

Lecture 17

Optimizing for the memory hierarchy

Announcements

- Peer Review Survey
 - Worth 1.5% of your final exam grade
 - Separate from CAPE
 - Run by Center for Teacher Development
 - Will be announced on Piazza next week
- A Performance Puzzle

A Performance Puzzle

- When we fuse the ODE and PDE loops in A3, we slow down the code...
- But, we actually observe fewer memory reads and cache misses!
- What happened?!

For a specified number of iterations, using supplied initial conditions repeat

```
for (j=1; j < m+1; j++){  
  for (i=1; i < n+1; i++) {  
    // PDE SOLVER  
    E[j,i] = E_p[j,i]+ $\alpha$ *(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);  
    // ODE SOLVER  
    E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);  
    R[j,i] += dt*( $\epsilon$ +M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));  
  }  
}
```

swap E_p and E

End repeat

Unfused loops

Command: `./apf -n 400 -i 1000`

Data file: `cachegrind.out.18715`

Dr

Dlmr

```
-----
1,684,151,925 61,178,280    PROGRAM TOTALS
1,683,255,028 61,140,008    solve.cpp:solve( ...)
.      .      .           // Fills in the TOP Ghost Cells
...
// Solve for the excitation, a PDE
  400,000          0    for(j=innerBlkRowStartIndx;j <=innerBlkRowEndIndx; j+=(n+2)){
                        E_tmp = E + j;
                        E_prev_tmp = E_prev + j;
        400,000          0    for(i = 0; i <= n; i++) {
880,404,002 20,202,001      E_tmp[i] = E_prev_tmp[i]+alpha*(E_prev_tmp[i+1]...)
                        }
                        // Solve the ODEs
        2,000        2,000    for(j=innerBlkRowStartIndx; j <= innerBlkRowEndIndx;j+=(n+2)){
                        E_tmp = E + j;
                        R_tmp = R + j;
                        for(i = 0; i <= n; i++) {
320,000,000 40,202,000      E_tmp[i] += -dt*(kk*E_tmp[i]*(E_tmp[i]-a)..)*R_tmp[i]);
480,000,000        2,000      R_tmp[i] +=  dt*(ε+M1*R_tmp[i]/(E_tmp[i]+M2))*(...);
                        }
}
```

Fused loops

- Last level cache miss traffic is insignificant
- Write traffic is virtually identical

Command: `./apf -n 400 -i 1000`

Data file: `cachegrind.out.18560`

Dr

Dlmr

```
-----  
1,122,541,940 41,076,278 PROGRAM TOTALS  
1,121,645,025 41,038,005 solve.cpp:solve( ...)
```

```
// Solve for the excitation, a PDE
```

```
    18,000      4,000  for(j=innerBlockRowStartIndex;j<=innerBlockRowEndIndex;j+=(n+2)){  
        0          0      for (i = 0; i <= n; i++) {  
800,000,000 20,201,000      E_tmp[i] = E_prev_tmp[i]+alpha*(E_prev_tmp[i+1]...)  
160,000,000 20,101,000      E_tmp[i] += -dt*(kk*E_tmp[i]*(E_tmp[i]-a)..)*R_tmp[i]);  
160,000,000          0      R_tmp[i] += dt*(ε+M1*R_tmp[i]/(E_tmp[i]+M2))*(...);  
                                }  
                                }
```

Fusing the loops

- Slows down the simulation: running time grows from 3.8 sec to 9.0 sec!
- # data references and L1 misses drop by 50%
- What happened?

For a specified number of iterations, using supplied initial conditions repeat

```
for (j=1; j < m+1; j++){  
  for (i=1; i < n+1; i++) {  
    // PDE SOLVER  
    E[j,i] = E_p[j,i]+ $\alpha$ *(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);  
    // ODE SOLVER  
    E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);  
    R[j,i] += dt*( $\epsilon$ +M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));  
  }  
}
```

swap E_p and E

End repeat

A plausible answer

- The number of instruction reads grew by about 40%
- We are executing 40% more instructions from 5.5g to 7.6g
(The number of instructions needed to execute the i loop grows from 25m to 322m)
- Most were intercepted in cache (high hit rate: 100%)

For a specified number of iterations, using supplied initial conditions repeat

```
for (j=1; j < m+1; j++){  
  for (i=1; i < n+1; i++) {  
    // PDE SOLVER  
    E[j,i] = E_p[j,i]+ $\alpha$ *(E_p[j,i+1]+E_p[j,i-1]-4*E_p[j,i]+E_p[j+1,i]+E_p[j-1,i]);  
    // ODE SOLVER  
    E[j,i] += -dt*(kk*E[j,i]*(E[j,i]-a)*(E[j,i]-1)+E[j,i]*R[j,i]);  
    R[j,i] += dt*( $\epsilon$ +M1* R[j,i]/(E[j,i]+M2))*(-R[j,i]-kk*E[j,i]*(E[j,i]-b-1));  
  }  
}
```

swap E_p and E

End repeat

Examining the assembler

- All SSE instructions are “unpacked” (64 bits only)
- The same number of instructions in the innermost loop ~ 45 vs 50
 - Same number of arithmetic SSE instructions
 - 3 extra loop control instructions for the unfused code
 - 1 extra data move (register-register)
- The assembler code for the loop isn't telling us the complete picture

```
for (j=1; j < m+1; j++){  
  for (i=1; i < n+1; i++) {  
     $E[j,i] = E\_p[j,i] + \alpha * (E\_p[j,i+1] + E\_p[j,i-1] - 4 * E\_p[j,i] + E\_p[j+1,i] + E\_p[j-1,i]);$   
     $E[j,i] += -dt * (kk * E[j,i] * (E[j,i] - a) * (E[j,i] - 1) + E[j,i] * R[j,i]);$   
     $R[j,i] += dt * (\epsilon + M1 * R[j,i] / (E[j,i] + M2)) * (-R[j,i] - kk * E[j,i] * (E[j,i] - b - 1));$   
  }  
}
```

The assembler for the fused loop

```

..B3.29:                # Preds ..B3.29 ..B3.28      movsd      (%r10,%r11,8), %xmm1
#solve.cpp:112.28        movaps      %xmm5, %xmm13
#solve.cpp:115.84        movsd      8(%r10,%r11,8), %xmm9
#solve.cpp:112.49        movsd      .L_2il0floatpacket.6(%rip), %xmm8
#solve.cpp:112.83        mulsd      %xmm1, %xmm8
#solve.cpp:112.83        addsd      -8(%r10,%r11,8), %xmm9
#solve.cpp:112.65        movsd      .L_2il0floatpacket.10(%rip), %xmm15
#solve.cpp:115.45        subsd      %xmm8, %xmm9
#solve.cpp:112.83        movsd      .L_2il0floatpacket.11(%rip), %xmm12
#solve.cpp:115.65        addsd      16(%rdx,%r11,8), %xmm9
#solve.cpp:112.97        addsd      -16(%rbp,%r11,8), %xmm9
#solve.cpp:112.117       mulsd      %xmm6, %xmm9
#solve.cpp:112.117       addsd      %xmm9, %xmm1
#solve.cpp:112.117       movaps      %xmm1, %xmm11
#solve.cpp:114.56        movaps      %xmm1, %xmm10
#solve.cpp:114.69        movsd      %xmm1, (%r8,%r11,8)
#solve.cpp:112.17        subsd      %xmm4, %xmm11
#solve.cpp:114.56        subsd      %xmm3, %xmm10

