CSE 160
Lecture 8

NUMA
OpenMP

Scott B. Baden

Today's lecture

- OpenMP
- NUMA Architectures

OpenMP

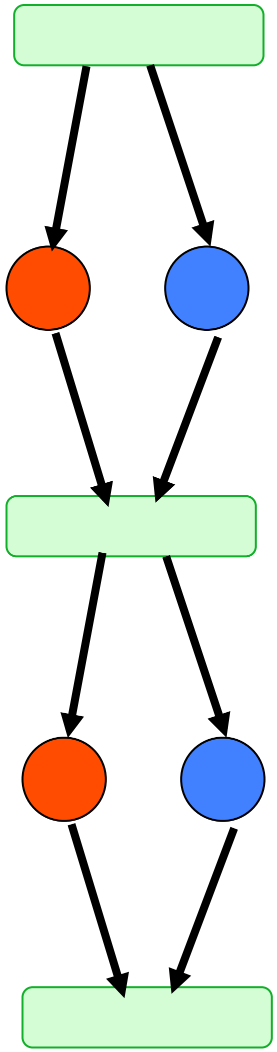
- A higher level interface for threads programming
- Parallelization handled via source code annotations
- See http://www.openmp.org
- Compare with explicit threads programing

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

```
i0 = $TID*n/$nthreads;
i1 = i0 + n/$nthreads;
for (i=i0; i< i1; i++)
   work(i);
```

OpenMP's Fork-Join Model

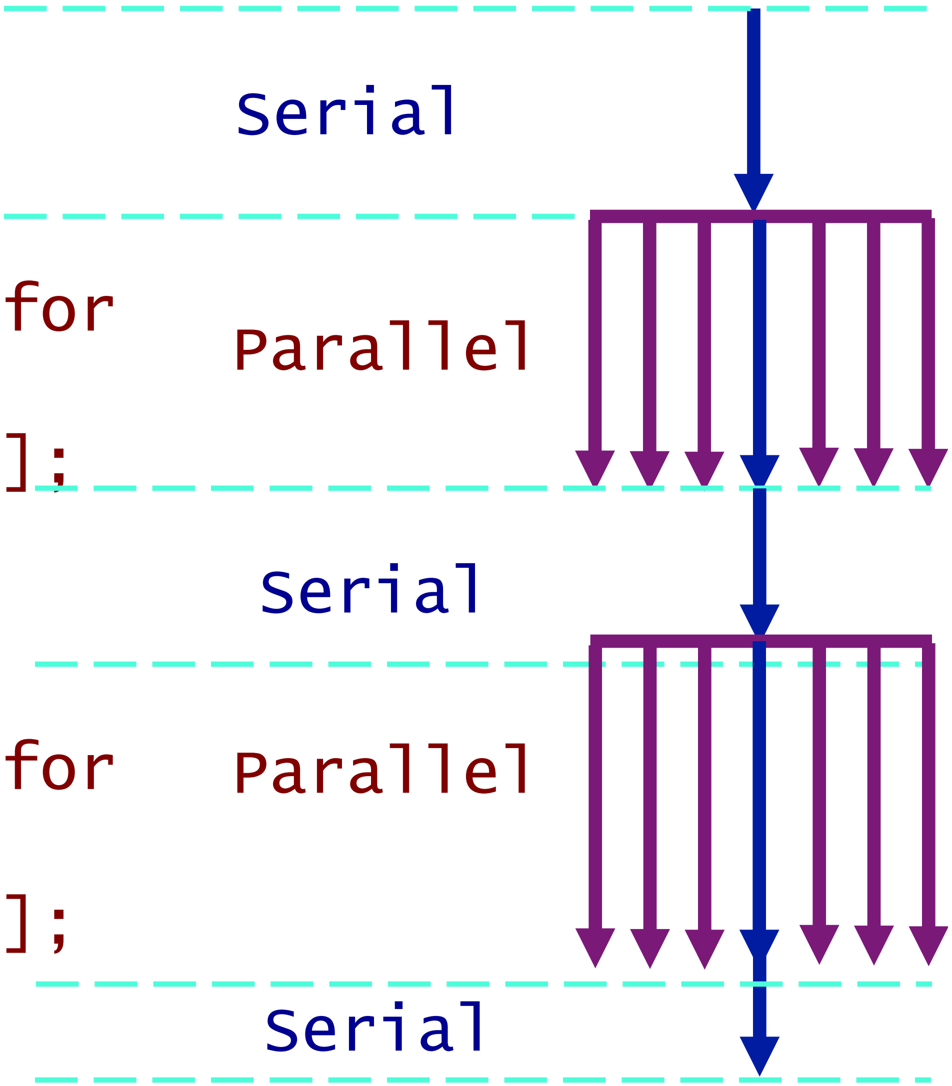
- A program begins life as a single thread
- Parallel regions spawn work groups of multiple threads
- The lexically enclosed program statements execute in parallel by all team members
- When we reach the end of the scope…
  - The team of threads synchronize at a barrier and are disbanded; they enter a wait state
  - Only the initial thread continues
- Thread teams can be created and disbanded many times during program execution, but this can be costly
- A clever compiler can avoid so many thread creations and joins



Fork join model with loops

```
cout << “Serial\n”;
N = 1000;

#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];

M = 500;

#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];

Cout << “Finish\n”;
```

Seung-Jai Min



Serial

Parallel

Serial

Parallel

Serial

Workload decomposition

- Translator automatically generates appropriate local loop bounds
- Also inserts any needed barriers
- We use private/shared pragmas to distinguish thread private from global variable
- Decomposition can be static or dynamic
- Dynamic assignment for irregular problems

