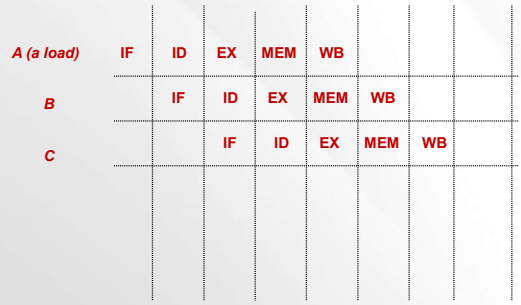
- ≈ half from circuit improvement (smaller transistors, faster clock, etc.)
- ≈ half from architecture/organization:
- Instruction-level parallelism (ILP)
  - Pipelining: RISC, CISC with RISC back-end
  - Superscalar
  - Out-of-order execution
- Memory hierarchy (caches)
  - Exploit spatial and temporal locality
  - Multiple cache levels

But uniprocessor perf. growth has stalled

- Source of performance growth had been ILP
  - Parallel execution of independent instructions from a single thread
- But ILP improvement has slowed abruptly
  - Memory wall: Processor speed grows at 55%/year, memory speed grows at 7% per year
  - ILP wall: achieving higher ILP requires quadratically increasing complexity (and power)
- Power efficiency
- Thermal packaging limit vs. cost

Types of parallelism

- Instruction level (cf. ECE 563)
  - Pipelining



Types of parallelism, cont.

- Superscalar/VLIW
- Original:

```
LD    F0, 34(R2)
ADDD  F4, F0, F2
LD    F7, 45(R3)
ADDD  F8, F7, F6
```

- Schedule as:

```
LD    F0, 34(R2)    | LD    F7, 45(R3)
ADDD  F4, F0, F2    | ADDD  F8, F0, F6
```

+ Moderate degree of parallelism
– Requires fast communication (register level)

Why ILP is slowing

- Number of pipeline stages is already deep (≈ 20–30 stages)
  - But critical dependence loops do not change
  - Memory latency requires more clock cycles to satisfy
- Branch-prediction accuracy is already > 90%
  - Hard to improve it even more
- Cache size
  - Effective, but also shows diminishing returns
  - In general, size must be doubled to reduce miss rate by half.

Current trends: multicore and manycore

Aspect	Intel Clovertown	AMD Barcelona	IBM Cell
# cores	4	4	8+1
Clock frequency	2.66 GHz	2.3 GHz	3.2 GHz
Core type	OOO Superscalar	OOO Superscalar	2-issue SIMD
Caches	2x4MB L2	512KB L2 (private), 2MB L3 (sh'd)	256KB local store
Chip power	120 watts	95 watts	100 watts

Exercise: Browse the Web (or the textbook ☺) for information on more recent processors, and for each processor, fill out this form. (You can view the submissions.)

Scope of CSC/ECE 506

- ***Parallelism***
  - ***Loop-level and task-level parallelism***
- Flynn taxonomy
  - SIMD (vector architecture)
  - MIMD
    - Shared memory machines (SMP and DSM)
    - Clusters
- Programming Model
  - Shared memory
  - Message-passing
  - Hybrid
  - Data parallel

Loop-level parallelism

- Sometimes each iteration can be performed independently.

```
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
```

- Sometimes iterations cannot be performed independently ⇒ no loop-level parallelism.

```
for (i=0; i<8; i++)
  a[i] = b[i] + a[i-1];
```

+ Very high parallelism > 1K
+ Often easy to achieve load balance
– Some loops are not parallel
– Some apps do not have many loops

Task-level parallelism

- Arbitrary code segments in a single program
- Across loops:

```
…
for (i=0; i<n; i++)
  sum = sum + a[i];
for (i=0; i<n; i++)
  prod = prod * a[i];
…
```

- Subroutines:

```
Cost = getCost();
A = computeSum();
B = A + Cost;
```

- Threads: e.g., editor: GUI, printing, parsing

+ Larger granularity ⇒ low overheads, communication
– Low degree of parallelism
– Hard to balance

Program-level parallelism

- Various independent programs execute together
- gmake:
  - `gcc -c code1.c` // assign to proc1
  - `gcc -c code2.c` // assign to proc2
  - `gcc -c main.c` // assign to proc3
  - `gcc main.o code1.o code2.o`

+ No communication
– Hard to balance
– Few opportunities

Scope of CSC/ECE 506

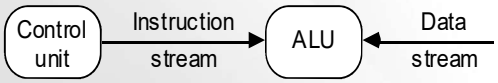
- Parallelism
  - Loop-level and task-level parallelism
- ***Flynn taxonomy***
  - ***SIMD (vector architecture)***
  - ***MIMD***
    - ***Shared-memory machines (SMP and DSM)***
    - ***Clusters***
- Programming Model
  - Shared memory
  - Message-passing
  - Hybrid
  - Data parallel

Taxonomy of parallel computers

The Flynn taxonomy

- *Single* or *multiple instruction* streams.
- *Single* or *multiple data* streams.

- **1. SISD machine**
  - Only one instruction fetch stream
  - Some not-too-ancient laptops or desktops

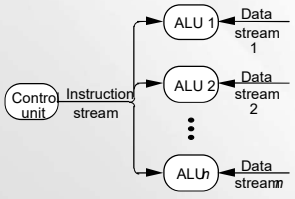


SIMD

- Examples: Vector processors, SIMD extensions (MMX), GPUs
- A single instruction operates on multiple data items.

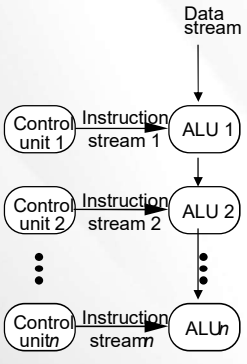
```
SISD:
for (i=0; i<8; i++)
  a[i] = b[i] + c[i];
```

```
SIMD:
a = b + c;  // vector addition
```



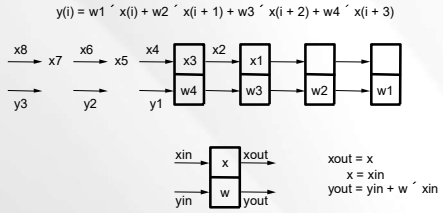
MISD

- Example: CMU Warp
- Systolic arrays



Systolic arrays (contd.)

Example: Systolic array for 1-D convolution

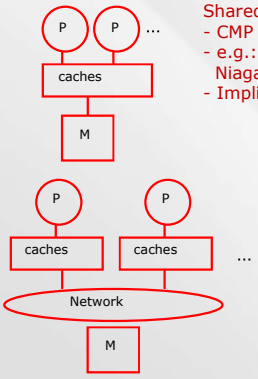


- Practical realizations (e.g. iWARP) use quite general processors
  - Enable variety of algorithms on same hardware
- But dedicated interconnect channels
  - Data transfer directly from register to register across channel
- Specialized, and same problems as SIMD
  - General-purpose systems work well for same algorithms (locality etc.)

MIMD

- Independent processors connected together to form a *multiprocessor* system.
- Physical organization
  - Determines which memory hierarchy level is shared
- Programming abstraction
  - Shared Memory:
    - on a chip: Chip Multiprocessor (CMP)
    - Interconnected by a bus: Symmetric multiprocessors (SMP)
    - Point-to-point interconnection: Distributed Shared Memory (DSM)
  - Distributed Memory:
    - Clusters, Grid

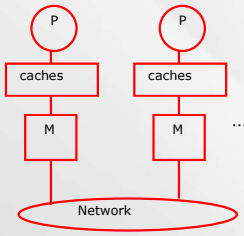
MIMD Physical Organization



Shared-cache architecture:
- CMP (or Simultaneous Multi-Threading)
- e.g.: Pentium 4 chip, IBM Power4 chip, Sun Niagara, Pentium D, etc.
- Implies shared-memory hardware

UMA (Uniform Memory Access)
Shared Memory :
- Pentium Pro Quad, Sun Enterprise, etc.
- What interconnection network?
  - Bus
  - Multistage
  - Crossbar
  - etc.
- Implies shared-memory hardware

MIMD Physical Organization (2)



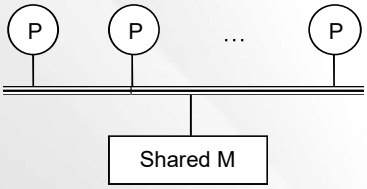
NUMA (Non-Uniform Memory Access)
Shared Memory :
- SGI Origin, Altix, IBM p690, AMD Hammer-based system
- What interconnection network?
  - Crossbar
  - Mesh
  - Hypercube
  - etc.

Scope of CSC/ECE 506

- Parallelism
  - Loop-level and task-level parallelism
- Flynn taxonomy
  - MIMD
    - Shared memory machines (SMP and DSM)
- ***Programming Model***
  - ***Shared memory***
  - ***Message-passing***
  - ***Hybrid***
  - ***Data parallel***

Programming models: shared memory

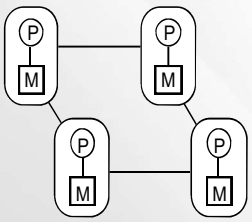
- Shared Memory / Shared Address Space:
  - Each processor can see the entire memory



  - Programming model = thread programming in uniprocessor systems

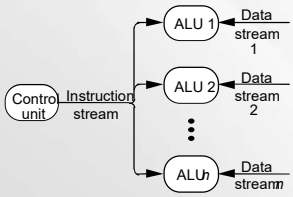
Programming models: message-passing

- Distributed Memory / Message Passing / Multiple Address Space:
  - A processor can directly access only its local memory.
  - All communication happens by explicit messages.



Programming models: data parallel

- Programming model
  - Operations performed in parallel on each element of data structure
  - Logically single thread of control, performs sequential or parallel steps
  - Conceptually, a processor associated with each data element



Data parallel (cont.)

- Architectural model
  - Array of many simple, cheap *processing elements* (PEs) each with little memory
    - Processing elements don't sequence through instructions
  - PEs are attached to a control processor that issues instructions
  - Specialized and general communication, cheap global synchronization
- Original motivation
  - Matches simple differential equation solvers
  - Centralize high cost of instruction fetch/sequencing

Top 500 supercomputers

- http://www.top500.org
- Let's look at the Earth Simulator, #1 in 2004
- Hardware:
  - 5,120 (640 8-way nodes) 500 MHz NEC CPUs
  - 8 GFLOPS per CPU (41 TFLOPS total)
    - 30s TFLOPS sustained performance!
  - 10 TB total memory
- Now (Nov. 2021)
  - Fugaku, at Fujitsu RIKEN Ctr. for Computational Science, is #1
  - 7.6 million cores
  - 5.1 PB total memory
  - 442.0 TFLOP/s max performance (Rmax)
  - 537.2 TFLOP/s peak performance (Rpeak)

Exploring the Top 500 list …

- Lists > Top500 > November 2022 > The list
  - See a list of the top systems
- Statistics > List Statistics > Vendors
  - Lenovo is top vendor, more than double HPE
- Statistics > List Statistics > Architecture
  - Clusters are overwhelmingly dominant
- Statistics > Developm't over Time > Countries
  - China comes from nowhere to lead in # of systems
  - But US still leads in performance share

Exercise

- Go to http://www.top500.org and look at the Lists and Statistics menus in the top menu bar.
- From the Statistics dropdown,
  - choose either List Statistics or Development over time,
  - then select one of the statistics, e.g., Vendors, Processor Architecture, and
  - examine what kind of systems are prevalent. Then do the same for earlier lists, and report on the trend.
- You can go all the way back to the first list from 1993.
- Submit your results here.

