15-213/15-513 Lecture 27: Thread-Level Parallelism

Learning Objectives

- Identify the effects of false sharing and ways of preventing it.
- Quantify parallel performance using speedup and Amdahl's Law.
- Use a sequential consistency model to reason about possible and impossible program execution order/outputs.

1 Getting Started

The directions for today's activity are on this sheet, but refer to accompanying programs that you'll need to download. To get set up, run these commands on a shark machine:

```
1. $ wget http://www.cs.cmu.edu/~213/activities/threads.tar
2. $ tar xf threads.tar
3. $ cd threads
```

2 Memory Models: An Example With Sequential Consistency

Recall that a consistency model defines the ordering of writes and reads to different memory locations — the hardware guarantees a certain consistency model and the programmer attempts to write correct programs with those assumptions.

An example of a memory model is sequential consistency. One way to think about sequential consistency is as a switch. At each time step, the switch selects a thread to run, and runs its next event completely. This model preserves the rules of sequential consistency: events are accessing a single main memory, and so happen in order; and by always running the next event from a selected thread, each thread's events happen in program order.

P0	P1
(a) A = 1	(c) x = Ready
(b) Ready = 1	(d) y = A

** All variables are initialized to 0.

1. What are some possible outcomes for (x,y) as a result of running P0 and P1? Write down the orderings of a/b/c/d that result in each outcome.
   (0,0) - c d a b
   (0,1) - c a d b
   (1,1) - a b c d

2. Is (1,0) a possible outcome? Explain why or why not.
   (1,0) is not a possible outcome.

   - We know a->b and c->d by program order.
   - b->c implies that a->d.
   - y==0 implies d->a which leads to a contradiction.

3. What are some shortcomings of an implementation adhering to a sequential memory consistency model?
   A problem with implementations adhering to a sequential consistency model is that they will likely be terribly slow. Only a single instruction can be run at a time, so such implementations lose most of the benefit of having multiple threads run in parallel. Worse, each instruction must finish before the next can be started — no more instructions can run until the current instruction's effects become visible to every other thread.

4. An advantage of implementations using a sequentially consistent model is that we can guarantee coherence. Why might this property be useful?
   All writes to the same location are seen in the same order by every thread. This significantly reduces programmer effort expended on reasoning about possible out-of-order interleavings of thread writes — a thread may miss certain intermediate values, but a thread will never observe a "previously" written value after a "later" written value. That is, all threads will see the same order of writes to a location.

3 False Sharing

Examine the file `falseSharing.c` using your editor of choice. Notice that the program has one thread read from and one thread write to different elements from an array of structs containing both fields. Once you're comfortable with the program, build (`$ make falseSharing`) and run it (`$ ./falseSharing`).

5. As you might have noticed, the code pays a significant performance penalty due to false sharing. Why does false sharing occur, and what is causing it in this specific example?
   False sharing occurs between the adjacent elements of the struct. When one thread writes to one element of the struct, it requires the other thread to reload its contents given that the data they are trying to access is adjacent, but not equal.

6. How could you fix this problem by changing a few lines of code? Implement the fix and comment on the performance change.
   Instead of an array of structs, create an array of student scores, and an array of student IDs so that they do not fall in the same cache block. Previously, the code took 9500 microseconds. After implementing this fix, the code takes 7000 microseconds.

4 Amdahl's Law + Parallel Performance

Recall that Amdahl's Law states that the max speedup attainable for a program with sequential execution time T is given by $p * T/k + (1 - p) * T$, where p is the percentage of the program that is parallelizable, and k is the number of parallel execution units.

7. Say our sequential program runs in 100s, but 80% of the program is parallelizable. Given that we run this program on a 16-core machine, what execution do we expect to achieve?
   100 * (0.2) + 100 * (0.8) / 16 = 20 + 5 = 25s

The concept of speedup relates closely to Amdahl's Law, where the speedup is a measure of the performance increase from sequential to parallel, and is computed as the best possible sequential performance divided by the best possible parallel performance.

8. If for a given program, we find that the sequential execution time is 50 minutes, but 5 minutes of the program cannot be parallelized, what is the maximum speedup we can achieve for this program?
   50 / 5 = 10x, because if we have "infinite" parallelism, the part of the program that can be parallelized will take "no" time while 5 minutes are still required for the sequential part of the program.

9. Consider a program that can be split into two tasks: A and B. The values for T (the sequential execution time) and p (the percentage of the program that is parallelizable) are given below:

   - T(A) = 40 hours, p(A) = 0.6
   - T(B) = 20 hours, p(B) = 0.8

   Assuming we are running this on an 8-core machine, what is the expected speedup?
   Parallel Time of A = 40 * (0.6) / 8 + 40 * (0.4) = 19
   Parallel Time of B = 20 * (0.8) / 8 + 20 * (0.2) = 6
   Speedup = 60 / (19 + 6) = 2.4x

10. Could we answer the previous question without knowing the sequential execution times of A and B, but rather just knowing the ratio between the two (T(A) = 2*T(B))?
    Yes. We can let T(B) be any time and T(A) be twice as much and still get the same answer.
    Parallel Time of A = 2*T(B) * (0.6) / 8 + 2*T(B) * (0.4) = T(B) * 0.95
    Parallel Time of B = T(B) * (0.8) / 8 + T(B) * (0.2) = T(B) * 0.3
    Speedup = 3*T(B) / ((0.95 + 0.3) * T(B)) = 2.4x