Multicore Computing – CS 5966 / 6966 – Week 3

`http://www.eng.utah.edu/~cs5966`

Contents

1 Overview of this week

This week Monday (1/26) we talked about Cilk using their PLDI 1998 paper (see 5966-lecture-1-26-09.pptx posted online). We understood at a high level how work-stealing works. We also discussed the basics of Cilk programming. We mentioned several times that

- Cilk and similar languages save the users from 'micromanaging' threads (creating, synchronizing, migrating for load balancing, aborting upon speculative computation failing, ...) with the concomitant bugs avoided.
- Yet, people have to worry about some low level bugs – esp. pertaining to memory models.

In today's lecture you will get an introduction to these topics. You will also get to practice these ideas using actual experiments. We will of course try to wind-up some discussions on Cilk. Each topic appears in a separate section below, with homeworks also assigned (to make life interesting!).

2 CILK Reading

Section 2.1 of Minicourse: Assignment 3 (due 2/3): Read Section 2.1 of the Minicourse ("A minicourse on multithreaded programming") which is posted online – Week2's LECTURES-AND-SLIDES area. **Post a one-paragraph summary.**

Section 2.2 of Minicourse: Assignment 3 (due 2/3): Why did the authors decide that `merge` must be parallelized, and how did the parallelism improve as a result. Calculate the parallelism for $n = 1024$ using the first version (without parallelizing merge) and the second version (after parallelizing merge). **Post a few lines of summary.**

3 Language Memory Models

Most compilers (e.g., `gcc`) are unaware of what is "safe" to do in a multithreaded/multi-core situation. To study this, let us experiment with `interlock1.c`

```
#include <pthread.h>
int p1=0, p2=0;
char aa=0, bb=0; // handshake bits
void * thread_routine1(void * arg)
{ do {
    aa  = 1;
    while(!bb);
    bb = 0;
        p1++; if (!(p1 % 10)) printf("progress1\n");
  } while(1);
}
void * thread_routine2(void * arg)
{ do {
    bb  = 1;
    while(!aa);
    aa = 0;
        p2++; if (!(p2 % 10)) printf("progress2\n");
  } while (1);
}
int main()
{ pthread_t  t1, t2;  pthread_create(&t1, 0, thread_routine1, 0);
  pthread_create(&t2, 0, thread_routine2, 0);  pthread_join(t1, 0);
  pthread_join(t2, 0);  return 0;
}
```

gcc, no optimization: Assignment 3 (due 2/1): `gcc -o interlock1 interlock1.c -lpthread` and run interlock1 on a uniprocessor and then a multiprocessor machine. Did you see the execution "hang"? Explain in a few sentences. **Post a brief summary**.

gcc, with optimization: Assignment 3 (due 2/1): Include the -O3 flag. Now what are the results you obtain? (It must hang!). Explain in a few sentences as to why. Will it hang on a uniprocessor? **Post a brief summary**.

Understanding what language memory model means: In a nutshell, language memory models define what a compiler is licensed to do when given multiprocessor/multi-threaded code. C has no such memory model specification that is widely agreed upon. C++ has something that is in the works. Java has gotten its act together.

Assignment 3 (due 2/1): Look at the assembly code generated. Can you find out why the unoptimized worked while the optimized did not?

Note: the unoptimized worked due to sheer luck!! There is no such guarantee that the store/load order will be preserved.

You can do gcc -S interlock1.c

and then look at interlock1.s

For more information, you can do

`gcc -c -g -Wa,-a,-ad interlock1.c > interlock1-unopt.lst`

`gcc -c -g -Wa,-a,-ad -O3 interlock1.c > interlock1-opt.lst`

See

`http://flatline.cs.washington.edu/orgs/acm/tutorials/dev-in-unix/compiler.html`

for details.

Can you now see why the optimized code hangs on a uniprocessor or a multiprocessor?. Post a brief summary.

4 Architectural Memory Models

Well, you think you somehow lucked out without using -O3? Tough luck! The machine can do a store/load reordering! Why? For efficiency of course. Loads are so critical to overall performance that they should not be held up (as far as possible) for previous stores to finish.

Danger without using mfence: Study the code below (obtained from the Cilk manual - so call it

`cel-ex1.c`).

Assignment 3 (due 2/1): Now run it on a uniprocessor, then a multiprocessor. You may have to run it multiple times. Did you get it to fail on a multiprocessor? Explain. **Post a brief summary**.

```
#include <pthread.h>
int aa=0, bb=0, zz=0;
int c1=0, c2=0;
void * thread_routine1(void * arg)
{ do {
    c1++;
    aa = 1;
    if (!bb) zz++;
    zz = 0;
    bb = 0;
  } while(1);
}
void * thread_routine2(void * arg)
{ do {
    c2++;
    bb = 1;
    if (!aa) zz++;
    zz = 0;
    aa = 0;
  } while (1);
}
void * thread_routine3(void * arg)
{ while(1)
  if (zz > 1) printf("Caught violation, c1=%d, c2=%d, zz=%d\n", c1, c2, zz);
}
int main()
{ pthread_t  t1, t2, t3; pthread_create(&t3, 0, thread_routine3, 0);
  pthread_create(&t1, 0, thread_routine1, 0); pthread_create(&t2, 0, thread_routine2, 0);
  pthread_join(t1, 0);  pthread_join(t2, 0);  pthread_join(t3, 0);   return 0;
}
```

Using mfence to rescue the ordering: Assignment 3 (due 2/1): To prevent the HW from reordering, you can do this. First get the assembly code. Then locate the place where the store/load ordering is being violated. Insert an `mfence`. Then

`gcc file.s -lpthread -o file`

to finish compiling.

Now repeatedly run on a multiprocessor. Did things fail?. **Post a brief summary**.

Try on producer/consumer: Assignment 3 (due 2/3): People often count on "producer/consumer" ordering. I show the bare-bones of this idea below.

```
while(1)  a=1 //produce; flag=1 // notify
```

```
while(1) await flag==1; consume
```

Think about whether `gcc` without optimization will preserve this ordering.

Experimentally confirm.

Then see whether the HW memory model will allow this to be preserved.

Show that by adding an `mfence` you can rescue the situation.

Post a brief summary.

5 Pthread verification using Inspect

Assignment 3 (due 2/3): Various Pthread programs have been posted in the Week3 directory. They are

- `deadlock01.c`
- `carter01.c`
- `Dining*.c`

1. First run these codes under "straight GCC" using instructions present in the `README` file. Did it reveal the deadlocks? In how many runs? **Post a few lines of summary.**
2. Next, run these codes under Inspect (instructions to be posted under 'software-updates' soon.)

   Summarize the bugs reported. Also discuss how formal verification (dynamic verification) tools help debug concurrent software.

   Post a few lines of summary.
3. Post some brief thoughts on how verification tools can help guard you from memory model related errors. Consider both language and architectural memory models.