CS3383 Unit 4: dynamic multithreaded algorithms

David Bremner

March 25, 2018

Outline

Dynamic Multithreaded Algorithms
- Fork-Join Model
- Span, Work, And Parallelism
- Parallel Loops
- Scheduling
- Race Conditions

Contents

Introduction to Parallel Algorithms

Dynamic Multithreading

- Also known as the *fork-join* model
- Shared memory, *multicore*
- Cormen et. al 3rd edition, Chapter 27

Introduction to Parallel Algorithms

Dynamic Multithreading

- Also known as the *fork-join* model
- Shared memory, *multicore*
- Cormen et. al 3rd edition, Chapter 27

Nested Parallelism

- Spawn a subroutine, carry on with other work.
- Similar to `fork` in POSIX.

Introduction to Parallel Algorithms

Nested Parallelism

- Spawn a subroutine, carry on with other work.
- Similar to `fork` in POSIX.

Parallel Loop

- iterations of a for loop *can* execute in parallel.
- Like `OpenMP`

Cilk+

- The multithreaded model is based on `Cilk+`, available in the latest versions of gcc.

Cilk+

- The multithreaded model is based on `Cilk+`, available in the latest versions of gcc.
- Programmer specifies *possible* paralellism

Cilk+

- The multithreaded model is based on `Cilk+`, available in the latest versions of gcc.
- Programmer specifies *possible* paralellism
- Runtime system takes care of mapping to OS threads

Cilk+

- The multithreaded model is based on `Cilk+`, available in the latest versions of gcc.
- Programmer specifies *possible* paralellism
- Runtime system takes care of mapping to OS threads
- Cilk+ contains several more features than our model, e.g. parallel vector and array operations.

Cilk+

- The multithreaded model is based on `Cilk+`, available in the latest versions of gcc.
- Programmer specifies *possible* paralellism
- Runtime system takes care of mapping to OS threads
- Cilk+ contains several more features than our model, e.g. parallel vector and array operations.
- Similar primitives are available in `java.util.concurrent`

Writing parallel (pseudo)-code

Keywords

parallel	Run the loop (potentially) concurrently
spawn	Run the procedure (potentially) concurrently
sync	Wait for all spawned children to complete.

Writing parallel (pseudo)-code

Keywords

parallel	Run the loop (potentially) concurrently
spawn	Run the procedure (potentially) concurrently
sync	Wait for all spawned children to complete.

Serialization

- remove keywords from parallel code yields correct serial code
- Adding parallel keywords to correct serial code might break it

Writing parallel (pseudo)-code

Keywords

parallel	Run the loop (potentially) concurrently
spawn	Run the procedure (potentially) concurrently
sync	Wait for all spawned children to complete.

Serialization

- remove keywords from parallel code yields correct serial code
- Adding parallel keywords to correct serial code might break it
  - missing sync

Writing parallel (pseudo)-code

Keywords

parallel	Run the loop (potentially) concurrently
spawn	Run the procedure (potentially) concurrently
sync	Wait for all spawned children to complete.

Serialization

- remove keywords from parallel code yields correct serial code
- Adding parallel keywords to correct serial code might break it
  - missing sync
  - loop iterations not independent

Fibonacci Example

function $\textsc{Fib}(n)$
    if $n \leq 1$ **then**
        return n
    else
        $x = \mathrm{Fib}(n-1)$
        $y = \mathrm{Fib}(n-2)$

        return $x + y$
    end if
end function

Fibonacci Example

function $\textsc{Fib}(n)$
    if $n \leq 1$ **then**
        return n
    else
        $x =$ spawn $\mathrm{Fib}(n-1)$
        $y = \mathrm{Fib}(n-2)$
        sync
        return $x + y$
    end if
end function

- Code in C, Java, Clojure and Racket available from `http://www.cs.unb.ca/~bremner/teaching/cs3383/examples`

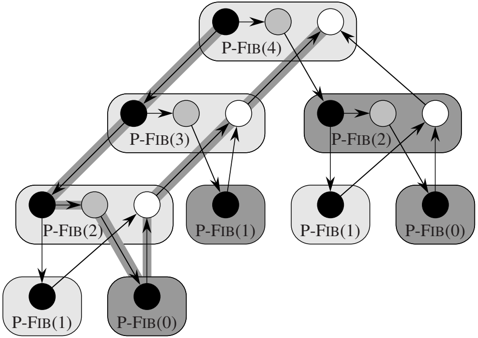
Contents

Computation DAG

Strands

Seq. inst. with no *parallel*, *spawn*, return from *spawn*, or *sync*.