```

```

111:    for(i = 0; i < n; i++) {
112:    E_tmp[i] = E_prev_tmp[i]+alpha*(E_prev_tmp[i+1]+E_prev_tmp[i-1]-
        4*E_prev_tmp[i]+E_prev_tmp[i+(n+2)]+E_prev_tmp[i-(n+2)]);
// Solve the ODE
114:    E_tmp[i] += -dt*(kk*E_tmp[i]*(E_tmp[i]-a)*(E_tmp[i]-1)+
        E_tmp[i]*R_tmp[i]);
115:    R_tmp[i] += dt*(epsilon+M1* R_tmp[i]/( E_tmp[i]+M2))*
        (-R_tmp[i]-kk*E_tmp[i]*(E_tmp[i]-b-1));
}

```

The assembler for the fused loop

```
#solve.cpp:114.69      mulsd      %xmm5, %xmm11
#solve.cpp:114.56      mulsd      %xmm10, %xmm11
#solve.cpp:114.69      addsd      (%r9,%r11,8), %xmm11
#solve.cpp:114.81      mulsd      %xmm1, %xmm11
#solve.cpp:114.81      mulsd      %xmm0, %xmm11
#solve.cpp:114.81      addsd      %xmm11, %xmm1
#solve.cpp:114.17      movsd      %xmm1, (%r8,%r11,8)
#solve.cpp:114.17      addsd      %xmm1, %xmm12
#solve.cpp:115.65      mulsd      %xmm1, %xmm13
#solve.cpp:115.84      subsd      %xmm4, %xmm1
#solve.cpp:115.103     movsd      (%r9,%r11,8), %xmm8
#solve.cpp:115.17      subsd      %xmm3, %xmm1
#solve.cpp:115.105     mulsd      %xmm8, %xmm15
#solve.cpp:115.45      mulsd      %xmm1, %xmm13
#solve.cpp:115.105     divsd      %xmm12, %xmm15
#solve.cpp:115.65      movaps     %xmm8, %xmm14
#solve.cpp:115.72      addsd      %xmm2, %xmm15
#solve.cpp:115.65      xorps      .L_2il0floatpacket.13(%rip), %xmm14
#solve.cpp:115.72      mulsd      %xmm7, %xmm15
#solve.cpp:115.65      subsd      %xmm13, %xmm14
#solve.cpp:115.105     mulsd      %xmm14, %xmm15
#solve.cpp:115.105     addsd      %xmm15, %xmm8
#solve.cpp:115.17      movsd      %xmm8, (%r9,%r11,8)
#solve.cpp:115.17      incq       %r11
#solve.cpp:111.13      cmpq       %r13, %r11
#solve.cpp:111.13      jb         ..B3.29      # Prob 82%
#solve.cpp:111.13
```

The assembler for the fused loop

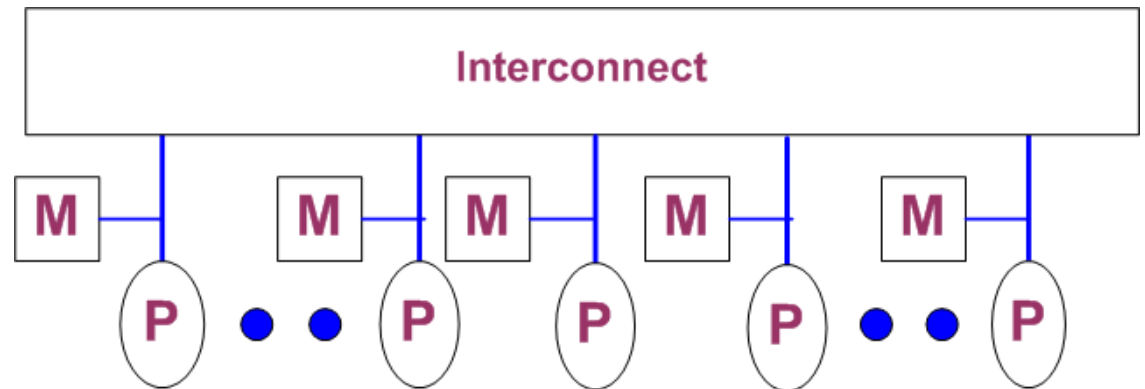
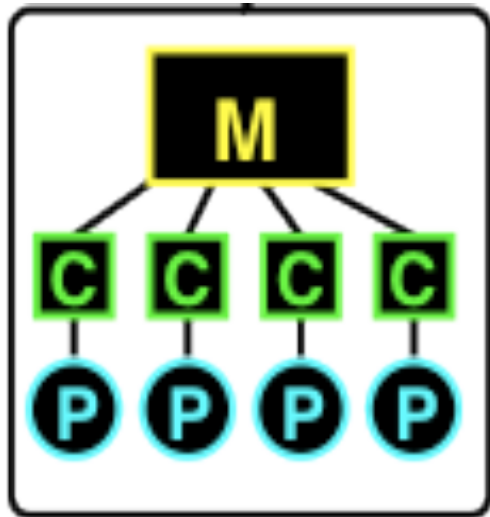
```
1.92g UNCHANGED  E_tmp[i] = E_prev_tmp[i] + ...;
1.52g 2.24g      E_tmp[i] += -dt*(...);
1.92b 3.2g      R_tmp[i] += dt*(...);
```

Today's lecture

- NUMA Architectures

Multiprocessor organization

- Recall that the CPU hardware (via virtual memory address mapping) automatically performs the global to local mapping of shared memory references
- 2 types, depending on the uniformity of memory access times
 - **UMA:** *Uniform* Memory Access time
Also called a Symmetric Multiprocessor (SMP)
 - **NUMA:** *Non-Uniform* Memory Access time

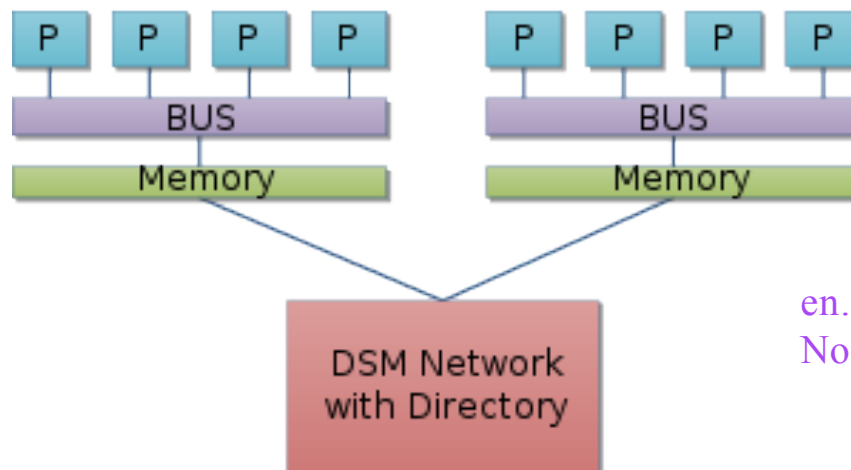


NUMA Architectures

- The address space is global to all processors, but memory is physically distributed

AKA distributed shared memory architectures

- Software/hardware support for monitoring sharers



en.wikipedia.org/wiki/Non-Uniform_Memory_Access

Managing Coherence in NUMA architectures

- Snooping
 - Not scalable
- Directory based
 - Intel Core i7
 - Stanford Dash, SGI, Altix (up to 512 cores), Origin 2000, Gordon system at San Diego Supercomputer Ctr
 - Hybrid: AMD

