

Lecture 2

Memory locality optimizations
Address space organization

Announcements

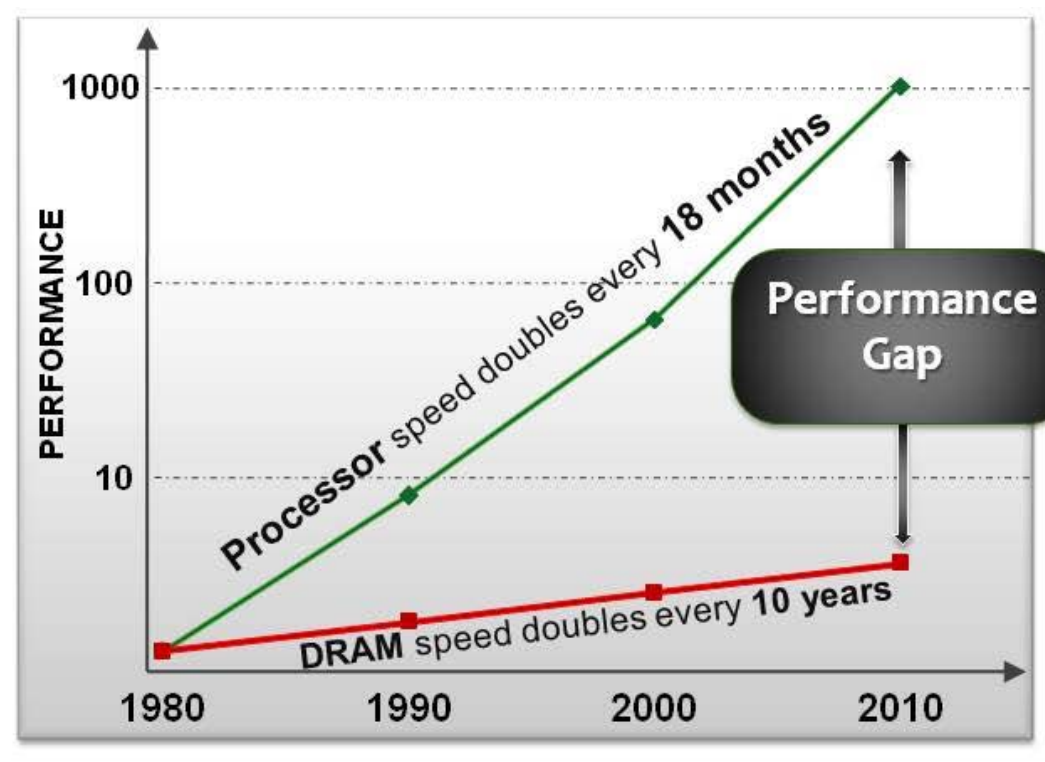
- Programming Lab #1 will be posted on Friday at noon
- Be sure you can login to Bang and Moodle
- Submit NERSC form

Today's Lecture

- Memory hierarchies
- Address space organization
 - ♦ Shared memory
 - ♦ Distributed memory
- Control mechanisms
- Shared memory hierarchy
 - ♦ Cache coherence
 - ♦ False sharing
- HW #1

The processor-memory gap

- The result of technological trends
- Difference in processing and memory speeds growing exponentially over time



Source: acm.org & MoSys

©MoSys, Inc. 2010. All Rights Reserved.

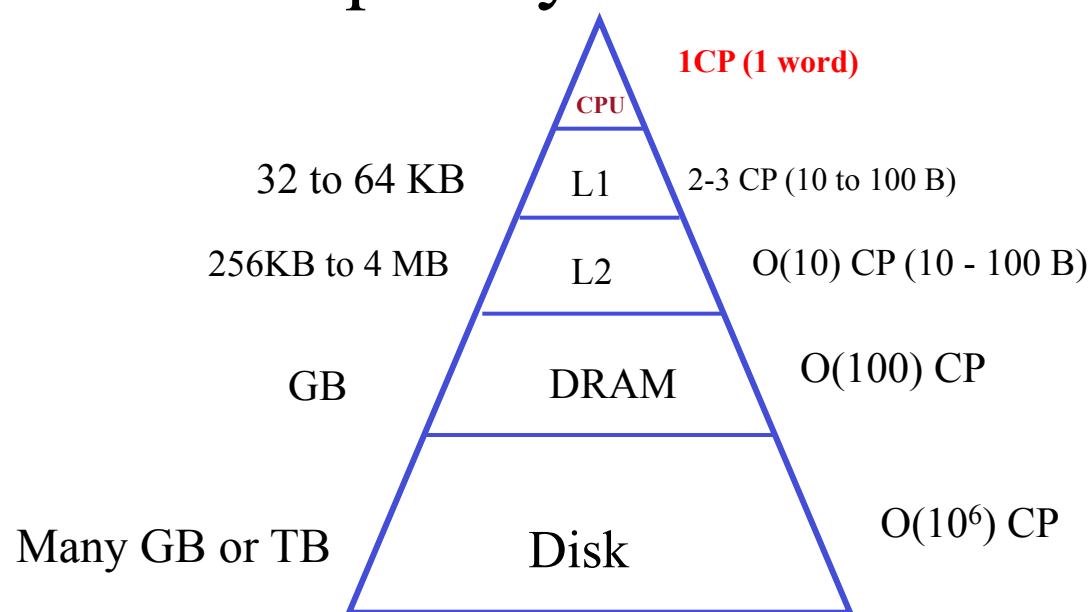
An important principle: locality

- Programs generally exhibit two forms of locality in accessing memory
 - ♦ Temporal locality (time)
 - ♦ Spatial locality (space)
- Often involves loops
- Opportunities for reuse

```
for t=0 to T-1
  for i = 1 to N-2
     $u[i] = (u[i-1] + u[i+1]) / 2$ 
```

Memory hierarchies

- Exploit reuse through a hierarchy of smaller but faster memories
- Put things in faster memory if we reuse them frequently

