Lecture 16

NUMA Architecture and Programming
Announcements
NUMA Architectures

- The address space is global to all processors, but memory is physically distributed
- Point-to-point messages manage coherence
- A directory keeps track of sharers, one for each block of memory
- Stanford Dash; SGI UV, Altix, Origin 2000
- Node architectures of modern supercomputers

Source: en.wikipedia.org/wiki/Non-Uniform_Memory_Access

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.
- Initially home = owner, but this can change …
- … if a processor other than the home 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

- 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
Change of ownership

P0 stores into A (home & owner)
P1 stores into A (becomes owner)
P2 loads A

Store A, #y
A ← dirty

Store A, #x
(home & owner)

Directory

Load A
Forwarding

P0 stores into A (home & owner)
P1 stores into A (becomes owner)
P2 loads A
home (P0) forwards request to owner (P1)

Store A, #y
(home & owner)
Performance issues

• False sharing
• Locality, locality, locality
  ✷ Page placement
  ✷ Page migration
  ✷ Copying v. redistribution
  ✷ Layout
An Example

• Processor node on the Cray XE-6 supercomputer
Cray XE6 node

- 24 cores sharing 32GB main memory
- Packaged as 2 AMD Opteron 6172 processors “Magny-Cours”
- Each processor is a directly connected Multi-Chip Module: two hex-core dies living on the same socket
- Each die has 6MB of shared L3, 512KB L2/core, 64K L1/core
  - 1MB of L3 is used for cache coherence traffic
  - Direct access to 8GB main memory via 2 memory channels
  - 4 Hyper Transport (HT) links for communicating with other dies
- Asymmetric connections between dies and processors

www.nersc.gov/users/computational-systems/hopper/configuration/compute-nodes/
Processor memory interconnect (node)

http://www.hector.ac.uk/cse/documentation/Phase2b/#arch
Case study

- SGI Origin 2000
Origin 2000 Interconnect

32 processor system

64 processor system
Locality
Poor Locality
Quick primer on paging

• We group the physical and virtual address spaces into units called *pages*
• Pages are backed up on disk
• Virtual to physical mapping done by the Translation Lookaside Buffer (TLB), backs up page tables set up by the OS
• When we allocate a block of memory, we don’t need to allocate physical storage to pages; we do it on demand
Remote access latency

• When we allocate a block of memory, which processor(s) is (are) the owner(s)?
• We can control memory locality with the same kind of data layouts that we use with message passing
• Page allocation policies
  ◆ First touch
  ◆ Round robin
• Page placement and Page migration
• Copying v. redistribution
• Layout
Example

• Consider the following loop

\[
\text{for } r = 0 \text{ to nReps}
\]
\[
\text{for } i = 0 \text{ to } n-1
\]
\[
a[i] = b[i] + q*c[i]
\]
Page Migration

\[ a[i] = b[i] + q^*c[i] \]

- Parallel initialization
- Serial initialization

Parallel Initialization, First touch
(No migration)

Round robin initialization, w/ migration

Iteration Time

Iteration Number

fixed placement in one node

initial placement in one node

round-robin initial place

fixed round-robin placement

parallel initialization with round-robin placement

optimal placement or migration enabled

https://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/0650/bks/SGI_Developer/books/OrOn2_PfTune/sgi_html/ch08.html#id5224855

Migration eventually reaches the optimal time

Round robin initial, w/ migration

• Parallel initialization
• Serial initialization

Parallel initialization, first touch, migration

Parallel initialization, first touch (optimal)

One node initial placement, w/ migration
Cumulative effect of Page Migration
Migration Level

![Graph showing iteration time and migration levels over iterations.](image-url)
Bisection Bandwidth and Latency

\[ a(i) = b(i) + q \cdot c(i) \]
Coping with false sharing
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache.
An example of false sharing

```c
float a[m,n], s[m]
// Outer loop is in parallel
// Consider m=4, 128 byte cache line size
// Thread i updates element s[i]
#pragma omp parallel for private(i,j), shared(s,a)
    for i = 0, m-1
        s[i] = 0.0
        for j = 0, n-1
            s[i] += a[i,j]
        end for
    end for
```

Avoiding false sharing

```c
float a[m,n], s[m,32]
#pragma omp parallel for private(i,j), shared(s,a)
for i = 0, m-1
    s[i,1] = 0.0
for j = 0, n-1
    s[i,1] += a[i,j]
end for
end for
```

False sharing in higher dimension arrays

Jacobi’s method for solving Laplace’s Eqn

\[
\begin{align*}
    & \text{for } j=1 : N \\
    & \text{for } i=1 : M \\
    & u'[i,j] = (u[i-1,j] + u[i+1,j] + u[l,j-1] + u[l,j+1])/4;
\end{align*}
\]

Copying in lieu of data distribution

```fortran
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)
endo
da(1:n1,j,1:n3) = tmp(1:n1,1:n3)
enddo
```

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 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
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.

\[
P \xrightarrow{X:=2} \quad \text{Memory} \xrightarrow{X:=2} \quad \quad P \xrightarrow{X:=2} \quad \text{Memory} \xrightarrow{X:=2}
\]
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

• **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
Fin