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

Parallelizing a nested loop with OpenMP

- We parallelize the outer loop index, indicated shared and private (local) variables
- Not all implementations can parallelize inner loops

```
#pragma omp parallel private(i) shared(n)
  #pragma omp for
  for(i=0; i < n; i++)
     for(j=0; j < n; j++) {
        u^new[i,j] = (u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1] – h^2
  }
```

- Generated code
- There is an implicit barrier after the loop

```
  mymin = 1 + ($TID * n/nprocs),     mymax = mymin + n/nprocs -1
  for(i=mymin; i < mymax; i++)
     for(j=0; j < n; j++)
        u^new[i,j] = (u[i-1,j] + u[i+1,j]+ u[i,j-1]+ u[i, j+1] – h^2f[i,j])/4
  Barrier();
```

Variable scoping

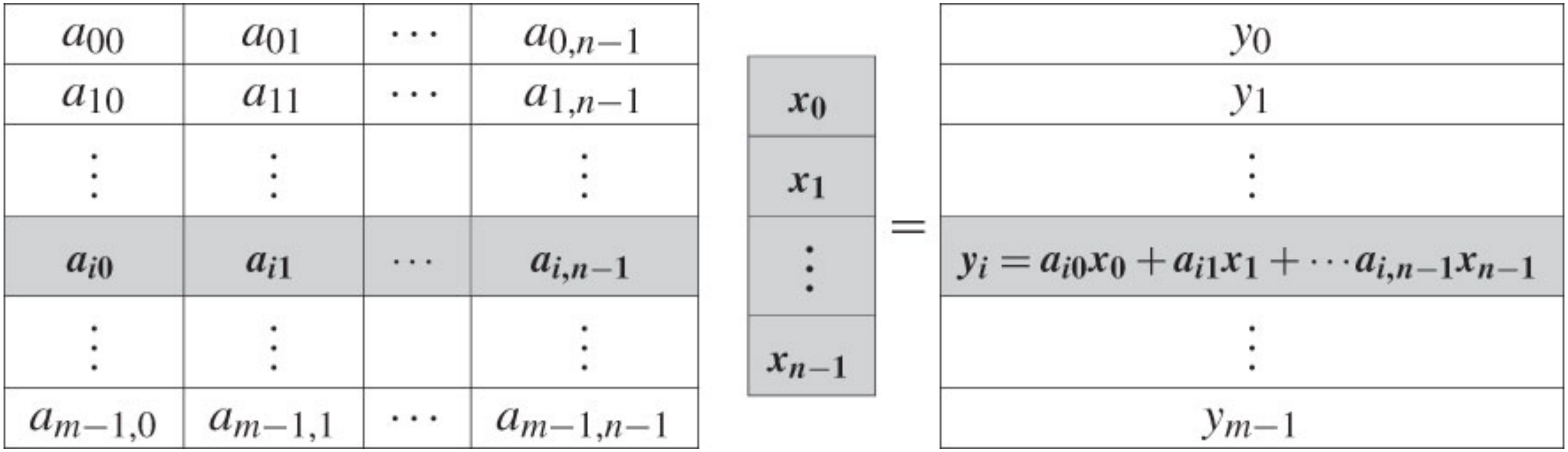
- Any variables declared outside a parallel region are shared by all threads
- Variables declared inside the region are private
- Used shared and private declarations to override the defaults

```
double c = 1/ 6.0, h = 1.0, c2 = h * h;
double ***c = …;
for (it= 0; it<nIters; it++) {
#pragma omp parallel shared(U,Un,b,nx,ny,nz,c2,c) private(i,j,k)
#pragma omp for
   for (int i=1; i<=nx; i++)
     for (int j=1; j<=ny; j++)
       for (int k=1; k<=nz+1; k++)
            Un[i][j][k]=c* (U[i-1][j][k] + U[i+1][j][k] + U[i][j-1][k] + U[i][j+1][k] +
                           U[i][j][k-1] + U[i][j][k+1] - c2*b[i-1][j-1][k-1]);
  U ↔Un;
}
```

OpenMP is also an API

```
#ifdef _OPENMP
#include <omp.h>

int nthreads = 1;
#pragma omp parallel
{
   int tid = omp_get_thread_num();
   if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of openMP threads: %d\n”, nthreads);
   }
}
#endif
```

An application: Matrix Vector Multiplication



$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & & \vdots \\ a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \vdots & \vdots & & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

Initialization

- We allocate and initialize storage outside a parallel region

```
double **A;
A = (double**) \
   malloc(sizeof(double*)*N + sizeof(double)*N*N);
assert(A);

for(j=0;j<N;j++) A[j] = (double *)(A+N) + j*N;

for ( j=0; j<N; j++ )
    for ( i=0; i<N; i++ )
        A[i][j] = 1.0 / (double) (i+j-1);
```

Computation

```
 double **A, *x, *y;                          // GLOBAL
// Start timer
   double t0 = -getTime();

#pragma omp parallel shared(A,x,N)
   for (int k = 0; k<reps; k++)
#pragma omp for
      for (i=0; i<N; i++){
         y[i] = 0.0;
         for (j=0; j<N; j++)
            y[i] += A[i][j] * x[j];
      }

   // Take time
   t0 += getTime();
```

Compare with threads coding

- "Outline" the computation into a thread function
- Spawn and join threads
- Partition the rows of the matrix over processors
- Insert critical sections, barriers when needed

Dealing with loop carried dependences

- OpenMP will dutifully parallelize a loop when you tell it to, even if doing so “breaks” the correctness of the code

```
     int* fib = new int[N];
       fib[0] = fib[1] =  1;
    #pragma omp parallel for num_threads(2)
       for (i=2; i<N; i++)
           fib[i] = fib[i-1]+ fib[i-2];
```

