Multithreaded algorithms

9/23/24

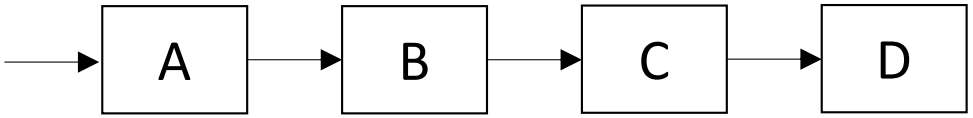
Administrivia

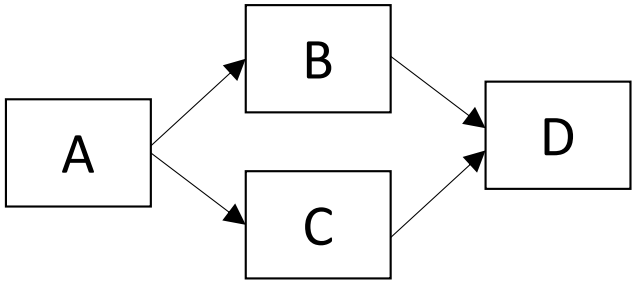
- HW 2 (AVL trees and Divide & Conquer) due tomorrow night
- Reading: Rest of 26.1 due Wednesday
- Join the CS Club! (Tuesdays 4pm in SMC A201)

Big Picture

- Essentially every computer is now multicore
  - Means it can run multiple parts of a program at the same time
- Threads
  - Main abstraction for shared memory computing
  - Each can run independently

Computation as a DAG (directed acyclic graph)



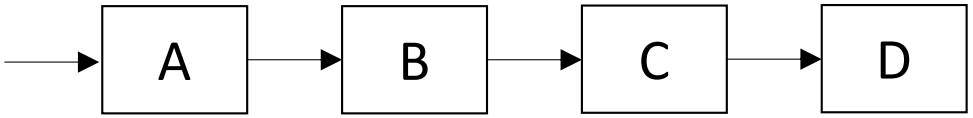


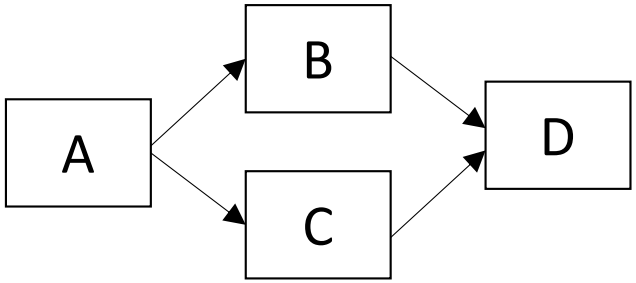
Can represent runtime behavior using a DAG (directed acyclic graph)

Edges show precedence constraints

Vertices are strands (sequence of instructions that don't have an outside precedence constraint or satisfy one)

Computation as a DAG (directed acyclic graph)





Can represent runtime behavior using a DAG (directed acyclic graph)

Edges show precedence constraints

Vertices are strands (sequence of instructions that don't have an outside precedence constraint or satisfy one)

Metrics:

- Work T_1 = time on one processor
- Span T_∞ = length of longest path

Naïve Fibonacci implementation

```
Fib(n)
        if(n <= 1)
                return n
        else
                x = Fib(n-1)
                y = Fib(n-2)
                return x + y
```

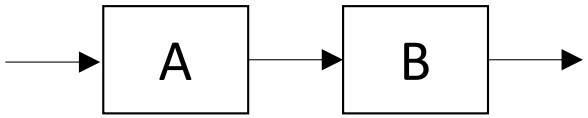
Parallel version

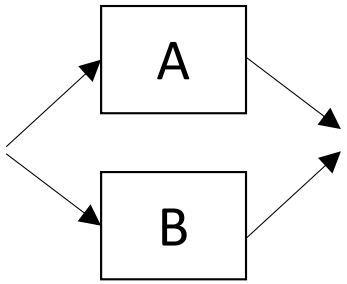
```
P_Fib(n)
        if(n <= 1)
                return n
        else
                x = spawn P_Fib(n-1)
                y = P_Fib(n-2)
                sync
                return x + y
```

Allows child procedure to run in parallel with its parent

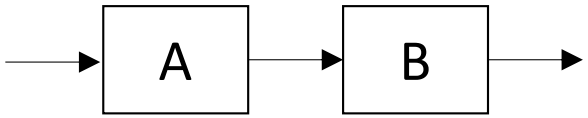
Causes parent to wait for all children to complete