function $\textsc{Fib}(n)$
    if $n \leq 1$ **then** $\triangleright$
        return n
    else
        $x =$ spawn $\mathrm{Fib}(n-1)$
        $y = \mathrm{Fib}(n-2)$ $\triangleright$
        sync
        return $x + y$ $\triangleright$
    end if
end function



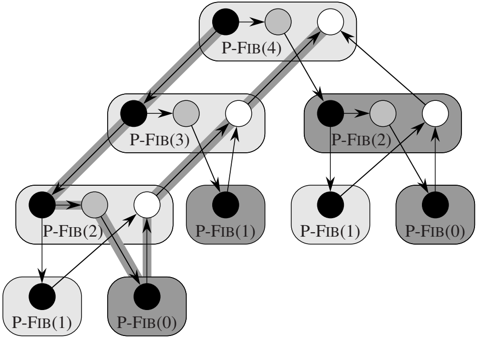
Computation DAG

Strands

Seq. inst. with no *parallel*, *spawn*, return from *spawn*, or *sync*.

nodes strands

down edges spawn

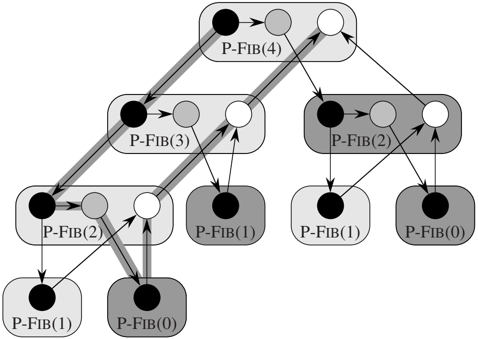


Computation DAG

Strands

Seq. inst. with no *parallel*, *spawn*, return from *spawn*, or *sync*.

nodes	strands
down edges	spawn
up edges	return

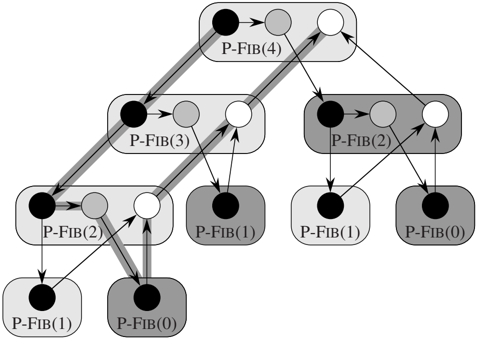


Computation DAG

Strands

Seq. inst. with no *parallel*, *spawn*, return from *spawn*, or *sync*.

nodes	strands
down edges	spawn
up edges	return
horizontal edges	sequential

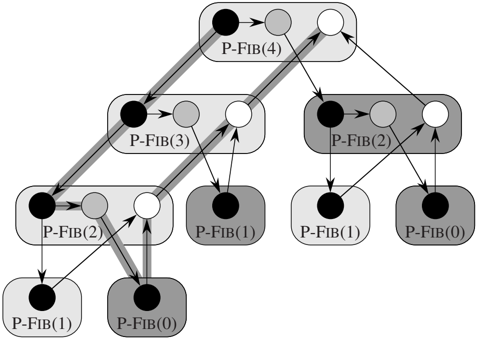


Computation DAG

Strands

Seq. inst. with no *parallel*, *spawn*, return from *spawn*, or *sync*.

nodes strands
down edges spawn
up edges return
horizontal edges sequential
critical path longest path in DAG

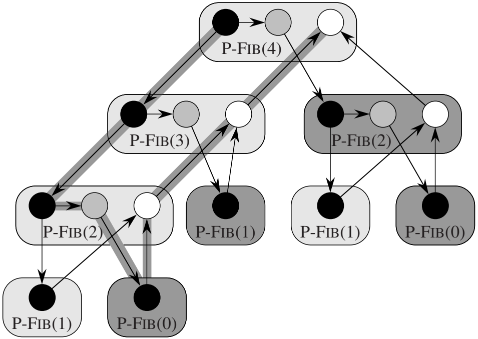


Computation DAG

Strands

Seq. inst. with no *parallel*, *spawn*, return from *spawn*, or *sync*.

nodes	strands
down edges	spawn
up edges	return
horizontal edges	sequential
critical path	longest path in DAG
span	weighted length of critical path ≡ lower bound on time



Work and Speedup

T_1 *Work*, sequential time.

Work and Speedup

T_1 *Work*, sequential time.

T_p Time on p processors.

Work and Speedup

T_1 *Work*, sequential time.

T_p Time on p processors.