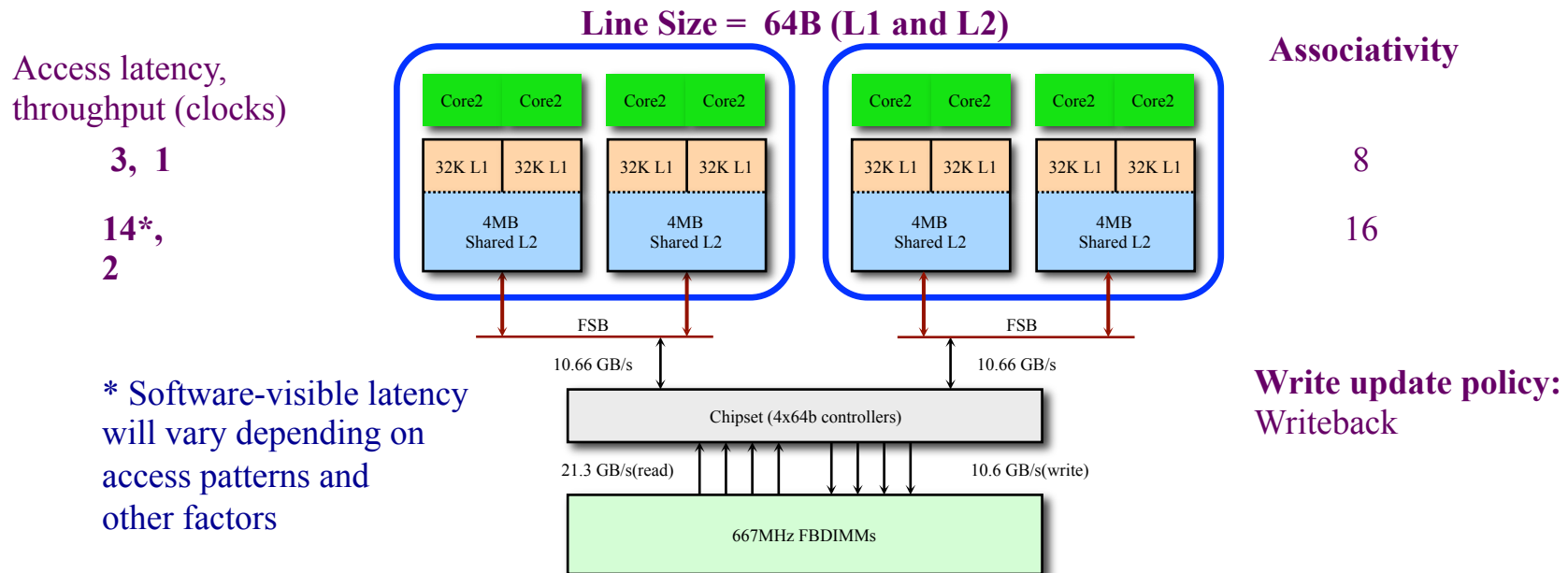


Bang's Memory Hierarchy

- Intel “Clovertown” processor
- Intel Xeon E5355 (Introduced: 2006)
- Two “Woodcrest” dies (Core2) on a multichip module
- Two “sockets”
- *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Tab 2.16



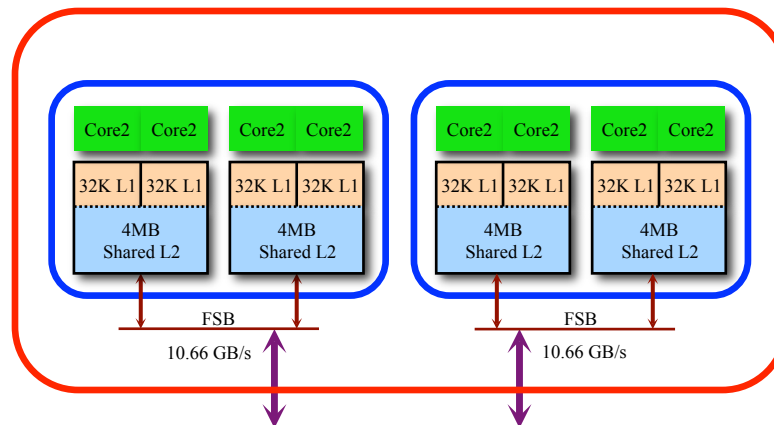
techreport.com/articles.x/10021/2



Sam Williams et al.

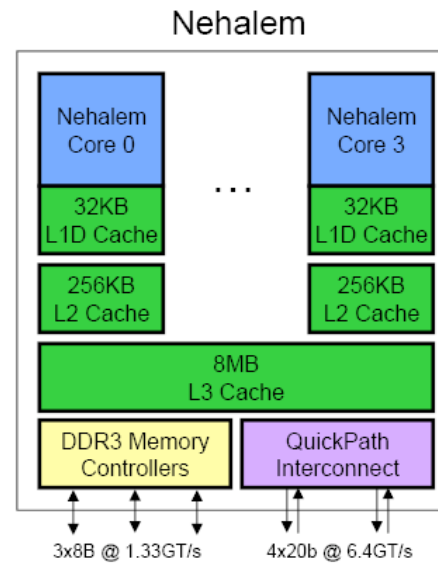
Examining Bang's Memory Hierarchy

- `/proc/cpuinfo` summarizes the processor
 - ♦ `vendor_id` : GenuineIntel
 - ♦ `model name` : Intel(R) Xeon(R) CPU E5345 @2.33GHz
 - ♦ `cache size` : 4096 KB
 - ♦ `cpu cores` : 4
- `processor : 0 through processor : 7`
- Detailed information at
`/sys/devices/system/cpu/cpu*/cache/index*/*`



The 3 C's of cache misses

- Cold Start
- Capacity
- Conflict



realworldtech.com

Managing locality with loop interchange

- The success of caching depends on the ability to *re-use* previously cached data
- Data access order affects re-use
- Assume a cache with 2 entries, each 2 words wide

for (i=0; i<N; i++)

for (j=0; j<N; j++)

a[i][j] += b[i][j];

for (j=0; j<N; j++)

for (i=0; i<N; i++)

a[i][j] += b[i][j];

0	1
4	5

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The 3 C's
Cold Start
Capacity
Conflict

Testbed

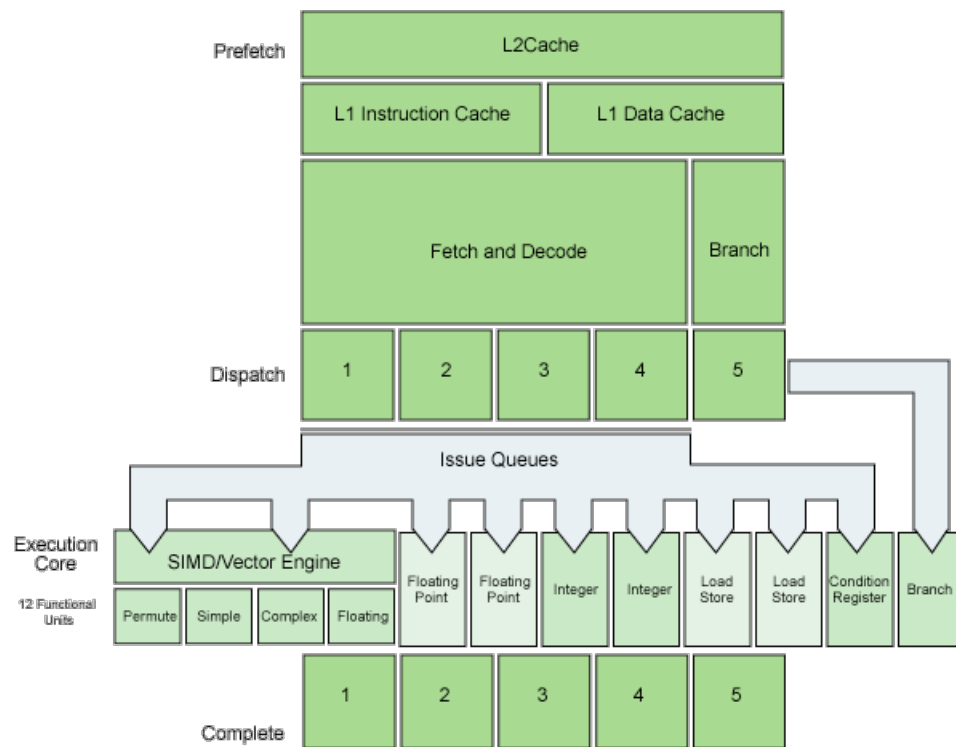
- 2.7GHz Power PC G5 (970fx)
- [http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/DC3D43B729FDAD2C00257419006FB955/\\$file/970FX_user_manual.v1.7.2008MAR14_pub.pdf](http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/DC3D43B729FDAD2C00257419006FB955/$file/970FX_user_manual.v1.7.2008MAR14_pub.pdf)
- Caches: 128 Byte line size
 - ♦ 512KB L2 (8-way, 12 CP hit time)
 - ♦ 32K L1 (2-way, 2 CP hit time)
- TLB: 1024 entries, 4-way
- gcc version 4.0.1 (Apple Computer, Inc. build 5370), -O2 optimization
- Single precision floating point