Three parallel-programming models

- *Shared-memory* programming is like using a "bulletin board" where you can communicate with colleagues.
- *Message-passing* is like communicating via e-mail or telephone calls. There is a well defined event when a message is sent or received.
- *Data-parallel* programming is a "regimented" form of cooperation. Many processors perform an action separately on different sets of data, then exchange information globally before continuing en masse.

User-level communication primitives are provided to realize the programming model

- There is a mapping between language primitives of the programming model and these primitives

These primitives are supported directly by hardware, or via OS, or via user software.

In the early days, the kind of programming model that could be used was closely tied to the architecture.

Today—

- Compilers and software play important roles as bridges
- Technology trends exert a strong influence

The result is convergence in organizational structure, and relatively simple, general-purpose communication primitives.

A shared address space

In the shared-memory model, processes can access the same memory locations.

Communication occurs implicitly as result of loads and stores

This is convenient.

- Wide range of granularities supported.
- Similar programming model to time-sharing on uniprocessors, except that processes run on different processors
- Wide range of scale: few to hundreds of processors

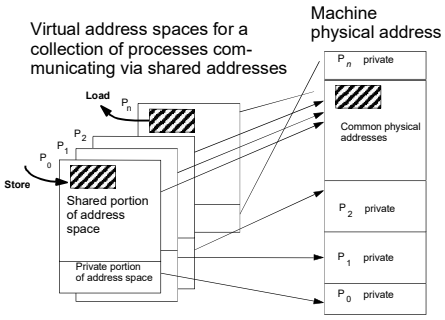
Good throughput on multiprogrammed workloads.

This is popularly known as the *shared memory* model, even though memory may be physically distributed among processors.

The shared-memory model

A process is a virtual address space plus

Portions of the address spaces of tasks are shared.



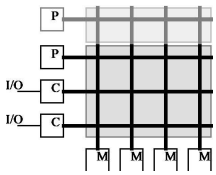
What does the private region of the virtual address space usually contain?

Conventional memory operations can be used for communication.

Special atomic operations are used for synchronization.

The interconnection structure

The interconnect in a shared-memory multiprocessor can take several forms.



It may be a *crossbar switch*.

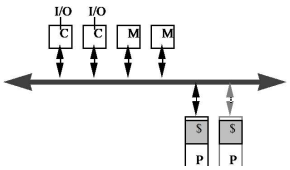
Each processor has a direct connection to each memory and I/O controller.

Bandwidth scales with the number of processors.

Unfortunately, cost scales with

This is sometimes called the "mainframe approach."

At the other end of the spectrum is a *shared-bus* architecture.



All processors, memories, and I/O controllers are connected to the bus.

Such a multiprocessor is called a symmetric multiprocessor (SMP).

What are some advantages and disadvantages of organizing a multiprocessor this way? List them here.

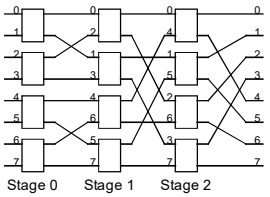
-
-
-

A compromise between these two organizations is a *multistage interconnection network*.

The processors are on one side, and the memories and controllers are on the other.

Each memory reference has to traverse the stages of the network.

Why is this called a compromise between the other two strategies?



For small configurations, however, a shared bus is quite viable.

Message passing

In a message-passing architecture, a complete computer, including the I/O, is used as a building block.

Communication is via explicit I/O operations, instead of loads and stores.

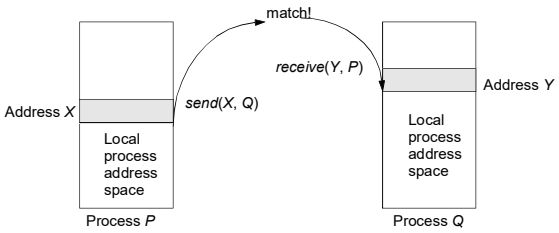
- A program can directly access only its private address space (in local memory).
- It communicates via explicit messages (*send* and *receive*).

It is like a network of workstations (clusters), but more tightly integrated.

Easier to build than a scalable shared-memory machine.

Send-receive primitives

The programming model is further removed from basic hardware operations.



Library or OS intervention is required to do communication.

- *send* specifies a buffer to be transmitted, and the receiving process.
- *receive* specifies sending process, and a storage area to receive into.
- A memory-to-memory copy is performed, from the address space of one process to the address space of the other.
- There are several possible variants, including whether *send* completes—
  - when the *receive* has been executed,
  - when the send buffer is available for reuse, or
  - when the message has been sent.
- Similarly, a *receive* can wait for a matching *send* to execute, or simply fail if one has not occurred.

There are many overheads: copying, buffer management, protection. Let's describe each of these. Submit your descriptions here.

- Why is there an overhead to copying, compared to a share-memory machine?

- Describe the overhead of buffer management.

- What is the overhead for protection?

Here's an example from the textbook of the difference between shared address-space and message-passing programming.

A shared-memory system uses the _____ model:

```
int a, b, signal;
…
void dosum(<args>) {
  while (signal == 0) {}; // wait until instructed to work
  printf("child thread> sum is %d", a + b);
  signal = 0;  // my work is done
}

void main() {
  signal = 0;
  thread_create(&dosum);  // spawn child thread
  a = 5, b = 3;
  signal = 1;             // tell child to work
  while (signal == 1) {}  // wait until child done
  printf("all done, exiting\n");
}
```

Message-passing uses the _____ model:

```
int a, b;
…
void dosum() {
  recvMsg(mainID, &a, &b);
  printf("child process> sum is %d", a + b);
}

void main() {
  if (fork() == 0)  // I am the child process
    dosum();
  else {            // I am the parent process
    a = 5, b = 3;
    sendMsg(childID, a, b);
```



```
    wait(childID);
    printf("all done, exiting\n");
  }
}
```

Here's the relevant section of documentation on the **`fork()`** function: "Upon successful completion, **`fork()`** and **`fork1()`** return 0 to the child process and return the process ID of the child process to the parent process."

Interconnection topologies

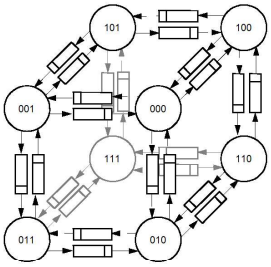
Early message-passing designs provided hardware primitives that were very close to the message-passing model.

Each node was connected to a fixed set of neighbors in a regular pattern by point-to-point links that behaved as FIFOs.

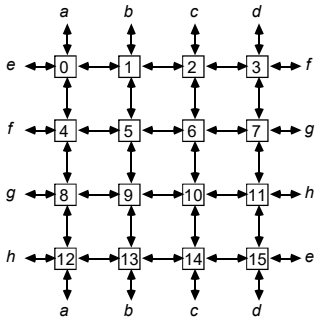
A common design was a *hypercube*, which had $2 \times n$ links per node, where n was the number of dimensions.

The diagram shows a 3D cube.

One problem with hypercubes was that they were difficult to lay out on silicon.



Because of that, 2D meshes eventually supplanted hypercubes.



Here is an example of a 16-node mesh. Note that the last element in one row is connected to the first element in the next.

If the last element in each row were connected to the first element in the same row, we would have a *torus* instead.

Early message-passing machines used a FIFO on each link.

- Thus, software sends were implemented as synchronous hardware operations at each node.

  What was the problem with this, for multi-hop messages?

- Synchronous ops were replaced by DMA, enabling non-blocking operations
  - A DMA device is a special-purpose controller that transfers data between memory and an I/O device without processor intervention.
  - Messages were buffered by the message layer of the system at the destination until a *receive* took place.
  - When a *receive* took place, the data was

The diminishing role of topology.

- With store-and-forward routing, topology was important.

Parallel algorithms were often changed to conform to the topology of the machine on which they would be run.

- Introduction of pipelined ("wormhole") routing made topology less important.

In current machines, it makes less difference how far the data travels.

This simplifies programming; cost of interprocessor communication is essentially independent of which processor is receiving the data.

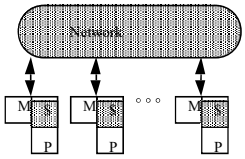
Toward architectural convergence

In 1990, there was a clear distinction between message-passing and shared-memory machines. Today, there isn't a distinct boundary.

- Message-passing operations are supported on most shared-memory machines.
- A shared virtual address space can be constructed on a message-passing machine, by sharing *pages* between processors.
  - When a missing page is accessed, a page fault occurs.
  - The OS fetches the page from the remote node via message-passing.

At the machine-organization level, the designs have converged too.

The block diagrams for shared-memory and message-passing machines look essentially like this:



In shared memory, the network interface is integrated with the memory controller.

It initiates a transaction to access memory at a remote node.

In message-passing, the network interface is essentially an I/O device.

What does Solihin say about the ease of writing shared-memory and message-passing programs on these architectures?

- Which model is easier to program for initially?
- Why doesn't it make much difference in the long run?

The limits of parallelism: Amdahl's law

Speedup is defined as

$$\frac{\text{time for serial execution}}{\text{time for parallel execution}}$$

or, more precisely, as

$$\frac{\text{time for serial execution of best serial algorithm}}{\text{time for parallel execution of our algorithm}}$$

Give two reasons why it is better to define it the second way than the first.

[§4.3.1] If some portions of the problem don't have much concurrency, the speedup on those portions will be low, lowering the average speedup of the whole program.

Exercise: Submit your answers to the questions below.

Suppose that a program is composed of a serial phase and a parallel phase.

- The whole program runs for 1 time unit.
- The serial phase runs for time *s*, and the parallel phase for time 1–*s*.

Then regardless of how many processors *N* are used, the execution time of the program will be at least ___

and the speedup will be no more than ___. This is known as *Amdahl's law.*

For example, if 25% of the program's execution time is serial, then regardless of how many processors are used, we can achieve a speedup of no more than __.

