Cache Coherence

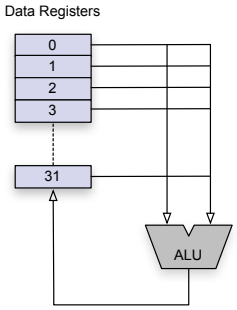
Overview of cache designs
Cache coherence in SMP systems
Performance studies

Reading

- Chapter 8 of the Wilkinson and Allen book
  - chapter is on shared memory programming
  - section 8.6 is on performance issues
- Books on computer architecture
  - *Computer Organization and Design: The Hardware/Software Inteface*
    Patterson and Hennessey
    (CIS 314)
  - *Computer Architecture: A Quantitative Approach*
    Hennessey and Patterson
    (CIS 629)

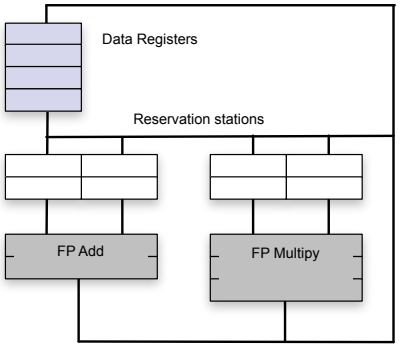
Inside a CPU

- The heart of a CPU is the data path
  - small set of data items stored in registers
  - arithmetic/logic unit does the calculations
- The data path is controlled by a clock
  - the simple design at right produces one result per cycle



Inside a CPU (cont'd)

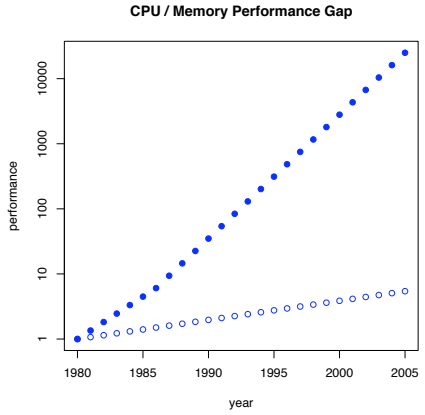
- More complex designs have multiple data processing units
  - integer operations
  - floating point (FP)
  - others
- Data units may be pipelined
  - instruction level parallelism
  - overlap instances of basic operation



Main Memory

- Memory (for instructions and data) is stored separately
  - would be nice to have 1GB of register storage, but...
  - data and instructions fetched as needed
- CPU performance has improved much more dramatically than memory performance since 1980

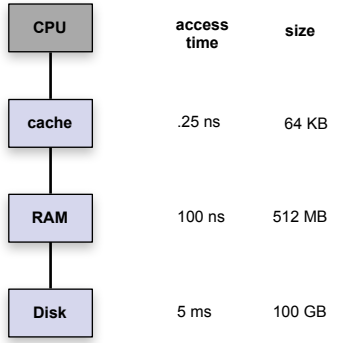
Note log scale...



H&P Fig. 5.2

Memory Hierarchy

- Computers have several different types of memories
- Memory closer to the CPU is
  - faster
  - smaller
  - more expensive



Times, sizes ca. 2001

Locality of Reference

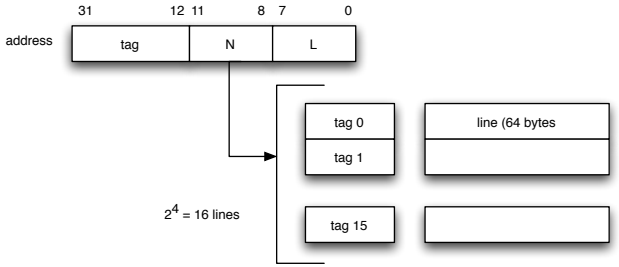
- Efficient implementation of a memory hierarchy relies on two common properties of applications
- *Temporal locality*
  - if an item is used once, it is very likely to be used again in the near future
- *Spatial locality*
  - if item x is used, items x+1, x+2, ... are likely to be used in the near future
- Applies to both instructions and data
  - loops, subroutines
  - vectors, arrays
  - trees? lists? objects?

