

# Shared Memory Architectures

Chris Kauffman

*Last Updated:  
Tue Mar 15 09:34:53 AM CDT 2022*

# Logistics

## Assignments + Grading

- ▶ Will combine A2 and A3 to do an MPI and OpenMP versions of K-Means clustering
- ▶ Post later this week - again, apologies for the delay
- ▶ Mini-Exam 1 and 2 grades will be up by next week

## Today

- ▶ Shared Memory Architecture Theory + Practicalities
- ▶ Cache Performance Effects
- ▶ Next Week: PThreads + OpenMP for shared memory machines

## Reading

- ▶ Grama 2.4.1 (PRAM), 2.4.6 (cache)
- ▶ Gram 7.1-9 (PThreads)
- ▶ Grama 7.10 (OpenMP)
- ▶ [OpenMP Tutorial at Laurence Livermore](#)

# PRAM: Parallel Random Access *Machine*, Grama Ch 2.4.1

## RAM: Random Access *Machine*

- ▶ An unfortunate name as every other use of “RAM” is random access memory
- ▶ Single CPU attached to random access memory
- ▶ Simplistic model for a real machine: CPU reads memory, performs operations in registers, writes to memory, repeats

## PRAM: Parallel Extension to RAM

- ▶ Again, theoretical model for a real parallel machine
- ▶ Multiple CPUs attached to memory, share clock but can execute different instructions
- ▶ Must clarify behavior of PRAM machine that is not possible in single PE situation: how are conflicts between processors resolve for **simultaneous memory access**

# Theoretical Flavors of PRAM

## Exclusive-Read, Exclusive-Write (EREW) PRAM

Multiple CPUs cannot touch same memory at all. No parallelism possible for reads / writes of the same memory location.

## Concurrent-Read, Exclusive-Write (CREW) PRAM

Multiple CPUs may read same memory location at same time. Writes to same location must be resolved.

## Exclusive-Read, Concurrent-Write (ERCW) PRAM

Multiple write accesses are allowed to a memory location, but multiple read accesses are serialized. (This is just weird)

## Concurrent-Read, Concurrent-Write (CRCW) PRAM

Simultaneous Read AND Write of the same memory location: the most “powerful” PRAM model for some definition of “power”.

*Q: What else must be specified for the xxCW models?*

# Resolution Schemes for Concurrent Reads/Writes

*A: How concurrent writes resolve.*

- ▶ **Common:** concurrent writes are allowed if all the values that the processors are attempting to write are identical.
- ▶ **Arbitrary:** an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- ▶ **Priority:** procs are organized in a predefined prioritized list; proc with the highest priority writes its value while others fail.
- ▶ **Sum:** the sum of all the quantities is written

Above categories do not resolve concurrent read+write such as:

MEM[#1024] is 10

P0 reads MEM[#1024] into REG1

P1 writes 20 to MEM[#1024]

Proper treatments of PRAM specify results for this such as

- ▶ All Reads resolve first, then Writes resolve
- ▶ Concurrent Reads/Writes occur in arbitrary order
- ▶ etc.

However, we'll proceed with more practical matters

# Pros and Cons of PRAM

## Why the PRAM Model?

- ▶ It's simple
- ▶ Much literature devoted to studying benefits of algorithms under different variants (e.g. *Parallel Array Sum with CRCW + summing on concurrent write*)
- ▶ Has significant theoretical importance

## Why Not PRAM

- ▶ No real, general machines currently implement any PRAM models
- ▶ Seen some references that GPUs might sort of implement but would require some more work
- ▶ Conclusions one might draw about “good” algorithms is skewed (e.g. *multi-core machines do NOT behave as a CRCW-summing machine; far from it*)

## Exercise: Recall the Memory Cache

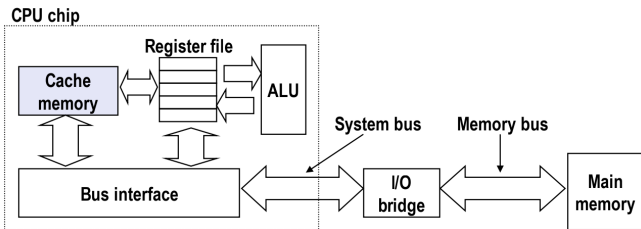
- ▶ Parallel programs are driven towards performance
- ▶ Optimize serial performance first: requires understanding of the memory hierarchy

### Questions

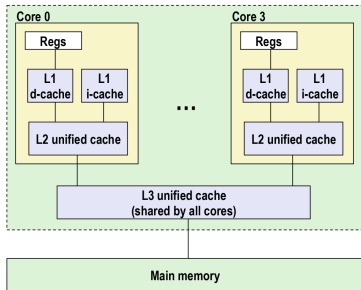
From your computer architecture experience...