The results

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    a[i][j] += b[i][j];
```

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    a[i][j] += b[i][j];
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15



N	IJ (ms)	JI (ms)
64	0.007	0.007
128	0.027	0.083
512	1.1	37
1024	4.9	284
2048	18	2,090

Blocking for Cache

Matrix Multiplication

- An important core operation in many numerical algorithms
- Given two *conforming* matrices A and B , form the matrix product $A \times B$
 - A is $m \times n$
 - B is $n \times p$
- Operation count: $O(n^3)$ multiply-adds for an $n \times n$ square matrix
- Discussion follows from Demmel

www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect02.html

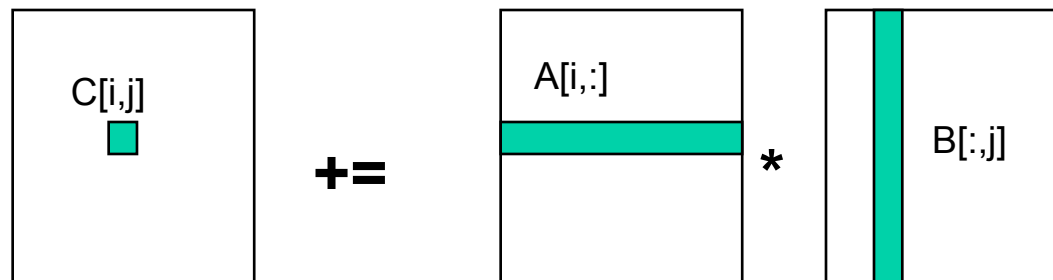
Unblocked Matrix Multiplication

for $i := 0$ **to** $n-1$

for $j := 0$ **to** $n-1$

for $k := 0$ **to** $n-1$

$C[i,j] += A[i,k] * B[k,j]$



Analysis of performance

for $i = 0$ to $n-1$

// for each iteration i , load all of B into cache

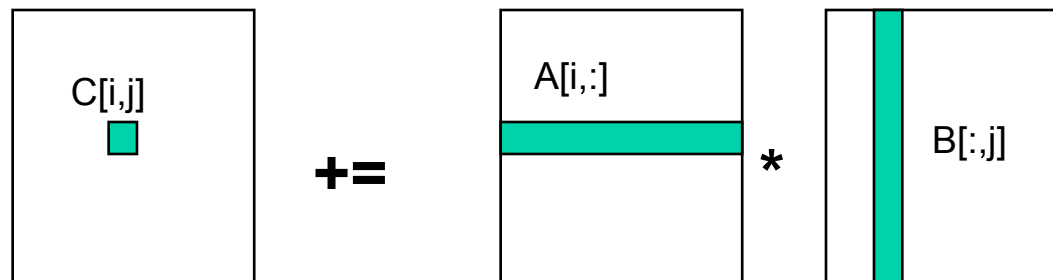
for $j = 0$ to $n-1$

// for each iteration (i,j) , load $A[i,:]$ into cache

// for each iteration (i,j) , load and store $C[i,j]$

for $k = 0$ to $n-1$

$$C[i,j] += A[i,k] * B[k,j]$$



Analysis of performance

for $i = 0$ to $n-1$

// $n \times n^2 / L$ loads = n^3/L , L =cache line size $B[:,:]$

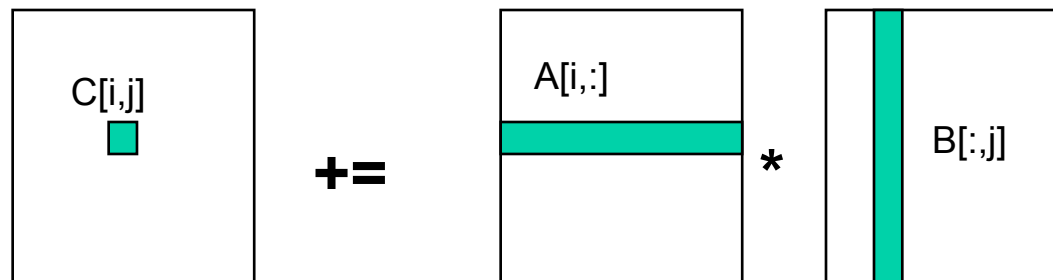
for $j = 0$ to $n-1$

// n^2 / L loads = n^2/L $A[i,:]$

// n^2 / L loads + n^2 / L stores = $2n^2 / L$ $C[i,j]$

for $k = 0$ to $n-1$

$C[i,j] += A[i,k] * B[k,j]$ **Total: $(n^3 + 3n^2) / L$**



Flops to memory ratio

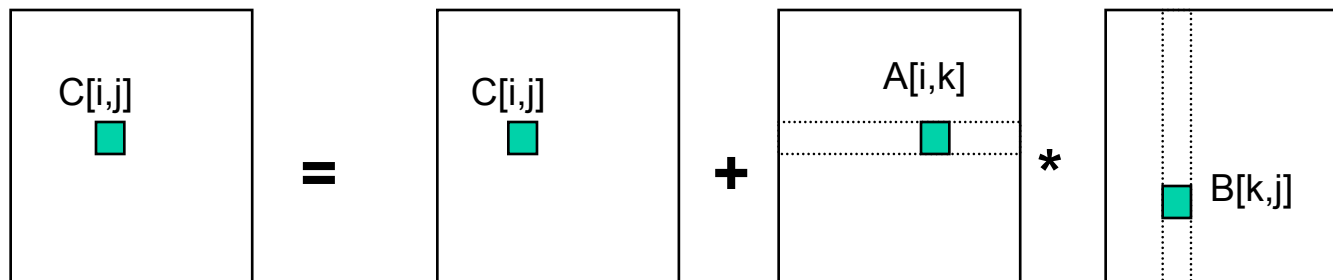
Let $q = \# \text{ flops} / \text{main memory reference}$

$$q = \frac{2n^3}{n^3 + 3n^2}$$

$$\approx 2 \text{ as } n \rightarrow \infty$$

Blocked Matrix Multiply

- Divide A, B, C into $N \times N$ sub blocks
- Assume we have a good quality library to perform matrix multiplication on subblocks
- Each sub block is $b \times b$
 - ♦ $b=n/N$ is called the **block size**
 - ♦ How do we establish b?



Blocked Matrix Multiplication