Terminology

hit	CPU accesses an item currently in the cache
miss	Requested item is not currently in the cache
line	A block of items moved between main memory and cache
line size	Number of bytes in a line
directory	Device used to locate an item in cache (or let CPU know the item needs to be fetched)

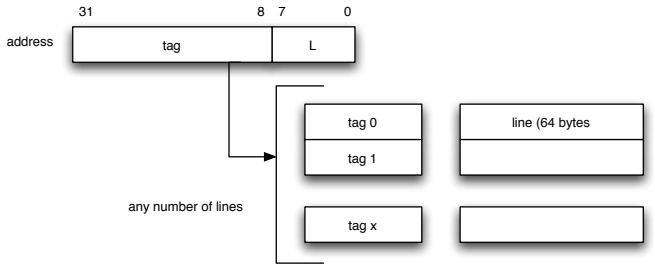
Cache Directory (I)

- The main responsibility of a cache directory: locating items in the cache
  - when the CPU requests item X, the directory needs to determine if X is present
- Direct-mapped cache: use function f(X) of address of X to locate line



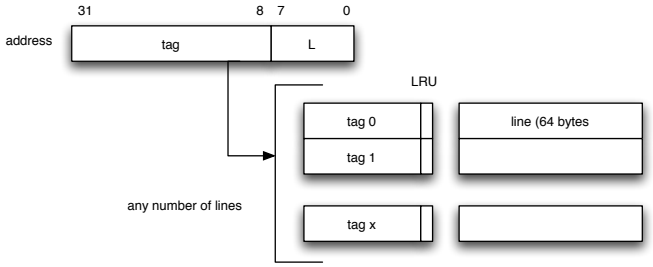
Cache Directory (cont'd)

- An alternative to a direct-mapped cache is an associative cache
  - the directory is a small associative memory
    - given address X, find the location i that contains X
  - lines can be anywhere in cache -- fewer collisions, better hit rate
  - set-associative: collection of small (2-8 element) associative sets



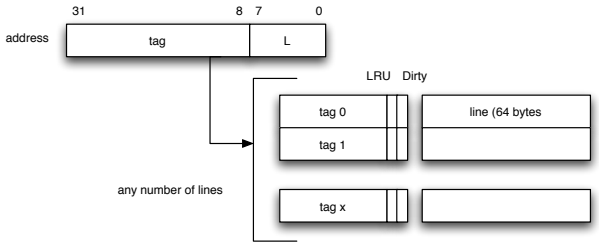
Cache Directory (II)

- Another decision made by the directory: replacement
  - if the cache is full when a new line arrives, which old line is thrown out?
  - direct-mapped: old line at f(X)
  - associative: random, or least recently used, or ...



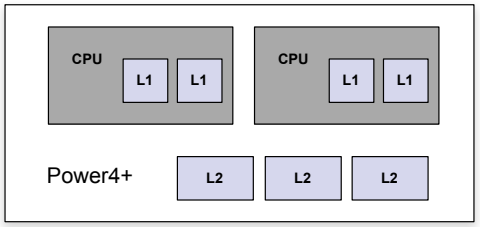
Cache Directory (III)

- Yet another set of design decisions: how to handle writes (store instructions)
- Write-through vs write-back
  - Update RAM immediately, or save all updates until line replaced?
- If the first reference to a line is a write instruction, should it be considered a miss?