Efficiency is defined as

$$\frac{\text{speedup}}{\text{number of processors}}$$

Let us normalize computation time so that

- the serial phase takes time 1, and
- the parallel phase takes time *p* if run on a single processor.

Then if run on a machine with *N* processors, the parallel phase takes *p*/*N*.

Let α be the ratio of serial time to total execution time. Thus

$$\alpha = \frac{1}{1 + p/N} = \frac{N}{N + p}.$$

For large *N*, α approaches __, so efficiency approaches ___.

Does it help to add processors?

Gustafson's law: But this is a pessimistic way of looking at the situation.

In 1988, Gustafson et al. noted that as computers become more powerful, people run larger and larger programs.

Therefore, as *N* increases, *p* tends to increase too. Thus, as *N* increases, α does not get very close to 1, and efficiency remains reasonable.

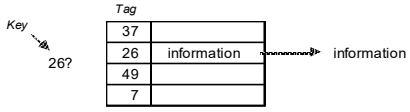
There may be a maximum to the amount of speedup for a given problem size, but since the problem is "scaled" to match the processing power of the computer, there is no clear maximum to "scaled speedup."

Gustafson's law states that any sufficiently large problem can be efficiently parallelized.

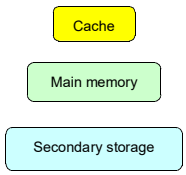
Cache memories

[§5.1] A *cache* is a small, fast memory which is *transparent* to the processor.

- The cache duplicates information that is in main memory.
- With each data block in the cache, there is associated an *identifier* or *tag.* This allows the cache to be *content addressable.*



- Caches are smaller and faster than main memory.
- *Secondary storage*, on the other hand, is larger and slower.



- A *cache miss* is the term analogous to a page fault. It occurs when a referenced word is not in the cache.
  - Cache misses must be handled much more quickly than page faults. Thus, they are handled in hardware.
- Caches can be *organized* according to four different strategies:
  - Direct
  - Fully associative
  - Set associative
  - Sectored

- A cache implements several different *policies* for retrieving and storing information, one in each of the following categories:
  - *Placement policy*—determines where a block is placed when it is brought into the cache.
  - *Replacement policy*—determines what information is purged when space is needed for a new entry.
  - *Write policy*—determines how soon information in the cache is written to lower levels in the memory hierarchy.

Cache memory organization

[§5.2] Information is moved into and out of the cache in *blocks.* When a block is in the cache, it occupies a cache *line*. Blocks are usually larger than one byte,

- to take advantage of locality in programs, and
- because memory may be organized so that it can overlap transfers of several bytes at a time.

The block size is the same as the line size of the cache.

A *placement policy* determines where a particular block can be placed when it goes into the cache. E.g., is a block of memory eligible to be placed in any line in the cache, or is it restricted to a single line?

In our examples, we assume—

- The cache contains 2048 bytes,
  with 16 bytes per line
  Thus it has lines.
- Main memory is made up of 256K bytes, or 16384 blocks. Thus an address consists of

We want to structure the cache to achieve a high *hit ratio.*

- *Hit*—the referenced information is in the cache.
- *Miss*—referenced information is not in cache, must be read in from main memory.

$$\text{Hit ratio} \equiv \frac{\text{Number of hits}}{\text{Total number of references}}$$

We will study caches that have three different placement policies (direct, fully associative, set associative).

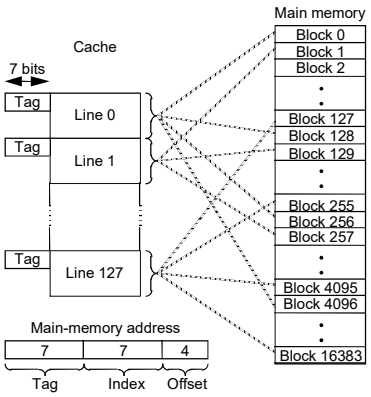
Direct

Only 1 choice of where to place a block.

$$\text{block } i \rightarrow \text{line } i \bmod 128$$

Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order seven bits of the main-memory address of the block.



To search for a word in the cache,

1. Determine what line to look in (easy; just select bits 10–4 of the address).
2. Compare the leading seven bits (bits 17–11) of the address with the tag of the line. If it matches, the block is in the cache.
3. Select the desired bytes from the line.

Advantages:

Fast lookup (only one comparison needed).

Cheap hardware (only one tag needs to be checked).

Easy to decide where to place a block

Disadvantage: Contention for cache lines.

Exercise: What would the size of the tag, index, and offset fields be if—

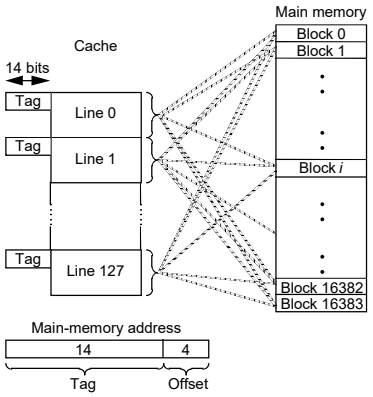
- the line size from our example were doubled, without changing the size of the cache?
- the cache size from our example were doubled, without changing the size of the line?
- an address were 32 bits long, but the cache size and line size were the same as in the example?

Fully associative

Any block can be placed in *any* line in the cache.

This means that we have 128 choices of where to place a block.

block $i \rightarrow$ any free (or purgeable) cache location



Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order *fourteen* bits of the main-memory address of the block.

To search for a word in the cache,

1. Simultaneously compare the leading 14 bits (bits 17–4) of the address with the tag of all lines. If it matches any one, the block is in the cache.
2. Select the desired bytes from the line.

Advantages:

Minimal contention for lines.

Wide variety of replacement algorithms feasible.

Exercise: What would the size of the tag and offset fields be if—

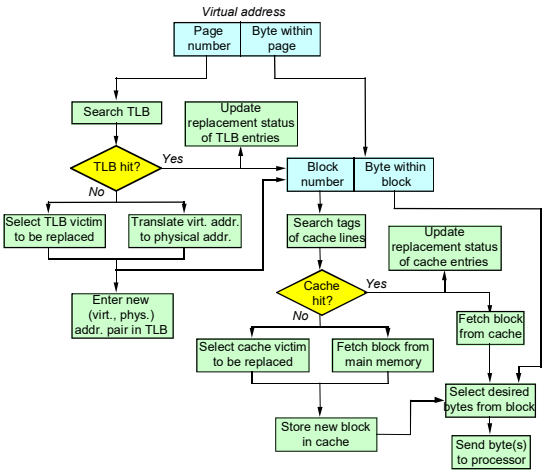
- the line size from our example were doubled, without changing the size of the cache?

- the cache size from our example were doubled, without changing the size of the line?
- an address were 32 bits long, but the cache size and line size were the same as in the example?

Disadvantage:

The most expensive of all organizations, due to the high cost of associative-comparison hardware.

A flowchart of cache operation: The process of searching a fully associative cache is very similar to using a directly mapped cache. Let us consider them in detail.



Which steps would be different if the cache were directly mapped?

Set associative

$1 < n < 128$ choices of where to place a block.

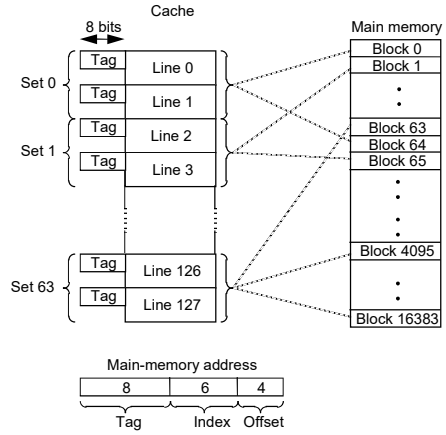
A compromise between direct and fully associative strategies.

The cache is divided into s sets, where s is a power of 2.

$$\text{block } i \rightarrow \text{ any line in set } i \bmod s$$

Each line has its own tag associated with it.

When the line is in use, the tag contains the high-order *eight* bits of the main-memory address of the block. (The next six bits can be derived from the set number.)



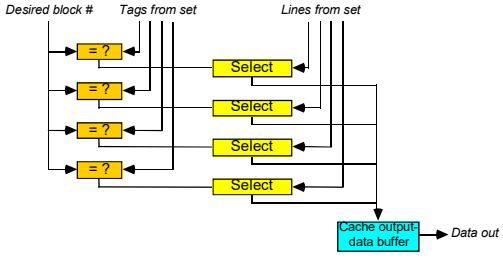
Exercise: What would the size of the tag, index, and offset fields be if—

- the line size from our example were doubled, without changing the size of the cache?
- the set size from our example were doubled, without changing the size of a line or the cache?
- the cache size from our example were doubled, without changing the size of the line or a set?
- an address were 32 bits long, but the cache size and line size was the same as in the example?

To search for a word in the cache,

1. Select the proper set (i mod s).
2. Simultaneously compare the leading 8 bits (bits 17–10) of the address with the tag of all lines in the set. If it matches any one, the block is in the cache.

   At the same time, the (first bytes of) the lines are also being read out so they will be accessible at the end of the cycle.
3. If a match is found, gate the data from the proper block to the cache-output buffer.
4. Select the desired bytes from the line



- All reads from the cache occur as early as possible, to allow maximum time for the comparison to take place.
- Which line to use is decided late, after the data have reached high-speed registers, so the processor can receive the data fast.

Factors influencing line lengths:

- Long lines ⇒ higher hit ratios.
- Long lines ⇒ less memory devoted to tags.
- Long lines ⇒ longer memory transactions (undesirable in a multiprocessor).
- Long lines ⇒ more write-backs (explained below).

For most machines, line sizes between 32 and 128 bytes perform best.

If there are *b* lines per set, the cache is said to be *b-way* set associative. How many way associative was the example above?

The logic to compare 2, 4, or 8 tags simultaneously can be made quite fast.

But as *b* increases beyond that, cycle time starts to climb, and the higher cycle time begins to offset the increased associativity.

Almost all L1 caches are less than 8-way set-associative. L2 caches often have higher associativity.

Two-level caches

Write policy

[§5.2.3] Answer these questions, based on the text.

What are the two write policies mentioned in the text?

Which one is typically used when a block is to be written to main memory, and why?

Which one can be used when a block is to be written to a lower level of the cache, and why?

Can you explain what error correction has to do with the choice of write policy?

Explain what a parity bit has to do with this.

Principle of inclusion

[§5.2.4] To analyze a second-level cache, we use the *principle of inclusion*—a large second-level cache includes everything in the first-level cache.

We can then do the analysis by assuming the first-level cache did not exist, and measuring the hit ratio of the second-level cache alone.

