## Parallel Algorithms (vs. serial algorithms)

Random-access machine model used for serial algorithms

Many models for parallel algorithms, no general agreement.

## Model used: Dynamic Multithreading

- appropriate for multicore machines, shared memory programming
- not appropriate for distributed memory programs.

Ex:

```
    Fib (n)
A  | if n < 2
   |    then return n
   | x ← spawn Fib(n-1)
B  [ y ← spawn Fib(n-2)
empty [ sync
C  [ return (x+y)
```

spawn, sync, or return terminate current thread

threads

A: - includes n-1 computation
 - upto spawn

B: - includes n-2 computation
 - upto spawn

empty: ignored now

C: - includes x+y computation
 - upto return, after sync

spawn: - subroutine can execute at same time as parent

sync: - wait until all children are done

Description of logical parallelism, not actual (does not describe # processors)

A scheduler determines how to map dynamically unfolding execution onto processors.

Serial instruction stream: when in a loop, chain of subsequent instructions.

Logical serial instruction stream is actually not executed sequentially by a processor → instruction-level parallelism not a focus here. The focus is on logical parallelism.

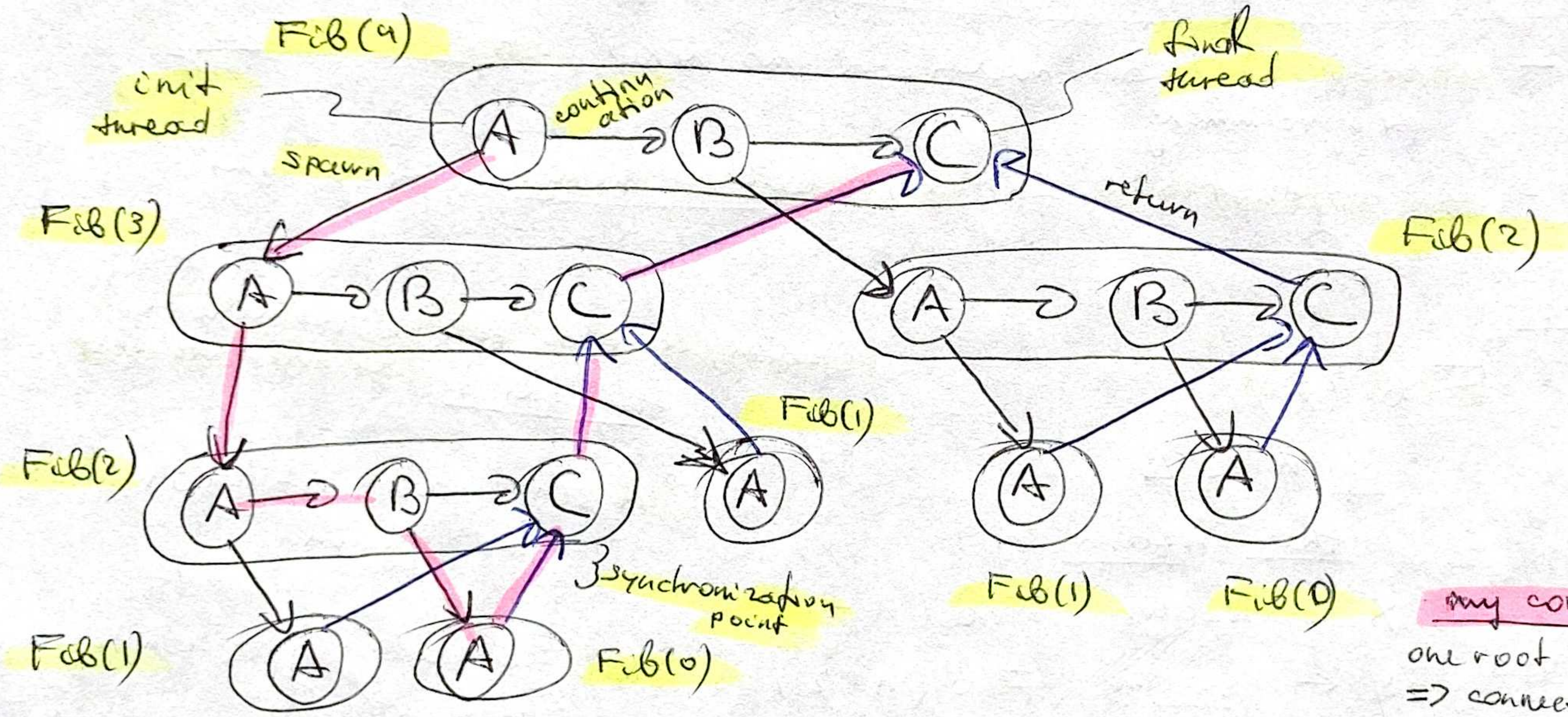
