Lecture 13

Shared memory:
Architecture and programming
Announcements

• Special guest lecture on Parallel Programming Language Uniform Parallel C
• Thursday 11/2, 2:00 to 3:20 PM
• EBU3B 1202
• See
  www.cse.ucsd.edu/classes/fa06/cse260/Lectures/Lec13
Sidebar on Memory hierarchies
The Advance of Technology

- Until about 2002, processor speeds doubled every ~18 months
- Today’s Laptop was Yesterday’s Supercomputer
- Improved at a much faster rate than DRAM access times
  - 52% per year vs. 7% per year
    (memory bandwidth 20% per year)
- Memory bandwidth is decreasing relative to processing speed and this has lead to a phenomenon called the “memory wall”
- Multi-core processors
Processing speeds

Computer Architecture: A Quantitative Approach
Fourth Edition, Hennessy and Patterson, Morgan Kaufman
(2006)
The processor-memory gap
An important universal: the locality principle

- Programs generally exhibit two forms of locality when accessing memory:
  - Temporal locality: we are more likely to reference more recently accessed memory locations than less recently accessed ones
  - Spatial locality: we are more likely to reference nearby addresses than far away ones

- Memory hierarchies rely on locality to improve memory access times through re-use

- Underlying principles for improving cache re-use also apply to local and remote data in a distributed memory computer

- Locality often involves loops in computationally intensive problems
Memory and Storage Technology

• Large memories are slow and cheap
  – Disk
  – DRAM (Dynamic random access memory)
  – Thousands of times faster than disk

• Small memories are fast but expensive
  – SRAM (Static random access memory)
  – 5 or 10+ times faster than DRAMs
Memory hierarchy pyramid

Disk

10s to 100s GB

106 CP

Dram

100 to 500 CP

100s of MB to a few GB

16 to 64 KB

256KB to 4 MB

L1$

1 CP (10 to 100 B)

L2$

10 CP (10 to 100 B)

CPU

Faster, smaller, more expensive

10 to 100 B
Why do we have memory hierarchies?

• Recall
  – Processor speeds are improving at a much faster rate then DRAM access times
  – 52% per year v. 7% per year

• Faster (and more expensive) memories are smaller than slower (and less costly ones)

• We may use a hierarchy to effectively reduce the cost of accessing memory: cache, TLB, etc.
Idea

• Put recently accessed data in a small, fast memory
• If this memory is 10 times faster (access time) than main memory …
• And if we find what we are looking for 90% of the time…
• Access time = 0.90 \times 1 + (1-0.9) \times 10
  \quad = \quad 1.9
• Memory is now 5 times faster
Sidebar

• If cache memory access time is 10 times faster than main memory …

• $T_{cache} = \frac{T_{main}}{10}$

• And if we find what we are looking for $f \times 100\%$ of the time…

• Access time $= f \times T_{cache} + (1-f) \times T_{main}$
  $= f \times T_{main} /10 + (1-f) \times T_{main}$
  $= (1-(9f/10)) \times T_{main}$

• We are now $1/(1-(9f/10))$ times faster

• To simplify, we use $T_{cache} = 1$, $T_{main} = 10$
Some terminology

- If we find what we are looking for, we have a cache hit
- If we don’t find what we are looking for, we have a cache miss
- The time it takes to access cache on a hit is the hit time
- The time it takes to obtain the data we need from the next higher level of the hierarchy is the miss penalty
More terminology

• The percentage of time that we hit in the cache is called the **hit ratio**
• The **miss ratio** is \(1 - \text{hit ratio}\)
• In today’s architectures we have **on-chip cache**(s) and **external cache**(s)
• Internal cache often contains a separate **data cache** and **instruction cache**
• External cache is often **unified**
• The unit of transfer is called the **block**
Valkyrie’s memory hierarchy

• There are two levels of cache: L1 and L2
• If what we are looking for isn’t in L1…
• We look in L2
• If what we are looking for isn’t in L2..
• We look to main memory
• Which is larger?
  – Size of L1: 32 KB
  – Size of L2: 256 KB
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 \textit{eventually} see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write
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 [Lamport]
  – Expensive to implement

• Relaxed consistency
  – 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 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

Parallel Computer Architecture, Culler, Singh, & Gupta
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
Forwarding

Store A, #1  
(home & owner)

