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 Livermore

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, repeats

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 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**

```c
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**

```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?
Edited Excerpt of Jeff Dean’s talk on data centers.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Time</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>-</td>
<td>Your brain</td>
</tr>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
<td>Your desk</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
<td>Neighbor’s Desk</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
<td>This Room</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
<td>Salt Lake City</td>
</tr>
</tbody>
</table>

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

with

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

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

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

Consider cache coherence

// 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

(a)

P0
load x
x = 1
P1
load x
x = 1
Memory

x = 1

Invalidate

P0
write #3, x
x = 3
P1
x = 1
Memory

(b)

P0
load x
x = 1
P1
load x
x = 1
Memory

x = 1

Update

P0
write #3, x
x = 3
P1
x = 3
Memory

x = 3
Cache Coherence

Each element in the ProcX’s cache is one of:

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Instruction at Processor 0</th>
<th>Instruction at Processor 1</th>
<th>Variables and their states at Processor 0</th>
<th>Variables and their states at Processor 1</th>
<th>Variables and their states in Global mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read x</td>
<td>read y</td>
<td>x = 5, S</td>
<td>y = 12, S</td>
<td>x = 5, S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x = x + 1</td>
<td>y = y + 1</td>
<td>x = 6, D</td>
<td>y = 13, D</td>
<td>x = 5, I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read y</td>
<td></td>
<td>y = 13, S</td>
<td>y = 13, D</td>
<td>y = 12, I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read x</td>
<td></td>
<td>x = 6, S</td>
<td>x = 6, S</td>
<td>y = 13, S</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x = x + y</td>
<td>y = x + y</td>
<td>x = 19, D</td>
<td>x = 6, I</td>
<td>x = 6, I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x = x + 1</td>
<td>y = y + 1</td>
<td>x = 20, D</td>
<td>y = 20, D</td>
<td>y = 13, I</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 are lots of memory requests
- Can alleviate somewhat through caches but that leads to trouble
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)
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

```c
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;
        }
    }
}
```
#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

```c
#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;
}
```

**Figure 9.3** Memory layout with extra padding