How should the line length in the second-level cache relate to the line length in the first-level cache?

When we measure a two-level cache system, two miss ratios are of interest:

- The *local miss rate* for a cache is the

  $$\frac{\text{\# misses experienced by the cache}}{\text{number of incoming references}}$$

  To compute this ratio for the L2 cache, we need to know the number of misses in

- The *global miss rate* of the cache is

  $$\frac{\text{\# L2 misses}}{\text{\# of references made by processor}}$$

  This is the primary measure of the L2 cache.

What conditions need to be satisfied in order for inclusion to hold?

- L2 associativity must be ≥ L1 associativity, irrespective of the number of sets.

  Otherwise, more entries in a particular set could fit into the L1 cache than the L2 cache, which means the L2 cache couldn't hold everything in the L1 cache.
- The number of L2 sets has to be ≥ the number of L1 sets, irrespective of L2 associativity.

  (Assume that the L2 line size is ≥ L1 line size.)

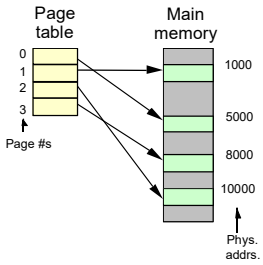
  If this were not true, multiple L1 sets would depend on a single L2 set for backing store. So references to one L1 set could affect the backing store for another L1 set.
- All reference information from L1 is passed to L2 so that it can update its replacement bits.

Even if all of these conditions hold, we still won't have logical inclusion if L1 is write-back. (However, we will still have *statistical inclusion*—L2 *usually* contains L1 data.)

[§5.2.6] Translation Lookaside Buffers

The CPU generates *virtual* addresses, which correspond to locations in virtual memory.

In principle, the virtual addresses are translated to physical addresses using a page table.

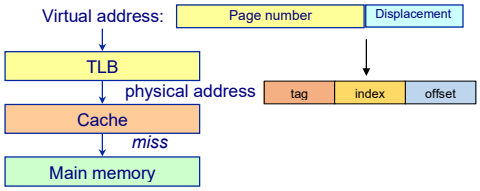


But this is too slow, so in practice, a *translation lookaside buffer* (TLB) is used.

It is like a special cache that is indexed by page number.

If there is a hit on a page number, then the address of the page in memory (called the *page-frame address*) is immediately obtained.

Therefore, the TLB and the cache must be accessed sequentially.

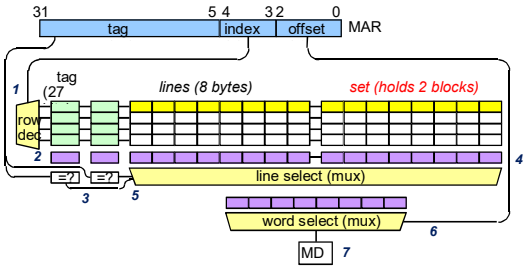


This adds an extra cycle in case of a hit.

(The page *displacement* is sometimes called the "page offset." But we will call it the displacement to avoid confusion with the block offset," which we just call "offset.")

How can we avoid wasting this time?

Let's look at what happens when a memory address is accessed.



What are the steps in cache access?

1.
2.
3.
4.
5.
6.
7.

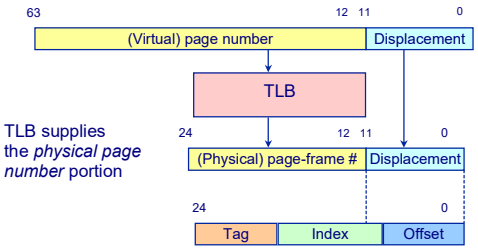
We always need to read lines into the sense amplifiers and then select the word (cf. the direct-mapped cache diagram in Lecture 4).

Now, if we know the index *before* address translation takes place, we can perform steps _____ while address translation is occurring.

There is a tradeoff between speed and power efficiency.

- For power efficiency, which order should should steps 1 through 4 be performed in?
- For maximum speed, which of steps 1 through 4 can be performed in parallel?

Let's take a look at address translation.



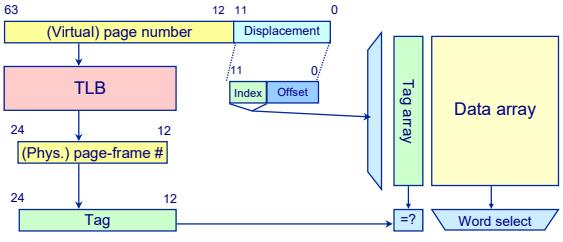
In this example, what is the page size?

How much physical memory is there?

Our goal is to allow the cache to be indexed before address translation completes.

In order to do that, we need to have the index field be *entirely contained* within the page displacement.

So, if the displacement is d bits wide, the width of the index is j bits, and the offset is k bits, we must have $j + k \leq d$.



Cache hit time reduces from two cycles to one!

> … because the cache can now be *indexed* in parallel with TLB (although the tag match uses output from the TLB).

But there are some constraints...

- Suppose our cache is direct mapped. Then the index field just contains the line number. So, (line number || block offset) must fit inside the page displacement.

  What is the largest the cache can be?
- If we want to increase the size of the cache, what can we do?

Options:

- For new machines, select page size such that—

$$\text{page size} \geq \frac{\text{cache size}}{\text{associativity}}$$

- If page size is fixed, select associativity so that—

$$\text{associativity} \geq \frac{\text{cache size}}{\text{page size}}$$

Example: MC88110

- Page size = 4KB
- I-cache, D-cache are both: 8KB, 2-way set-associative (4KB = 8KB / 2)

Example: VAX series

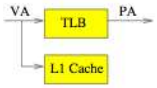
- Page size = 512B
- For a 16KB cache, need assoc. = (16KB / 512B) = 32-way set. assoc.!

The textbook gives these three alternatives for cache indexing and tagging. Answer some questions about them.

Physically Indexed and Tagged

What's the main disadantage of physically indexed and tagged?

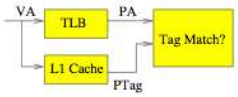
Virtually Indexed and Tagged



What is the organization we have just been discussing (in the last diagram)?

What is the main disadvantage of virtually indexed and tagged?

Virtually Indexed but Physically Tagged



Multilevel cache design

What are distinguishing features of the different cache levels of the four-level design (from 2013) illustrated on p. 135 of the textbook?

	Distinguish-ing feature	Size	Access time	Implement'n techology
L1 cache				
L2 cache				
L3 cache				
L4 cache				
Main mem.				

What are some advantages of a centralized cache?

What are some advantages of a banked structure?

Inclusion in multilevel caches

Answer these questions about inclusion policies.

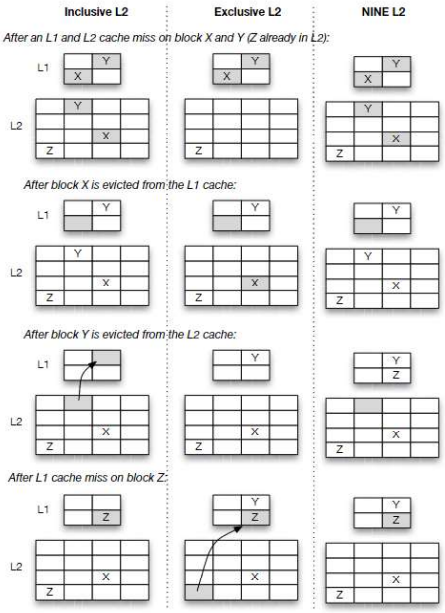
Which kind(s) of caches move a block from one level to the other?

Which kind(s) of caches propagate up an eviction from the L2 to the L1?

Which kind(s) of caches have to inform the L2 about a write to the L1?

In an inclusive cache, can L2 associativity be greater than L1 associativity?

Find and describe the typo in this diagram.



Replacement policies

LRU is a good strategy for cache replacement.

In a set-associative cache, LRU is reasonably cheap to implement. Why?

With the LRU algorithm, the lines can be arranged in an *LRU stack*, in order of recency of reference. Suppose a string of references is—

a b c d a b e a b c d e

and there are 4 lines. Then the LRU stacks after each reference are—

a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*	*d*	*e*
	a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*	*d*
		a	*b*	*c*	*d*	*a*	*b*	*e*	*a*	*b*	*c*
			a	*b*	*c*	*d*	*d*	*d*	*e*	*a*	*b*
*	*	*	*			*			*	*	*

Notice that at each step:

- The line that is referenced moves to the top of the LRU stack.
- All lines below that line keep their same position.
- All lines above that line move down by one position.

How many bits per set are required to keep track of LRU status in both of the implementations described in the text?

- Matrix
- Pseudo-LRU

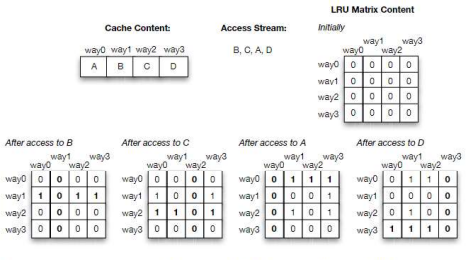


Figure 5.6: Illustrating matrix implementation of the least recently used (LRU) replacement policy.

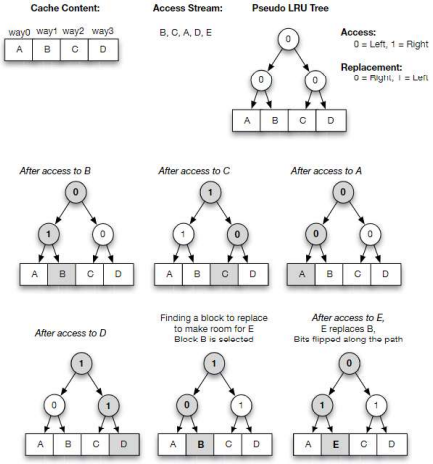


Figure 5.7: Illustration of pseudo-LRU replacement on a 4-way set associative cache.

The Cache-Coherence Problem

Lecture 6
(Chapter 6)

Outline

- **Bus-based multiprocessors**
- The cache-coherence problem
- Peterson's algorithm
- Coherence vs. consistency

Shared vs. Distributed Memory

- What is the difference between …
  - SMP
  - NUMA
  - Cluster ?