- ▶ Describe a **memory cache** and why most CPUs have several layers of them
- ▶ Give an example of “strange” cache effects where similar algorithms have very different performance

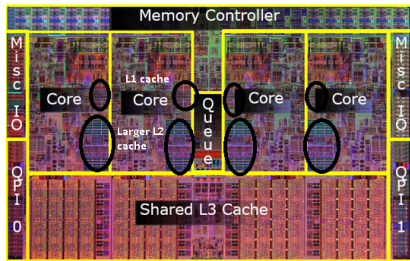
# Diagrams of Memory Interface and Cache Levels



Source: Bryant/O'Hallaron CS:APP 3rd Ed.



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition



Source: SO "Where exactly L1, L2 and L3 Caches located in computer?"



# Numbers Everyone (in Computing) Should Know

Edited Excerpt of [Jeff Dean's](#) talk on data centers.

Operation	Time (ns)
L1 cache reference	0.5
Branch mispredict	5
L2 cache reference	7
Mutex lock/unlock	100
Main memory reference	100
Compress 1K bytes with Zippy	10,000
Send 2K bytes over 1 Gbps network	20,000
Read 1 MB sequentially from memory	250,000
Round trip within same datacenter	500,000
Disk seek	10,000,000
Read 1 MB sequentially from network	10,000,000
Read 1 MB sequentially from disk	30,000,000
Send packet CA->Netherlands->CA	150,000,000

Numbers are likely out of date now but scales are worth knowing and explain why Cache is useful

# Matrix Summing Examples

## Sum R

```
double X[N][N]; // N by N mat
...
sum = 0;
for(i=0; i<N; i++){
    for(j=0; j<N; j++){
        sum += X[i][j]
    }
}
```

## Sum C

```
double X[N][N]; // N by N mat
...
sum = 0;
for(j=0; j<N; j++){
    for(i=0; i<N; i++){
        sum += X[i][j]
    }
}
```

- ▶ What's the Big O complexity of each?
- ▶ What happens with cache?
- ▶ Will one be faster than the other?

## Cache Affects Performance

As measured by hardware counters using linux's perf on

model name : Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz

cache size : 6144 KB

with

```
perf stat $opts java MatrixSums 8000 4000 row
```

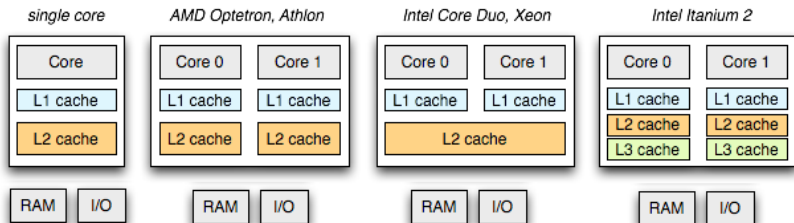
```
perf stat $opts java MatrixSums 8000 4000 col
```

Measurement	row	col
cycles	3,507,364,715	5,605,621,966
instructions	2,353,887,029	2,543,165,478
L1-dcache-loads	527,694,054	561,540,169
L1-dcache-load-misses	25,638,014	122,663,199
Runtime (seconds)	1.001	1.620

L1 data cache load misses

- ▶ Row:  $25\text{K}/548\text{K} = 4\%$  main memory access
- ▶ Col:  $122/585\text{K} = 20\%$  main memory access

# Cache Issues in Shard Memory Machines



Source: Multi-core, Threads & Message Passing by Ilya Grigorik

Consider the following sequence of operations:

```
// MEM[#1024] has value 5
P0: load R1 MEM[#1024] // slow, populates cache
P0: load R2 MEM[#1024] // fast, from cache
P0: ADD R1 R1 R2       // R1 is 10
P0: store R1 MEM[#1024] // cache dirty, MEM[#1024] unchanged
    a short time later
P1: load R3 MEM[#1024] // read 5 or 10?
```

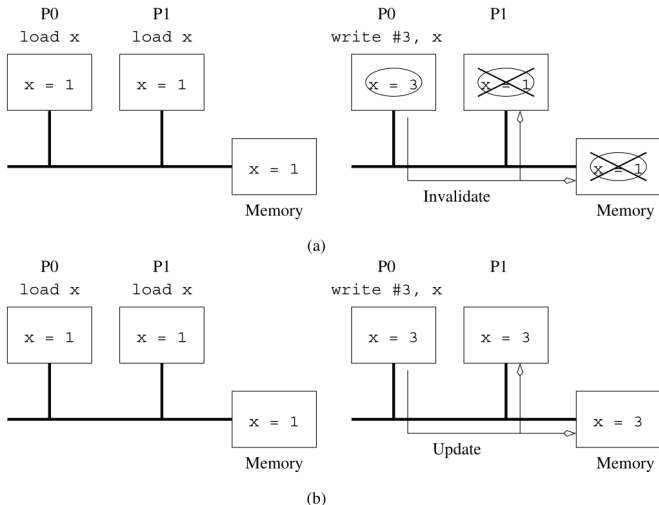
Illustrates **Cache Coherence** Problem: how do multiple PEs maintain the illusion of a single block of shared memory?