Cache Structure of the IBM p690

- The p690 uses IBM PowerPC processor chips
  - two CPUs per chip (Power4+)
  - separate level 1 instruction and data caches within each CPU
  - shared level 2 cache on chip
  - level 3 cache (32MB) on a separate chip



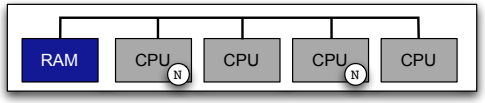
Line size:
128 bytes

L1: 64KB, 2-way SA

L2: 1.5MB, 8-way SA

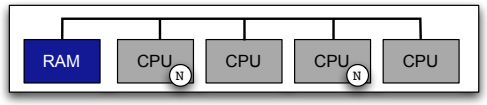
Coherence

- The big issue for shared memory multiprocessing: cache coherence
- Suppose several processes want to update a shared counter by executing `N += 1`
  1. CPU A: cache miss, `N` loaded to L1 cache
  2. CPU B: cache miss, `N` loaded to L1 cache
  3. CPU A: cache hit: update previous value of `N`...



Coherence (cont'd)

- Coherence is related to synchronization
  - previous lecture: need for critical regions in updates of shared variables
- Coherence is an issue even when programs are synchronized correctly
  - critical regions won't fix the problem
  - needs hardware support to detect multiple copies, make sure they are consistent across all caches
  - helped (but not solved) by write-through caches



Snoopy Cache

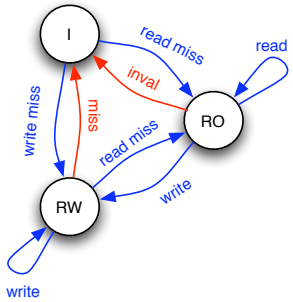
- A common method for implementing coherence in small scale SMPs is a "snoopy cache"
  - When a CPU updates the item at address X it puts X on the bus
  - Another CPU that has the line for X in its cache knows X has changed
- Two basic methods for handling writes to shared lines:
  - *write-invalidate*: all other CPUs that have X in their cache remove X from the directory; the next reference to X will be a miss
  - *write-update*: put X and the new value of X on the bus; all other CPUs replace their copies with the new value

Snoopy Cache (cont'd)

- Pros/cons of write-invalidate (WI) vs write-update (WU):
  - WU requires more overhead per write (broadcast address and data)
  - If one CPU updates more than once, WI needs only one transaction (first invalidate), but WU does one transaction per update
  - WI invalidates an entire line; in WU writes to neighboring words require additional transactions
  - WI causes more misses in other processors (with WU they have the new value, don't need to re-fetch the line)
- Bus bandwidth is the critical resource, and in general WI causes less bus traffic

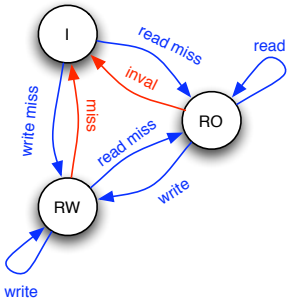
Implementation

- This is too much detail to absorb at once, but should give you a general sense of how the method works...
- The cache controller implements a small state machine for each line
- State transitions occur in response to CPU actions (blue) or actions seen on the bus (red)
  - I: invalid (no copy here)
  - RO: clean copy here
  - RW: updated copy here



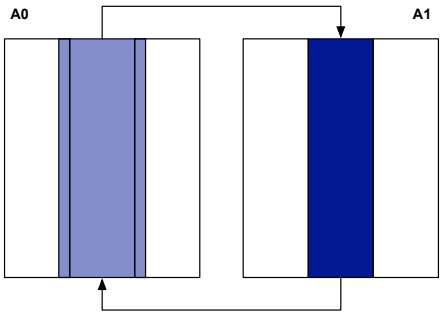
Implementation (cont'd)

- Each CPU has a write-back cache (i.e. it can accumulate updates, but has to send line back when requested)
- Examples:
  - read miss in I: fetch copy
  - write in RO: update copy
  - miss from bus in RW: write back copy
- p690 has variation of this scheme with 7 states (e.g. shared read-only vs single-copy read-only)



