CS 450: Operating Systems Lecture 7: Concurrent Programming II

Spring 2014, J. Sasaki
Dept of Computer Science
Illinois Institute of Technology

Amdahl's Law

How Much Speedup?

- Matrix multiplication is a rare example of a perfectly parallelizable algorithm.
  - Even so, with *N* cores, we don't get *N* times faster results — overhead.
- What about algorithms that aren't perfectly parallelizable?

Best Runtime

- Let S be the fraction of our program that is serial-only (not parallelizable); $0 \leq S \leq 1$.
    - So $1-S$ is parallelizable.
- If we have N cores, then the best runtime we can hope for is $S+(1-S)/N$.
- For perfectly parallelizable program $S=0$, $1-S=1$, so with N cores our best runtime is $1/N$.

Amdahl's Law

- Our new runtime $\geq S+(1-S)/N$
- Amdahl's law:

$$\text{Speedup} \leq 1/(S+(1-S)/N)$$

- As $N \to \infty$, runtime $\to S$; speedup $\to 1/S$.
- Speedup severely limited by S. Examples:
    - S = 20%; speedup ≤ 5
    - S = 10%; speedup ≤ 10

5

Estimating S

- If T_{new} and T_{old} are the new and old runtimes, then
  - $T_{new}/T_{old} = S+(1-S)/N$
  - So $S=(N\times T_{new}/T_{old}-1)/(N-1)$
- Matrix multiplication example
  - T_{old} = 355.4 ms
  - For N=2, T_{new}=207.3 ms, so S=17%
  - For N=3, T_{new}=210.7 ms, so S=39%

Process Synchronization

(Chapter 5)

Concurrent Execution and Race Conditions

Concurrent Execution

- Concurrent or parallel execution of computation sequences (call them "threads" for short):
  - Each sequence executes sequentially.
  - But the two sequences are interleaved nondeterministically.

Cooperation is Good

- Concurrent/parallel programs need their computation sequences to cooperate.
  - Communicate data: Messages, shared data
  - Synchronize (transfer pgm counter info)

10

Cooperation is Hard

- Cooperation is hard because any shared state can change nondeterministically.
- What does reading or setting a variable *V* tell you about the value of *V*?
- Even if you inspect or update a shared variable, you have no idea what its current value is unless you know something about the programs involved.

Synchronization

- We might know how do individual threads behave in isolation…
- But behavior together can be totally unlike behavior in isolation.
- Plus, # potential interaction points is large!
- Synchronization problems are HARD.

Combining Behaviors

- Example: If `x=10` before thread 0 runs `x++` and thread 1 runs `x--` then what is `x` afterwards?
- Thread 0: `reg0←x; reg0++; x←reg0;`
- Thread 1: `reg1←x; reg1--; x←reg1;`
- Result depends on sizes of basic operations and on their order of interleaving.

Granularity of Interleaving

- We broke up `x++` into
  `reg0 ← x; reg0++; x ← reg0`
  because these (probably) correspond to hardware instructions.
- Can hardware instructions be interleaved?
- Can memory accesses be interleaved?

A Nice Interleaving

Thread 0	*Thread 1*	**x**	**reg0**	**reg1**
`reg0 ← x;`		10	10	
`reg0++;`		10	11	
`x ← reg0;`		11	11	
	`reg1 ← x;`	11	11	11
	`reg1--;`	11	11	10
	`x ← reg1;`	10	11	10

A Less-Nice Interleaving

Thread 0	*Thread 1*	**x**	**reg0**	**reg1**
reg0 ← x;		10	10	
	reg1 ← x;	10	10	10
	reg1--;	10	10	9
	x ← reg1;	9	10	9
reg0++;		9	11	9
x ← reg0;		11	11	9

Race Condition

- A ***race condition*** occurs when the correctness of a program depends on the relative speed of its threads.
- Avoid race conditions by making sure that all allowed execution interleavings produce acceptable results.
- Control granularity of interleaving.
- Stop threads when they shouldn't continue.

Use a Flag to Signal OK to Go?

```
x = 10;
ok_to_go = true;
```

```
/* Thread 0 */            /* Thread 1 */
while (!ok_to_go) ;       while (!ok_to_go) ;
ok_to_go = false;         ok_to_go = false;
x++;                      x--;
ok_to_go = true;          ok_to_go = true;
```

Uh, oh

Thread 0	***Thread 1***	***`ok_to_go`***
`while(!ok_to_go);`		`true`
	`while(!ok_to_go);`	`true`
`ok_to_go = false;`		`false`
	`ok_to_go = false;`	`false`
etc	*etc*	`false`