Shared Memory Architectures

Chris Kauffman

CS 499: Spring 2016 GMU

Logistics

Today

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

Reading

- Grama 2.4.1 (PRAM), 2.4.6 (cache)
- Grama 7.10 (OpenMP)
- OpenMP Tutorial at Laurence Livermorem

HW 3: Up later today

- Problem 1: IPC Heat
- Problems 2&3: Textbook
- Problem 4: OpenMP Exercises
- Due: Fri 4/1 (8 days)

Mini-exam 3

Thu: 4/7

PRAM: Parallel Random Access *Machine*

Grama Ch 2.4.1

RAM: Random access machine

- An unfortunate name, but so it goes
- Single CPU attached to random access memory
- Simplistic model for a real machine: CPU reads memory, performs operations in registers, writes to memory, repeates

Parallel alternative to RAM: PRAM

- Again, theoretical model for a real parallel machine
- Multiple CPUs attached to memory, share clock but can execute different instructions
- In some version of PRAM, allowed *infinite* processors
- Question: What immediate problems are there with PRAM that don't exist in RAM?

Theoretical Flavors of PRAM

Exclusive-read, exclusive-write (EREW) PRAM

Multiple CPUs cannot touch same memory at all. No concurrency possible for reads or writes

Concurrent-read, exclusive-write (CREW) PRAM

Multiple CPUs can read same 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

Multiple read and write accesses to a common memory location.
This is the most "powerful" PRAM model.
What is meant by powerful here?
Anything flaws in the above classification?

Resolution Schemes for Concurrent Reads/Writes

- Common, in which the concurrent write is allowed if all the values that the processors are attempting to write are identical.
- Arbitrary, in which an arbitrary processor is allowed to proceed with the write operation and the rest fail.
- Priority, in which all processors are organized into a predefined prioritized list, and the processor with the highest priority succeeds and the rest fail.
- Sum, in which the sum of all the quantities is written

None of these deal with resolution resolution of concurrent read/write

```
MEM[1024] is 10
P0 reads MEM[1024] into R1
P1 writes 20 to MEM[1024]
```

But deeper studies of PRAM might resolve this (everyone reads first, then writes if needed. . .)

Pros and Cons of PRAM

Why the PRAM Model?

- It's simple
- Lots of study of different algorithms
- Has significant theoretical importance

Why Not PRAM

- No general machine currently implements the model
- Seen some references that GPUs might sort of implement but would require some more work
- Conclusions one might draw about "good" algorithms is skewed

Recall the Cache

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

Matrix Multiplication 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?

Numbers Everyone Should Know

Edited Excerpt of Jeff Dean's talk on data centers.

Reference	Time	Analogy
Register	-	Your brain
L1 cache reference	0.5 ns	Your desk
L2 cache reference	7 ns	Neighbor's Desk
Main memory reference	100 ns	This Room
Disk seek	10,000,000 ns	Salt Lake City

Does Big-O analysis capture these effects?

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: 25K/548K = 4% main memory access
- Col: 122/585K = 20% main memory access

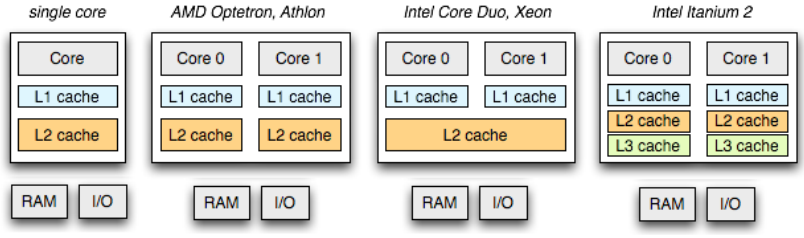
Caches Strike Back

Consider a Typical Shared Memory machine

- Single hunk of RAM (random access *memory*)
- Several CPUs (2, 4, 8 typical)

Where does the cache live and why is this a problem?