Parallel instruction stream: DAG

# Multithreaded computation

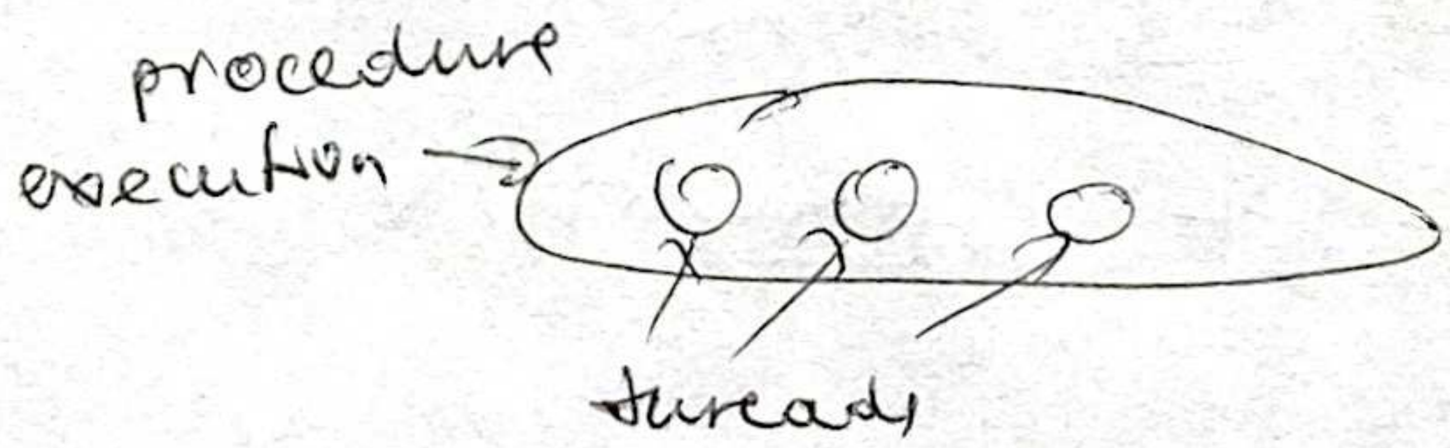
Parallel instruction stream = DAG

**vertices are threads** : maximal sequence of instructions
not containing parallel control
( spawn, sync, return)

**edges** : spawn , return, continuation




my comment
one root
=> connected
DAG




**Performance measures** :

$T_P$ = running time on P processors

$T_1$ = work = serial time (just like getting rid of spawn, sync)

$T_\infty$ = critical path length = longest path in DAG

my comment
longest path across topologically sorted DAG
⇓
other threads can be parallel

Ex Fib(4) [redacted] $T_1 = 17$ (assume each thread is unit time)
\# of threads

$T_\infty = 8$ (longest path in DAG
threads that must be sequentially executed)

$\underbrace{\qquad\qquad}_{\text{\# of unit-time threads}}$

this model does not take communication into account

## Lower bounds on $T_p$

$T_p \ge \frac{T_1}{p}$

- $p$ proc can do $\le p$ work in 1 step
- if [redacted], [redacted] $T_p < \frac{T_1}{p}$, Processor can do $> p$ work in 1 step.

my comment

[redacted]
Suppose $T_p < \frac{T_1}{p}$, then $\exists$ a thread that is not executed. contradiction to $T_p$ def.

