CS3383 Unit 4: dynamic multithreaded algorithms lecture 1

David Bremner

March 20, 2024

Outline

- 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.

Race Conditions

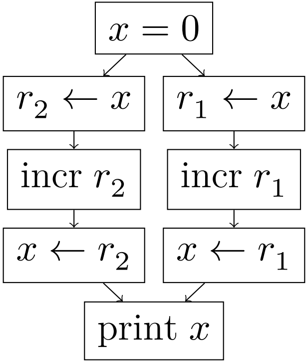
Non-Determinism

- 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
```

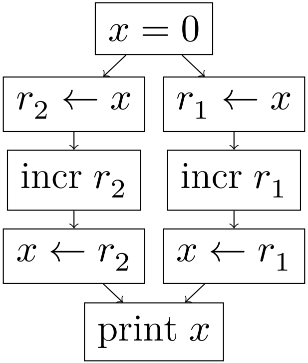
- nondeterministic unless incrementing x is atomic

Racy execution



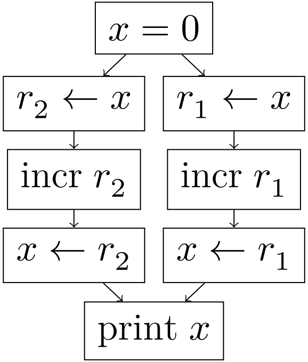
- all possible topological sorts are valid execution orders

Racy execution



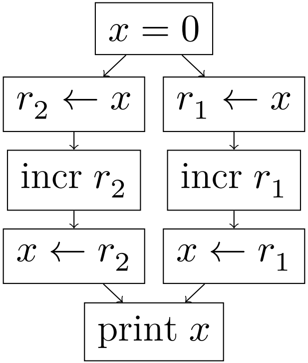
- all possible topological sorts are valid execution orders
- In particular it's not hard for both loads to complete before either store

Racy execution



- all possible topological sorts are valid execution orders
- In particular it's not hard for both loads to complete before either store
- In practice there are various synchronization strategies (locks, etc...).

Racy execution



- all possible topological sorts are valid execution orders
- In particular it's not hard for both loads to complete before either store
- In practice there are various synchronization strategies (locks, etc...).
- Here we will insist that parallel strands are independent

Racy demo

```
#pragma omp parallel for
  for (int i=0; i<10000; i++){
    x++;
  }
```

- what is the final value of x?

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.

spawn race demo

```
static void
sum(long i, long j, long *out) {
  if (i>j)
    return;
  if (i==j) {
    (*out)++;
  } else {
    long m=(i+j)/2;
#pragma omp task
    sum(i,m,out);
    sum(m+1,j, out);
#pragma omp taskwait
  }
```

Being more *functional* helps

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

  m ← (i+j)/2;

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

- each strand writes into different variables

Being more *functional* helps

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

  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

functional sum demo