Some terminology

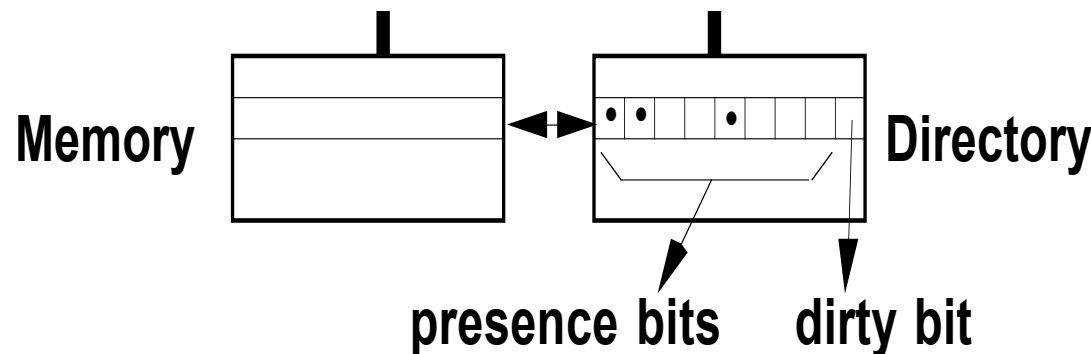
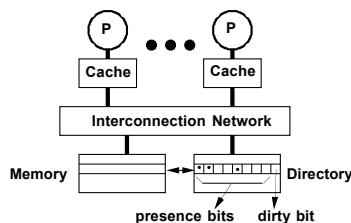
- Every block of memory has an associated **home**: the specific processor that physically holds the associated portion of the global address space
- Every block also has an **owner**: the processor whose memory contains the actual value of the data
- Initially home = owner

How can the owner change?

- A. If a processor other than the home processor writes a block
- B. If a processor other than the home processor reads a block
- C. Both

How we build directory based NUMA

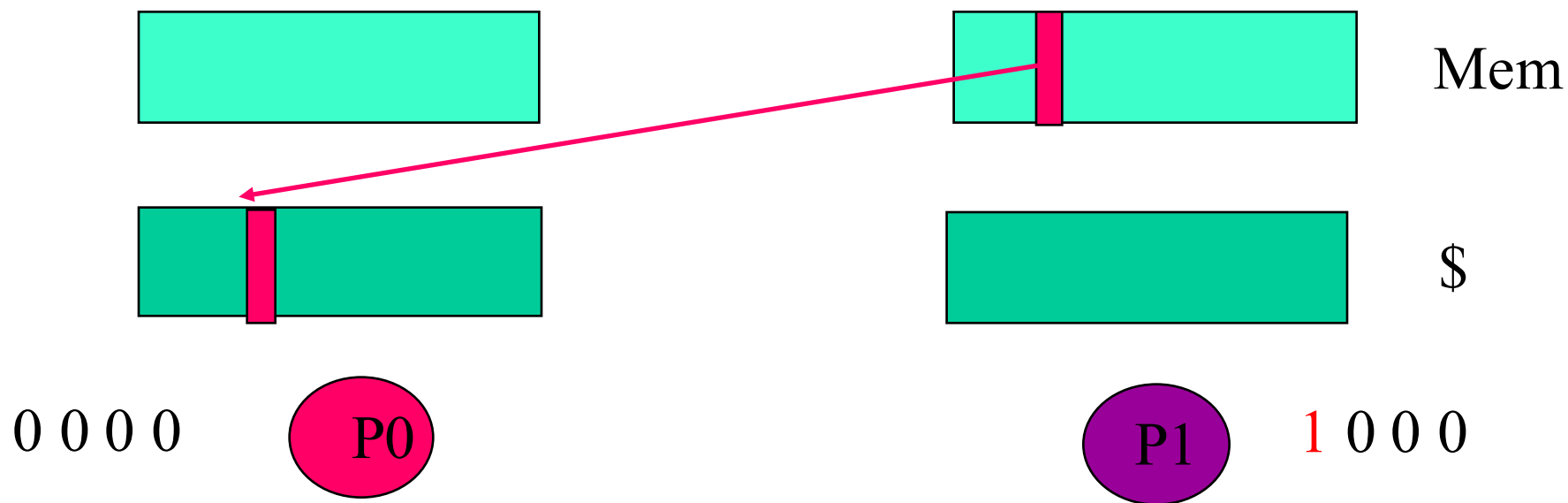
- A **directory** keeps track of sharers, one for each block of memory (usually a cache line)
- Each processor has a 1-bit “sharer” entry in the directory
- There is also a dirty bit and a PID identifying the owner in the case of a dirty block
- Processors send messages to manage coherence



*Parallel
Computer
Architecture,
Culler,
Singh, &
Gupta*

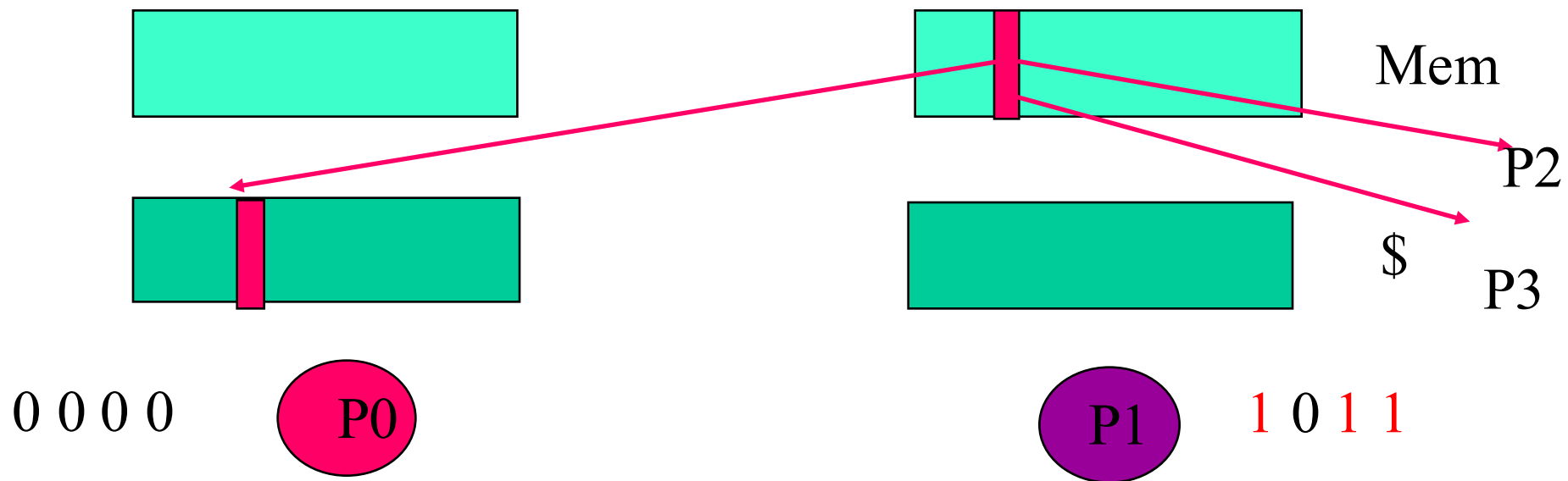
Operation of a directory

- P0 loads **A**
- Set directory entry for **A** (on P1) to indicate that P0 is a sharer