- Sometimes we can restructure an algorithm, as we saw in odd/even sorting
- OpenMP may warn you when it is doing something unsafe, but not always

Reductions in OpenMP

- In some applications, we reduce a set of values down to a single value
    - Taking the sum of a list of numbers
    - Decodomg when Odd/Even sort has finished
- OpenMP avoids the need for an explicit serial section

```
int     Sweep(int *Keys, int OE, int lo, int hi){
bool done = true;
#pragma omp parallel for reduction(&:done)
     for (int i = OE+lo; i <= hi; i+=2) {
        if (Keys[i] > Keys[i+1]){
            Keys[i] ↔ Keys[i+1];
            done &= false;
        }
     }   // All threads 'and' their done flag into the local
        variable
    return done;
}
```

In class exercises – for OpenMP

Questions

1. Iteration to thread mapping
2. Removing data dependencies
3. Dependence analysis
4. Tree Summation

1. Iteration to thread mapping

In $PUB/Examples/OpenMP/Assign
Compile with omp=1 on "make" line

```
#pragma omp parallel shared(N,iters) private(i)
#pragma omp for
for (i = 0; i < N; i++)
        iters[i] = omp_get_thread_num();
```

N = 9, # of openMP threads = 3

```
0 0 0 1 1 1 2 2 2
```

N = 16, # of openMP threads = 4, schedule(static,2)

```
0 0 1 1 2 2 3 3 0 0 1 1 2 2 3 3
```

N=9: `0 0 1 1 2 2 0 0 1`

N = 16, # of openMP threads = 4, schedule(dyanmic,2)

```
3 3 0 0 1 1 2 2 3 3 3 3 3 3 3 3
2 2 3 3 0 0 1 1 2 2 2 2 2 2 2 2
```

2. Removing data dependencies

- B initially: 0 1 2 3 4 5 6 7
- B on 1 thread: 7 7 7 7 11 12 13 14
- How can we split into 2 loops so that each loop parallelizes, the result it correct?

```
#pragma omp for shared (N,B)
  for i  = 0 to N-1
    B[i] += B[N-1-i];
```

B[0] += B[7], B[1] += B[6], B[2] += B[5]

B[3] += B[4], B[4] += B[2], B[5] += B[2]

B[7] += B[0]

Splitting a loop

- For iterations i=N/2+1 to N, B[N-i] reference newly computed data
- All others reference "old" data
- B initially: 0 1 2 3 4 5 6 7
- Correct result: 7 7 7 7 11 12 13 14

```
for i  = 0 to N-1
    B[i] += B[N-i];
```

⇒

```
for i  = 0 to N/2-1
    B[i] += B[N-1-i];
for i  = N/2+1 to N-1
    B[i] += B[N-1-i];
```

In $PUB/Examples/OpenMP/Assign
Compile with omp=1 on "make" line

3. Loop Dependence Analysis

- Which loop(s) can we correctly parallelize with OpenMP?

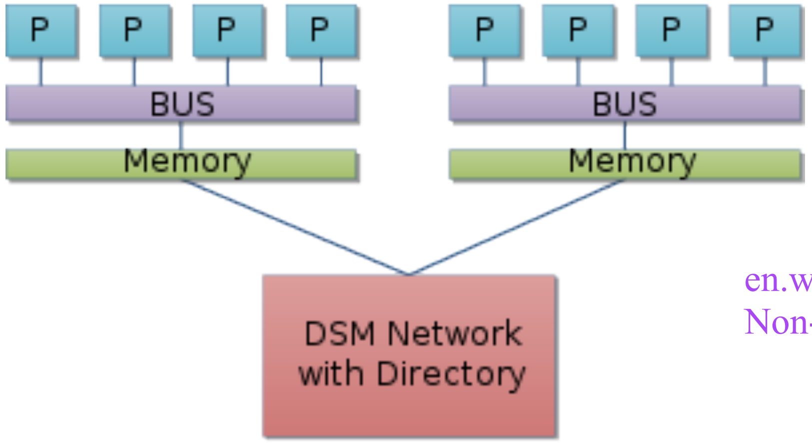
```
1.  for i = 1 to N-1
        A[i] = A[i] + B[i-1];

2.  for i = 0 to N-2
        A[i+1] = A[i] + 1;

3.  for i = 0 to N-1 step 2
        A[i] = A[i-1] + A[i];

4.  for i = 0 to N-2{
        A[i] = B[i];
        C[i] = A[i] + B[i];
        E[i] = C[i+1];
    }
```

Today’s lecture

- OpenMP
- NUMA Architectures

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



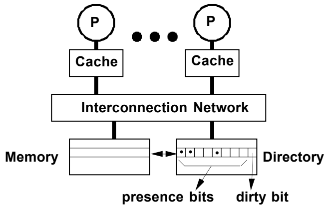
en.wikipedia.org/wiki/Non-Uniform_Memory_Access