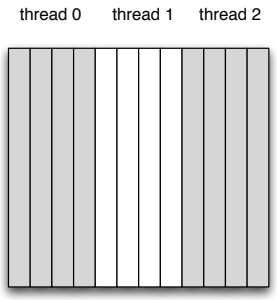
Applications

- The next few slides will examine the impact of cache coherence on SMP applications
  - array indexing
  - shared data structures
  - synchronization
  - false sharing
  - scalability
- Many applications are similar to your Laplace program, which successively updates two arrays



Array Indexing

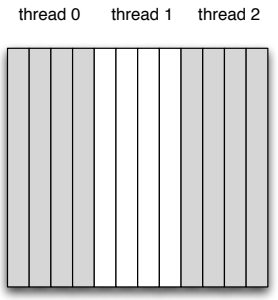
- Make sure loops that iterate over arrays know whether the arrays are allocated in row-major or column-major order
- For column-major (shown here):
  - `A[i+1,j]` is in the word immediately following `A[i,j]`
  - a miss to `A[i,j]` loads `A[i+1,j]`, `A[i+2,j]`, ... into the cache
  - make sure your code takes advantage of this



Array Indexing (cont'd)

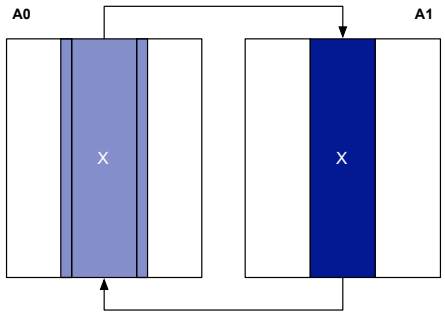
- Make sure parallel regions are organized as sets of columns
- The best loop structure for 2D column-major arrays has the column index (`j`) in the outer loop

```
for (int j = 0; ...) {
  for (int i = 0; ...) {
    ... A(i,j) ...
  }
}
```



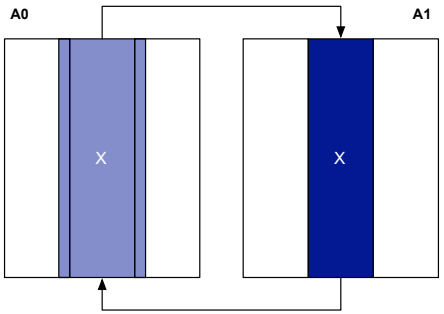
Shared Data Structures

- When filling A1, the program reads from corresponding cells in A0
- Each process also reads the adjacent columns in regions before, after its own region
- The process that fills region X has exclusive access to these cells in A1
- It has shared access to columns neighboring X in A0



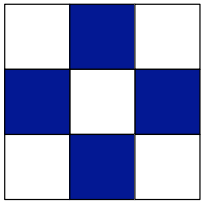
Shared Data Structures (cont'd)

- When filling A0:
  - get write access to the two border columns inside region X in A0
  - give up exclusive access to region X in A1
  - both involve rounds of invalidations and write-backs



Shared Data Structures (cont'd)

- What can you do?
  - arrange the "geometry" of the application to minimize the size of border regions
  - have each process do as much work as possible when it gains write access



Synchronization

- An example from the OpenMP slides: a set of processes cooperates to set a global variable `a` to the inner product of vectors `x` and `y`

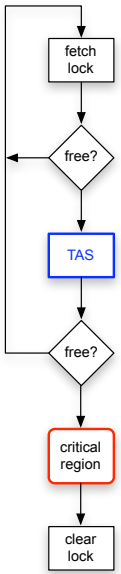
```
#pragma pfor iterate(T=0; 4; 1)
    for (T = 0; T < 3; T++) {
       for (i = T*250; i < (T+1)*250; i++)
          ax[T] += x[i] * y[i];
       #pragma critical
       a += ax[T]
    }
```