Work Law

$$T_p \geq T_1/p$$

$$\text{speedup} := T_1/T_p \leq p$$

Parallelism

T_p Time on p processors.

Parallelism

T_p Time on p processors.

T_∞ *Span*, time given unlimited processors.

We could idle processors:

$$T_p \geq T_\infty \tag{1}$$

Parallelism

T_p Time on p processors.

T_∞ *Span*, time given unlimited processors.

We could idle processors:

$$T_p \geq T_\infty \qquad (1)$$

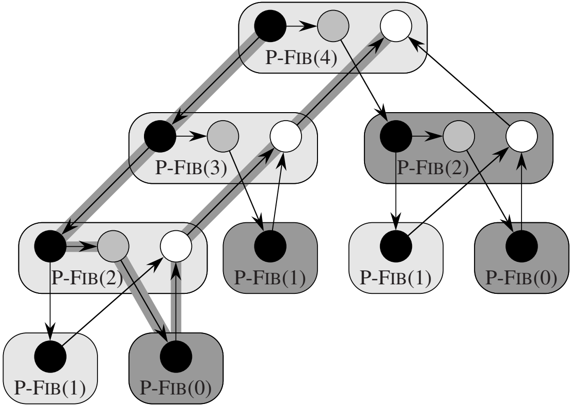
Best possible speedup:

$$\text{parallelism} = T_1/T_\infty \geq T_1/T_p = \text{speedup}$$

Span and Parallelism Example

Assume strands are unit cost.

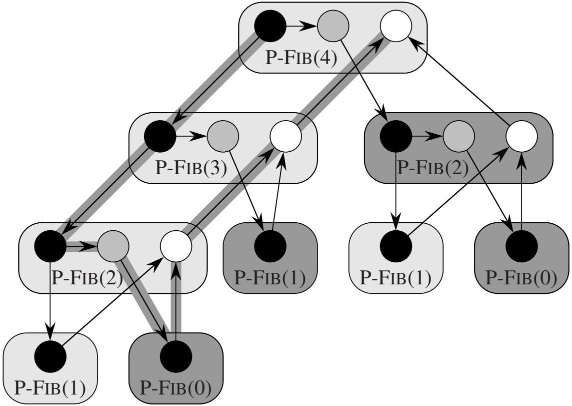
- $T_1 = 17$



Span and Parallelism Example

Assume strands are unit cost.

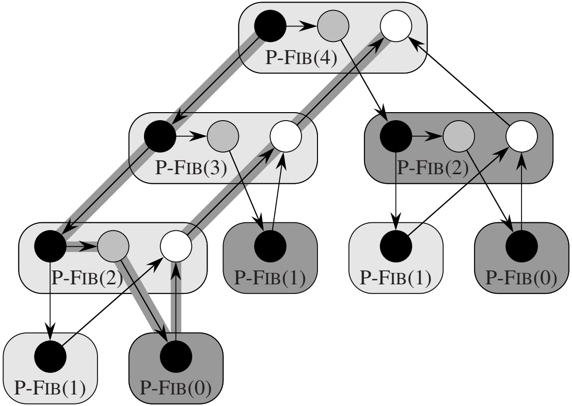
- $T_1 = 17$
- $T_\infty = 8$



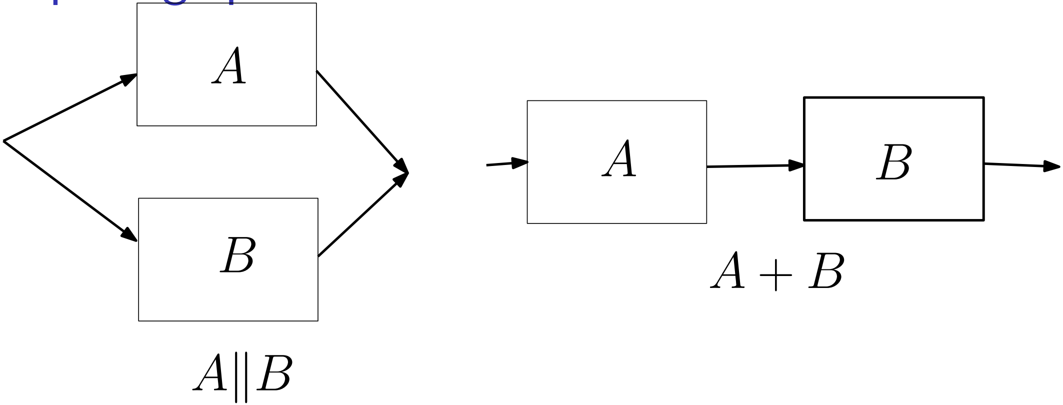
Span and Parallelism Example

Assume strands are unit cost.