# Cache Coherence Protocols

Gramma 2.4.6 offers two theoretical protocols to maintain coherence

- ▶ **Invalidate:** shared memory written by one PE is marked as invalid in cache of others
- ▶ **Update:** shared memory written in by one PE is updated in other PEs

# Invalidate and Update Protocol Diagrams

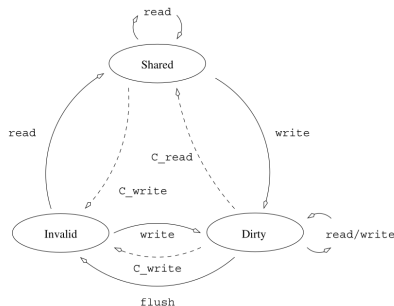


**Figure 2.21** Cache coherence in multiprocessor systems: (a) Invalidate protocol; (b) Update protocol for shared variables.

# Cache State and Coherence

Each element in the ProcX's cache is one of

- Shared** valid for to read/write
- Dirty** written by me, must eventually write to main memory
- Invalid** someone else wrote it in their cache, must reload



**Figure 2.22** State diagram of a simple three-state coherence protocol.

# Demonstration

Time  
↓

Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.
				x = 5, D y = 12, D
read x	read y	x = 5, S	y = 12, S	x = 5, S y = 12, S
x = x + 1	y = y + 1	x = 6, D	y = 13, D	x = 5, I y = 12, I
read y	read x	y = 13, S x = 6, S	y = 13, S x = 6, S	y = 13, S x = 6, S
x = x + y	y = x + y	x = 19, D y = 13, I	x = 6, I y = 19, D	x = 6, I y = 13, I
x = x + 1	y = y + 1	x = 20, D	y = 20, D	x = 6, I y = 13, I

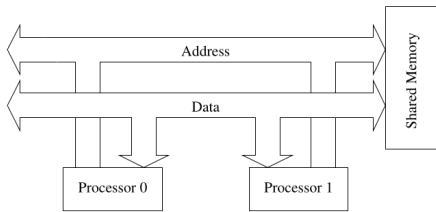
**Figure 2.23** Example of parallel program execution with the simple three-state coherence protocol discussed in Section 2.4.6.



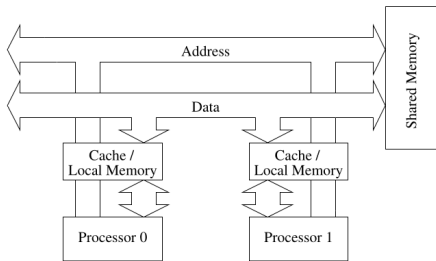
# The Memory Bus

- ▶ Cache coherence protocols involve communication between procs, obtaining information about changes to memory
- ▶ The **Memory Bus** is the communication channel that enables Procs/Memory chips to “talk” to each other
- ▶ Hardware construct to move data around, usually across wires connected to each Proc and DRAM chip
- ▶ Memory Buses use a communication protocol which includes **device identifiers** so messages about changes are directed to individual PEs or DRAM
- ▶ Bus can get “crowded” if lots of Procs make memory requests
- ▶ All hardware can “see” messages on the bus: allows **snooping** of messages intended for others
- ▶ Example: PE1 sees that PE0 read address #1024 so PE0 knows it may share #1024 now

# Diagram of Typical Memory Buses



(a)



(b)

**Figure 2.7** Bus-based interconnects (a) with no local caches; (b) with local memory/caches.

# Snoopy Cache

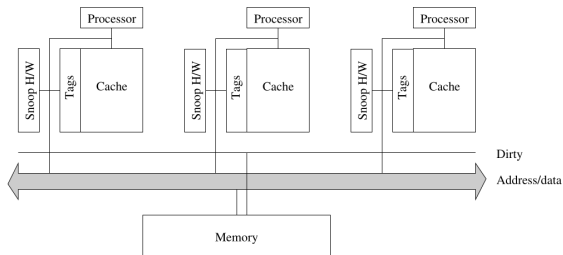


Figure 2.24 A simple snoopy bus based cache coherence system.