for i = 0 to N-1

for j = 0 to N-1

// load each block C[i,j] into cache, once : n^2

// b = n/N = block size

for k = 0 to N-1

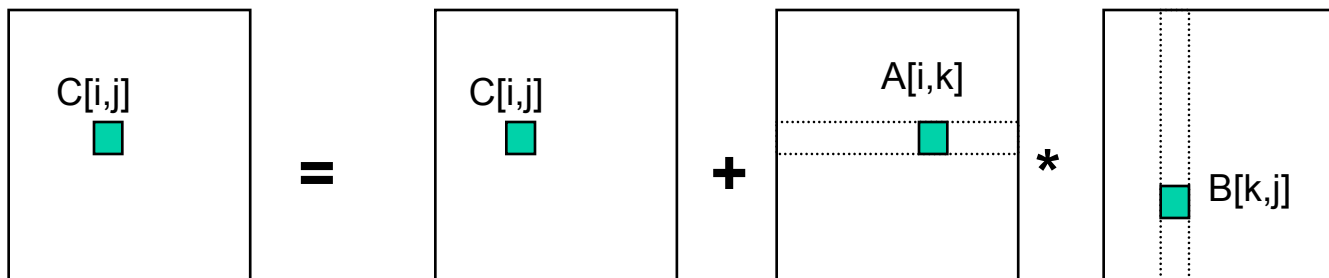
// load each block A[i,k] and B[k,j] N^3 times

// = $2N^3 \times (n/N)^2 = 2Nn^2$

C[i,j] += A[i,k] * B[k,j] // do the matrix multiply

// write each block C[i,j] once : n^2

Total: $(2*N+2)*n^2$



Flops to memory ratio

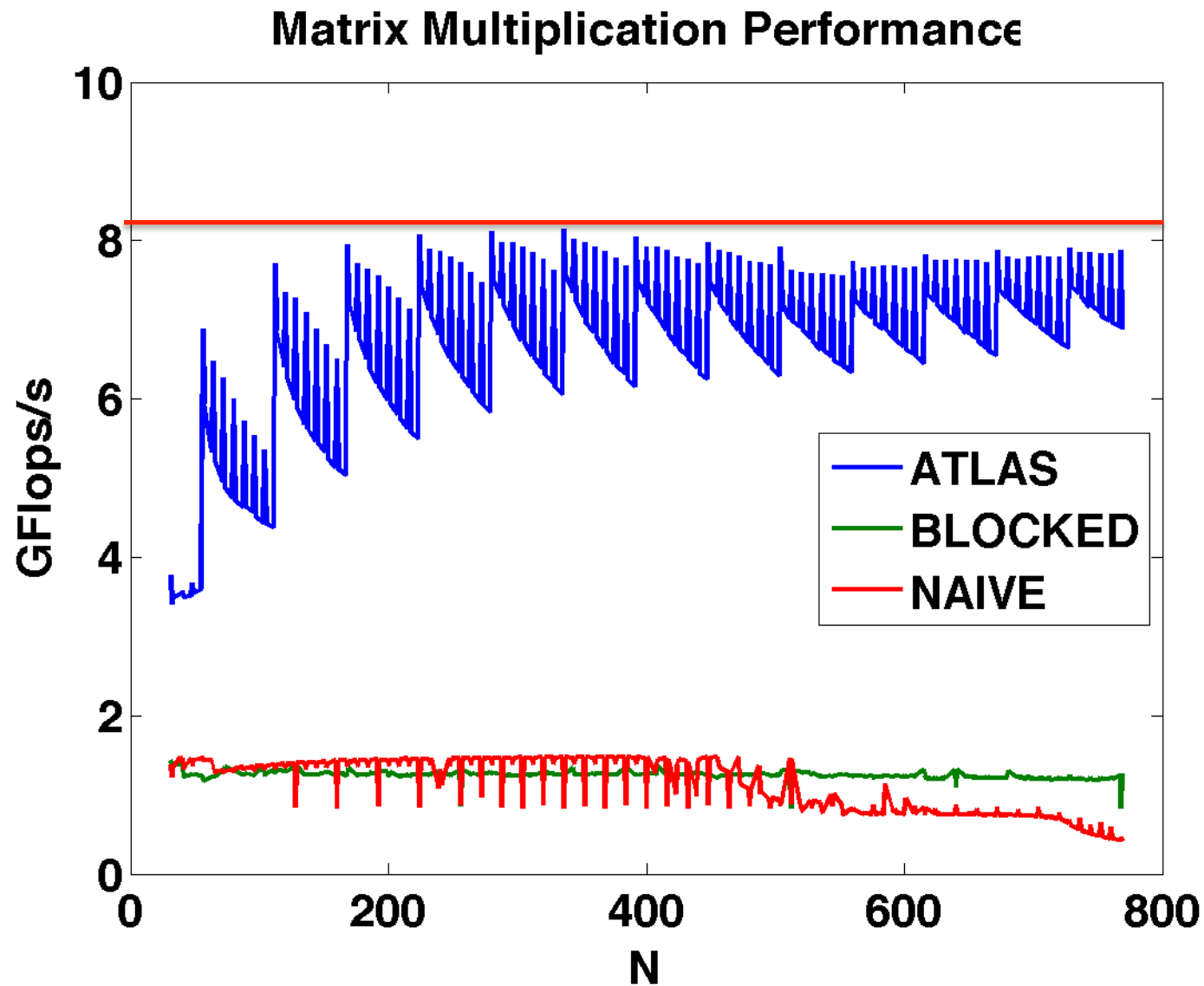
Let $q = \# \text{ flops} / \text{main memory reference}$

$$q = \frac{2n^3}{(2N+2)n^2} = \frac{n}{N+1}$$

$$\approx n/N = \mathbf{b}$$

as $n \rightarrow \infty$

The results



**8.14
GFlops**

$R_{\infty} = 4 * 2.33 =$
9.32 Gflops
~87% of peak

Programming Assignment #1

- Implement high performance matrix multiplication
- Provided code
 - ♦ Single level blocked matrix multiplication code
 - ♦ Reference: calls highly optimized matrix multiply,, provided by ATLAS
- Further optimization required
 - ♦ 2 level blocking
 - ♦ SSE (vectorization, next week)
 - ♦ Compiler optimizations
 - ♦ Other hand optimizations may be needed
- Will post the results

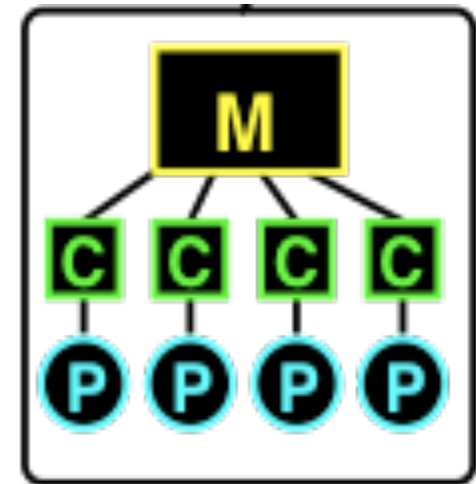
Address space organization Control

Address Space Organization

- We classify the address space organization of a parallel computer according to whether or not it provides global memory
- When there is a global memory we have a “shared memory” or “shared address space” architecture
 - ♦ *multiprocessor* vs *partitioned global address space*
- Where there is no global memory, we have a “shared nothing” architecture, also known as a *multicomputer*

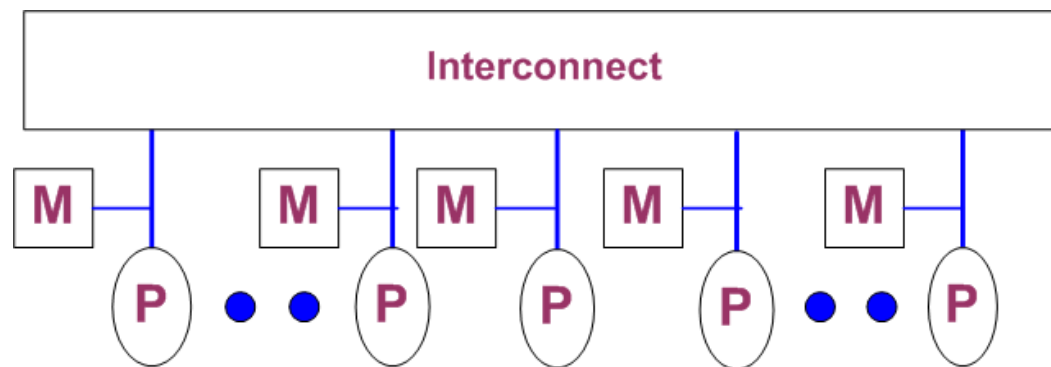
Multiprocessor organization

- The address space is global to all processors
- Hardware automatically performs the global to local mapping using address translation mechanisms
- Two types, according to the uniformity of memory access times
- **UMA: Uniform Memory Access time**
 - ♦ In the absence of contention all processors observe the same memory access time
 - ♦ Also called **Symmetric Multiprocessors**
 - ♦ Usually bus based



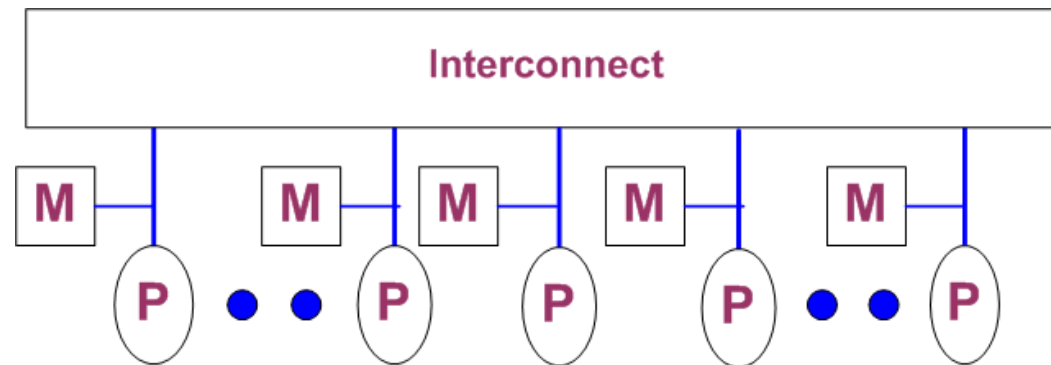