- $T_1 = 17$
- $T_\infty = 8$
- Parallelism $= 2.125$ for this input size.

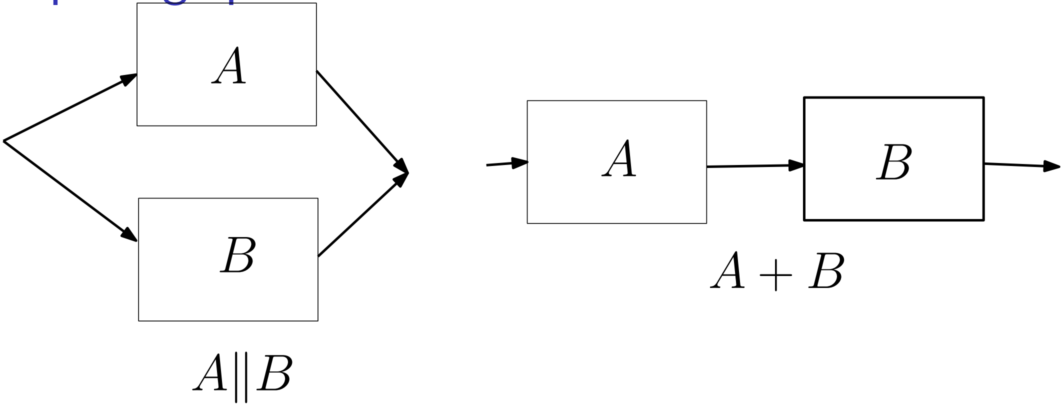


Composing span and work



$$\text{series} \quad T_\infty(A+B) = T_\infty(A) + T_\infty(B)$$

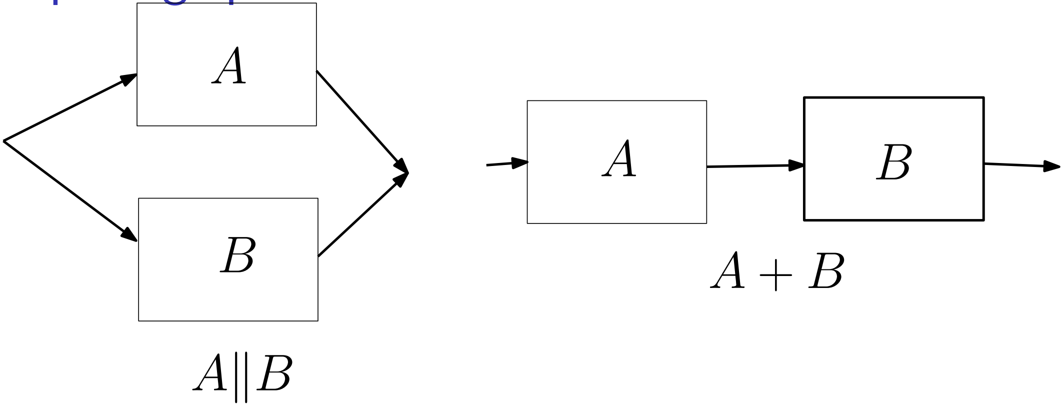
Composing span and work



series $T_\infty(A+B) = T_\infty(A) + T_\infty(B)$

parallel $T_\infty(A \| B) = \max(T_\infty(A), T_\infty(B))$

Composing span and work



series $T_\infty(A + B) = T_\infty(A) + T_\infty(B)$

parallel $T_\infty(A \| B) = \max(T_\infty(A), T_\infty(B))$

series or parallel $T_1 = T_1(A) + T_1(B)$

Work of Parallel Fibonacci

Write $T(n)$ for T_1 on input n.

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \qquad \text{(I.H.)}$$

Work of Parallel Fibonacci

Write $T(n)$ for T_1 on input n.

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \qquad \text{(I.H.)}$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

Work of Parallel Fibonacci

Write $T(n)$ for T_1 on input n.

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \qquad \text{(I.H.)}$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

We can show by induction (twice) that

$$T(n) \in \Theta(\phi^n)$$

Work of Parallel Fibonacci

Write $T(n)$ for T_1 on input n.

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \quad T(n) \leq a\phi^n - b \qquad \text{(I.H.)}$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

We can show by induction (twice) that

$$T(n) \in \Theta(\phi^n)$$

Work of Parallel Fibonacci

Write $T(n)$ for T_1 on input n.