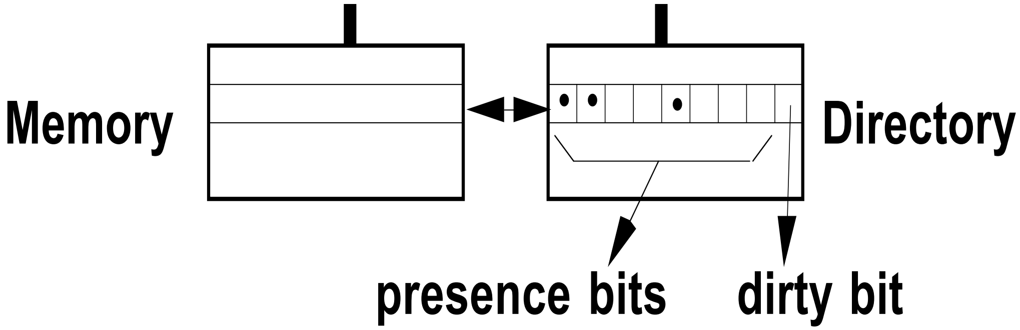
Some terminology

- 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

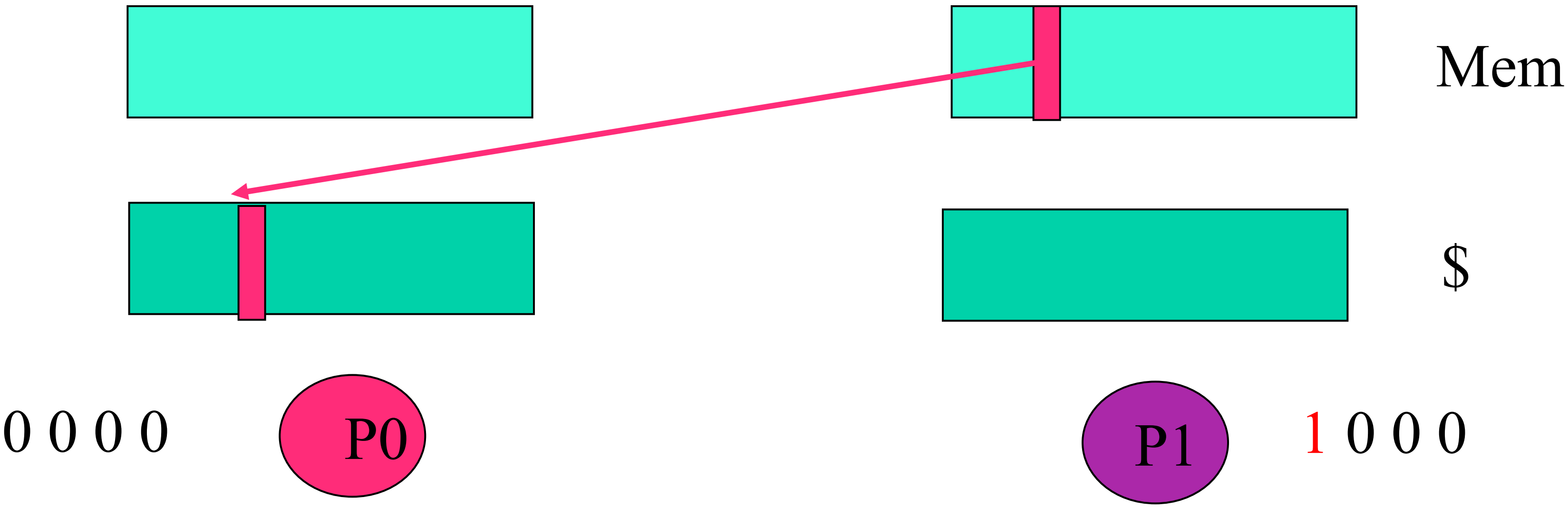




Parallel Computer Architecture, Culler, Singh, & Gupta

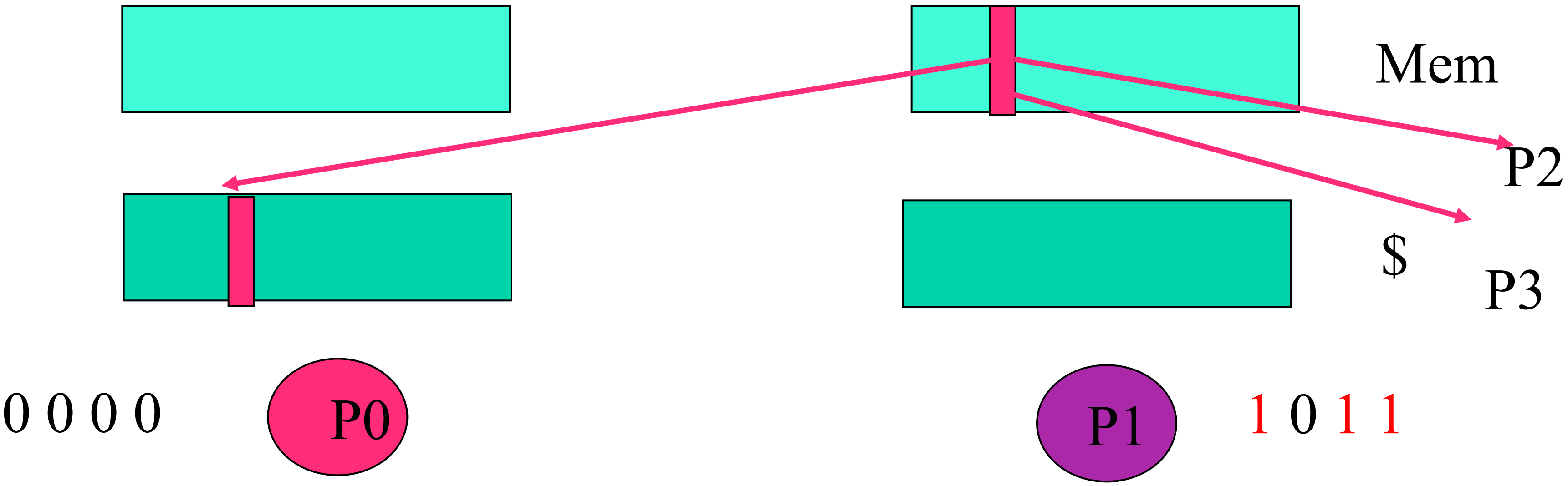
Operation of a directory