```
long sum(long i, long j) {
  if (i>j) return 0;
  if (i==j) {
    return i;
  } else {
    long left,right,m=(i+j)/2;
#pragma omp task shared(left)
    left = sum(i,m);
    right = sum(m+1,j);
#pragma omp taskwait
    return left+right;
  }
}
```

Single Writer races

- arguments to spawned routines are evaluated in the parent context

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

Single Writer races

- arguments to spawned routines are evaluated in the parent context
- but this isn't enough to be race free.

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

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.

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.

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.

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

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

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

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

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 Steps

- Let k be the number of complete steps.

Counting Complete Steps

- Let k be the number of complete steps.
- At each complete step we do p units of work.

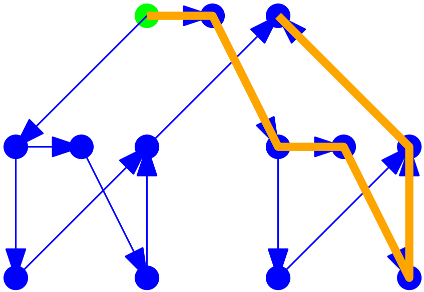
Counting Complete Steps

- Let k be the number of complete steps.
- At each complete step we do p units of work.
- Every unit of work corresponds to one step of the serialization, so $kp \leq T_1$.

Counting Complete Steps

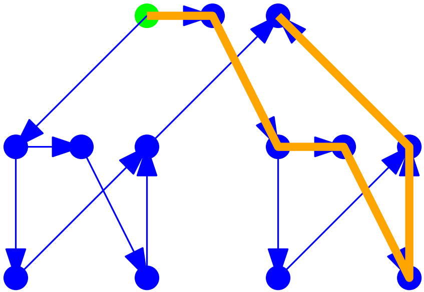
- Let k be the number of complete steps.
- At each complete step we do p units of work.
- Every unit of work corresponds to one step of the serialization, so $kp \leq T_1$.
- Therefore $k \leq T_1/p$

Counting Incomplete Steps



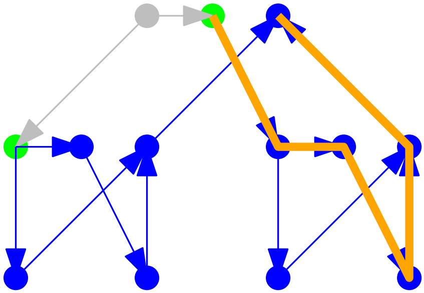
- Let G be the DAG of *remaining strands*.

Counting Incomplete Steps



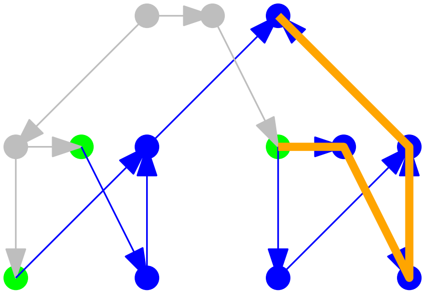
- Let G be the DAG of *remaining strands*.
- ready queue $=$ the set of sources in G

Counting Incomplete Steps



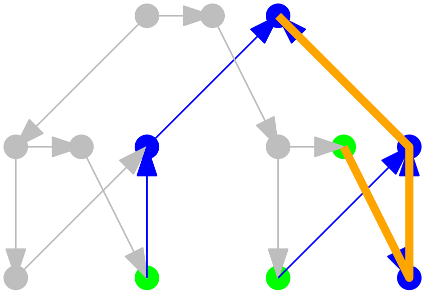
- Let G be the DAG of *remaining strands*.
- ready queue $=$ the set of sources in G
- In incomplete step runs *all* sources in G

Counting Incomplete Steps



- Let G be the DAG of *remaining strands*.
- ready queue $=$ the set of sources in G
- In incomplete step runs *all* sources in G
- Every longest path starts at a source

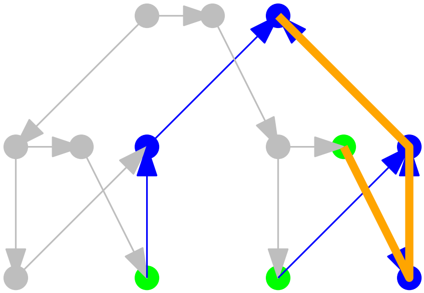
Counting Incomplete Steps



- Let G be the DAG of *remaining strands*.
- ready queue $=$ the set of sources in G
- In incomplete step runs *all* sources in G
- Every longest path starts at a source

- After an incomplete step, length of longest path shrinks by 1

Counting Incomplete Steps



- Let G be the DAG of *remaining strands*.
- ready queue $=$ the set of sources in G
- In incomplete step runs *all* sources in G
- Every longest path starts at a source

- After an incomplete step, length of longest path shrinks by 1
- There can be at most T_∞ steps.

Parallel Slackness

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

- If slackness < 1, speedup $< p$

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 suf. large slackness, greedy scheduler approaches time T_1/p.

Slackness and Scheduling

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

Theorem

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

Suppose

$$T_1/(p \times T_\infty) \geq c$$

Slackness and Scheduling

$$\text{slackness} := \frac{T_1}{p \times T_\infty} \qquad (1) \qquad T_\infty \leq \frac{T_1}{cp}$$

Theorem

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

Suppose

$$T_1/(p \times T_\infty) \geq c$$

Slackness and Scheduling

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

Theorem

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

Suppose

$$T_1/(p \times T_\infty) \geq c$$

(1) $$T_\infty \leq \frac{T_1}{cp}$$

With the greedy scheduler,

$$T_p \leq \left(\frac{T_1}{p} + T_\infty\right)$$

Slackness and Scheduling

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

Theorem

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

Suppose

$$T_1/(p \times T_\infty) \geq c$$

(1) $$T_\infty \leq \frac{T_1}{cp}$$

With the greedy scheduler,

$$T_p \leq \left(\frac{T_1}{p} + T_\infty\right)$$

Substituting (1),

$$T_p \leq \frac{T_1}{p}\left(1 + \frac{1}{c}\right)$$