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

We can show by induction (twice) that

$$T(n) \in \Theta(\phi^n)$$

$$T(n) \leq a\phi^n - b \qquad \text{(I.H.)}$$

Substitute the I.H.

$$T(n) \leq a(\phi^{n-1} + \phi^{n-2}) - 2b + \Theta(1)$$

Work of Parallel Fibonacci

Write $T(n)$ for T_1 on input n.

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

We can show by induction (twice) that

$$T(n) \in \Theta(\phi^n)$$

$$T(n) \leq a\phi^n - b \qquad \text{(I.H.)}$$

Substitute the I.H.

$$T(n) \leq a(\phi^{n-1} + \phi^{n-2}) - 2b + \Theta(1)$$
$$= a\frac{\phi + 1}{\phi^2}\phi^n - b + (\Theta(1) - b)$$

Work of Parallel Fibonacci

Write $T(n)$ for T_1 on input n.

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

We can show by induction (twice) that

$$T(n) \in \Theta(\phi^n)$$

$$T(n) \leq a\phi^n - b \qquad \text{(I.H.)}$$

Substitute the I.H.

$$\begin{aligned} T(n) &\leq a(\phi^{n-1} + \phi^{n-2}) - 2b + \Theta(1) \\ &= a\frac{\phi+1}{\phi^2}\phi^n - b + (\Theta(1) - b) \\ &\leq a\frac{\phi+1}{\phi^2}\phi^n - b \qquad \text{for } b \text{ large} \end{aligned}$$

Work of Parallel Fibonacci

Write $T(n)$ for T_1 on input n.

$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$

Let $\phi \approx 1.62$ be the solution to

$$\phi^2 = \phi + 1$$

We can show by induction (twice) that

$$T(n) \in \Theta(\phi^n)$$

$$T(n) \leq a\phi^n - b \qquad \text{(I.H.)}$$

Substitute the I.H.

$$\begin{aligned} T(n) &\leq a(\phi^{n-1} + \phi^{n-2}) - 2b + \Theta(1) \\ &= a\frac{\phi+1}{\phi^2}\phi^n - b + (\Theta(1) - b) \\ &\leq a\frac{\phi+1}{\phi^2}\phi^n - b \qquad \text{for } b \text{ large} \\ &= a\phi^n - b \end{aligned}$$

Span and Parallelism of Fibonacci

$$
\begin{aligned}
T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\
&= T_\infty(n-1) + \Theta(1)
\end{aligned}
$$

Span and Parallelism of Fibonacci

$$
\begin{aligned}
T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\
&= T_\infty(n-1) + \Theta(1)
\end{aligned}
$$

Transforming to sum, we get

$$T_\infty \in \Theta(n)$$

Span and Parallelism of Fibonacci

$$
\begin{aligned}
T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\
&= T_\infty(n-1) + \Theta(1)
\end{aligned}
$$

Transforming to sum, we get

$$T_\infty \in \Theta(n)$$

$$\text{parallelism} = \frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{\phi^n}{n}\right)$$

Span and Parallelism of Fibonacci

$$\begin{aligned} T_\infty(n) &= \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1) \\ &= T_\infty(n-1) + \Theta(1) \end{aligned}$$

Transforming to sum, we get

$$T_\infty \in \Theta(n)$$

$$\text{parallelism} = \frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{\phi^n}{n}\right)$$

- So an inefficient way to compute Fibonacci, but very parallel

Contents

Parallel Loops

parallel for $i = 1$ to n **do**
    statement...
    statement...
end for

- Run n copies in parallel with local setting of i.

Parallel Loops

parallel for $i = 1$ to n **do**
  statement...
  statement...
end for

- Run n copies in parallel with local setting of i.
- Effectively n-way spawn

Parallel Loops

parallel for $i = 1$ to n **do**
    statement...
    statement...
end for

- Run n copies in parallel with local setting of i.
- Effectively n-way spawn
- Can be implemented with spawn and sync

Parallel Loops

parallel for $i = 1$ to n **do**

  statement...

  statement...

end for

- Run n copies in parallel with local setting of i.
- Effectively n-way spawn
- Can be implemented with spawn and sync
- Span

$$T_\infty(n) = \Theta(\log n) + \max_i T_\infty(\text{iteration i})$$

Parallel Matrix-Vector product

To compute $y = Ax$, in parallel

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

Parallel Matrix-Vector product

To compute $y = Ax$, in parallel

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

function $\textsc{RowMult}$(A,x,y,i)
  $y_i = 0$
  for $j = 1$ to n **do**
    $y_i = y_i + a_{ij} x_j$
  end for
end function

