Lecture 11

Shared memory:
Architecture and programming
Announcements

- Datastar accounts
- UPC lecture next week
Shared memory architecture

- Every processor has direct access to all of memory
- The address space is global to all processors
- Hardware automatically performs the global to local mapping using virtual to physical address translation
Two kinds of shared memory architectures

- Distinguished by whether or not memory access time is uniform

  **UMA**: Uniform Memory Access time
  - In the absence of contention, all processors see the same access time to memory (approximates a PRAM)
  - Also called a *Symmetric Multiprocessor (SMP)*
  - Usually bus based: not a scalable solution

- **NUMA**: Non-Uniform Memory Access time
  - Memory access time depends on distance to memory
  - Also called *Distributed Shared Memory (DSM)*
  - Elaborate interconnect structure
Cache Coherence

- A central design issue in shared memory architectures
- Processors may read and write the same cached memory location
- If one processor writes to the location, all others must eventually see the write

\[
X := 1
\]

Memory
Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system doesn’t have a coherent value for X
Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write

```plaintext
X:=2
Memory
P2
```

```
P1
X:=2
```

```
P1
X:=2
```

```
P2
X:=2
```
SMP architectures

• Employ a *snooping protocol* to ensure coherence

• Processors listen to bus activity

---

*Parallel Computer Architecture, Culler, Singh, Gupta*
Memory consistency and correctness

- Cache coherence tells us that memory will *eventually* be consistent
- The memory consistency policy tells us *when* this will happen
- Even if memory is consistent, changes don’t propagate instantaneously
- These give rise to correctness issues involving program behavior
Memory consistency

• A memory system is consistent if the following 3 conditions hold
  – Program order
  – Definition of a coherent view of memory
  – Serialization of writes
Program order

- If a processor writes and then reads the same location $X$, and there are no other intervening writes by other processors to $X$, then the read will always return the value previously written.
Definition of a coherent view of memory

- If a processor P reads from location X that was previously written by a processor Q, then the read will return the value previously written, if a sufficient amount of time has elapsed between the read and the write.
Serialization of writes

- If two processors write to the same location $X$, then other processors reading $X$ will observe the same the sequence of values in the order written.
- If 10 and then 20 is written into $X$, then no processor can read 20 and then 10.
Memory consistency model

• The memory consistency model determines when a written value will be seen by a reader

• **Sequential Consistency** maintains a linear execution on a parallel architecture that is consistent with the sequential execution of some interleaved arrangement of the separate concurrent instruction streams

• Expensive to implement

• **Relaxed consistency**
  – Enforce consistency only at well defined times
  – Useful in handling false sharing
Relaxed memory consistency

• We enforce consistency only at well defined times
• Useful in handling false sharing
False sharing

- Consider two processors that write to different locations mapping to different parts of the same cache line.
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory
False sharing

• P1’s write updates main memory
• Snooping protocol invalidates the corresponding block in P0’s cache
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Eliminating false sharing

• Cleanly separate locations updated by different processors
  – Manually assign scalars to a preallocated region of memory using pointers
  – With a block partitioned array, we want partition boundaries to coincide with a cache line boundary

• Compilers can perform some of these optimizations
False sharing in higher dimension arrays

- Large memory access strides, conflict misses
- Compare with distributed memory solution

Parallel Computer Architecture, Culler, Singh, & Gupta
NUMA Architectures

• Unlike UMAs, do not rely on a broadcasting
• Point-to-point messages manage coherence
• A directory keeps track of sharers, one for each block of memory
• Stanford Dash; SGI Origin 2000
Some basic terminology

- Every block of memory has an associated **home**: the specific processor that physically holds the associated portion of the global address space.
- Every block also has an **owner**: the processor whose memory contains the actual value of the data.
- Normally these are the same.
- But they can be different if a processor other than the home’s processor writes a block.
Inside a directory

- Each processor has a 1-bit “sharer” entry in the directory
- There is also a dirty bit and a PID identifying the owner in the case of a dirt block
Operation of a directory

- Assume a 4 processor system (only P0 & P1 shown)
- A is a location with home P1
- Initial directory entry for block containing A is empty
Operation of a directory

• P0 loads A
• Set directory entry for A (on P1) to indicate that P0 is a sharer
Operation of a directory

- P2, P3 load A (not shown)
- Set directory entry for A (on P1) to indicate that P0 is a sharer
Acquiring ownership of a block

- P0 writes A
- P0 becomes the owner of A
Acquiring ownership of a block

- P0 becomes the owner of A
- P1’s directory entry for A is set to Dirty
- Outstanding sharers are invalidated
- Access to line is blocked until all invalidations are acknowledged

Diagram:

- P0
- P1
- Mem
- P2
- P3
- $
Forwarding

Store A, #1  
(home & owner)

Store A, #2

P1

P2

P0

Load A

A ← dirty

Directory

1 1 D P0
Performance issues with DSMs

- When we allocate a block of memory, which processor(s) is (are) the owner(s)?
- Page allocation policies
  - First touch
  - Round robin
- We can control memory locality with the same kind of data layouts that we use with message passing
Shared memory programming

• Shared memory introduces a new set of programming issues
  – Synchronization
  – Cache management
• Certain applications can be automatically parallelized with a compiler
• Others cannot
• We’ll start with the most primitive programming model: threads
• We’ll then describe a high level model which is becoming popular: openmp
Shared memory programming model

- A collection of concurrent instruction streams, called *threads*, that share memory
- Each thread has a unique thread ID: ~ MPI rank
- We get a new kind of storage class: shared data
- Synchronization is needed when updating shared state
  - mutual exclusion
  - barriers
Why threads?

- Processes are “heavy weight” objects scheduled by the OS
  - Protected address space, open files, and other state
- A thread, AKA a lightweight process (LWP) is sometimes more appropriate
  - Threads share the address space and open files of the parent, but have their own stack
  - Reduced management overheads
  - Kernel scheduler multiplexes threads
More on Threads

- A thread is similar to a procedure call with notable differences
  - A procedure call is “synchronous;” a return indicates completion
  - With a thread we get an independently schedulable unit, since a spawned thread executes asynchronously until it completes
- With a procedure call and a thread we get a local stack, and we share global storage with the caller
- A common interface is the POSIX Threads “standard” (pthreads): IEEE POSIX 1003.1c-1995
  - But there are non-standard features some of which are outside the scope of the standard!
- Another approach is to use program annotations via openMP
Programming model

- We start with a single root thread
- We employ fork-join parallelism to create new concurrently executing threads
- The root thread may execute concurrently with the spawned threads
- But at the end it waits for the spawned threads to complete
- Whether or not threads execute on different processors, and at the same time isn’t said
- The scheduler might interleave many threads on a single CPU or some other subset of the hardware
- Scheduling behavior is usually specified separately