NUMA

- Non-Uniform Memory Access time
 - ♦ Processors see distant-dependent access times to memory
 - ♦ Implies physically distributed memory
- We often call these *distributed shared memory architectures*
 - ♦ Commercial example: SGI Origin Altix, up to 512 cores
 - ♦ Elaborate interconnect with a directory structure to monitor sharers



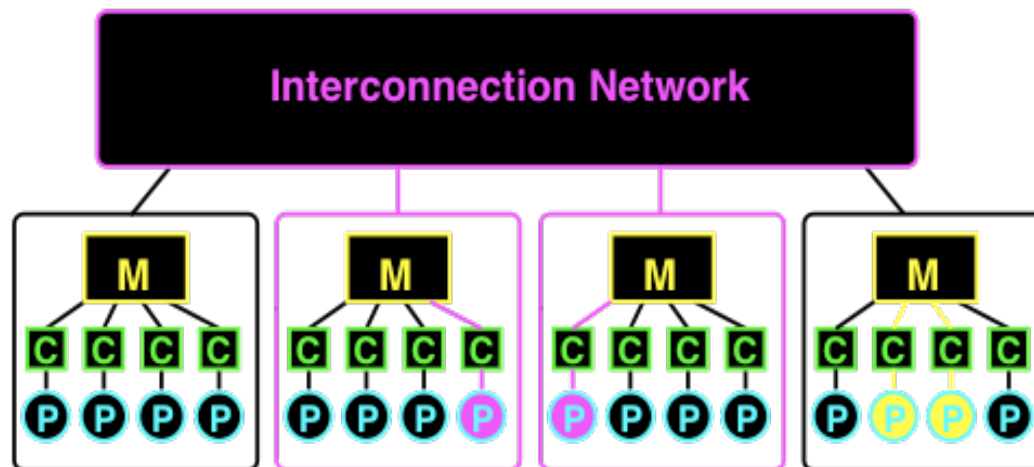
Architectures without shared memory

- Each processor has direct access to local memory only
- Send and receive messages to obtain copies of data from other processors
- We call this a *shared nothing* architecture, or a *multicomputer*



Hybrid organizations

- Multi-tier organizations are hierarchically organized
- Each node is a multiprocessor, usually and SMP
- Nodes communicate by passing messages, processors within a node communicate via shared memory
- All clusters and high end systems today



Today's Lecture

- Memory hierarchies
- Address space organization
 - ♦ Shared memory
 - ♦ Distributed memory
- **Control mechanisms**
- Shared memory hierarchy
 - ♦ Cache coherence
 - ♦ False sharing

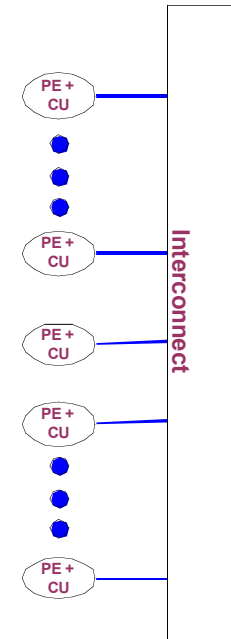
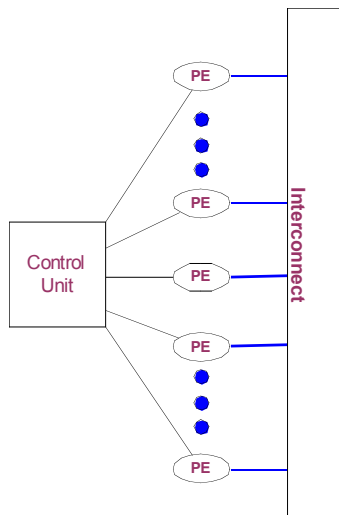
Control Mechanism

Flynn's classification (1966)

How do the processors issue instructions?

SIMD: Single Instruction, Multiple Data

Execute a global instruction stream in lock-step



MIMD: Multiple Instruction, Multiple Data
Clusters and servers processors execute
instruction streams independently

SIMD (Single Instruction Multiple Data)

- Operate on regular arrays of data
- Two landmark SIMD designs
 - ♦ ILIAC IV (1960s)
 - ♦ Connection Machine 1 and 2 (1980s)
- Vector computer: Cray-1 (1976)
- Intel and others support SIMD for multimedia and graphics
 - ♦ SSE
 - Streaming SIMD extensions, AltiVec
 - ♦ Operations defined on vectors
- GPUs, Cell Broadband Engine
- Reduced performance on data dependent or irregular computations

$$\begin{bmatrix} 2 \\ 4 \\ 8 \\ 7 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 5 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \\ 5 \\ 2 \end{bmatrix}$$