$T_p \ge T_\infty$

- $p$ processors can't do more work than $\infty$ processors

## Speedup

$T_1 / T_p$ = speedup of $p$ processors

$T_1 / T_p = \Theta(p) \Rightarrow$ linear speedup, each processor contributes within a constant factor its measure of full support

$T_1 / T_p = p \Rightarrow$ perfect linear speedup

$T_1 / T_p > p \Rightarrow$ superlinear speedup
NOT possible in this model ($T_p < \frac{T_1}{p}$ contr.)
In other models possible
(e.g. caching effects)

Max possible speedup, given $T_1, T_\infty$, is $T_1 / T_\infty$ = parallelism

= average amount of work that can be done in parallel along each step of critical path.

adding more processors does not improve speedup [redacted]

$= \bar{P}$

[redacted]

# Scheduling

Map computation to $p$ processors
Done by runtime system (scheduler algorithm)
typically language runtime system

On-line schedulers are complex (randomized schedulers with guarantees!!!)
Illustrate ideas using off-line scheduler.

## Greedy scheduler ($p$ processors)

in DAG: cannot execute a node until nodes preceding it are executed

- Do as much as possible on every step.
  ⇒ do not guess if something is worth delaying
  - Complete step: $\geq p$ threads ready to run.
    execute any $p$ threads. May be not optimal.
    There maybe a particular thread, if executed now, enables more parallelism later.
  - Incomplete step: $< p$ threads ready to run.
    Execute all of them.

!! Scheduling optimally a DAG on $p$ processors is NP-complete.

Theorem (Graham, Brent):
A greedy scheduler executes any computation with work $T_1$ and critical path length $T_\infty$ in time

$$\boxed{T_p \leq T_1/p + T_\infty} \leq 2OPT \qquad T_p \geq \frac{T_1}{p}, \; T_p \geq T_\infty \Rightarrow 2\,OPT$$

on a computer with $p$ processors.

2-competitive.

$$OPT \geq \max\left(\frac{T_1}{p}, T_\infty\right)$$

my comment

$$\frac{T_1}{p} \geq T_\infty \Rightarrow \frac{T_1}{p} + T_\infty \leq 2\frac{T_1}{p} \leq 2OPT$$

my comment
[redacted]

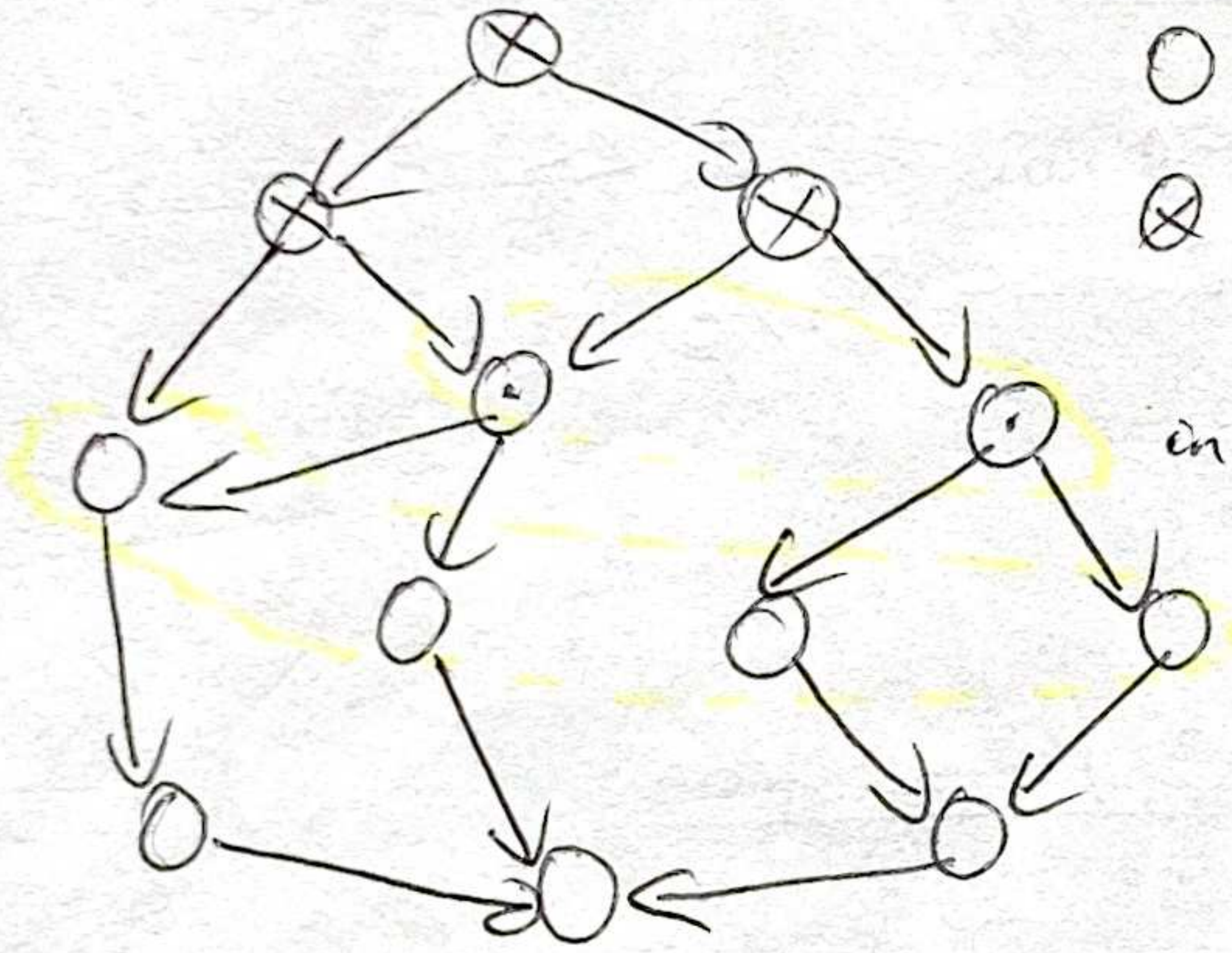
**Proof**: # complete steps $\leq \frac{T_1}{P}$
since otherwise more than $T_1$ work would be done.