Cache Problems



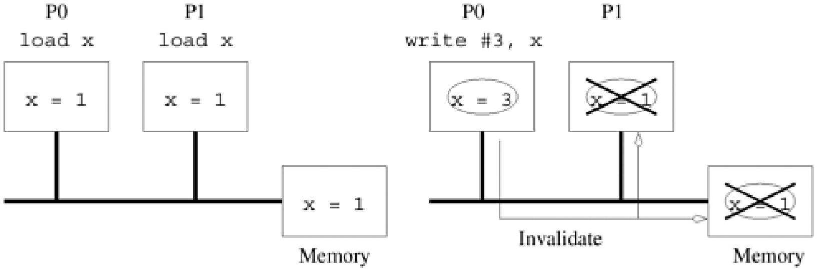
Source: Multi-core, Threads & Message Passing by Ilya Grigorik

Consider cache coherencef

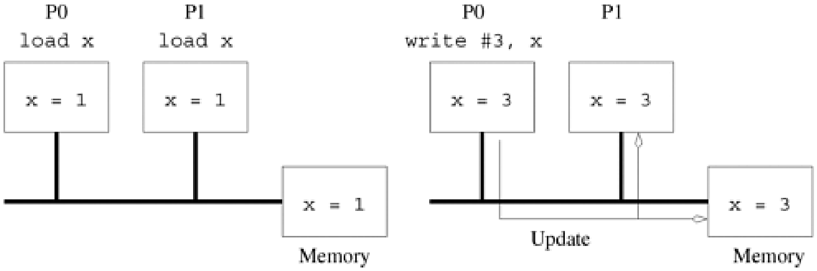
```
// MEM[1024] has value 5
P0: load R1 MEM[1024]   // slow, populates cache
P0: load R2 MEM[1024]   // fast, from cache
P0: ADD R1 R1 R2        // R1 is 10
P0: store R1 MEM[1024]  // cache dirty, MEM[1024] unchanged

P1: load R3 MEM[1024]   // read 5 or 10?
```

Cache Coherence Protocols: Invalidate and Update

Grama 2.4.6



(a)

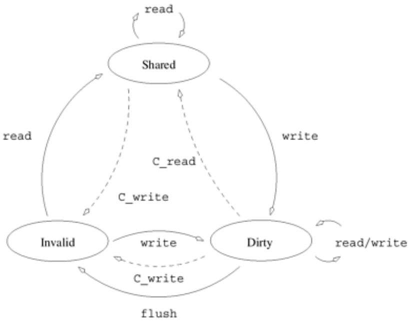


(b)

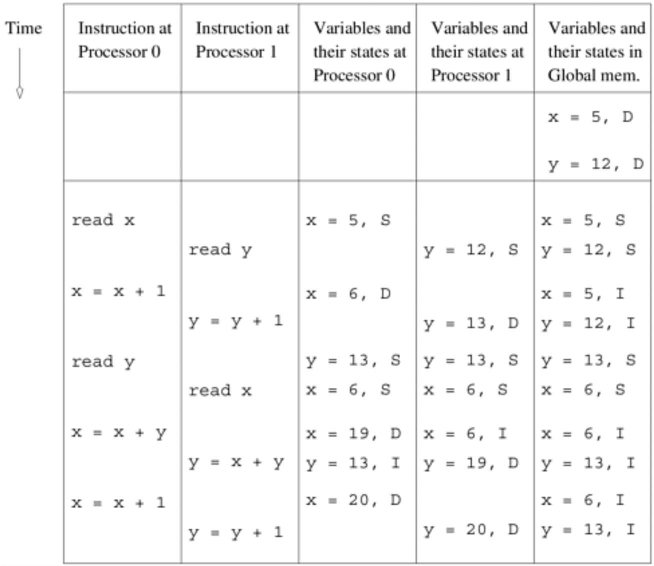
Cache Coherence

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



Demonstration



Time

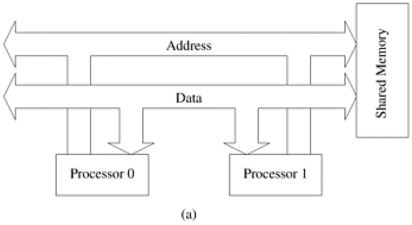
Instruction at Processor 0	Instruction at Processor 1	Variables and their states at Processor 0	Variables and their states at Processor 1	Variables and their states in Global mem.