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



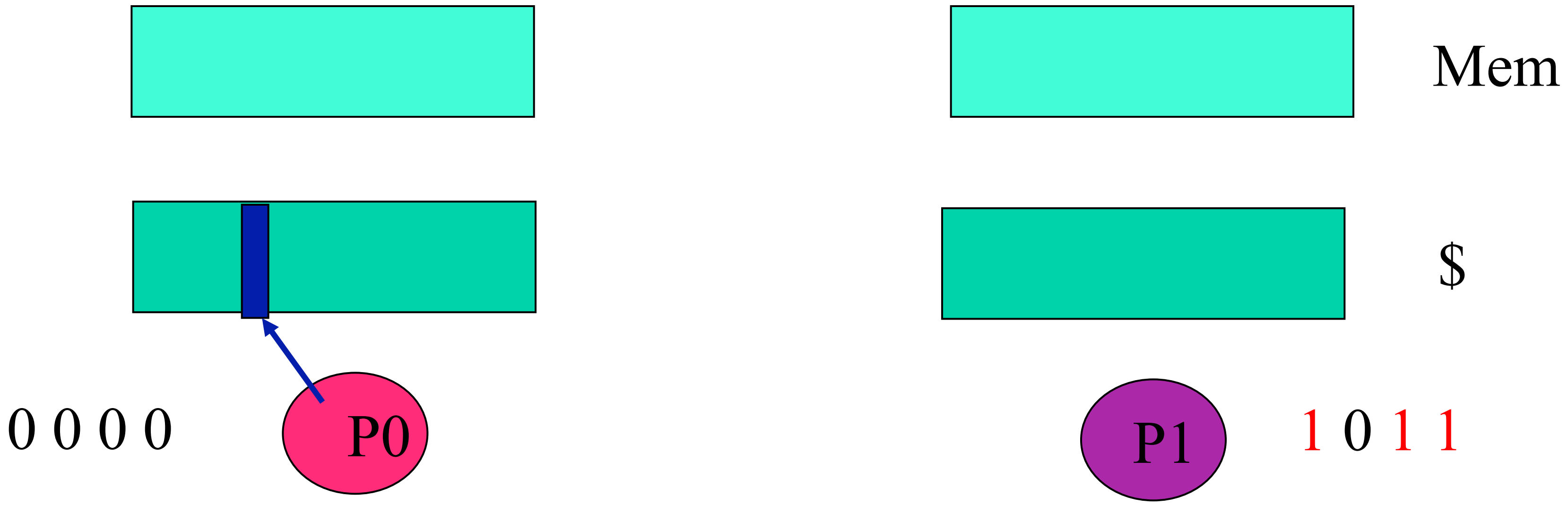
Operation of a directory

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



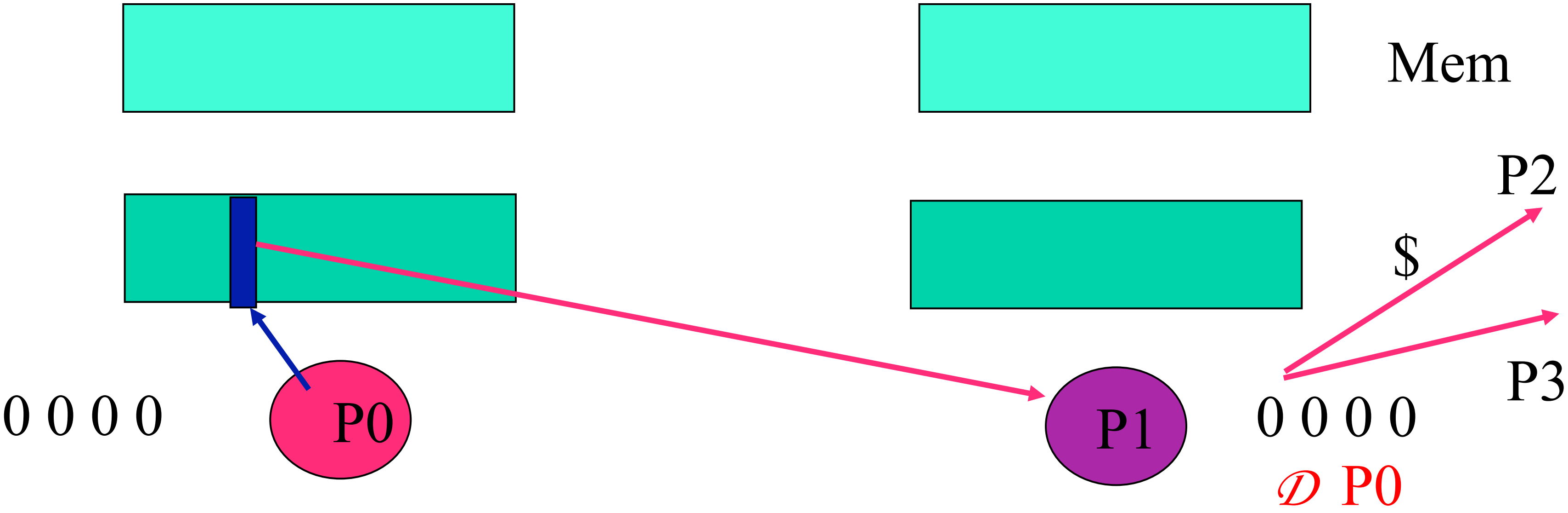
Acquiring ownership of a block

- P0 writes A
- P0 becomes the owner of A