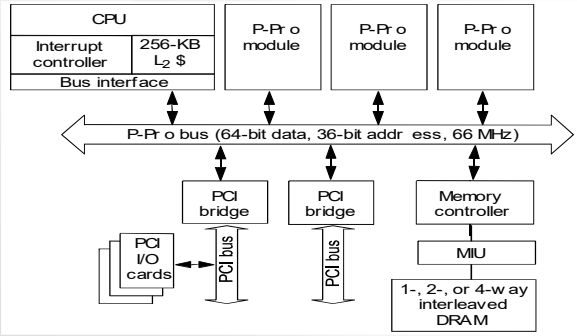
Small to Large Multiprocessors

- **Small scale** (2–30 processors): shared memory
  - Often on-chip: shared memory (+ perhaps shared cache)
  - Most processors have MP support out of the box
  - Most of these systems are bus-based
  - Popular in commercial as well as HPC markets
- **Medium scale** (64–256): shared memory and clusters
  - Clusters are cheaper
  - Often, clusters of SMPs
- **Large scale** (> 256): few shared memory and many clusters
  - SGI Altix 3300: 512-processor shared memory (NUMA)
  - Large variety on custom/off-the-shelf components such as interconnection networks.
    - Beowulf clusters: fast Ethernet
    - Myrinet: fiber optics
    - IBM SP2: custom

Shared Memory vs. No Shared Memory

- Advantages of shared-memory machines (vs. distributed memory w/same total memory size)
  - Support shared-memory programming
    - Clusters can also support it via *software shared virtual memory*, but with much coarser granularity and higher overheads
  - Allow fine-grained sharing
    - You can't do this with messages—there's too much overhead to share small items
  - Single OS image
- Disadvantage of shared-memory machines
  - Cost of providing shared-memory abstraction

A Bus-Based Multiprocessor



Outline

- Bus-based multiprocessors
- **The cache-coherence problem**
- Peterson's algorithm
- Coherence vs. consistency

Will This Parallel Code Work Correctly?

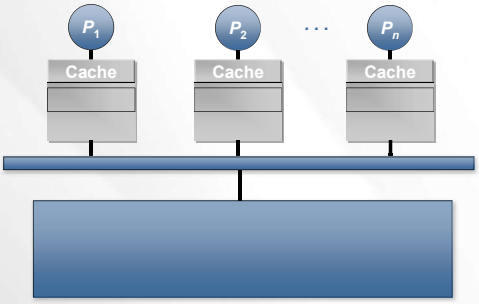
```
sum = 0;
begin parallel
for (i=1; i<=2; i++) {
    lock(id, myLock);
    sum = sum + a[i];
    unlock(id, myLock);
end parallel
print sum;

Suppose  a[1] = 3 and
         a[2] = 7
```

Two issues:

- **Will it print `sum = 10`?**
- **How can it support locking correctly?**

The Cache-Coherence Problem

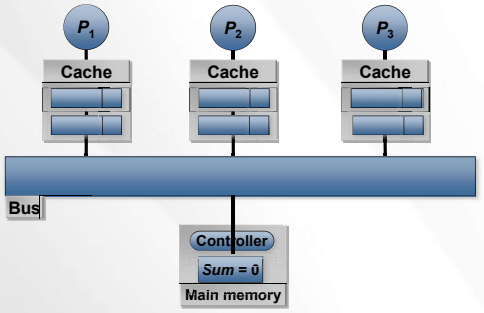
```
sum = 0;
begin parallel
for (i=1; i<=2; i++) {
    lock(id, myLock);
    sum = sum + a[i];
    unlock(id, myLock);
end parallel
print sum;

Suppose a[1] = 3 and
a[2] = 7
```



- **Will it print `sum = 10`?**

Cache-Coherence Problem Illustration

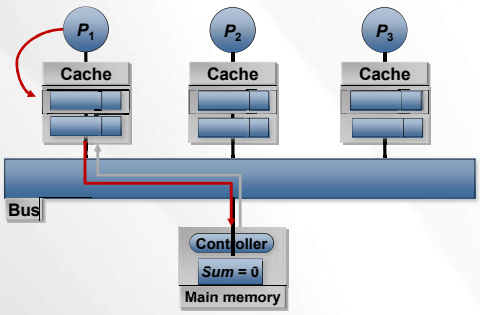
Start state. All caches empty and main memory has *Sum* = 0.



Trace
P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

Cache-Coherence Problem Illustration

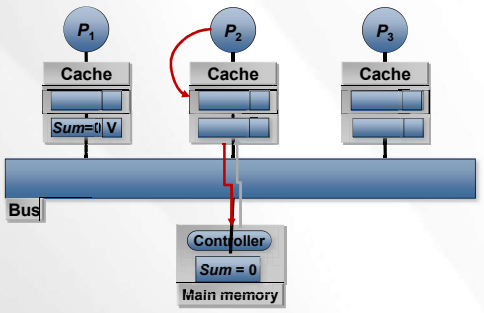
P_1 reads *Sum* from memory.



Trace
P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

Cache-Coherence Problem Illustration

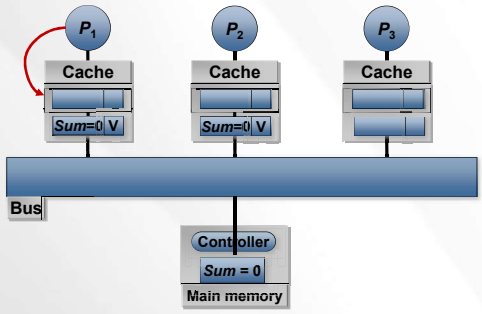
P_2 reads. Let's assume this comes from memory too.



Trace
P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

Cache-Coherence Problem Illustration

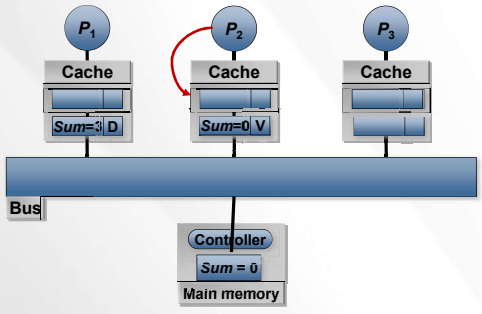
P_1 writes. This write goes to the cache.



Trace
P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

Cache-Coherence Problem Illustration

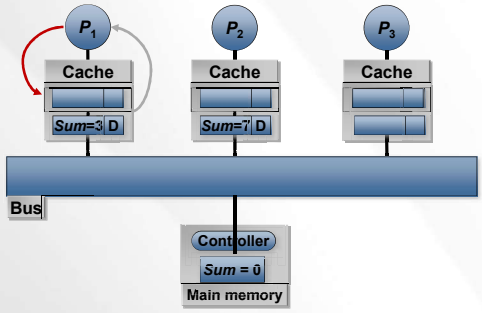
P_2 writes.



Trace
P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

Cache-Coherence Problem Illustration

P_1 reads.



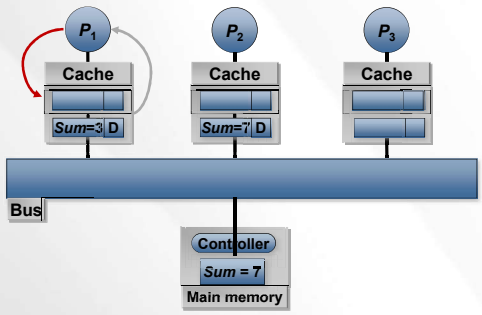
Trace
P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

Cache-Coherence Problem

- Do P1 and P2 see the same sum?
- Does it matter if we use a WT cache?
- What if we do not have caches, or sum is uncacheable. Will it work?
- The code given at the start of the animation does not exhibit the same coherence problem shown in the animation. Explain why.

Write-Through Cache Does Not Work

P_1 reads.



Trace
P_1 Read *Sum*
P_2 Read *Sum*
P_1 Write *Sum* = 3
P_2 Write *Sum* = 7
P_1 Read *Sum*

Software Lock Using a Flag

- Here's simple code to implement a lock:

```
void lock (int process, int lvar) {      // process is 0 or 1
  while (lvar == 1) {} ;
  lvar = 1;
}

void unlock (int process, int lvar) {
  lvar = 0;
}
```

- Will this guarantee mutual exclusion?
- Let's look at an algorithm that will …

Outline

- Bus-based multiprocessors
- The cache-coherence problem
- **Peterson's algorithm**
- Coherence vs. consistency

Peterson's Algorithm

```
int turn;
int interested[n];  // initialized to false

void lock (int process, int lvar) {     // process is 0 or 1
  int other = 1 - process;
  interested[process] = TRUE;
  turn = other;
  while (turn == other && interested[other] == TRUE) {} ;
}
// Post: turn != other or interested[other] == FALSE

void unlock (int process, int lvar) {
  interested[process] = FALSE;
}
```

- Acquisition of **`lock()`** occurs only if
  1. **`interested[other] == FALSE`**: either the other process has not competed for the lock, or it has just called **`unlock()`**, or
  2. **`turn != other`**: the other process is competing, has set the turn to *our* process, and will be blocked in the **`while()`** loop

No Race

```
// Proc 0
interested[0] = TRUE;
turn = 1;
while (turn==1 && interested[1]==TRUE)
  {};
// since interested[1] starts out FALSE,
// Proc 0 enters critical section
```

```
// Proc 1
interested[1] = TRUE;
turn = 0;
while (turn==0 && interested[0]==TRUE)
  {};
// since turn==0 && interested[0]==TRUE
// Proc 1 waits in the loop until Proc 0
// releases the lock
```

```
// unlock
interested[0] = FALSE;
```

```
// now Proc 1 can exit the loop and
// acquire the lock
```


Race

```
// Proc 0
interested[0] = TRUE;
turn = 1;

while (turn==1 && interested[1]==TRUE)
  {};
// since turn == 0,
// Proc 0 enters critical section

// unlock
interested[0] = FALSE;
```

```
// Proc 1
interested[1] = TRUE;

turn = 0;

while (turn==0 && interested[0]==TRUE)
  {};
// since turn==0 && interested[0]==TRUE
// Proc 1 waits in the loop until Proc 0
// releases the lock

// now Proc 1 can exit the loop and
// acquire the lock
```


When Does Peterson's Alg. Work?

- Correctness depends on the global order of

```
A: interested[process] = TRUE;
B: turn = other;
```

- Thus, it will not work if—
  - The *compiler* reorders the operations
    - There's no data dependence, so unless the compiler is notified, it may well reorder the operations
    - This prevents compiler from using aggressive optimizations used in serial programs
  - The *architecture* reorders the operations
    - Write buffers, memory controller
    - Network delay for statement A
    - If **`turn`** and **`interested[]`** are cacheable, A may result in cache miss, but B in cache hit
- This is called the memory-consistency problem.