```
forall i = 0 : n-1
    x[i] = y[i] + z [ K[i] ]
end forall
```

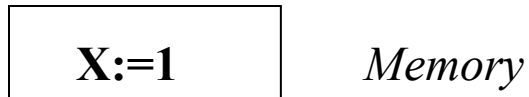
```
forall i = 0 : n-1
    if ( x[i] < 0) then
        y[i] = x[i]
    else
        y[i] = √x[i]
    end if
end forall
```

Today's Lecture

- Memory hierarchies
- Address space organization
 - ♦ Shared memory
 - ♦ Distributed memory
- Control mechanisms
- **Shared memory hierarchy**
 - ♦ Cache coherence
 - ♦ False sharing

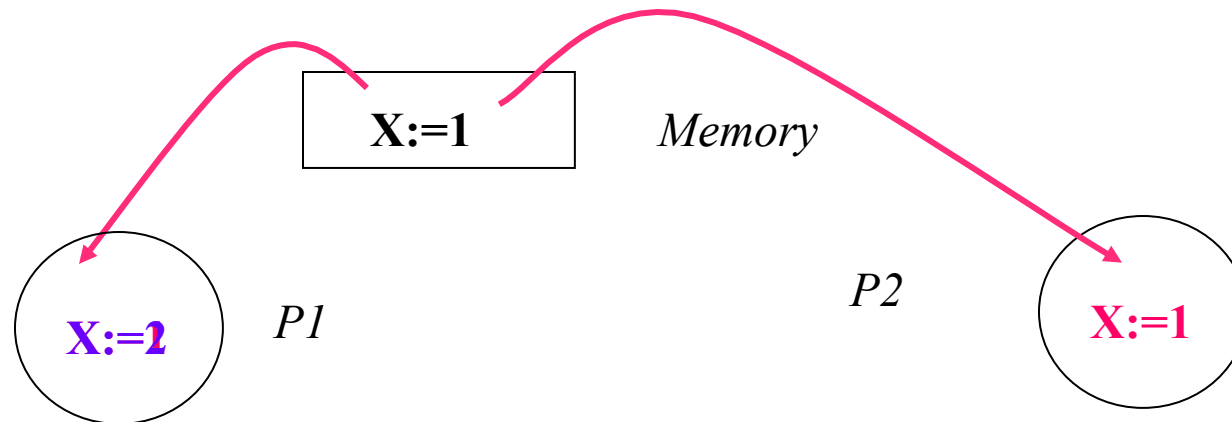
Cache Coherence

- A central design issue in shared memory architectures
- Processors may read and write the same cached memory location
- If one processor writes to the location, *all* others must *eventually* see the write



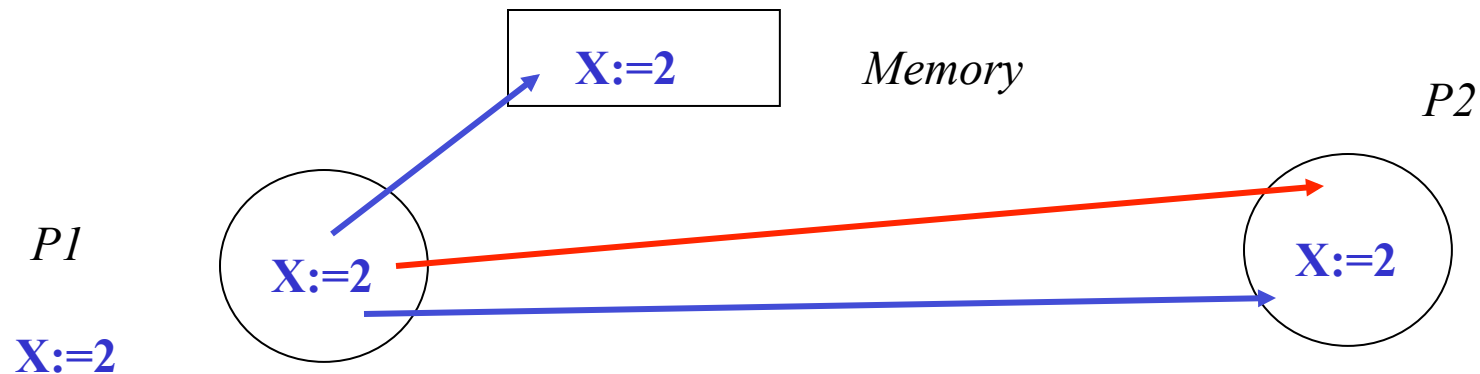
Cache Coherence

- 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



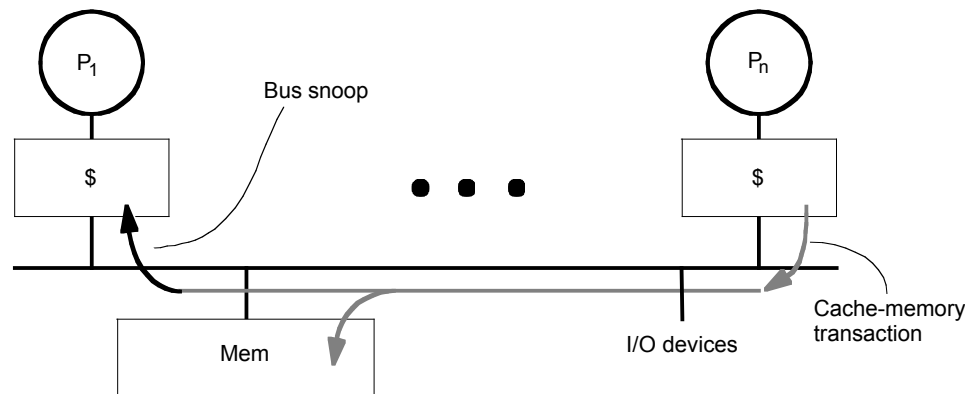
Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
 - ♦ Update-on-write (implies a write-through cache)
 - ♦ Invalidate-on-write



SMP architectures

- Employ a *snooping protocol* to ensure coherence
- Processors listen to bus activity



Memory consistency and correctness

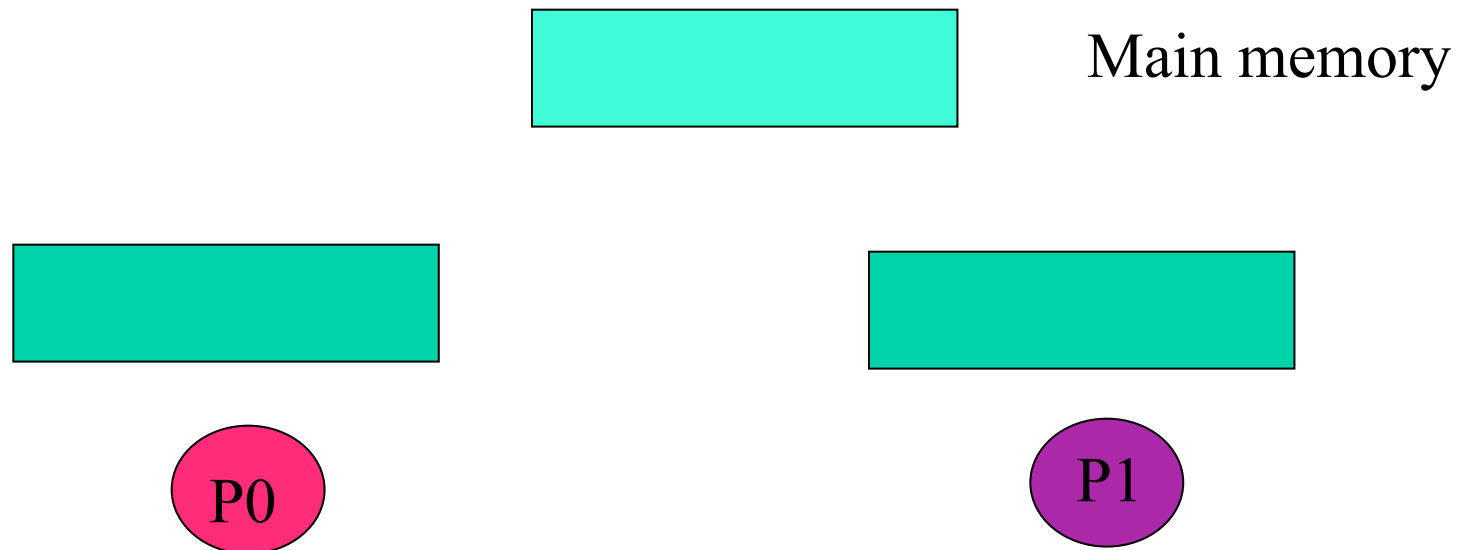
- Cache coherence tells us that memory will *eventually* be consistent