- This example uses four local counters (elements of `ax`) to store partial results, then thread `i` adds `ax[i]` to a in a critical region

Synchronization (cont'd)

- To implement a critical region, the OpenMP library uses a "spin lock"
  - a single shared integer
  - 1 = region in use
  - 0 = region free
- TAS = test and set
  - atomic operation

    ```
    tas $r, x
    ```
  - loads old value of `x` into `$r`
  - sets `x` to 1



P&H Fig 9.6

Synchronization (cont'd)

- Bus and cache operations as three processors use a lock (P&H 9.7)

T	P0	P1	P2	Bus
1	has lock	spinning	spinning	none
2	sets lock to 0	spinning	spinning	WI from P0
3		cache miss	cache miss	access to P2
4		waits for bus	fetches lock	cache line to P2
5		fetches lock	TAS sets lock	cache line to P1
6		TAS sets lock	lock to bus	WI from P2
7		lock to bus	owns lock	WI from P1
8		spinning		

False Sharing

- Back to the inner product example:

```
#pragma pfor iterate(T=0; 4; 1)
    for (T = 0; T < 3; T++) {
       for (i = T*250; i < (T+1)*250; i++)
          ax[T] += x[i] * y[i];
       #pragma critical
       a += ax[T]
    }
```

- There is actually a subtle problem in this program: every write to `ax` triggers a write-invalidate!

False Sharing (cont'd)

- With four threads, the temporary counter vector is defined as

```
int ax[4];
```

- All elements of `ax` are in the same cache line
- Even though each thread accesses a different element on the line, the entire line is invalidated and updated with the write-invalidate protocol described here
- What to do:
  - watch out for false sharing
  - use `#pragma omp reduce` and other library operations

Scalability

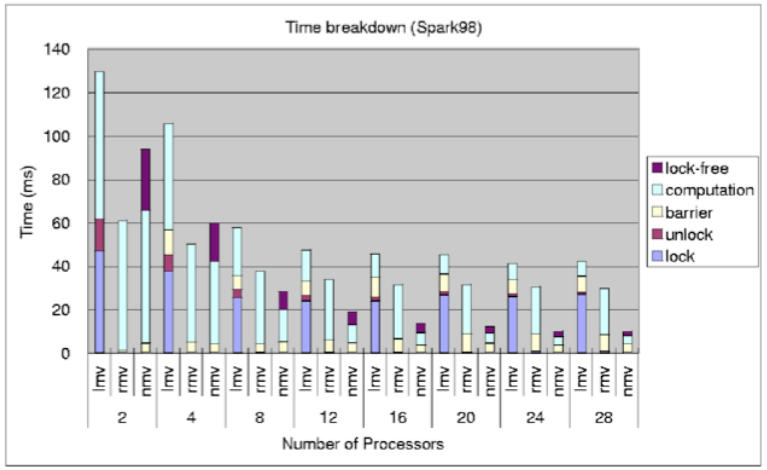
- All of the factors described on the previous slides limit the scalability of shared memory programs
  - access to boundary regions in shared data structures
  - spin locks and other mechanisms for implementing synchronization
  - false sharing
- One early study (1996) measured performance on small systems (16 CPUs), used simulator to predict speedups for up to 256 CPUs
- After extensive rearrangement of application code they were able to get up to 70% efficiency for most applications
  - data layout
  - load balancing

$$E = (S_n/S_1)/n$$

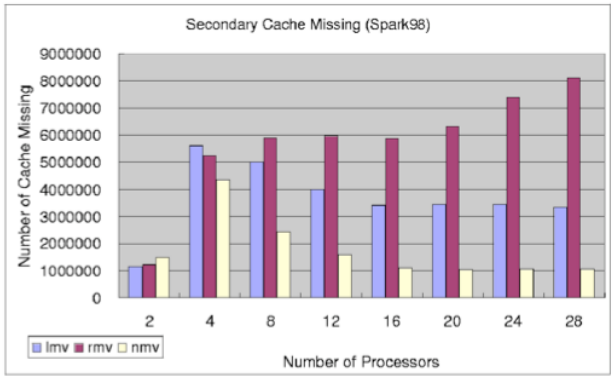
Scalability (cont'd)