Race on a Non-Sequentially Consistent Machine

```
// Proc 0
interested[0] = TRUE;
```

```
// Proc 1
interested[1] = TRUE;
turn = 0;
```

```
turn = 1;
while (turn==1 && interested[1]==TRUE)
  {};
```

```
while (turn==0 && interested[0]==TRUE)
  {};
```

Race on a Non-Sequentially Consistent Machine

```
// Proc 0                                  // Proc 1
interested[0] = TRUE;

                                           turn = 0;          reordered

turn = 1;
while (turn==1 && interested[1]==TRUE)
  {};
// since interested[1] == FALSE,           interested[1] = TRUE;
// Proc 0 enters critical section          while (turn==0 && interested[0]==TRUE)
                                             {};
                                           // since turn==1,
                                           // Proc 1 enters critical section
```

Can you explain what has gone wrong here?

Coherence vs. Consistency

Cache coherence	Memory consistency
Deals with the ordering of operations to a *single* memory location.	Deals with the ordering of operations to *different memory locations.*

Coherence vs. Consistency

Cache coherence	Memory consistency
Deals with the ordering of operations to a *single* memory location.	Deals with the ordering of operations to *different memory locations.*
Tackled by hardware • using coherence protocols. • Hw. alone guarantees correctness but with varying performance	Tackled by consistency models • supported by hardware, but • software must conform to the model.

Coherence vs. Consistency

Cache coherence	Memory consistency
Deals with the ordering of operations to a *single* memory location.	Deals with the ordering of operations to *different memory locations.*
Tackled by hardware • using coherence protocols. • Hw. alone guarantees correctness but with varying performance	Tackled by consistency models • supported by hardware, but • software must conform to the model.
All protocols realize same abstraction • A program written for 1 protocol can run w/o change on any other.	Models provide diff. abstractions • Compilers must be aware of the model (no reordering certain operations …). • Programs must "be careful" in using shared variables.

Two Approaches to Consistency

- Sequential consistency
  - Multi-threaded codes for uniprocessors automatically run correctly
  - How? Every shared R/W completes globally in program order
  - Most intuitive but worst performance
- Relaxed consistency models
  - Multi-threaded codes for uniprocessor need to be ported to run correctly
  - Additional instruction (memory fence) to ensure global order between 2 operations

Cache Coherence

- Do we need caches?
  - Yes, to reduce average data access time.
  - Yes, to reduce bandwidth needed for bus/interconnect.
- Sufficient conditions for coherence:
  - Notation: $\text{Request}_{proc}(data)$
  - ***Write propagation***:
    - $\text{Rd}_i(X)$ must return the "latest" $\text{Wr}_j(X)$
  - ***Write serialization***:
    - $\text{Wr}_i(X)$ and $\text{Wr}_j(X)$ are seen in the same order by everybody
      - e.g., if I see w2 after w1, you shouldn't see w2 before w1
      - → There must be a **global ordering** of memory operations *to a single location*
  - Is there a need for *read serialization*?

A Coherent Memory System: Intuition

- Uniprocessors
  - Coherence between I/O devices and processors
  - Infrequent, so software solutions work
    - uncacheable memory, uncacheable operations, flush pages, pass I/O data through caches
- But coherence problem much more critical in multiprocessors
  - Pervasive
  - Performance-critical
  - Necessitates a hardware solution
- * Note that "latest write" is ambiguous.
  - Ultimately, what we care about is that any write is propagated everywhere in the same order.
  - Synchronization defines what "latest" means.

Summary

- Shared memory with caches raises the problem of cache coherence.
  - Writes to the same location must be seen in the same order everywhere.
- But this is not the only problem
  - Writes to *different* locations must also be kept in order if they are being depended upon for synchronizing tasks.
  - This is called the memory-consistency problem

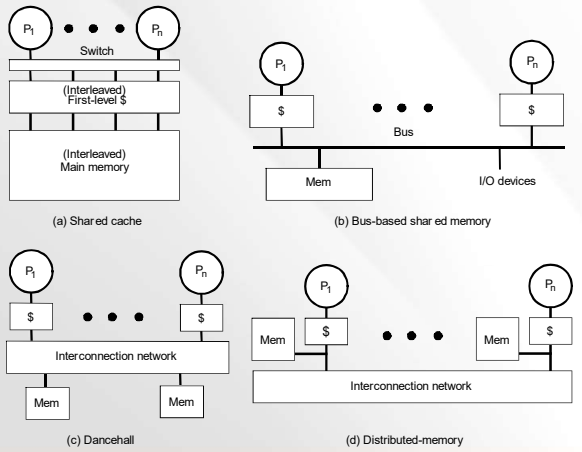
Coherence and Consistency

Lecture 7
(Chapter 7)

Outline

- **Bus-based coherence**
- Invalidation vs. update coherence protocols
- Memory consistency
  - Sequential consistency

Several Configurations for a Memory System



Assume a Bus-Based SMP

- Built on top of two fundamentals of uniprocessor system
  - Bus transactions
  - Cache-line finite-state machine
- Uniprocessor bus transaction:
  - Three phases: arbitration, command/address, data transfer
  - All devices observe addresses, one is responsible
- Uniprocessor cache states:
  - Every cache line has a finite-state machine
  - In WT+write no-allocate: Valid, Invalid states
  - WB: Valid, Invalid, Modified ("Dirty")
- Multiprocessors extend both these somewhat to implement coherence

Snoop-Based Coherence on a Bus

- *Basic Idea*
  - Assign a snooper to each processor so that all bus transactions are visible to all processors ("snooping").
  - Processors (via cache controllers) change line states on relevant events.

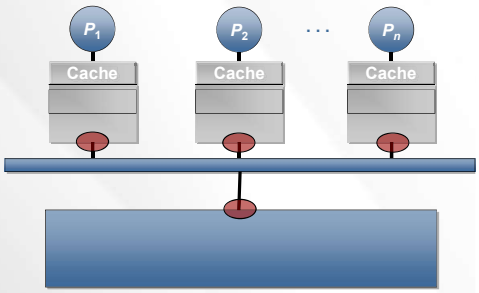
Snoop-Based Coherence on a Bus

- *Basic Idea*
  - Assign a snooper to each processor so that all bus transactions are visible to all processors ("snooping").
  - Processors (via cache controllers) change line states on relevant events.
- *Implementing a Protocol*
  - Each **cache controller** reacts to processor and bus events:
    - Takes actions when necessary
      - Updates state, responds with data, generates new bus transactions
  - The **memory controller** also snoops bus transactions and returns data only when needed
  - Granularity of coherence is typically cache line/block
    - Same granularity as in transfer to/from cache

Coherence with Write-Through Caches

```
sum = 0;
begin parallel
for (i=0; i<2; i++) {
  lock(id, myLock);
  sum = sum + a[i];
  unlock(id, myLock);
end parallel
Print sum;

Suppose a[0] = 3 and a[1] = 7
```

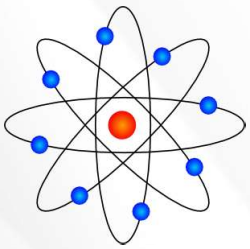
= Snooper



– What happens when we snoop a write?
  - Write-update protocol: write is immediately propagated ***or***
  - Write-invalidation protocol: causes miss on later access, and memory up-to-date via write-through

Snooper Assumptions

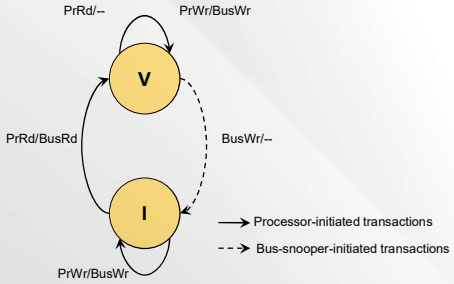
- Atomic bus
- Writes occur in program order



Transactions

- To show what's going on, we will use diagrams involving—
  – Processor transactions
    - PrRd
    - PrWr
  – Snooped bus transactions
    - BusRd
    - BusWr

Write-Through State-Transition Diagram



write-through
no-write-allocate
write invalidate

How does this protocol guarantee write propagation?

How does it guarantee write serialization?

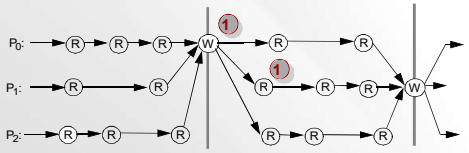
- Key: A write invalidates all other caches
- Therefore, we have:
  – Modified line: exists as V in only 1 cache
  – Clean line: exists as V in at least 1 cache
  – Invalid state represents invalidated line or not present in the cache

Is It Coherent?

- Write propagation:
  – through invalidation
  – then a cache miss, loading a new value
- Write serialization: Assume—
  – atomic bus
  – invalidation happens instantaneously
  – writes serialized by order in which they appear on bus (*bus order*)
    - So are invalidations
- Do reads see the latest writes?
  – Read misses generate bus transactions, so will get the last write
  – Read hits: do not appear on bus, but are preceded by
    - most recent write by this processor (self), or
    - most recent read miss by this processor
  – Thus, reads hits see latest written values (according to bus order)

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.
1. Read follows write W if read generates bus transaction that follows W's xaction.

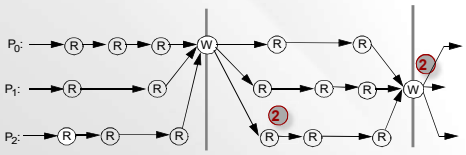


- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  –any order among reads between writes is fine, as long as in program order

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

1. Read follows write W if read generates bus transaction that follows W's xaction.
2. Write follows read or write M if M generates bus transaction and the transaction for the write follows that for M.

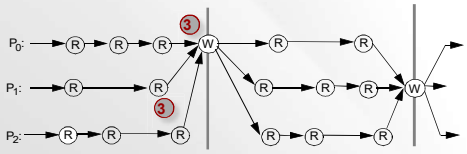


- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order

Determining Orders More Generally

A memory operation M2 follows a memory operation M1 if the operations are issued by the same processor and M2 follows M1 in program order.

1. Read follows write W if read generates bus transaction that follows W's xaction.
2. Write follows read or write M if M generates bus transaction and the transaction for the write follows that for M.
3. Write follows read if read does not generate a bus transaction and is not already separated from the write by another bus transaction.



- Writes establish a partial order
- Doesn't constrain ordering of reads, though bus will order read misses too
  - any order among reads between writes is fine, as long as in program order

Problem with Write-Through

- Write-through can guarantee coherence, but needs a lot of bandwidth.
  - Every write goes to the shared bus and memory
  - Example:
    - 200MHz, 1-CPI processor, and 15% instrs. are 8-byte stores
    - Each processor generates 30M stores, or 240MB data, per second
    - How many processors could a 1GB/s bus support without saturating?
  - Thus, unpopular for SMPs
- Write-back caches
  - Write hits do not go to the bus ⇒ reduce most write bus transactions
  - But now how do we ensure write propagation and serialization?

