Lecture 11

Shared memory:
Architecture and programming
NUMA Architectures

- The address space is global to all processors
- Point-to-point messages manage coherence
- A directory keeps track of sharers, one for each block of memory
- Stanford Dash; SGI Origin 2000
Cache Coherence

- If one processor writes to the location, all others must *eventually* see the write
- 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
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 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

• **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
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.
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

```
Mem

P0

P1

0 0 0 0 1 0 1 1

P2

P3
```

5/10/07 Scott B. Baden/CSE 262/Spring 2007
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)

Store A, #2

A ← dirty

Load A

Directory

1 1 D P0

P1

P2

P0
Forwarding

Store A, #1

 home & (P1)
  owner)
Forwarding

Store A, #1

\[ \downarrow \]

(P1) \[\text{---}\] (P2)

Store A, #2
Forwarding

Store A, #1

mark A as dirty

(P1) ← owner ← (P2)

Store A, #2
Forwarding

Store A, #1

⇓

mark A as dirty

(P1) ← owner

(P2)

Store A, #2

(P3) Load A
Forwarding

Store A, #1

mark A as dirty

(P1) owner

Store A, #2

(P3) Load A
Forwarding

Store A, #1

\[\text{mark A as dirty}\]

(P1)  \[\leftrightarrow\]  (P2) owner

Store A, #2

(P3) Load A
Performance issues

• False sharing
• Locality, locality, locality
  – Page placement
  – Page migration
  – Copying v. redistribution
  – Layout
Parallelization via the compiler

integer n, lda, ldr, lds, i, j, k
real*8 a(lda,n), r(ldr,n), s(lds,n)
 !$OMP PARALLEL DO private(j,k,i), shared(n,a,r,s),
 schedule(static)
    do j = 1, n
        do k = 1, n
            do i = 1, n
                a(i,j) = a(i,j) + r(i,k)*s(k,j)
            enddo
        enddo
    enddo
enddo
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
An example of false sharing

integer m, n, i, j
real    a(m,n), s(m)
!
! At each stage all CPUs update s(i)
!*omp parallel do private(i,j), shared(s,a)
    do i = 1, m
        s(i) = 0.0
        do j = 1, n
            s(i) = s(i) + a(i,j)
        enddo
    enddo
enddo
Removing false sharing

\begin{verbatim}
integer m, n, i, j 
real    a(m,n), s(32,m)

!$omp parallel do private(i,j), shared(s,a)
do i = 1, m 
  s(1,i) = 0.0 
do j = 1, n 
  s(1,i) = s(1,i) + a(i,j)
enddo 
enddo
\end{verbatim}
Motivating application

• Jacobi’s method for solving Poisson’s equation in two dimensions

for \( j=1 : N \)
for \( i=1 : M \)

\[
\text{unew}[i,j] = (\text{u}[i-1,j] + \text{u}[i+1,j] + \text{u}[l,j-1] + \text{u}[l,j+1])/4;
\]
False sharing in higher dimension arrays

- Large memory access strides, conflict misses
- Compare with distributed memory solution
Origin 2000 Interconnect

32 processor system

64 processor system
Bisection Bandwidth and Latency

\[ a(i) = b(i) + q \cdot c(i) \]
Locality
Poor Locality
Page Migration

\[ a(i) = b(i) + q \cdot c(i) \]
Cumulative effect of Page Migration
Migration Level
Data Affinity

real a(2*n+3,0:n)
!$sgi distribute a(block,cyclic(1))
do j=0,n; i=1,2*n+3
   a(i,j)=1000*i+j
!$omp parallel do private(i,j), shared(a)
!$sgi+   affinity(i) = data(a(2*i+3,j))
do i=1,n
   do j=1,n
      a(2*i+3,j)=a(2*i+3,j-1)
   enddo
endo
do enddo
end

Run iter i on CPU where A(2*i+3,j) is in local memory
Copying in lieu of data distribution

real*8 tmp(n1,n3):
!$omp parallel do private(j,i,k,sum,tmp), shared(a)
do j = 1, n2
  tmp(1:n1,1:n3) = a(1:n1,j,1:n3)
do i = 1, n1
  sum = 1.0d0/dasum(n3, tmp(i,1), n1)
call dscal(n3, sum, tmp(i,1), n1)
enddo
a(1:n1,j,1:n3) = tmp(1:n1,1:n3)
enddo