CS3383 Unit 4: dynamic multithreaded algorithms lecture 1

David Bremner

March 20, 2024

Dynamic Multithreaded Algorithms

- Race Conditions
- Scheduling

Race Conditions

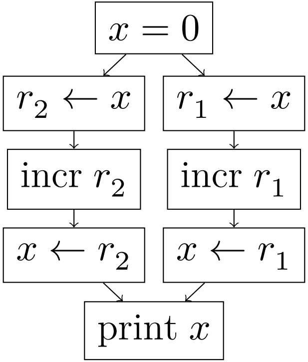
Non-Determinism

- result varies from run to run
- sometimes OK (in certain randomized algorithms)
- mostly a bug.

```
x = 0
parallel for i ← 1 to 2 do
    x ← x + 1
```

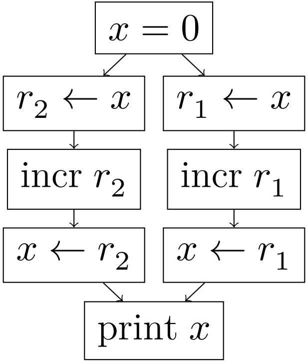
- nondeterministic unless incrementing x is atomic

Racy execution



- all topological sorts are possible
- both loads can complete before either store
- We will insist that parallel strands are independent

Racy execution



- all topological sorts are possible
- both loads can complete before either store
- We will insist that parallel strands are independent

We can write bad code with spawn too

```
sum(i, j)
  if (i>j)
    return;
  if (i==j)
    x++;
  else
    m=(i+j)/2;
    spawn sum(i,m);
    sum(m+1,j);
    sync;
```

- here we have the same non-deterministic interleaving of reading and writing x
- the style is a bit unnatural, in particular we are not using the return value of spawn at all.

Being more *functional* helps

```
sum(i, j)
  if (i>j) return 0;
  if (i==j) return 1;

  m ← (i+j)/2;

  left ← spawn sum(i,m);
  right ← sum(m+1,j);
  sync;
  return left + right;
```

- each strand writes into different variables
- sync is used as a barrier to serialize

Single Writer races

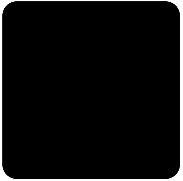
```
x ← spawn foo(x)
y ← foo(x)
sync
```

- arguments to spawned routines are evaluated in the parent context
- but this isn't enough to be race free.
- which value x is passed to the second call of 'foo' depends how long the first one takes.

Single Writer races

- arguments to spawned routines are evaluated in the parent context
- but this isn't enough to be race free.
- which value x is passed to the second call of 'foo' depends how long the first one takes.

```
x ← spawn foo(x)
y ← foo(x)
sync
```



Scheduling

Scheduling Problem

Abstractly Mapping threads to processors

Pragmatically Mapping logical threads to a thread pool.

Ideal Scheduler

On-Line No advance knowledge of when threads will spawn or complete.

Distributed No central controller.

- to simplify analysis, we relax the second condition

A greedy centralized scheduler

Maintain a *ready queue* of strands ready to run.

Scheduling Step

Complete Step If $\geq p$ (# processors) strands are ready, assign p strands to processors.

Incomplete Step Otherwise, assign all waiting strands to processors

- To simplify analysis, split any non-unit strands into a chain of unit strands
- Therefore, after one time step, we schedule again.

Optimal and Approximate Scheduling

Recall

(work law) $$T_p \geq T_1/p$$

(span) $$T_p \geq T_\infty$$

Therefore

$$T_p \geq \max(T_1/p, T_\infty) = \text{opt}$$

With the greedy algorithm we can achieve

$$T_p \leq \frac{T_1}{p} + T_\infty \leq 2\max(T_1/p, T_\infty) = 2 \times \text{opt}$$

Counting Complete and Incomplete Steps

We can show

- There are at most T_1/p complete steps (easy)
- There are at most T_∞ incomplete steps (shrinking longest path)

Counting Complete and Incomplete Steps

We can show

- There are at most T_1/p complete steps (easy)
- There are at most T_∞ incomplete steps (shrinking longest path)

Parallel Slackness

$$\text{parallel slackness} = \frac{\text{parallelism}}{p} = \frac{T_1}{pT_\infty}$$

$$\text{speedup} = \frac{T_1}{T_p} \leq \frac{T_1}{T_\infty} = p \times \text{slackness}$$

- If slackness < 1, speedup $< p$
- If slackness ≥ 1, linear speedup achievable for given number of processors

Slackness and Scheduling

$$\text{slackness} := \frac{T_1}{p \times T_\infty}$$

Theorem

For sufficiently large slackness, greedy scheduler approaches time T_1/p.