Lecture 7 Outline

- Bus-based coherence
- **Invalidation vs. update coherence protocols**
- Memory consistency
  - Sequential consistency

Dealing with "Dirty" Lines

- What does it mean to say a cache line is "dirty"?
  - That at least one of its words has been changed since it was brought in from main memory.
- Dirty in a uniprocessor vs. a multiprocessor
  - Uniprocessor:
    - Only need to keep track of *whether* a line has been modified.
  - Multiprocessor:
    - Keep track of *whether* line is modified.
    - Keep track of which cache *owns* the line.
  - Thus, a cache line must know whether it is—
    - **Exclusive**: "I'm the only one that has it, other than possibly main memory."
    - The **Owner**: "I'm responsible for supplying the block upon a request for it."

Invalidation vs. Update Protocols

- Question: What happens to a line if *another* processor changes one of its words?
  - It can be *invalidated.*
  - It can be *updated.*

Invalidation-Based Protocols

- Idea: When I write the block, invalidate everybody else ⇒ I get exclusive state.
- "Exclusive" means …
  - Can modify without notifying anyone else (i.e., without a bus transaction)
- But, before writing to it,
  - Must first get block in exclusive state
  - Even if block is already in state V, a bus transaction (Read Exclusive = RdX) is needed to invalidate others.
- What happens when a block is ejected from the cache?
  - if the block is not dirty?
  - if the block *is* dirty?

UPDATE! -Based Protocols

- Idea: If this block is written, send the new word to all other caches.
  - New bus transaction: Update
- Compared to invalidate, what are advs. and disads.?
- Advantages
  - Other processors don't miss on next access
  - Saves refetch: In invalidation protocols, they would miss & bus transaction.
  - Saves bandwidth: A single bus transaction updates several caches
- Disadvantages
  - Multiple writes by same processor cause multiple update transactions
    - In invalidation, first write gets exclusive ownership, other writes local

Invalidate versus Update

- Is a block written by one processor read by other processors before it is rewritten?
- Invalidation:
  - Yes → Readers will take a miss.
  - No → Multiple writes can occur without additional traffic.
    - Copies that won't be used again get cleared out.
- Update:
  - Yes → Readers will not miss if they had a copy previously
    - A single bus transaction will update all copies
  - No → Multiple useless updates, even to dead copies
- Invalidation protocols are much more popular.
  - Some systems provide both, or even hybrid

Lecture 7 Outline

- Bus-based coherence
- Invalidation vs. update coherence protocols
- **Memory consistency**
  - Sequential consistency

Let's Switch Gears to Memory Consistency

Coherence: Writes to a single location are visible to all in the same order

Consistency: Writes to multiple locations are visible to all in the same order

- Recall Peterson's algorithm (`turn= …; interested[process]=…`)
- When "multiple" means "all", we have sequential consistency (SC)

P_1	P_2
/*Assume initial values of A and flag are 0*/	
`A = 1;`	`while (flag == 0);` /*spin idly*/
`flag = 1;`	`print A;`

- Sequential consistency (SC) corresponds to our intuition.
- Other memory consistency models do not obey our intuition!
- Coherence doesn't help; it pertains only to a single location

Another Example of Ordering

P_1	P_2
/*Assume initial values of `A` and `B` are 0 */	
(1a) `A = 1;`	(2a) `print B;`
(1b) `B = 2;`	(2b) `print A;`

- What do you think the results should be? You may think:
  - 1a, 1b, 2a, 2b ⇒ `{A=1, B=2}`
  - 1a, 2a, 2b, 1b ⇒ `{A=1, B=0}`
  - 2a, 2b, 1a, 1b ⇒ `{A=0, B=0}`

  } programmers' intuition: sequential consistency
  - Is `{A=0, B=2}` possible? Yes, suppose P2 sees: 1b, 2a, 2b, 1a e.g. evil compiler, evil interconnection.
- Whatever our intuition is, we need
  - an **ordering model** for clear semantics across different locations
  - as well as **cache coherence**!

  so programmers can reason about what results are possible.

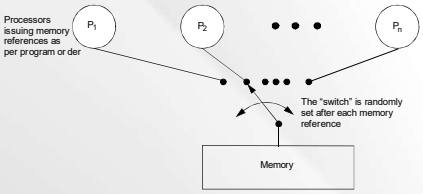
A Memory-Consistency Model …

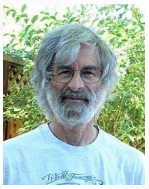
- Is a contract between programmer and system
  - Necessary to reason about correctness of shared-memory programs
- Specifies constraints on the order in which memory operations (from any process) can *appear to execute* with respect to one another
  - Given a load, constrains the possible values returned by it
- Implications for programmers
  - Restricts algorithms that can be used
  - e.g., Peterson's algorithm, home-brew synchronization will be incorrect in machines that do not guarantee SC
- Implications for compiler writers and computer architects
  - Determines how much accesses can be reordered.

Lecture 7 Outline

- Bus-based coherence
- Memory consistency
  - **Sequential consistency**
- Invalidation vs. update coherence protocols

Sequential Consistency





"A multiprocessor is sequentially consistent if 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." [Lamport, 1979]

- (as if there were no caches, and a single memory)
- Total order achieved by *interleaving* accesses from different processes
- Maintains *program order*, and memory operations, from all processes, appear to [issue, execute, complete] atomically w.r.t. others

What Really Is Program Order?

- Intuitively, the order in which operations appear in source code
- *Thus, we assume order as seen by programmer,*
  - *the compiler is prohibited from reordering memory accesses to shared variables.*
- Note that this is one reason parallel programs are less efficient than serial programs.

What Reordering Is Safe in SC?

What matters is the order in which code *appears to execute*, not the order in which it actually *executes*.

P_1	P_2
/*Assume initial values of `A` and `B` are `0` */	
(1a) `A = 1;`	(2a) `print B;`
(1b) `B = 2;`	(2b) `print A;`

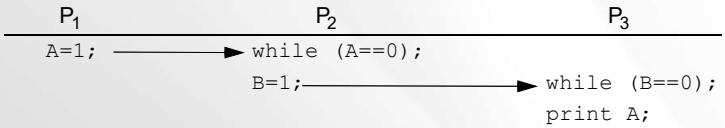
- Possible outcomes for `(A,B)`: `(0,0)`, `(1,0)`, `(1,2)`; impossible under SC: `(0,2)`
- *Proof:* By program order we know 1a → 1b and 2a → 2b
  - `A = 0` implies 2b → 1a, which implies 2a → 1b
  - `B = 2` implies 1b → 2a, which leads to a contradiction
- BUT, actual execution 1b →1a → 2b → 2a is SC, despite not being in program order
  - It produces the same result as 1a → 1b → 2a → 2b.
  - Actual execution 1b → 2a → 2b → 1a is not SC, as shown above
  - Thus, some reordering is possible, but difficult to reason that it ensures SC

Conditions for SC

- Two kinds of requirements
  - **Program order**
    - Memory operations issued by a process must appear to become visible (to others and itself) in program order.
  - **Global order**
    - Atomicity: One memory operation should appear to complete with respect to all processes before the next one is issued.
    - Global order: The same order of operations is seen by all processes.
- Tricky part: how to make writes atomic?
  - → Necessary to detect write completion
  - Read completion is easy: a read completes when the data returns
- Who should enforce SC?
  - Compiler should not change program order
  - Hardware should ensure program order and atomicity

Write Atomicity

- *Write Atomicity* ensures same write ordering is seen by all procs.
  - In effect, extends write serialization to writes from multiple processes



- Under SC, transitivity implies that **A** should print as 1. Without SC, why might it not?

Is the Write-Through Example SC?

- Assume no write buffers, or load-store bypassing
- Yes, it is SC, because of the atomic bus:
  - Any write and read misses (to *all locations)* are serialized by the bus into bus order.
  - If a read obtains value of write W, W is guaranteed to have completed since it caused a bus transaction
  - When write W is performed *with respect to any processor*, all previous writes in bus order have completed

Summary

- One solution for small-scale multiprocessors is a shared bus.
- State-transition diagrams can be used to show how a cache-coherence *protocol* operates.
  - The simplest protocol is write-through, but it has performance problems.
- Sequential consistency guarantees that memory operations are seen in order throughout the system.
  - It is fairly easy to show whether a result is or is not sequentially consistent.
- The two main types of coherence protocols are invalidate and update.
  - Invalidate usually works better, because it frees up cache lines more quickly.

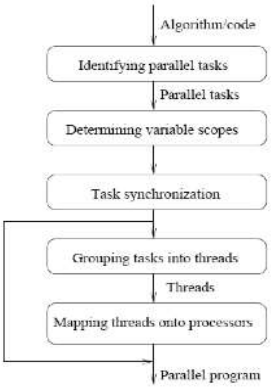
Shared-Memory Parallel Programming

[§3.1] Solihin identifies several steps in parallel programming.

The first step is identifying parallel tasks. Can you give an example?

The next step is identifying variable scopes. What does this mean?

The next step is grouping tasks into threads. What factors need to be taken into account to do this?



Then threads must be synchronized. How did we see this done in the three parallel-programming models?

What considerations are important in mapping threads to processors?

Solihin says that there are three levels of parallelism:

- program level
- algorithm level
- code level

Identifying loop-level parallelism

[§3.2] *Goal:* given a code, without knowledge of the algorithm, find parallel tasks.

Focus on loop-dependence analysis.

Notations:

- *S* is a statement in the source code
- *S*[*i*, *j*, …] denotes a statement in the loop iteration [*i*, *j*, …]
- "*S*1 then *S*2" means that *S*1 *happens before* *S*2
- If *S*1 then *S*2:

  *S*1 →T *S*2 denotes true dependence, i.e., *S*1 writes to a location that is read by *S*2

  *S*1 →*A* *S*2 denotes anti-dependence, i.e., *S*1 reads a location written by *S*2

  *S*1 →O *S*2 denotes output dependence, i.e., *S*1 writes to the same location written by *S*2

Example:

S1: *x* = 2;
S2: *y* = x;
S3: *y* = x + 4;
S4: *x* = *y*;

Exercise: Identify the dependences in the above code.

Loop-independent vs. loop-carried dependences

[§3.2] Loop-carried dependence: dependence exists across iterations; i.e., if the loop is removed, the dependence *no longer exists*.

Loop-independent dependence: dependence exists within an iteration; i.e., if the loop is removed, the dependence still exists.

