Lecture 17

NUMA Architecture and Programming
Announcements

• Extended office hours today until 6pm
• Weds after class?
• Partitioning and communication in Particle method project
Uniform vs. non-uniform partitioning

- Rectilinear partitioning maintains fixed connectivity, simplifying communication
Uniform vs. non-uniform partitioning

- Rectilinear partitioning maintains fixed connectivity, simplifying communication
Load imbalance drift increases running time

A comparison of static and dynamic load balancing of a vortex dynamics computation on 32 processors of the Intel iPSC-1. If the workloads are partitioned only at the beginning of the calculation (static load balancing, top curve), the loads drift gradually out of balance, and the time required to perform a velocity evaluation steadily increases with time. In contrast, a dynamic load balancing strategy (bottom curve) periodically rebalances the workloads and can maintain an almost steady running time; here loads were rebalanced every fourth velocity evaluation.
Processor $i$ is assigned $L_i$, a subregion of the work mesh, and an external interaction region $D_i$, which we have shaded. $D_i$ is a surrounding shell of bins and is $C$ bins thick, where we have chosen $C = 2$. This region divides into 6 subregions, one for each interacting processor. Subproblems $L_i$ and $L_k$ are locally dependent because $D_i$ and $L_k$ intersect. In comparison, subproblems $L_i$ and $L_j$ are locally independent, because $D_i$ and $L_j$ do not intersect.
Repatriating particles

Processor $i$'s assigned subproblem changes from $L_i$ to $\overline{L}_i$ as the result of repartitioning. The processor must give up the work contained in the region $L_i - \overline{L}_i$ to other processors.
Today’s lecture

• Consistency
• NUMA
• Example NUMA Systems
  ♦ Cray XE-6
  ♦ SGI
• Performance programming
Memory model

- 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.
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
Today’s lecture

• Consistency
• **NUMA**
• Example NUMA Systems
  ❖ Cray XE-6
  ❖ SGI
• Performance programming
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; NUMA nodes of the Cray XE-6, SGI UV, Altix, Origin 2000

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

Store A, #x
(home & owner)

A ← dirty

Load A

©2012 Scott B. Baden / CSE 260/ Fall 2012
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)

Load A

©2012 Scott B. Baden /CSE 260/ Fall 2012
Performance issues

• False sharing
• Locality, locality, locality
  ◆ Page placement
  ◆ Page migration
  ◆ Copying v. redistribution
  ◆ Layout
Today’s lecture

• Consistency
• NUMA
• Example NUMA Systems
  ✷ Cray XE-6
  ✷ SGI
• Performance programming
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/
XE-6 Processor memory interconnect (node)

http://www.hector.ac.uk/cse/documentation/Phase2b/#arch
XE-6 Processor memory interconnect (node)

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

©2012 Scott B. Baden /CSE 260/ Fall 2012
Today’s lecture

• Consistency
• Example NUMA Systems
  ♦ Cray XE-6
  ♦ SGI Origin 2000
• Performance programming
Origin 2000 Interconnect

32 processor system

64 processor system
Locality
Poor Locality
Today’s lecture

• Consistency
• NUMA
• Example NUMA Systems
  ❖ Cray XE-6
  ❖ SGI
• Performance programming
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

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

\[ a[i] = b[i] + q \times c[i] \]

Round robin initialization, w/ migration
- Parallel initialization
- Serial initialization

(No migration) parallel initialization with round-robin placement
(No migration) optimal placement or migration enabled
Migration eventually reaches the optimal time

- Round robin initial, w/ migration
- Parallel initialization
- Serial initialization
- One node initial placement, w/ migration
- Parallel initialization, first touch, migration
- Parallel initialization, first touch (optimal)
Cumulative effect of Page Migration

![Graph showing the cumulative effect of page migration with various migration strategies over iteration number.](image-url)
Bisection Bandwidth and Latency

\[ a(i) = b(i) + q \cdot c(i) \]
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
```

©2012 Scott B. Baden /CSE 260/ Fall 2012
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
```

![Graph showing speedup with processors]
False sharing in higher dimension arrays

Jacobi’s method for solving Laplace’s Eqn

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

Parallel Computer Architecture, Culler, Singh, & Gupta
Fin