Parallel Matrix-Vector product

To compute $y = Ax$, in parallel

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

function $\textsc{RowMult}$(A,x,y,i)
  $y_i = 0$
  for $j = 1$ to n **do**
    $y_i = y_i + a_{ij} x_j$
  end for
end function

function $\textsc{Mat-Vec}(A, x, y)$
  Let $n =$ rows(A)
  parallel for $i = 1$ to n **do**
    RowMult(A,x,y,i)
  end for
end function

Parallel Matrix-Vector product

To compute $y = Ax$, in parallel

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

function $\textsc{RowMult}$(A,x,y,i)
  $y_i = 0$
  for $j = 1$ to n **do**
    $y_i = y_i + a_{ij} x_j$
  end for
end function

function $\textsc{Mat-Vec}(A, x, y)$
  Let $n = \text{rows}(A)$
  parallel for $i = 1$ to n **do**
    RowMult(A,x,y,i)
  end for
end function

$$T_1(n) \in \Theta(n^2) \quad \text{(serialization)}$$

$$T_\infty(n) = \underbrace{\Theta(\log(n))}_{\text{parallel for}} + \underbrace{\Theta(n)}_{\text{RowMult}}$$

Parallel Matrix-Vector product

function $\textsc{RowMult}$(A,x,y,i)
    $y_i = 0$
    for $j = 1$ to n **do**
        $y_i = y_i + a_{ij}x_j$
    end for
end function

function $\textsc{Mat-Vec}(A, x, y)$
    Let $n =$ rows(A)
    parallel for $i = 1$ to n **do**
        RowMult(A,x,y,i)
    end for
end function

Parallel Matrix-Vector product

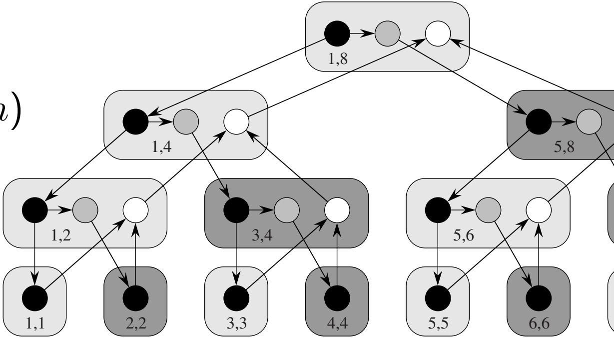
function $\textsc{RowMult}$(A,x,y,i)
    $y_i = 0$
    for $j = 1$ to n **do**
        $y_i = y_i + a_{ij}x_j$
    end for
end function

function $\textsc{Mat-Vec}(A, x, y)$
    Let $n =$ rows(A)
    parallel for $i = 1$ to n **do**
        RowMult(A,x,y,i)
    end for
end function

- Why is RowMult not using parallel for?

Divide and Conquer Matrix-Vector product

function $\mathrm{MVDC}(A, x, y, f, t)$
  if $f == t$ **then**
    RowMult(A,x,y,f)
  else
    $m = \lfloor (f + t)/2 \rfloor$
    spawn MVDC(A, x, y, f, m)
    MVDC$(A, x, y, m + 1, t)$
    sync
  end if
end function



Divide and Conquer Matrix-Vector product

```
function MVDC(A, x, y, f, t)
    if f == t then
        RowMult(A,x,y,f)
    else
        m = ⌊(f + t)/2⌋
        spawn MVDC(A, x, y, f, m)
        MVDC(A, x, y, m + 1, t)
        sync
    end if
end function
```

- $T_\infty(n) = \Theta(\log n)$ (binary tree)

Divide and Conquer Matrix-Vector product

function $\mathrm{MVDC}(A, x, y, f, t)$
    if $f == t$ **then**
        RowMult(A,x,y,f)
    else
        $m = \lfloor (f+t)/2 \rfloor$
        spawn MVDC(A, x, y, f, m)
        MVDC$(A, x, y, m+1, t)$
        sync
    end if
end function

- $T_\infty(n) = \Theta(\log n)$ (binary tree)
- $\Theta(n)$ leaves (one per row)

Divide and Conquer Matrix-Vector product

function $\mathrm{MVDC}(A, x, y, f, t)$
  if $f == t$ **then**
    RowMult(A,x,y,f)
  else
    $m = \lfloor (f+t)/2 \rfloor$
    spawn MVDC(A, x, y, f, m)
    MVDC$(A, x, y, m+1, t)$
    sync
  end if
end function

- $T_\infty(n) = \Theta(\log n)$ (binary tree)
- $\Theta(n)$ leaves (one per row)
- $\Theta(n)$ interior nodes (binary tree)