Example:


```
for (i=1; i<n; i++) {
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;

for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S4: a[i][j] = a[i-1][j] + 1;
```

S1[i] →T **S1[i+1]**: loop-carried

S1[i] →T **S2[i]**: loop-independent

S3[i,j] →T **S3[i,j+1]**:

- loop-carried on **for** j loop
- no loop-carried dependence in **for** i loop

S4[i,j] →T **S4[i+1,j]**:

- no loop-carried dependence in **for** j loop
- loop-carried on **for** i loop

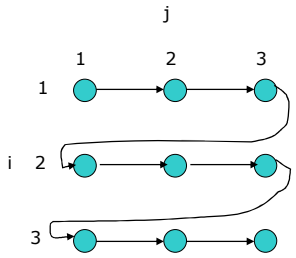
Iteration-space Traversal Graph (ITG)

[§3.2.1] The ITG shows graphically the order of traversal in the iteration space. This is sometimes called the *happens-before relationship*. In an ITG,

- A *node* represents a point in the iteration space
- A *directed edge* indicates the next point that will be encountered after the current point is traversed

Example:

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

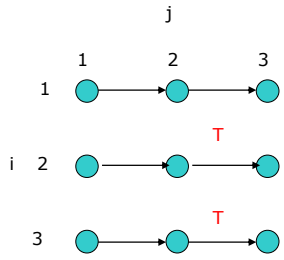



Loop-carried Dependence Graph (LDG)

- LDG shows the true/anti/output dependence relationship graphically.
- A *node* is a point in the iteration space.
- A *directed edge* represents the dependence.

Example:

```
for (i=1; i<4; i++)
  for (j=1; j<4; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```



Another example:

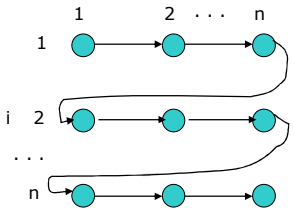
```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];

for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

- Draw the ITG
- List all the dependence relationships

Note that there are two "loop nests" in the code.

- The first involves S1.
- The other involves S2 and S3.

What do we know about the ITG for these nested loops?



Dependence relationships for Loop Nest 1

- True dependences:
  - **S1[i,j]** →T **S1[i,j+1]**
  - **S1[i,j]** →T **S1[i+1,j]**
- Output dependences:
  - None
- Anti-dependences:
  - **S1[i,j]** →A **S1[i+1,j]**
  - **S1[i,j]** →A **S1[i,j+1]**

Exercise: Suppose we dropped off the first half of S1, so we had

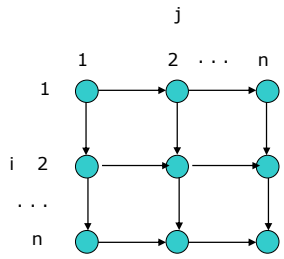
```
S1: a[i][j] = a[i-1][j] + a[i+1][j];
```

or the last half, so we had

```
S1: a[i][j] = a[i][j-1] + a[i][j+1];
```

Which of the dependences would still exist?

Draw the LDG for Loop Nest 1.

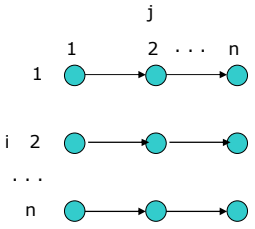


Note: each edge represents both true and anti-dependences

Dependence relationships for Loop Nest 2

- True dependences:
  - **S2[i,j]** →T **S3[i,j+1]**
- Output dependences:
  - None
- Anti-dependences:
  - **S2[i,j]** →A **S3[i,j]** (loop-independent dependence)

Draw the LDG for Loop Nest 2.



Note: each edge represents only true dependences

Why are there no vertical edges in this graph? Answer here.

Why is the anti-dependence not shown on the graph?

Exercise: Consider this code sequence.

```
for (i = 3; i < n; i++) {
     for (j = 0; j < n - 3; j++) {
          S1: A[i][j] = A[i - 3][j] + A[i][j + 3];
          S2: B[i][j] = A[i][j] / 2;
     }
}
```

List the dependences, and say whether they are loop independent or loop carried. Then draw the ITG and LDG (you don't need to submit these).

Finding parallel tasks across iterations

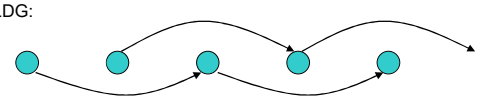
[§3.3.1] Analyze loop-carried dependences:

- Dependences must be enforced (especially true dependences; other dependences can be removed by privatization)
- There are opportunities for parallelism when some dependences are not present.

Example 1

```
for (i=2; i<=n; i++)
  S: a[i] = a[i-2];
```

LDG:



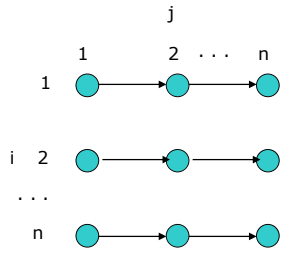
We can divide the loop into two parallel tasks (one with odd iterations and another with even iterations):

```
for (i=2; i<=n; i+=2)
  S: a[i] = a[i-2];
for (i=3; i<=n; i+=2)
  S: a[i] = a[i-2];
```


Example 2

```
for (i=0; i<n; i++)
  for (j=0; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;
```

LDG

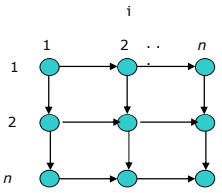


How many parallel tasks are there here?

Example 3

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] + a[i-1][j] + a[i+1][j];
```

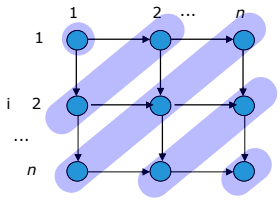
LDG



Note: each edge represents both true, and anti-dependences

Identify which nodes are not dependent on each other

In each anti-diagonal, the nodes are independent of each other



Note: each edge represents both true, and anti-dependences

We need to rewrite the code to iterate over anti-diagonals:

Calculate number of anti-diagonals
for each anti-diagonal do
  Calculate the number of points in the current anti-diagonal
  for_all points in the current anti-diagonal do
    Compute the value of the current point in the matrix

Parallelize the loops highlighted above.

```
for (i=1; i <= 2*n-1; i++) {// 2n-1 anti-diagonals
  if (i <= n) {
    points = i;           // number of points in anti-diag
    row = i;              // first pt (row,col) in anti-diag
    col = 1;              // note that row+col = i+1 always
  }
  else {
    points = 2*n - i;
    row = n;
    col = i-n+1;          // note that row+col = i+1 always
  }
  for_all (k=1; k <= points; k++) {
    a[row][col] = …       // update a[row][col]
    row--; col++;
  }
}
```


DOACROSS Parallelism

[§3.3.2] Suppose we have this code:

Can we execute anything in parallel?

```
for (i=1; i<=N; i++) {
  S: a[i] = a[i-1] + b[i] * c[i];
}
```

Well, we can't run the iterations of the `for` loop in parallel, because …

`S[i]` →T `S[i+1]` (There is a loop-carried dependence.)

But, notice that the `b[i]*c[i]` part has no loop-carried dependence.

This suggests breaking up the loop into two:

```
for (i=1; i<=N; i++) {
  S1: temp[i] = b[i] * c[i];
}
for (i=1; i<=N; i++) {
  S2: a[i] = a[i-1] + temp[i];
}
```

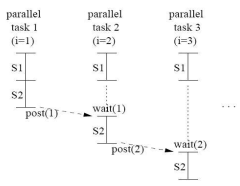
The first loop is ||izable. The second is not.

Execution time: $N\times(T_{S1}+T_{S2})$

What is a disadvantage of this approach?

Here's how to solve this problem:

```
post(0);
for (i=1; i<=N; i++) {
  S1: temp = b[i] * c[i];
  wait(i-1);
  S2: a[i] = a[i-1] + temp;
  post(i);
}
```



What is the execution time now? $T_{S1} + N\times T_{S2}$

Function parallelism

- [§3.3.3] Identify dependences in a loop body.
- If there are independent statements, can split/distribute the loops.

Example:

```
for (i=0; i<n; i++) {
  S1: a[i] = b[i+1] * a[i-1];
  S2: b[i] = b[i] * coef;
  S3: c[i] = 0.5 * (c[i] + a[i]);
  S4: d[i] = d[i-1] * d[i];
}
```

Loop-carried dependences:

Loop-indep. dependences:

Note that S4 has no dependences with other statements

After loop distribution:

```
for (i=0; i<n; i++) {
  S1: a[i] = b[i+1] * a[i-1];
  S2: b[i] = b[i] * coef;
  S3: c[i] = 0.5 * (c[i] + a[i]);
}

for (i=0; i<n; i++) {
  S4: d[i] = d[i-1] * d[i];
}
```

Each loop is a parallel task.

This is called *function parallelism.*

It can be distinguished from *data parallelism*, which we saw in DOALL and DOACROSS.

Further transformations can be performed (see p. 64 of text).

"`S1[i]` →A `S2[i+1]`" implies that S2 at iteration *i*+1 must be executed after S1 at iteration *i.* Hence, the dependence is not violated if all S2s execute after all S1s.

Characteristics of function parallelism:

-
-

Can use function parallelism along with data parallelism when data parallelism is limited.

DOPIPE Parallelism

[§3.3.4] Another strategy for loop-carried dependences is pipelining the statements in the loop.

Consider this situation:

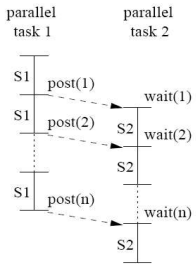
```
for (i=2; i<=N; i++) {
  S1: a[i] = a[i-1] + b[i];
  S2: c[i] = c[i] + a[i];
}
```

Loop-carried dependences:

Loop-indep. dependences:

To parallelize, we just need to make sure the two statements are executed in sync:

```
for (i=2; i<=N; i++) {
  a[i] = a[i-1] + b[i];
  post(i);
}

for (i=2; i<=N; i++) {
  wait(i);
  c[i] = c[i] + a[i];
}
```



Question: What's the difference between DOACROSS and DOPIPE?

Determining variable scope

[§3.6] This step is specific to the shared-memory programming model. For each variable, we need to decide how it is used. There are three possibilities:

- Read-only: *variable is only read by multiple tasks*
- R/W non-conflicting: *variable is read, written, or both by only one task*
- R/W conflicting: *variable is written by one task and may be read by another*

Intuitively, why are these cases different?

Example 1

Let's assume each iteration of the **for** *i* loop is a parallel task.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

Fill in the tableaus here.

Read-only	R/W non-conflicting	R/W conflicting

Now, let's assume that each **for** *j* iteration is a separate task.

Read-only	R/W non-conflicting	R/W conflicting

Do these two decompositions create the same number of tasks?

Example 2

Let's assume that each **for** *j* iteration is a separate task.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j] * d[i][j];
    S3: e[i][j] = a[i][j];
  }
```

Read-only	R/W non-conflicting	R/W conflicting

Exercise: Suppose each **for** *i* iteration were a separate task …

Read-only	R/W non-conflicting	R/W conflicting