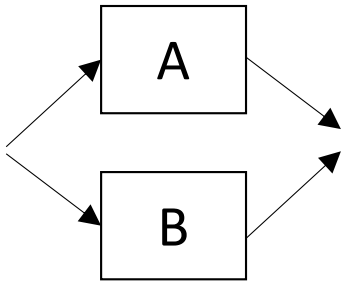
Combining subcomputations





Combining subcomputations

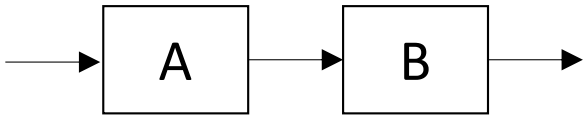


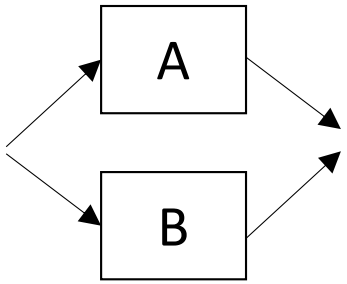


Work: $T_1(A) + T_1(B)$

Work: $T_1(A) + T_1(B)$

Combining subcomputations





Work: $T_1(A) + T_1(B)$

Span: $T_\infty(A) + T_\infty(B)$

Work: $T_1(A) + T_1(B)$

Span: $\max\{ T_\infty(A), T_\infty(B) \}$

Which of the following is a lower bound on T_p, the time on p processors?

A. $T_1 + p$

B. $T_1 - p$

C. $p\ T_1$

D. T_1 / p

E. Not exactly one of the above

Which of the following is a lower bound on T_p, the time on p processors?

A. $T_1 + p$

B. $T_1 - p$

C. $p\ T_1$

D. T_1 / p (called the work law)

E. Not exactly one of the above

Which of the following is a lower bound on T_p, the time on p processors?

A. T_∞

B. T_∞ - p

C. pT_∞

D. T_∞ / p

E. Not exactly one of the above

Which of the following is a lower bound on T_p, the time on p processors?

A. T_∞ (called the span law)

B. $T_\infty - p$

C. pT_∞

D. T_∞ / p

E. Not exactly one of the above

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Corollary: The running time of any such scheduler is within a factor of 2 of optimal

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Corollary: The running time of any such scheduler is within a factor of 2 of optimal

Another metric: parallelism = T_1 / T_∞

(intuitively, the number of processors we can use)

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Proof: Charge each time step to one of the terms.

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Proof: Charge each time step to one of the terms.
Charge complete steps (when all processors are busy) to T_1/p.

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Proof: Charge each time step to one of the terms.
Charge complete steps (when all processors are busy) to T_1/p.
Suppose to contrary that every processor is busy > $\lfloor T_1/p \rfloor$ time steps.

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Proof: Charge each time step to one of the terms.
Charge complete steps (when all processors are busy) to T_1/p.
Suppose to contrary that every processor is busy $> \lfloor T_1/p \rfloor$ time steps.
Then these steps do work $\geq p(\lfloor T_1/p \rfloor + 1) = p \lfloor T_1/p \rfloor + p$

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Proof: Charge each time step to one of the terms.
Charge complete steps (when all processors are busy) to T_1/p.
Suppose to contrary that every processor is busy $> \lfloor T_1/p \rfloor$ time steps.
Then these steps do work $\geq p(\lfloor T_1/p \rfloor + 1) = p \lfloor T_1/p \rfloor + p$

$= T_1 - (T_1 \bmod p) + p$

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Proof: Charge each time step to one of the terms.
Charge complete steps (when all processors are busy) to T_1/p.
Suppose to contrary that every processor is busy $> \lfloor T_1/p \rfloor$ time steps.
Then these steps do work $\geq p(\lfloor T_1/p \rfloor + 1) = p \lfloor T_1/p \rfloor + p$

$= T_1 - (T_1 \bmod p) + p$

$> T_1$

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Proof: Charge each time step to one of the terms.
Charge complete steps (when all processors are busy) to T_1/p.
Suppose to contrary that every processor is busy $> \lfloor T_1/p \rfloor$ time steps.
Then these steps do work $\geq p(\lfloor T_1/p \rfloor + 1) = p \lfloor T_1/p \rfloor + p$

$$= T_1 - (T_1 \bmod p) + p$$
$$> T_1$$

Charge incomplete steps (at least one processor idle) to T_∞.