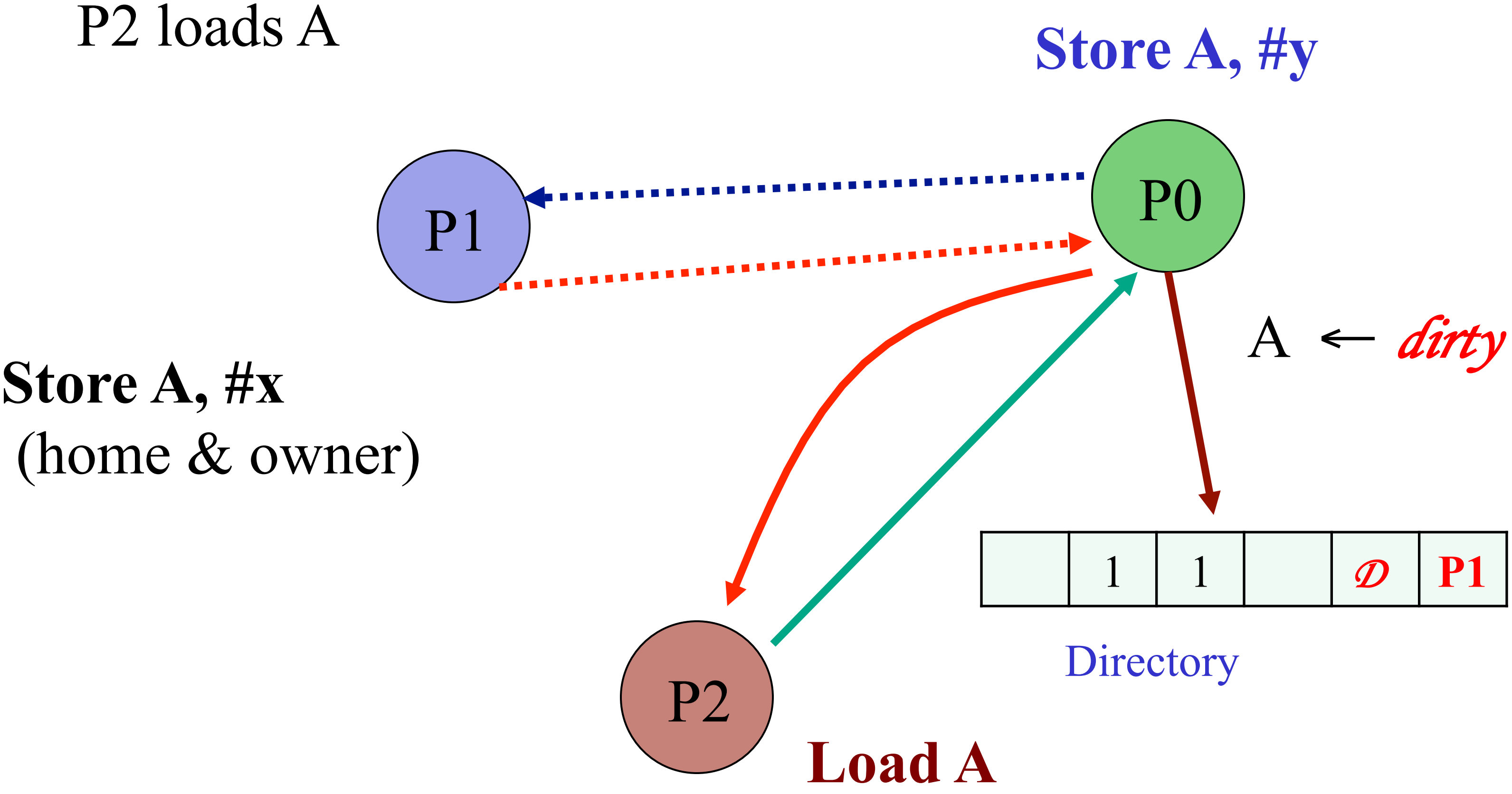
Acquiring ownership of a block

- 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



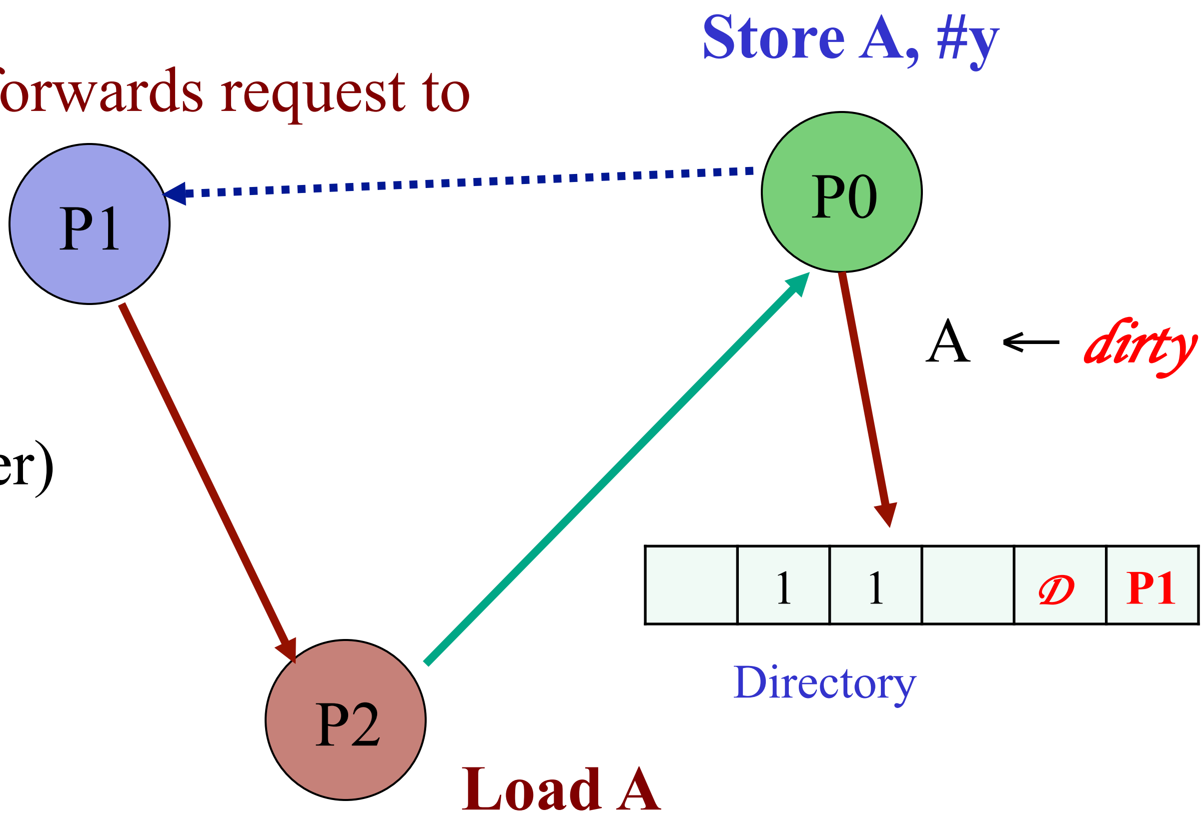
Change of ownership

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

P0

P1

Store A, #x
(home & owner)

P2

Load A

A ← *dirty*