[redacted] assume # complete steps $> T_1/P$
smallest size of a complete step is $P$, $> T_1$ work, contradiction

Consider an incomplete step, and let $G'$ be subgraph of $G$ that remains to be executed.

G:

$P = 3$



O : $G'$ (to be executed)
⊗ : already executed
WLOG each is a unit-time thread
incomplete step
complete step
O in $G'$, ready to be executed, in-degree 0

Threads with in-degree 0 in $G'$ are ready to be executed
The critical path length of $G'$ is reduced by 1

my comment [redacted]

$\Rightarrow$ # incomplete steps $\leq T_\infty$

consider a topologically sorted G, along [redacted] a critical path, [redacted] at each [redacted] slice there is at most one incomplete step that touches the slice

$$\underbrace{\Rightarrow T_P \leq \frac{T_1}{P} + T_\infty}$$

foundational theorem of Scheduling!

**Corollary**: $T_1/T_P = \Theta(P)$ $\quad \bar{P} = \frac{T_1}{T_\infty}$

linear speedup when $P = O(\bar{P})$ parallelism or fewer procs
with greedy scheduler

bound on P growth as $T_1$ and $T_\infty$ grow

$\bar{P} = T_1/T_\infty \Rightarrow P = O(T_1/T_\infty) \Rightarrow T_\infty = O(T_1/P)$

consider a class of DAGs [redacted]

Thus $T_P \leq \frac{T_1}{P} + O\left(\frac{T_1}{P}\right) = O\left(\frac{T_1}{P}\right)$ [redacted] when running on fewer processor than $\bar{P}$, can get speedup.

Cilk

Randomized online scheduler

$E[T_p] = T_1/p + O(T_\infty)$ provably

$T_p \approx T_1/p + T_\infty$ empirically

Near-perfect linear speedup if $p \ll \bar{P}$

i.e. $T_\infty \ll T_1/p$

Chessprograms vs. DeepBlue

Orig. program

$T_{32} = 65$ sec

$T_1 = 2048$

$T_\infty = 1$

$T_{32} = T_1/32 + T_\infty = 65$

Opt. program

$T'_{32} = 40$ sec

Reject.

$T_1' = 1024$

$T_1^\infty = 8$

$T'_{32} = T'_1/32 + T_\infty' = 40$

Extrapolate on a larger machine

$T_{512} = T_1/512 + T_\infty = 5$

$T'_{512} = T'_1/512 + T_\infty' = 10$

optimization vs. scale