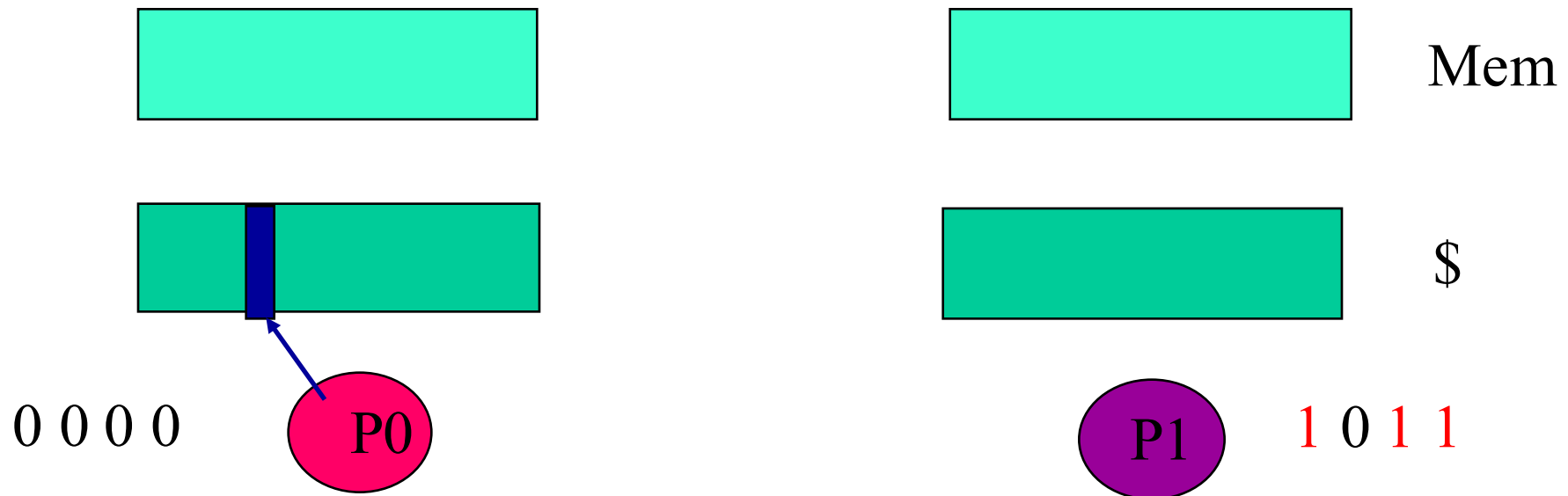
Operation of a directory

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



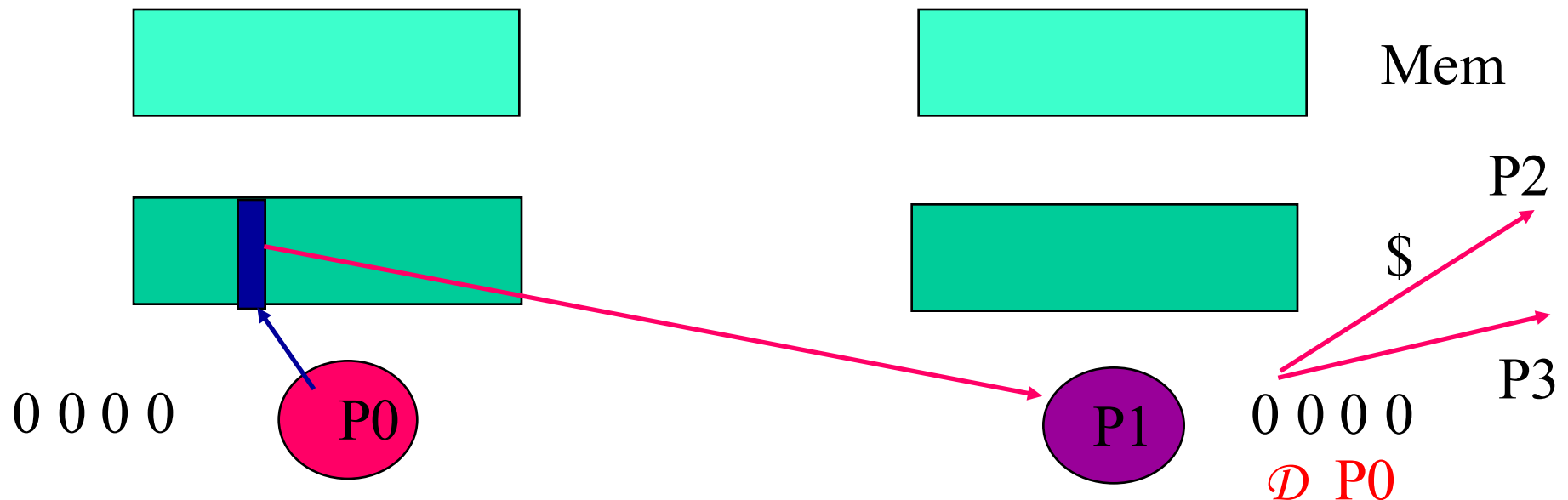
Acquiring ownership of a block

- P0 writes *A*
- P0 becomes the owner of *A*



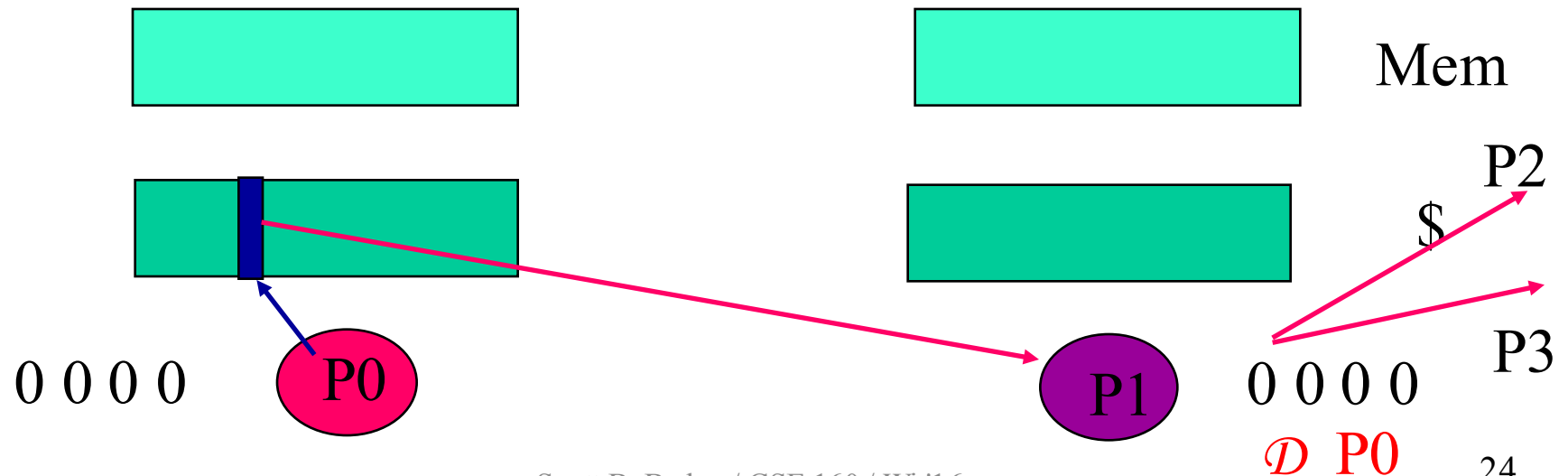
Acquiring ownership of a block

- 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



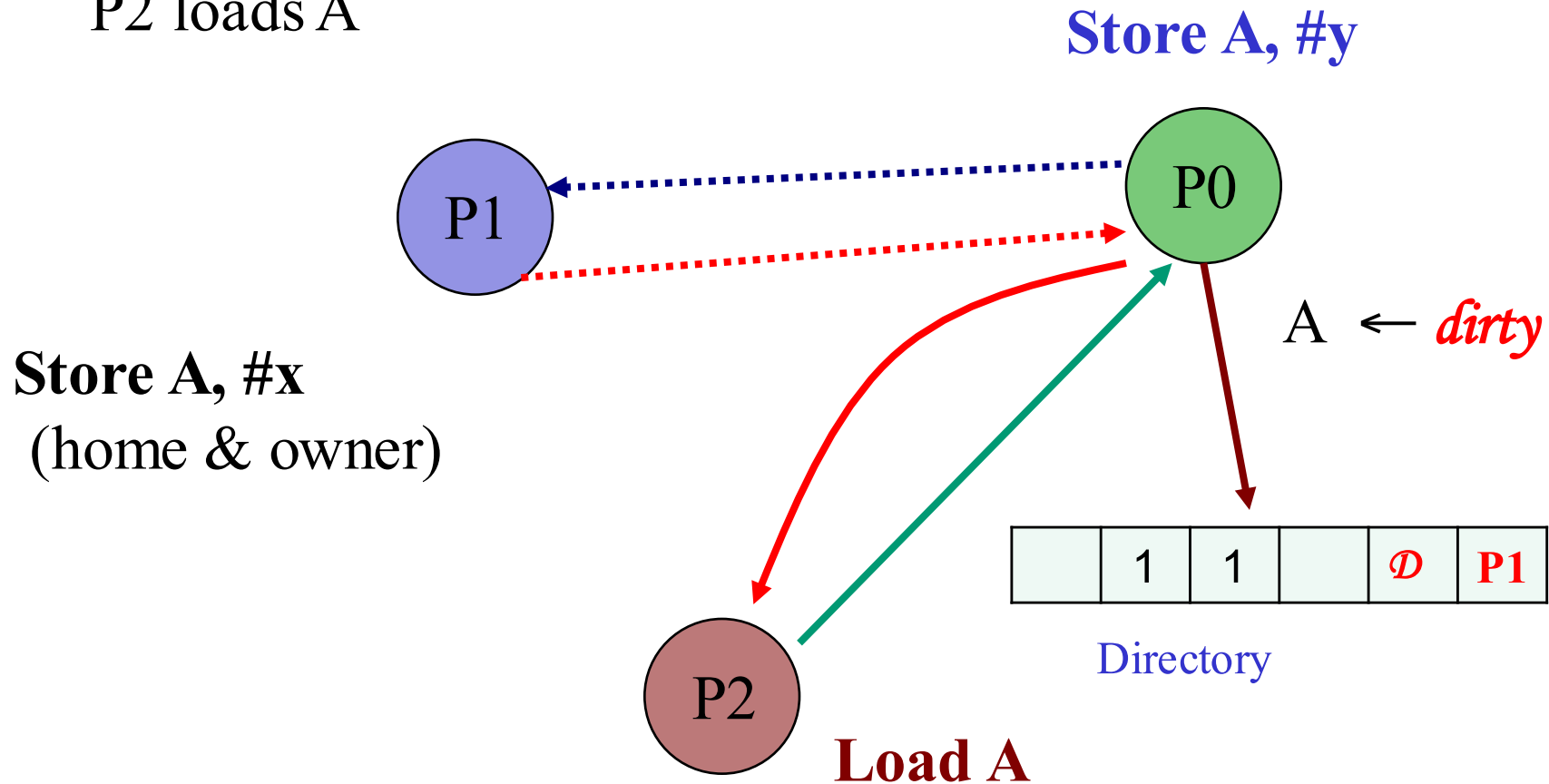
Why must the new owner block all access to the line it is acquiring until all sharer's invalidations have been acknowledged?