- The memory consistency policy tells us *when* this will happen
- Even if memory is consistent, changes don't propagate instantaneously
- These give rise to correctness issues involving program behavior

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

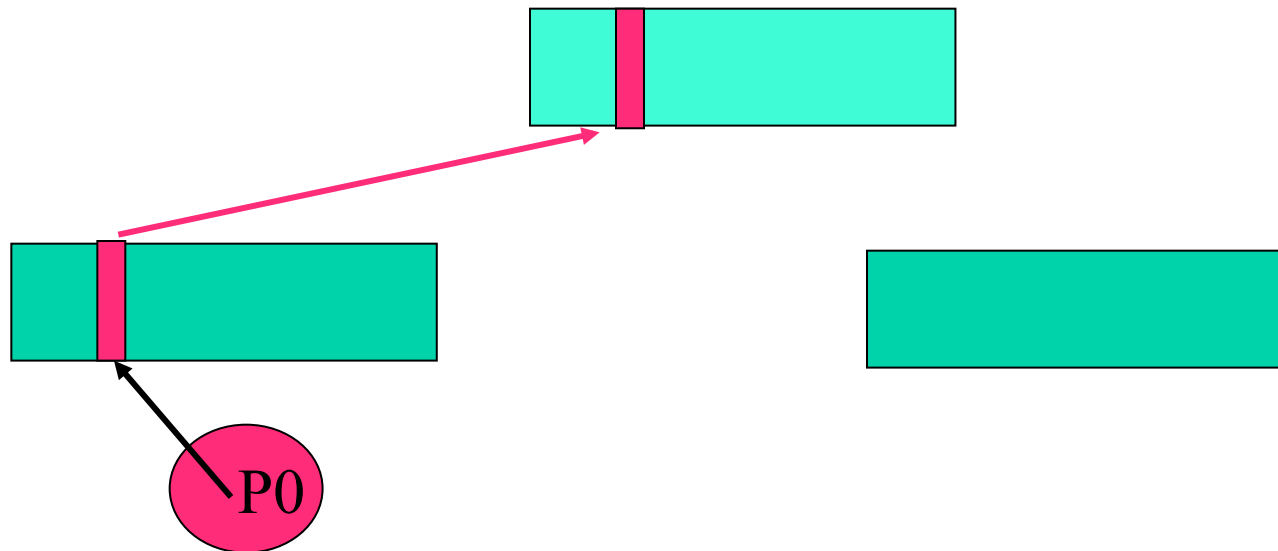
False sharing

- Consider two processors that write to different locations mapping to different parts of the same cache line



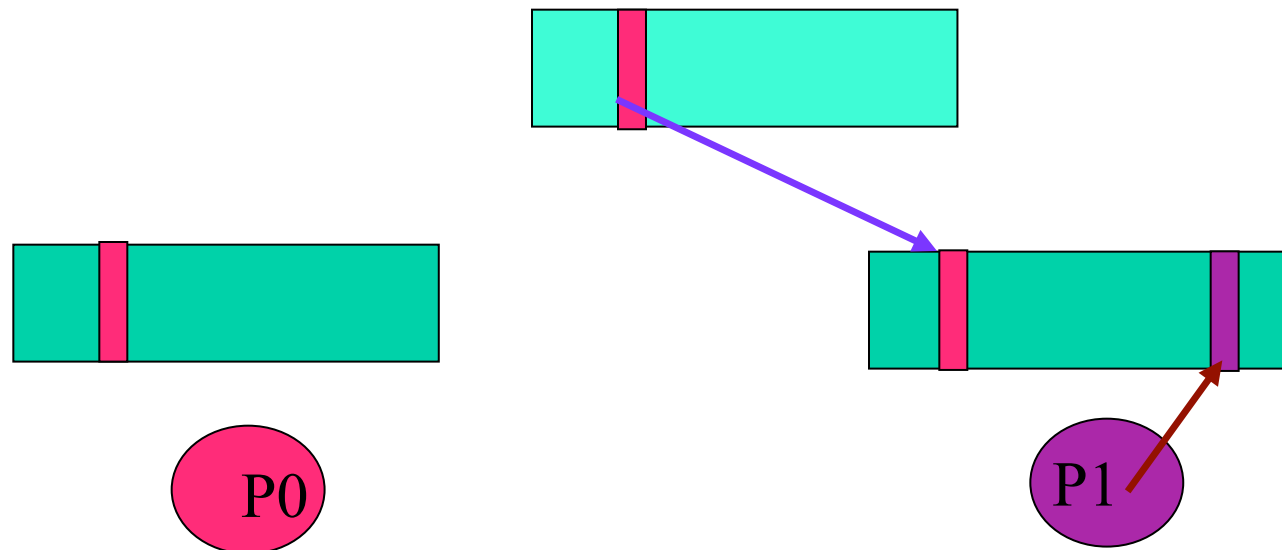
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated



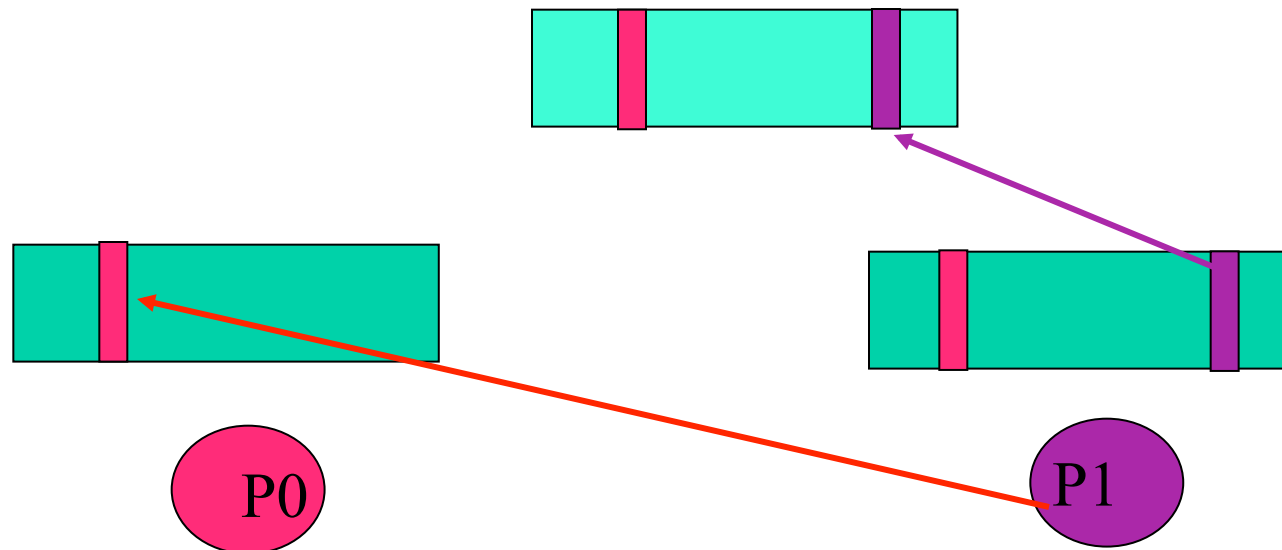
False sharing

- P1 reads the location written by P0
- P1 then writes a different location in the same block of memory



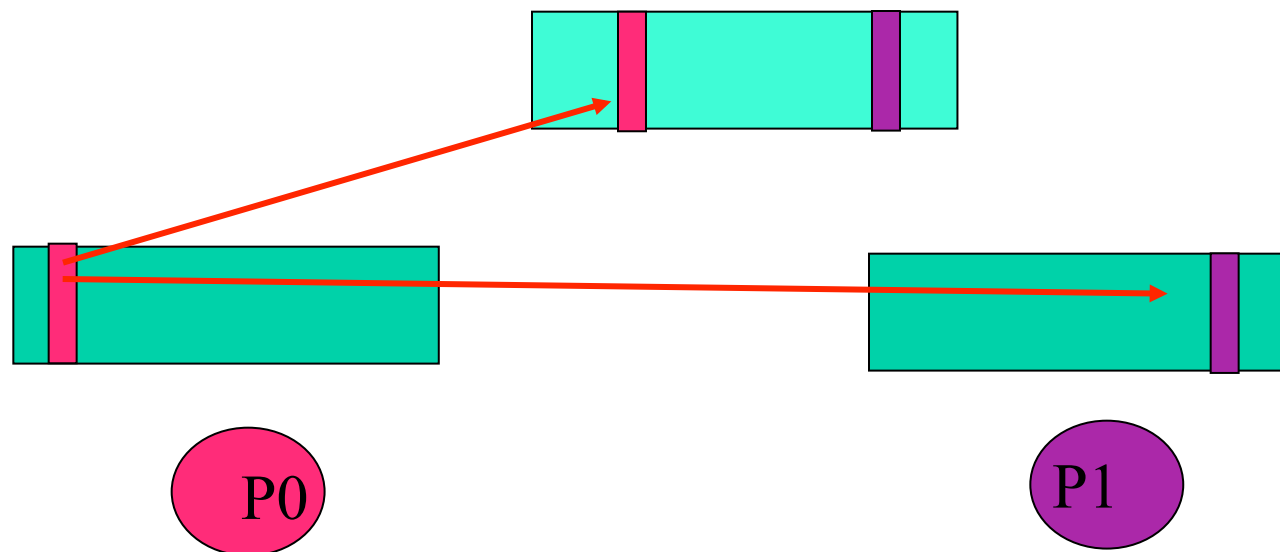
False sharing

- P1's write updates main memory
- Snooping protocol invalidates the corresponding block in P0's cache



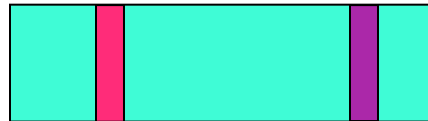
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another's cache



Eliminating false sharing

- Cleanly separate locations updated by different processors
 - ◆ Manually assign scalars to a pre-allocated region of memory using pointers
 - ◆ Spread out the values to coincide with a cache line boundaries



- Reduce number of accesses to shared state
- False sharing occurs a small fixed number of times