	1	1		*D*	**P1**

Directory

Performance issues

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

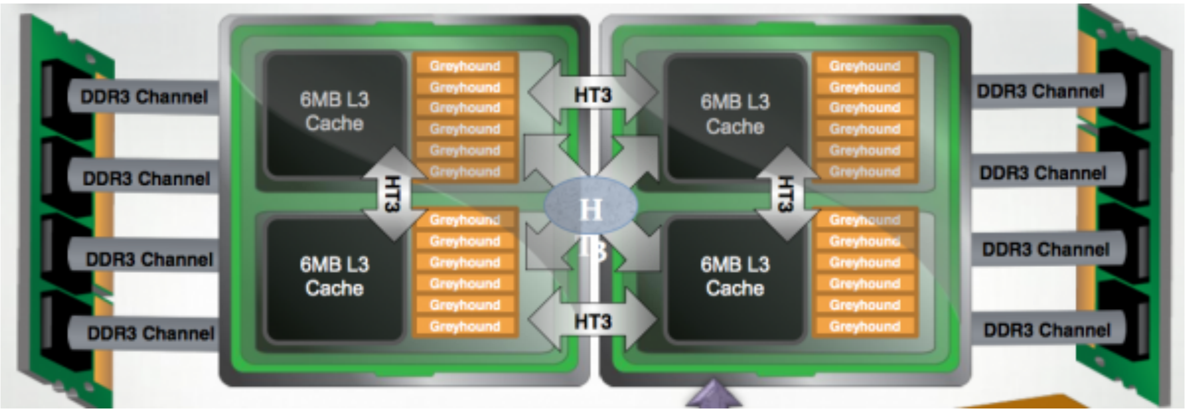
Today's lecture

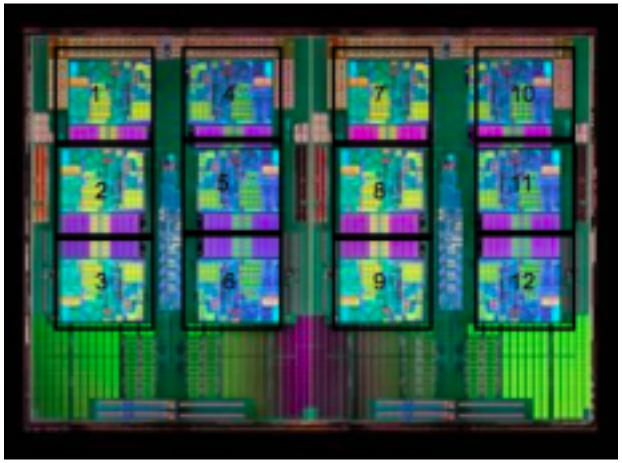
- Consistency
- NUMA
- Example NUMA Systems
  - Cray XE-6
  - SGI
- Performance programming