Th: With p processors, any scheduler that is never voluntarily idle completes a computation in time

$$T_p \leqslant T_1/p + T_\infty$$

Proof: Charge each time step to one of the terms.
Charge complete steps (when all processors are busy) to T_1/p.
Suppose to contrary that every processor is busy $> \lfloor T_1/p \rfloor$ time steps.
Then these steps do work $\geq p(\lfloor T_1/p \rfloor + 1) = p \lfloor T_1/p \rfloor + p$

$= T_1 - (T_1 \bmod p) + p$

$> T_1$

Charge incomplete steps (at least one processor idle) to T_∞.
If a processor is idle, it must schedule every strand w/o incomplete prerequisites. Every path must start with one of these, so this shortens every critical path, which can happen at most T_∞ times.

What is the span of the following code?

```
for(int i=0; i < n; i++)
    spawn A[i] = B[i];
sync
```

A. $\Theta(1)$
B. $\Theta(\log n)$
C. $\Theta(n^{1/2})$
D. $\Theta(n)$
E. None of the above

What is the span of the following code?

```
for(int i=0; i < n; i++)
    spawn A[i] = B[i];
sync
```

A. $\Theta(1)$
B. $\Theta(\log n)$
C. $\Theta(n^{1/2})$
D. $\Theta(n)$
E. None of the above

Alternate idea of a for loop

```
void do_it(int s, int e) {
        if(s == e)
                A[s] = B[s]
        else {
                spawn do_it(s, (s+e)/2)
                do_it((s+e)/2+1, e)
                sync
        }
}
…
do_it(0,n-1)
```

What recurrence gives the work of this?

```
void do_it(int s, int e) {
        if(s == e)
                A[s] = B[s]
        else {
                spawn do_it(s, (s+e)/2)
                do_it((s+e)/2+1, e)
                sync
        }
}
…
do_it(0,n-1)
```

(n set at the bottom)

A. $T_1(n) = T_1(n/2) + 1$
B. $T_1(n) = T_1(n/2) + n$
C. $T_1(n) = 2T_1(n/2) + 1$
D. $T_1(n) = 2T_1(n/2) + n$
E. None of the above

What recurrence gives the work of this?

```
void do_it(int s, int e) {
        if(s == e)
                A[s] = B[s]
        else {
                spawn do_it(s, (s+e)/2)
                do_it((s+e)/2+1, e)
                sync
        }
}
…
do_it(0,n-1)
```

(n set at the bottom)

A. $T_1(n) = T_1(n/2) + 1$

B. $T_1(n) = T_1(n/2) + n$

C. $T_1(n) = 2T_1(n/2) + 1$

D. $T_1(n) = 2T_1(n/2) + n$

E. None of the above

What is the work of this?

```
void do_it(int s, int e) {
        if(s == e)
                A[s] = B[s]
        else {
                spawn do_it(s, (s+e)/2)
                do_it((s+e)/2+1, e)
                sync
        }
}
…
do_it(0,n-1)
```

A. $\Theta(1)$
B. $\Theta(\log n)$
C. $\Theta(n)$
D. $\Theta(n \log n)$
E. None of the above

What is the work of this?

```
void do_it(int s, int e) {
        if(s == e)
                A[s] = B[s]
        else {
                spawn do_it(s, (s+e)/2)
                do_it((s+e)/2+1, e)
                sync
        }
}
…
do_it(0,n-1)
```

A. $\Theta(1)$
B. $\Theta(\log n)$
C. $\Theta(n)$
D. $\Theta(n \log n)$
E. None of the above

What is the span?

```
void do_it(int s, int e) {
        if(s == e)
                A[s] = B[s]
        else {
                spawn do_it(s, (s+e)/2)
                do_it((s+e)/2+1, e)
                sync
        }
}
…
do_it(0,n-1)
```

A. $\Theta(1)$
B. $\Theta(\log n)$
C. $\Theta(n^{0.5})$
D. $\Theta(n)$
E. None of the above

What is the span?

```
void do_it(int s, int e) {
        if(s == e)
                A[s] = B[s]
        else {
                spawn do_it(s, (s+e)/2)
                do_it((s+e)/2+1, e)
                sync
        }
}
…
do_it(0,n-1)
```

A. $\Theta(1)$
B. $\Theta(\log n)$
C. $\Theta(n^{0.5})$
D. $\Theta(n)$
E. None of the above

Parallel for loop

- New syntax:

  parallel for i = 1 to n

- Syntactic sugar for recursive routine that divides domain in half, running the calls in parallel, and then executes a single iteration