P1

P2

Directory

Load A

Store A, #2

P0

1

A ← dirty

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 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
- A thread is similar to a procedure call with notable differences
  - A procedure call is “synchronous:” a return indicates completion
  - A spawned thread executes asynchronously until it completes
  - Both share global storage with caller
  - Synchronization is needed when updating shared state

```
Private

Shared
```
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
Threads in practice

- 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, or 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
OpenMP programming

• Simpler interface than explicit threads
• Parallelization handled via annotations
• See http://www.openmp.org
• Parallel loop:

```c
#pragma omp parallel private(i) shared(n)
{
#pragma omp for
for(i=0; i < n; i++)
    work(i);
}
```
Parallel Sections

```c
#pragma omp parallel // Begin a parallel construct
{ // form a team
  // Each team member executes the same code
  #pragma omp sections // Begin worksharing
  { //
    #pragma omp section // One unit of work
    {x = x + 1;}
    #pragma omp section // Another unit of work
    {x = x + 1;}
  } // Wait until both units of work complete
} // End of Parallel Construct; disband team
// and continue serial execution
```
Race conditions

• Consider the statement, assuming \( x == 0 \)
  \[
x = x + 1;
\]

• Generated code
  – \( r1 \leftarrow (x) \)
  – \( r1 \leftarrow r1 + #1 \)
  – \( r1 \rightarrow (x) \)

• Possible interleaving with two threads

  P1
  \[
  r1 \leftarrow x
  r1 \leftarrow r1 + #1
  x \leftarrow r1
  \]

  P2
  \[
  r1 \leftarrow x
  r1 \leftarrow r1 + #1
  x \leftarrow r1
  \]

  P1 writes its R1

  r1(P1) gets 0
  r2(P2) also gets 0
  r1(P1) set to 1
  r1(P1) set to 1
  P1 writes its R1
  P2 writes its R1
Race conditions

- A *Race* condition arises within an application when the timing of accesses to shared memory can affect the outcome.
- We say we have a *non-deterministic* computation.
- Sometimes we can use non-determinism to advantage, but usually we want to avoid it.
- For the same input, we want to obtain the same results from operations that do not have side effects (like I/O and random number generators).
- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness.
- We need to take steps to avoid race conditions through appropriate program synchronization.
Mutual exclusion

• Each process samples and increments the shared variable $x$
• The code performing the operation is a critical section
• Only one thread at a time may access this code
• We use *mutual exclusion* to implement the critical section
• A critical section is non-parallelizing computation.. sensible guidelines?
Critical Sections

#pragma omp parallel // Begin a parallel construct
{ // form a team
  // Each team member executes the same code
  #pragma omp sections    // Begin worksharing
  { //
    #pragma omp critical    // Critical section
    {x = x + 1}
    #pragma omp critical    // Another critical section
    {x = x + 1}
  } // Wait until both units of work complete
  #pragma omp barrier // Wait for all members to arrive
} // End of Parallel Construct; disband team
// and continue serial execution
How does mutual exclusion work?

- A simple solution is to use a mutex variable
- E.g. provided by pthreads

```c
Mutex mtx;
mtx.lock();
    CRITICAL SECTION
mtx.unlock();
```
Implementation issues

• Hardware support
  – Test and set: atomically test a memory location and then set it
  – Cache coherence protocol provides synchronization

• Scheduling issues
  – Busy waiting or spinning
  – Yield process
  – Pre-emption by scheduler
Practical issues

- Thread creation costs 10s of μsecs
- Moving data in shared memory is cheaper than passing a message through shared memory

http://www.llnl.gov/computing/tutorials/pthreads

<table>
<thead>
<tr>
<th></th>
<th>create</th>
<th>MPI Shared Mem BW</th>
<th>Mem to CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel 2.4 GHz Xeon</td>
<td>34 μs</td>
<td>0.3 GB/s</td>
<td>4.3 GB/s</td>
</tr>
<tr>
<td>Intel 1.4 GHz Itanium 2</td>
<td>42 μs</td>
<td>1.8</td>
<td>6.4</td>
</tr>
<tr>
<td>IBM 1.5 GHz POWER4</td>
<td>30 μs</td>
<td>2.1</td>
<td>11</td>
</tr>
</tbody>
</table>
Thread safety

• Some libraries are not thread safe
• Others have thread safe versions, e.g. MPI
• We know that messages are non-overtaking
  Send(x,1); Send(y,1);  PROCESS 0
  Recv(A,0); Recv(B,0);  PROCESS 1
  A ← x , B ← y
• What if each Send/Recv is executed by a different thread?