Divide and Conquer Matrix-Vector product

function $\mathrm{MVDC}(A, x, y, f, t)$
    if $f == t$ **then**
        RowMult(A,x,y,f)
    else
        $m = \lfloor (f + t)/2 \rfloor$
        spawn MVDC(A, x, y, f, m)
        MVDC($A, x, y, m + 1, t$)
        sync
    end if
end function

- $T_\infty(n) = \Theta(\log n)$ (binary tree)
- $\Theta(n)$ leaves (one per row)
- $\Theta(n)$ interior nodes (binary tree)
- $T_1(n) = \Theta(n^2)$

Contents

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

$$T_p \geq T_1/p \qquad \text{(work law)}$$

$$T_p \geq T_\infty \qquad \text{(span)}$$

Therefore

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

Optimal and Approximate Scheduling

Recall

$$T_p \geq T_1/p \qquad \text{(work law)}$$
$$T_p \geq T_\infty \qquad \text{(span)}$$

Therefore

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

With the greedy algorithm we can achieve

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

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.

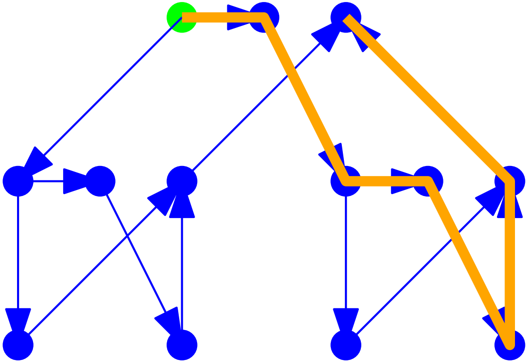
Counting Complete Steps

- 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

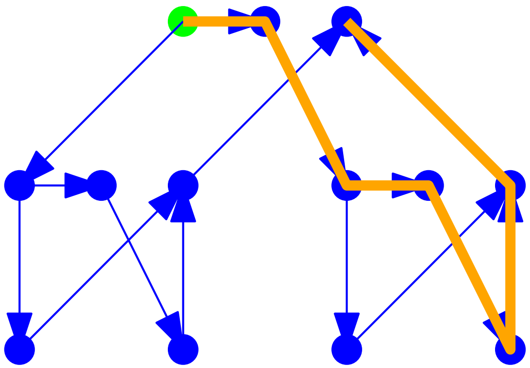
- Let k be the number of complete steps.
- At each complete step we do p units of work.
- Every unit of work corresponds to one step of the serialization, so $kp \leq T_1$.
- Therefore $k \leq T_1/p$

Counting Incomplete Steps



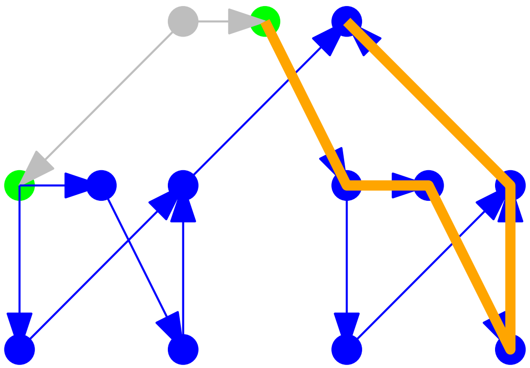
- Let G be the DAG of *remaining strands*.

Counting Incomplete Steps



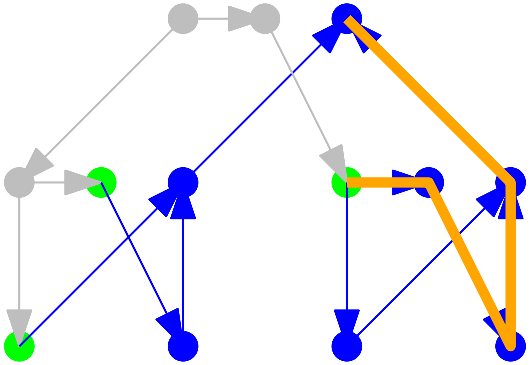
- Let G be the DAG of *remaining strands*.
- The ready queue of strands is exactly the set of sources in G

Counting Incomplete Steps



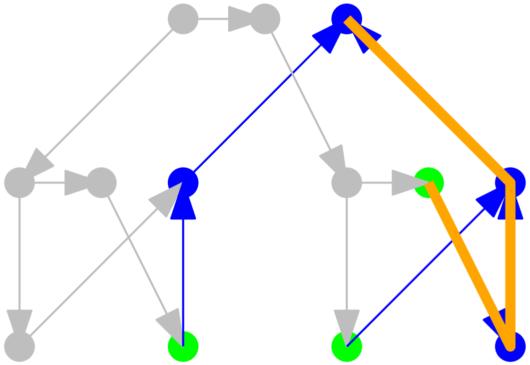
- Let G be the DAG of *remaining strands*.
- The ready queue of strands is exactly the set of sources in G
- In incomplete step runs *all* sources in G