Cray XE6 node

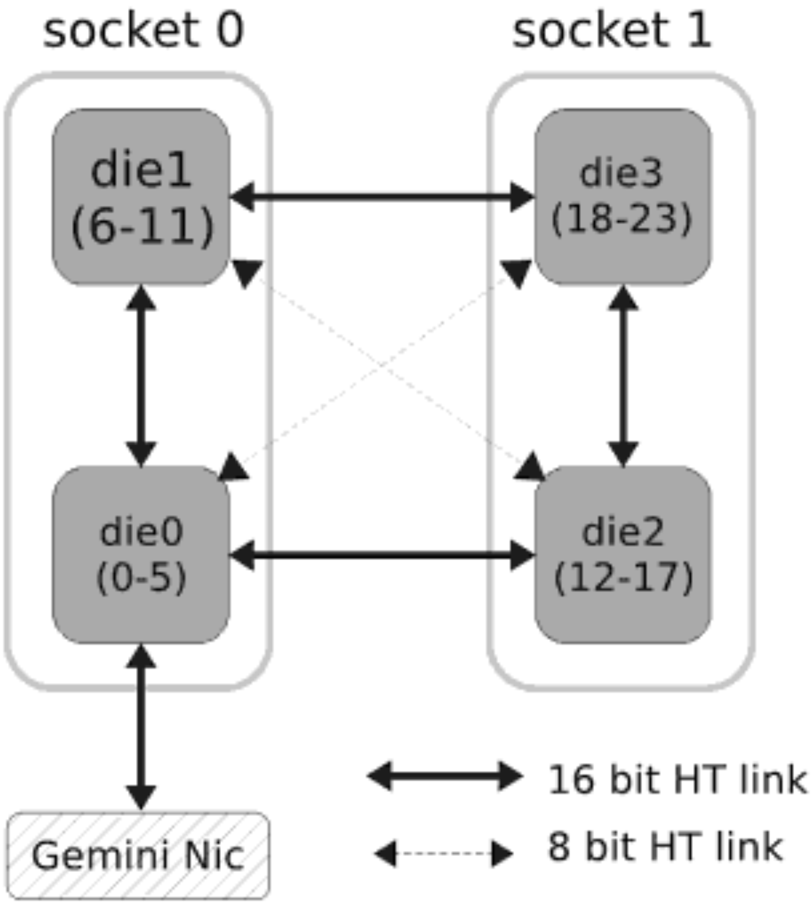
- 24 cores sharing 32GB main memory
- Packaged as 2 AMD Opteron 6172 processors "Magny-Cours"
- Each processor is a directly connected Multi-Chip Module: two hex-core dies living on the same socket
- Each die has 6MB of shared L3, 512KB L2/core, 64K L1/core
  - 1MB of L3 is used for cache coherence traffic
  - 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/



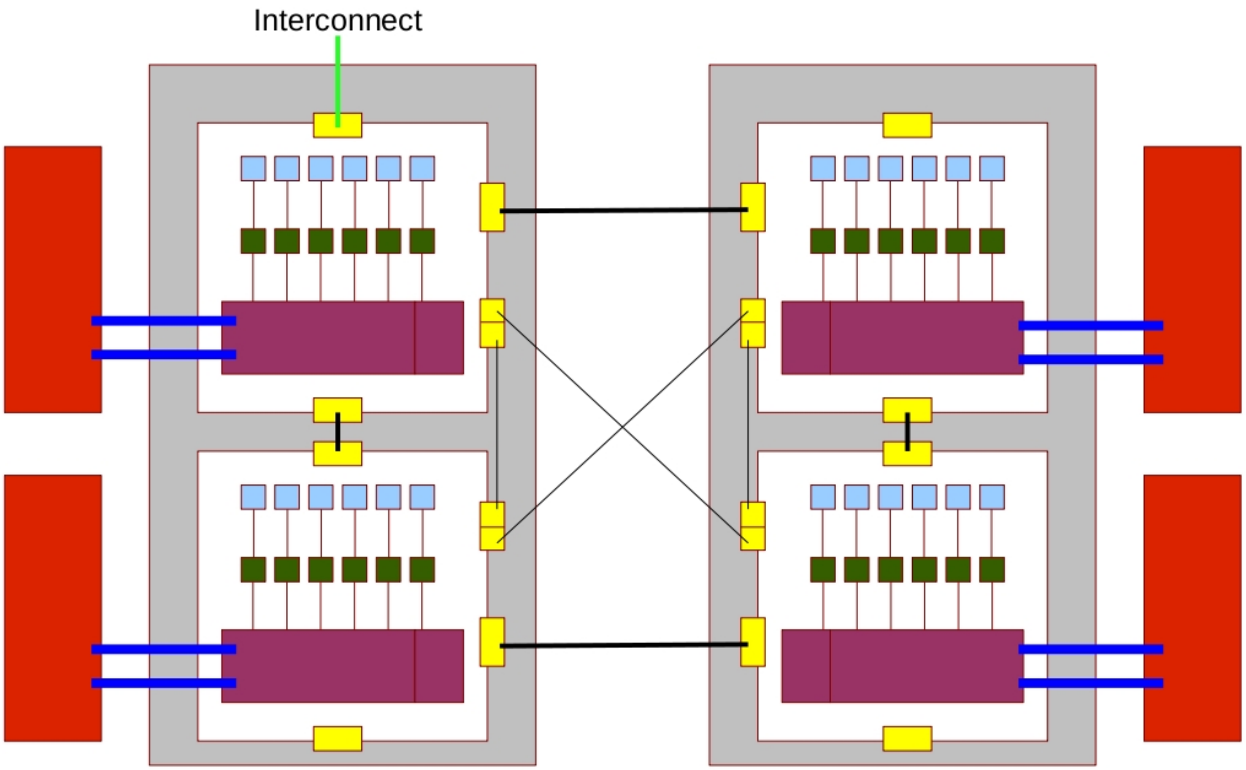


XE-6 Processor memory interconnect (node)



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

XE-6 Processor memory interconnect (node)



- 🟦 Single core (incl. 64KB L1)
- 🟩 L2 Cache 512KB
- 🟪 L3 Cache 6MB (~1MB used by HT Assist)
- ☐ Hex-core die
- ▨ G34 socket – Magny-Cours Opteron

- 🟨 Hyper Transport port
- ━ 16-bit Hyper Transport 1 link, 6.4GB/s
- ━ 16-bit Hyper Transport 3.1 link, 25.6GB/s
- — 8-bit Hyper Transport 3.1 link, 12.8GB/s
- 🟥 8GB DDR3 Memory @ 1333 MHz, 85.3GB/s across node
- ━ Memory channel

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