- A. Cache must be consistent before anyone can access it
- B. Results could be incorrect
- C. A processor could load the stale data from the old owner
- D. All of A, B and C**
- E is not correct**
- E. A & B



Change of ownership

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



Forwarding

P0 stores into A (home & owner)

P1 stores into A (becomes owner)

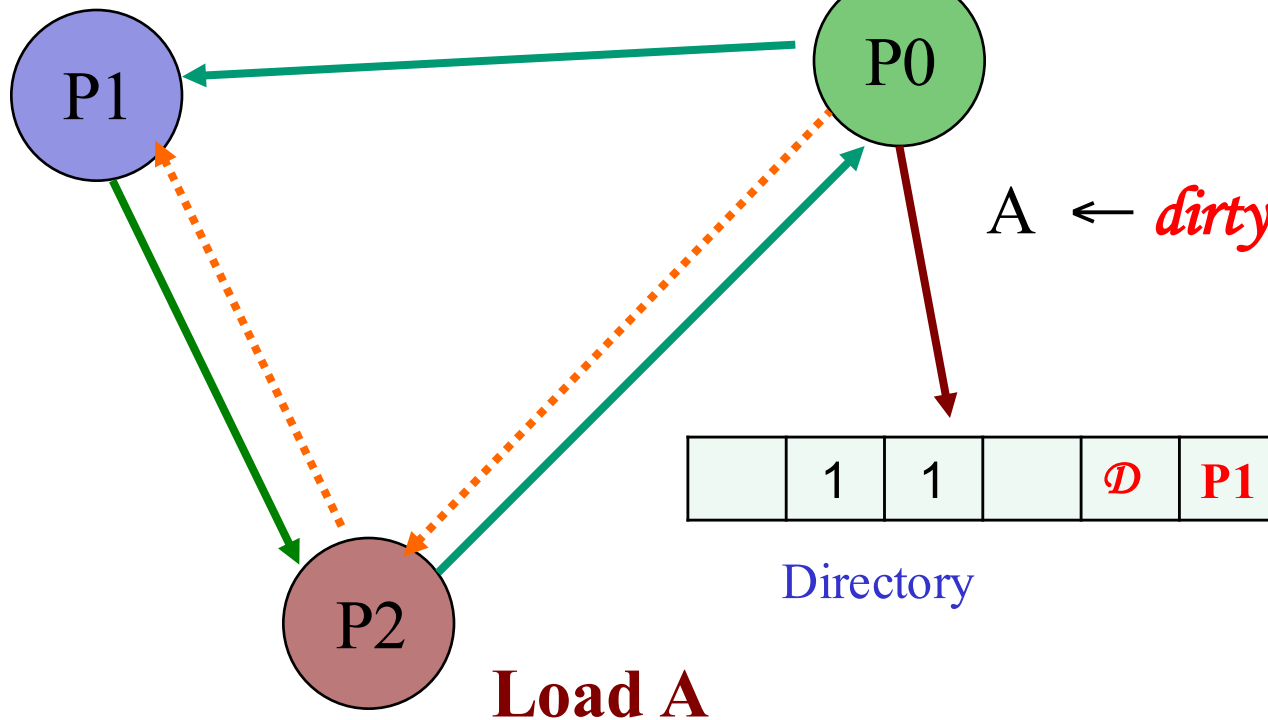
P2 loads A

home (P0) forwards request to
owner (P1)