## Basics

- ▶ Additional hardware watches messages on the bus
- ▶ Writing to cache invalidates global memory
- ▶ Message pertaining to a dirty memory address cause flush, state back to shared

## Example

- ▶ x in P0 cache Dirty
- ▶ x in Global mem Invalid
- ▶ P1 reads x
  - ▶ P0 “snoops” request
  - ▶ Flushes x to global mem
  - ▶ P1 can read x from global
- ▶ x is now Shared

## Different Variable but Same Cache Line → Collisions

- ▶ Performance problem: two processors grinding on different but close variables
- ▶ Consider the following program:  $x, y$  are adjacent in main memory, likely to share same cache line
- ▶ Proc0 and Proc1 each have own cache, will interfere with one another despite working on different variables

```
void collide(){
    int x=42;
    int y=31;
    if(proc_id == 0){
        int i;
        for(i=0; i<1000; i++){
            x = (x+1)*(x+3)/x;
        }
    }
    else{
        int i;
        for(i=0; i<1000; i++){
            y = y/2;
            y = y+2*y;
        }
    }
}
```

## Small Stacks for Threads → False Sharing Collisions

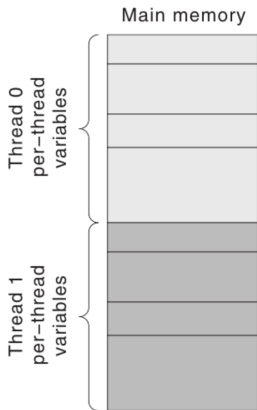
```
#include <pthread.h>
#include <stdio.h>
void *fx(void *param) {
    int i, x=(int) param;
    for(i=0; i<1000; i++){
        x = (x+1)*(x+3)/x;
        printf("x %d\n",x);
    }
    return (void *) x;
}

void *fy(void *param){
    int i, y=(int) param;
    for(i=0; i<1000; i++){
        y = y/2;
        y = y+2*y;
        printf("y %d\n",y);
    }
    return (void *) y;
}

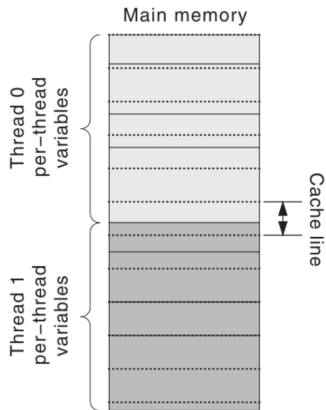
int main(int argc, char *argv[]) {
    pthread_t    thread_1;
    pthread_t    thread_2;
    pthread_create(&thread_1, NULL,
                  fx, 42);
    pthread_create(&thread_2, NULL,
                  fy, 31);

    int *xres, *yres;
    pthread_join(thread_1, &xres);
    pthread_join(thread_2, &yres);
    printf("x is %d\ny is %d\n",
          (int) xres,(int) yres);
}
```

# False Sharing of Thread Stacks



**Figure 9.1** Per-thread variable memory layout



**Figure 9.2** Memory layout showing cache line boundaries

Source: [Building Parallel Programs](#), Kaminsky

# Compiler /Padding Can fix This

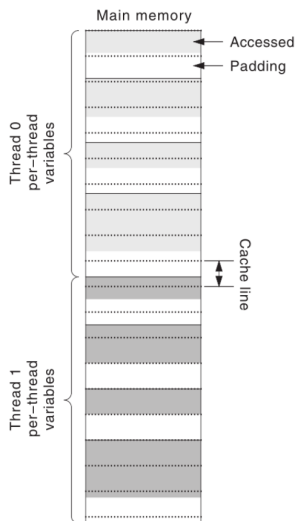


Figure 9.3 Memory layout with extra padding

```
#include <pthread.h>
#include <stdio.h>
void *fx(void *param) {
    int i, x=(int) param;
    int padding[32];    // PADDING
    for(i=0; i<1000; i++){
        x = (x+1)*(x+3)/x;
        printf("x %d\n",x);
    }
    return (void *) x;
}

void *fy(void *param){
    int i, y=(int) param;
    int padding[32];    // PADDING
    for(i=0; i<1000; i++){
        y = y/2;
        y = y+2*y;
        printf("y %d\n",y);
    }
    return (void *) y;
}
```

# Cache Coherence Overall

- ▶ Caches speed up individual processor execution in most cases
- ▶ Coordinating caches across several PEs is complex
- ▶ Requires additional hardware such for Snooping, alternatively Directory-based approach (textbook)
- ▶ Hardware manages most of this but uses techniques that are smack of distributed memory systems
- ▶ To eek out more performance, programmers should be aware of these things when using Thread-based programs