- Some results from a recent paper describing a new method for synchronization
  - P. Tsigas and Y. Zhang. The non-blocking programming paradigm in large scale scientific computations. PPAM'03, *LNCS 2019* 1114-24.
- Measured performance of two different programs on an SGI Origin with 28 CPUs
  - a sparse matrix kernel (SPARK98)
    - 3 versions: lock, reduce, non-blocking
  - a 3D volume rendering application
    - 2 versions: lock, non-blocking

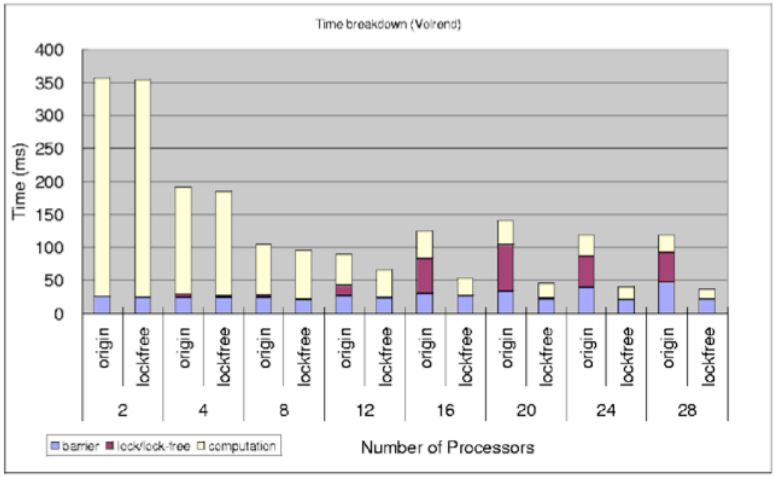
SPARK98



SPARK98 (cont'd)