Store A, #y

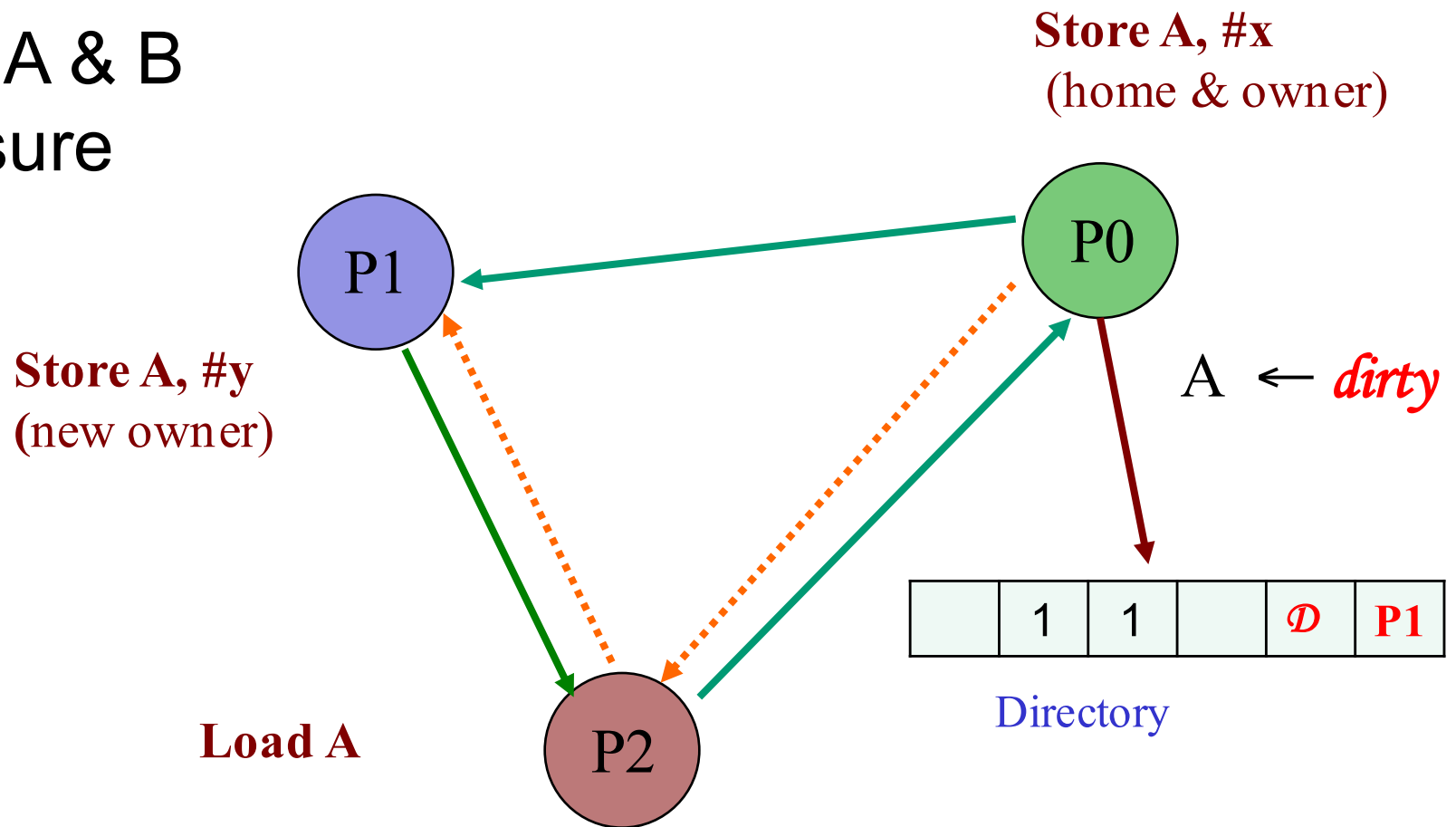
Store A, #x
(home & owner)

A ← *dirty*



Why does the home forward the request to the new owner?

- A. Reduces the cost of the handling the request
- B. Needed to enforce coherence
- C. Both A & B
- D. Not sure



How do processors implement NUMA?

- Intel i7
 - Directory lives in L3 cache (8MB+)
 - Bit vector to indicate those processors whose cache has a block copy in L3
 - Caches must be *inclusive*: anything stored in L1 must also reside in L2 & L3
- AMD Opteron takes a hybrid approach

<http://www.programcreek.com/2012/12/amd-versus-intel-successes-and-pitfalls-of-their-processor-architectures>

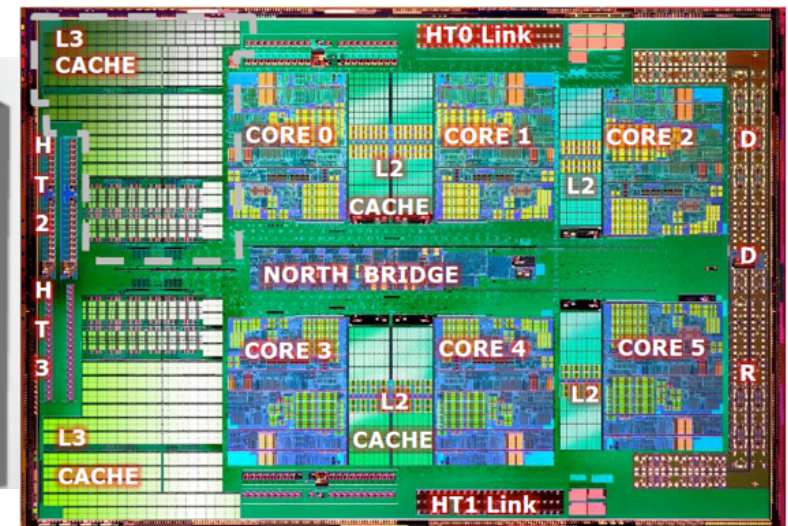
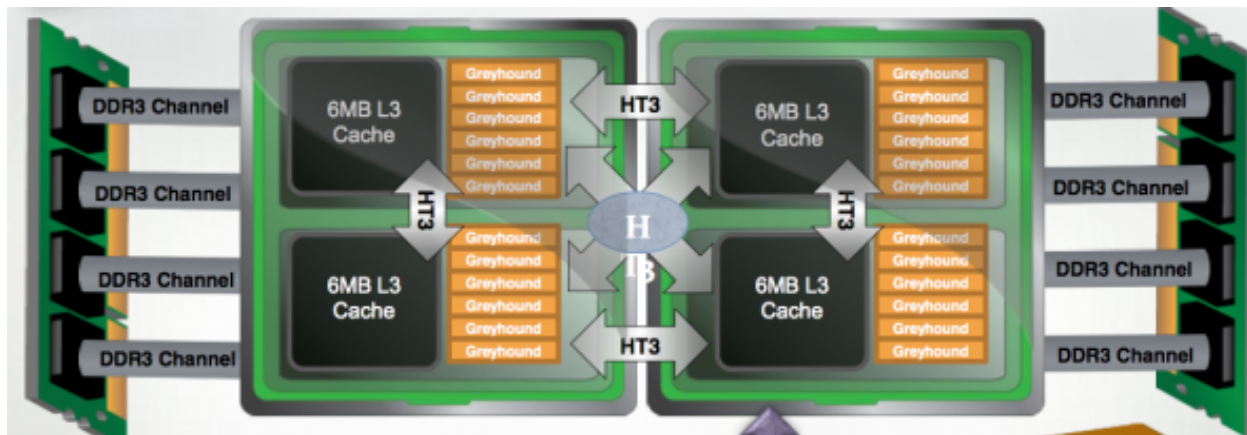
Why must L3 be inclusive?