```
static int counts[];  
for (int k = 0; k < reps; k++)  
    for (int r = first; r <= last; ++ r)  
        if ((values[r] % 2) == 1)  
            counts[TID]++;
```

4.7s, 6.3s, 7.9s, 10.4 [NT=1,2,4,8]

```
int _count = 0;  
for (int k = 0; k < reps; k++) {  
    for (int r = first; r <= last; ++ r)  
        if ((values[r] % 2) == 1)  
            _count++;  
    counts[TID] = _count;  
}
```

3.4s, 1.7s, 0.83, 0.43 [NT=1,2,4,8]

- Put each counter in its own cache line

Spreading

0	1	2	3	4	5	6	7	...	31
0	1	2	3	4	5	6	7	...	31
0	1	2	3	4	5	6	7	...	31
0	1	2	3	4	5	6	7	...	31

```
static int counts[];
for (int k = 0; k < reps; k++)
    for (int r = first; r <= last; ++ r)
        if ((values[r] % 2) == 1)
            counts[TID]++;
```

```
static int counts[][LINE_SIZE];
for (int k = 0; k < reps; k++)
    for (int r = first; r <= last; ++ r)
        if ((values[r] % 2) == 1)
            counts[TID][0]++;
```

	NT=1	NT=2	NT=4	NT=8
Unoptimized	4.7 sec	6.3	7.9	10.4
Optimized	4.7	5.3	1.2	1.3

Cache performance bottlenecks in nearest neighbor computations

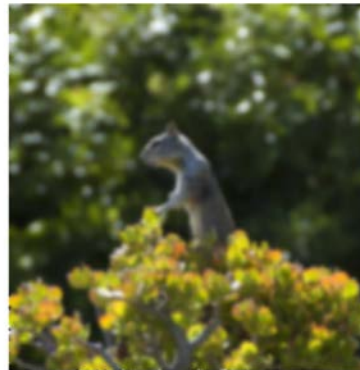
- Recall the image smoothing algorithm

for (i,j) in 0:N-1 x 0:N-1

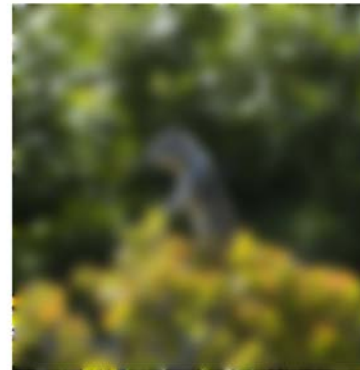
$$I^{\text{new}}[i,j] = (I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1])/4$$



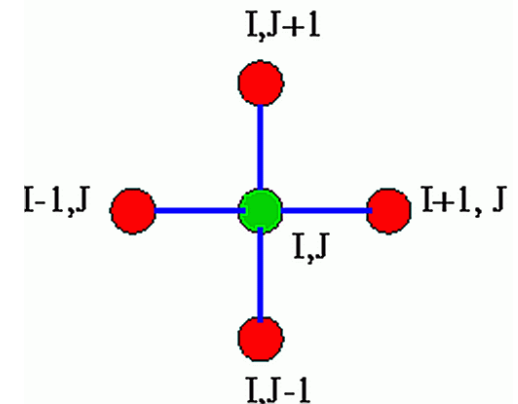
Original



100 iter



1000 iter

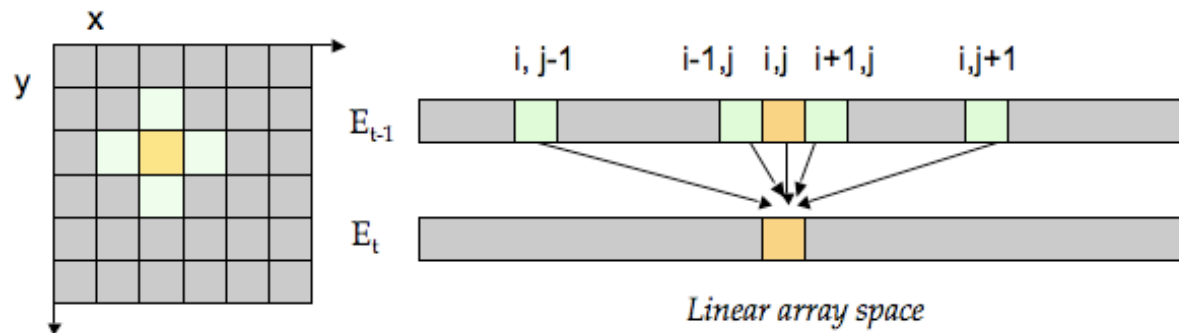


Memory access pattern

- Some nearest neighbors in space are far apart in memory
- Stride = N along the vertical dimension

for (i,j) in 0:N-1 x 0:N-1

$$I^{\text{new}}[i,j] = (I[i-1,j] + I[i+1,j] + I[i,j-1] + I[i,j+1])/4$$



False sharing and conflict misses

- False sharing involves internal boundaries, poor spatial locality, cache line internally fragmented
- Large memory access strides: conflict misses, poor cache locality
- Even worse in 3D: large strides of N^2
- Contiguous access on a single processor