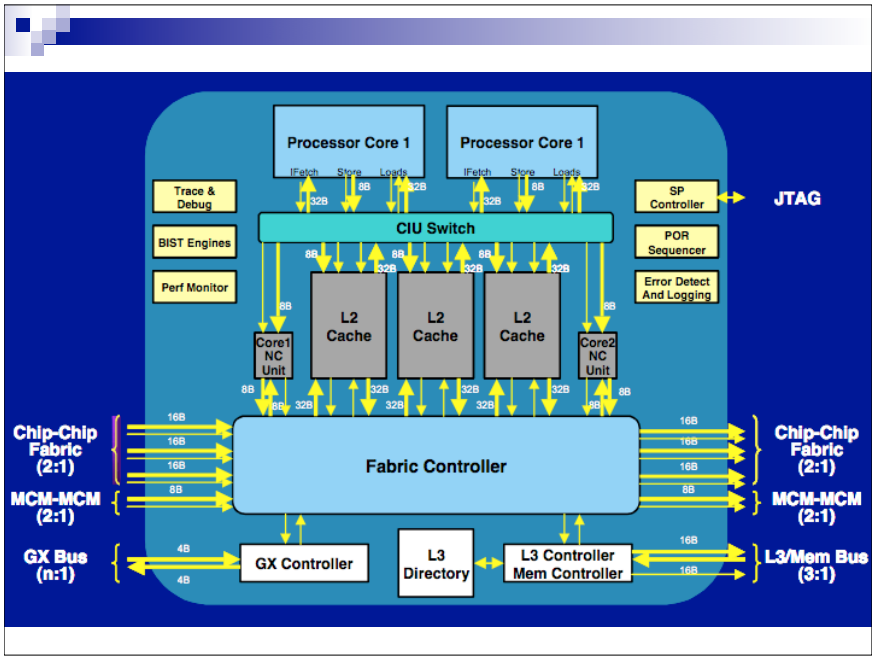


Volume Rendering



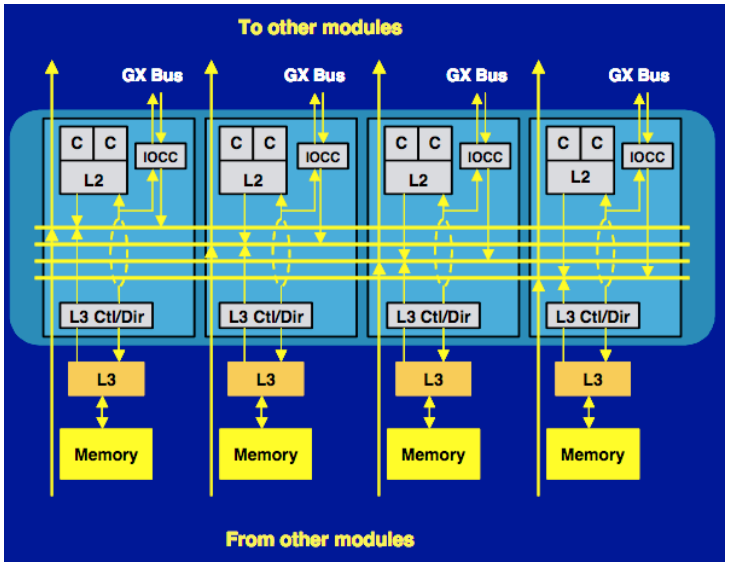
The CPU Chips in the p690

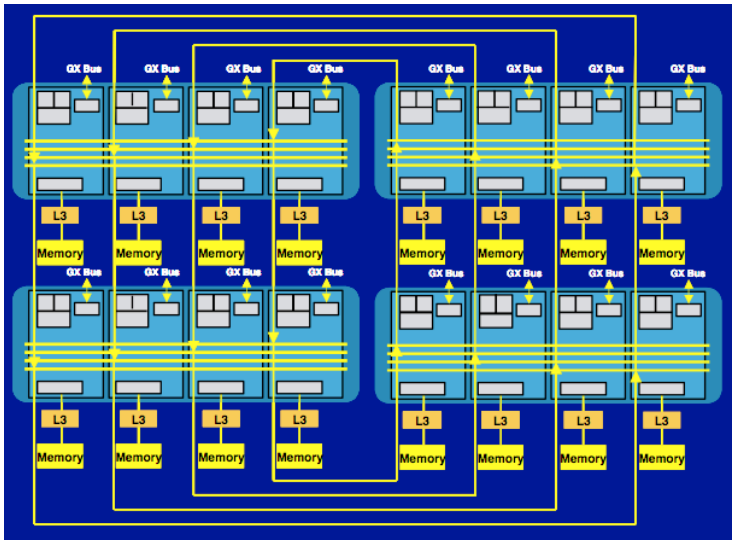
- The processors in our IBM p690 are PowerPC
  - 1.3GHz
  - Dual core (two independent CPUs per chip)
  - Each CPU has its own level one (L1) cache
  - The CPUs share an L2 cache
  - The directory for the L3 cache is on the chip, but the memory is separate
- See the picture from IBM documentation on the next slide



p690 SMP Interconnections

- The next two slides show 4-chip (8 CPU) and 16-chip (32 CPU) configurations of the p690
- An 8-processor module uses a local bus to connect the L2 caches
- Uses a variation of the basis write-invalidate protocol
- Connect 4 8-CPU modules in a ring to implement a 32-CPU system





Analyzing Performance

- For your own programs:
  - many processor chips now have programmable counters
  - count instructions issued, clock cycles, L1 misses, ...
  - find C/C++ library for resetting the counters, reading their values, etc
  - use TAU and other libraries
- Write different versions of the application, measure performance