- A. How would we know what's in L1 or L2?
- B. Avoids the need to access L1 and L2, which will slow down the processor
- C. Both A & B

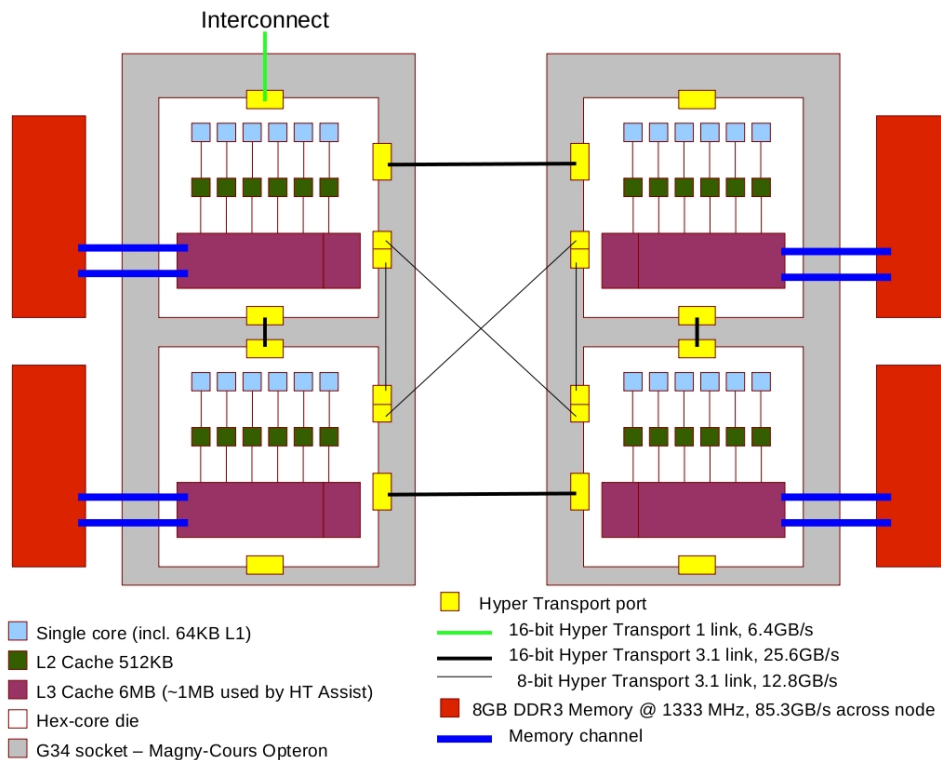
Operton's hybrid approach

- “Magny Cours” processors packaged as 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 to manage 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 accommodates at most 4 cores

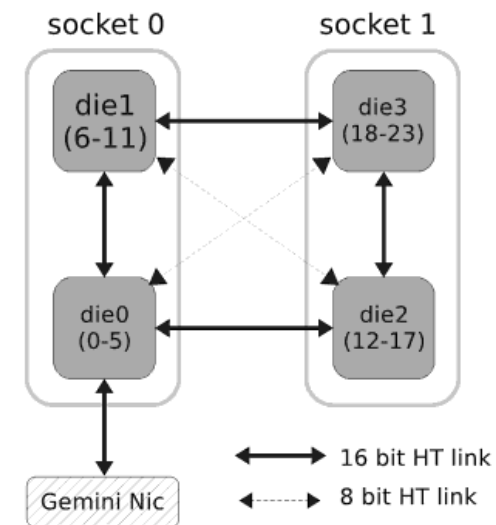
www.nersc.gov/users/computational-systems/hopper/configuration/compute-nodes/



XE-6 Processor memory interconnect (server node)



- Asymmetric connections between dies and processors
 - Direct access to 8GB main memory via 2 memory channels
 - 4 Hyper Transport (HT) links for communicating with other dies



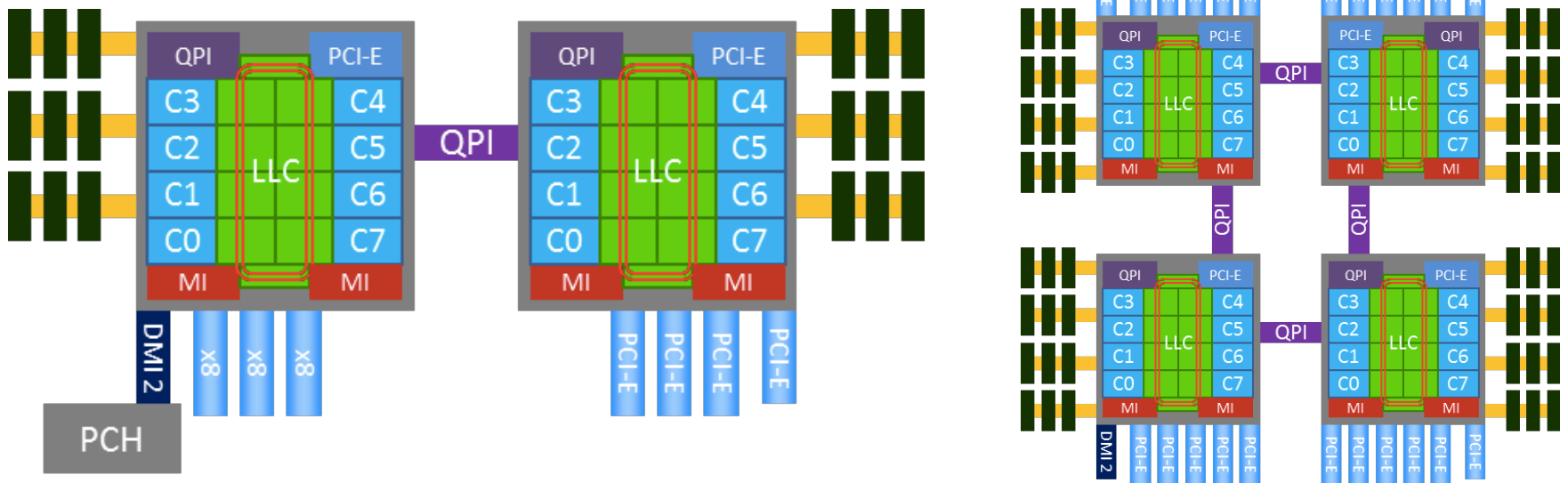
<http://www.hector.ac.uk/cse/documentation/Phase2b/#arch>

Performance issues

- Locality, locality, locality
 - Page placement
 - Page migration
 - Copying v. redistribution
 - Layout
- False sharing

Intel Sandy Bridge Processor

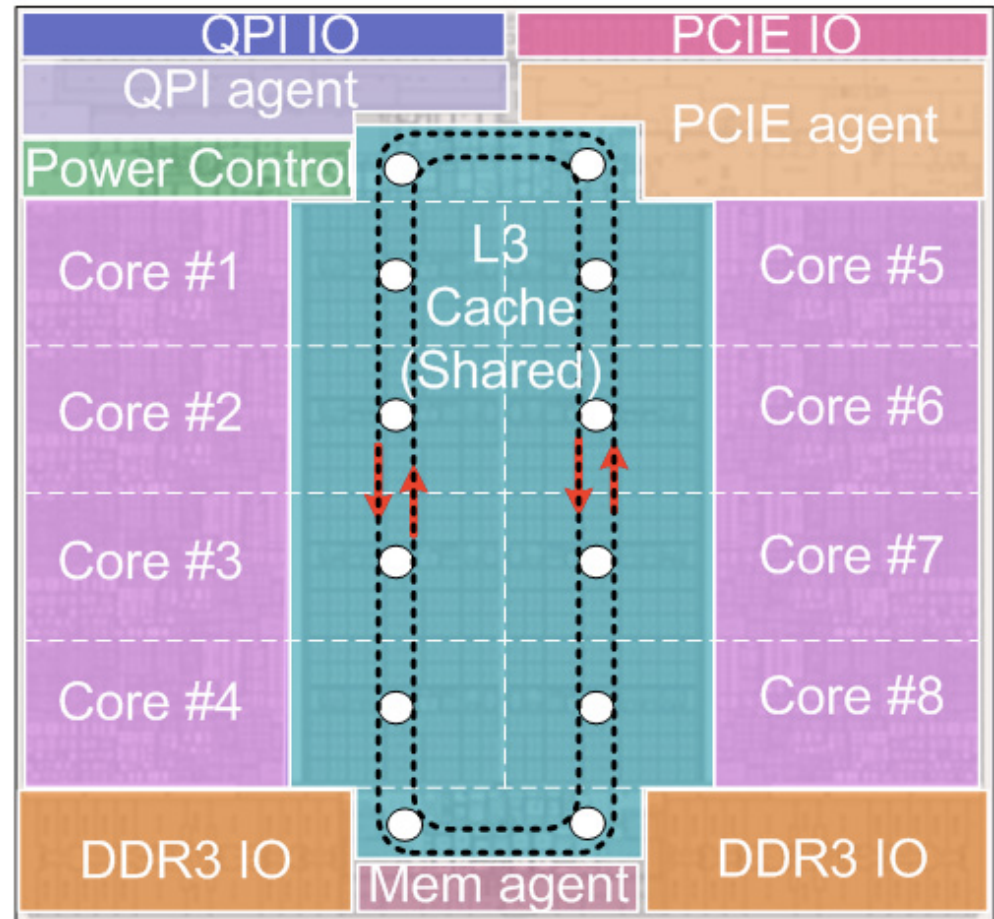
- QPI “Quick Path” interconnect links the processors, 1 per socket
- Up to 4 processors