				x = 5, D
				y = 12, D

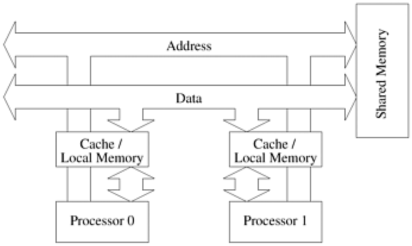
read x		x = 5, S		x = 5, S
	read y		y = 12, S	y = 12, S
x = x + 1		x = 6, D		x = 5, I
	y = y + 1		y = 13, D	y = 12, I
read y		y = 13, S	y = 13, S	y = 13, S
	read x	x = 6, S	x = 6, S	x = 6, S
x = x + y		x = 19, D	x = 6, I	x = 6, I
	y = x + y	y = 13, I	y = 19, D	y = 13, I
x = x + 1		x = 20, D		x = 6, I
	y = y + 1		y = 20, D	y = 13, I

The Magical Memory Bus

- Cache coherence protocols rely on the Memory Bus
- Handwavy hardware construct to move data around
- All PEs use the bus to communicate all other PEs
- Every PE has a way of knowing a bus message is for it
- Bus can get crowded if there there are lots of memory requests
- Can alleviate somewhat through caches but that leads to trouble

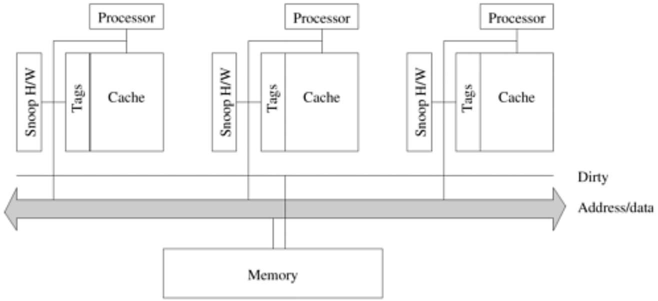


(a)



(b)

Snoopy Cache



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

Cache Coherence Overall

- Coordinating caches across several cores and main memory is complex
- Requires additional hardware such for Snooping, alternatively Directory-based approach (textbook)
- Be sensitive to read/write conflicts: avoid when possible
- Look for false sharing due to cache (next)

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

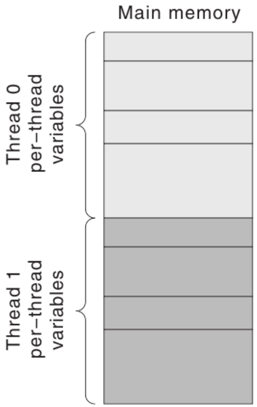


Figure 9.1 Per-thread variable memory layout

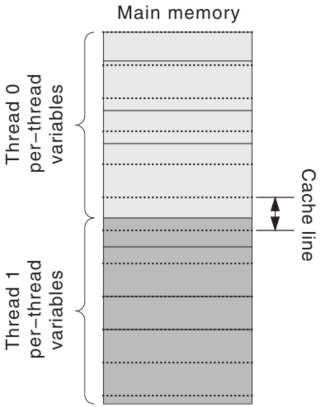


Figure 9.2 Memory layout showing cache line boundaries

Source: Building Parallel Programs, Kaminsky

Padding Can fix This

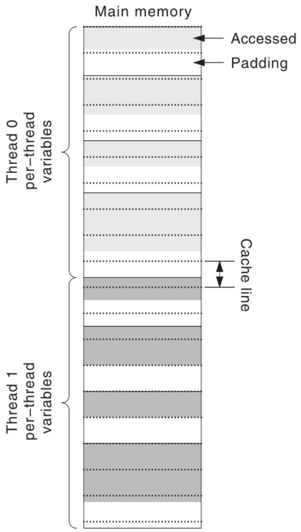


Figure 9.3 Memory layout with extra padding

```
#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;
}
```