Counting Incomplete Steps



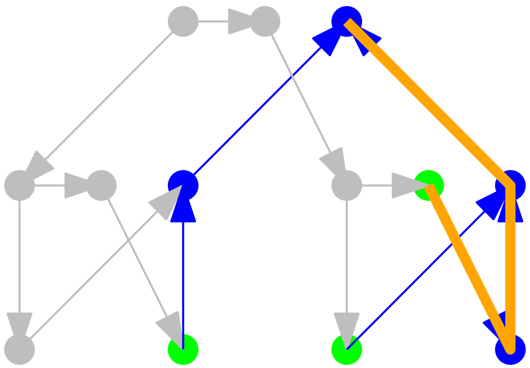
- Let G be the DAG of *remaining strands*.
- The ready queue of strands is exactly the set of sources in G
- In incomplete step runs *all* sources in G
- Every longest path starts at a source (otherwise, extend)

Counting Incomplete Steps



- Let G be the DAG of *remaining strands*.
- The ready queue of strands is exactly the set of sources in G
- In incomplete step runs *all* sources in G
- Every longest path starts at a source (otherwise, extend)
- After an incomplete step, length of longest path shrinks by 1

Counting Incomplete Steps



- Let G be the DAG of *remaining strands*.
- The ready queue of strands is exactly the set of sources in G
- In incomplete step runs *all* sources in G
- Every longest path starts at a source (otherwise, extend)
- 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 sufficiently large slackness, the greed scheduler approaches time T_1/p.

Slackness and Scheduling

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

Theorem

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

Suppose

$$\frac{T_1}{p \times T_\infty} \geq c$$

Slackness and Scheduling

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

Theorem

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

Suppose

$$\frac{T_1}{p \times T_\infty} \geq c$$

Then

$$T_\infty \leq \frac{T_1}{cp} \qquad (2)$$

Slackness and Scheduling

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

Theorem

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

Suppose

$$\frac{T_1}{p \times T_\infty} \geq c$$

Then

$$T_\infty \leq \frac{T_1}{cp} \tag{2}$$

Recall that 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 sufficiently large slackness, the greed scheduler approaches time T_1/p.

Suppose

$$\frac{T_1}{p \times T_\infty} \geq c$$

Then

$$T_\infty \leq \frac{T_1}{cp} \qquad (2)$$

Recall that with the greedy scheduler,

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

Substituting (2), we have

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

Contents

Race Conditions

Non-Determinism

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

Race Conditions

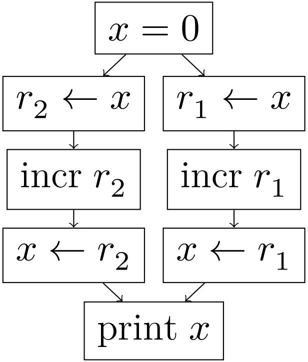
Non-Determinism

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

Example

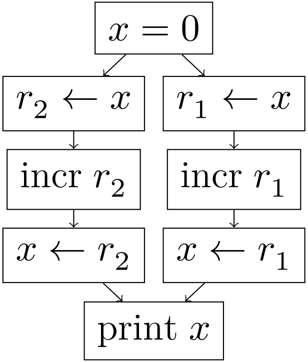
```
x = 0
parallel for i ← 1 to 2 do
    x ← x + 1
```

Racy execution



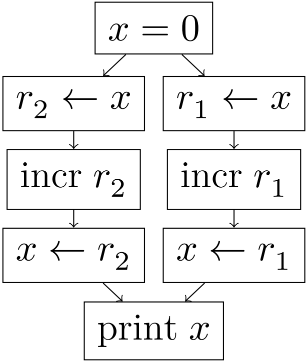
- all possible topological sorts are valid execution orders

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

$$\begin{tikzcd}
& \boxed{x = 0} \arrow[dl] \arrow[dr] & \\
\boxed{r_2 \leftarrow x} \arrow[d] & & \boxed{r_1 \leftarrow x} \arrow[d] \\
\boxed{\text{incr } r_2} \arrow[d] & & \boxed{\text{incr } r_1} \arrow[d] \\
\boxed{x \leftarrow r_2} \arrow[dr] & & \boxed{x \leftarrow r_1} \arrow[dl] \\
& \boxed{\text{print } x} &
\end{tikzcd}$$

- 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

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

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

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.

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.