www.qdpma.com/systemarchitecture/systemarchitecture_sandybridge.html

Ring Interconnect

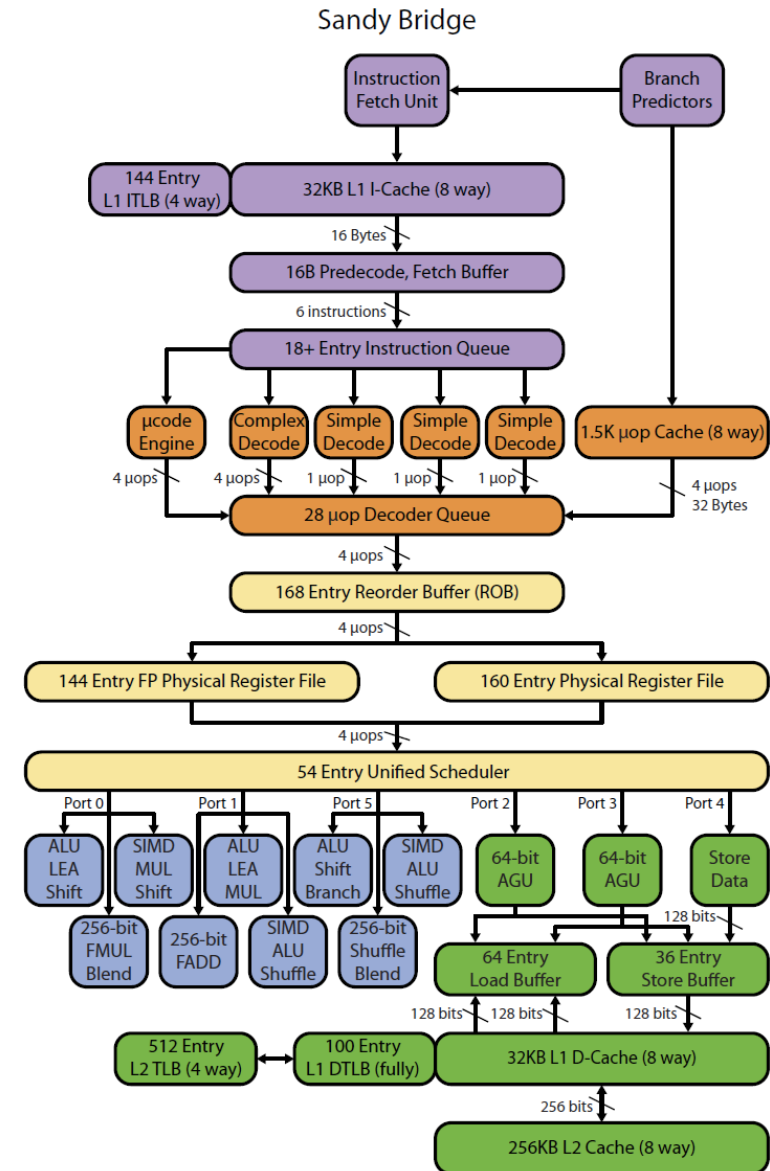
- Ring connects slices of L3
 - Snooping
 - Request
 - Acknowledge
 - 32B data ring
- Distributed communication protocol implements coherence
- Older designs used a crossbar (N x N), but not scalable



www.qdpma.com/systemarchitecture/systemarchitecture_sandybridge.html

Inside the processor

- Instructions can be reordered to hide data transfer latencies behind other operations (Instruction Level Parallelism)
- We can have outstanding cache misses
- We extend the register set to provide placeholders for in-flight instructions



<http://www.realworldtech.com/sandy-bridge/10/>

NUMA awareness

- When we allocate memory, a NUMA processor uses the *first-touch policy*, unless told otherwise
- Be sure to initialize data in parallel, using the same decomposition strategy

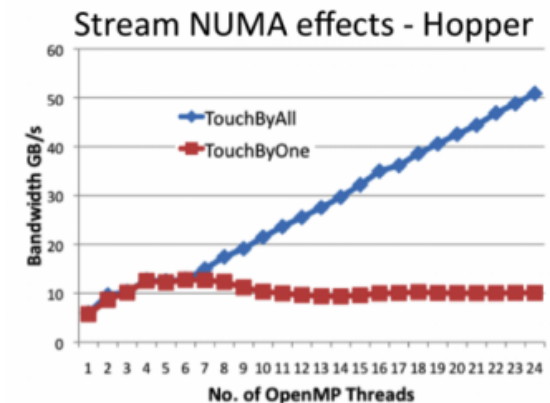
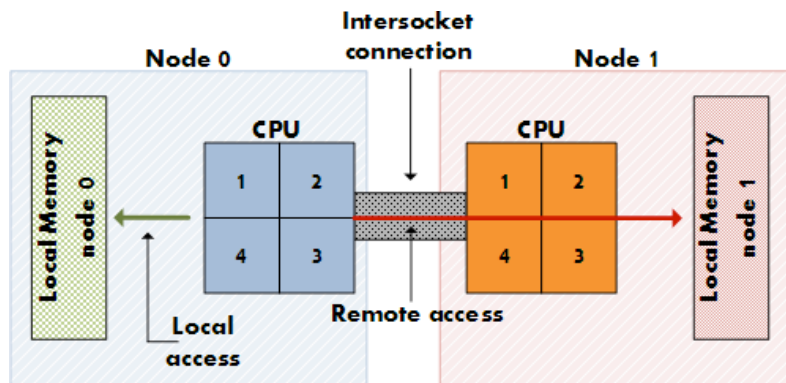
```
x, y, z = new double[n];
```

```
#pragma omp parallel for
```

```
for (i=0; i< n; i++) {x[i]=0; y[i] = ...; z[i] = ...}
```

```
#pragma omp parallel for
```

```
for (i=0; i<n; i++) {x[i] = y[i] + z[i]; }
```



tinyurl.com/olmxh23

What sorts of memory system behavior will result from not parallelizing the data initialization loop?

A. Many costly remote memory accesses

B. Few costly remote memory accesses

```
for (i=0; i< n; i++) {x[i]=0; y[i] = ...; z[i] = ...}  
#pragma omp parallel for  
for (i=0; i<n; i++) {x[i] = y[i] + z[i]; }
```