Colorado State University

CS 320: Algorithms
Week 12: Dynamic multithreading
aka: Parallel Algorithms

Sanjay Rajopadhye

Cormen et al Ch 27

Nov 2023

Drill down into Moore's Law

Transistor density doubles every two years

- *Sustained for ~50 years, despite many predictions of its end*

Real Moore's Law

- *Society does not care for transistors, all it wants is performance: double it every two years (or else)*

Our challenges (all of CS)

- Sustain Moore's law of density (more and more difficult). Eventually, it must(?) come to an end
- Continue to deliver performance (translate the transistors into performance)
  - Faster processors
  - Better architectures, run-time systems, languages, compilers, networks.
  - Better algorithms

Translating transistors into performance

Denard scaling: small transistors also use less power

- *Reduce operating voltage*
- *Increase clock frequency*
- *Maintain constant total power*

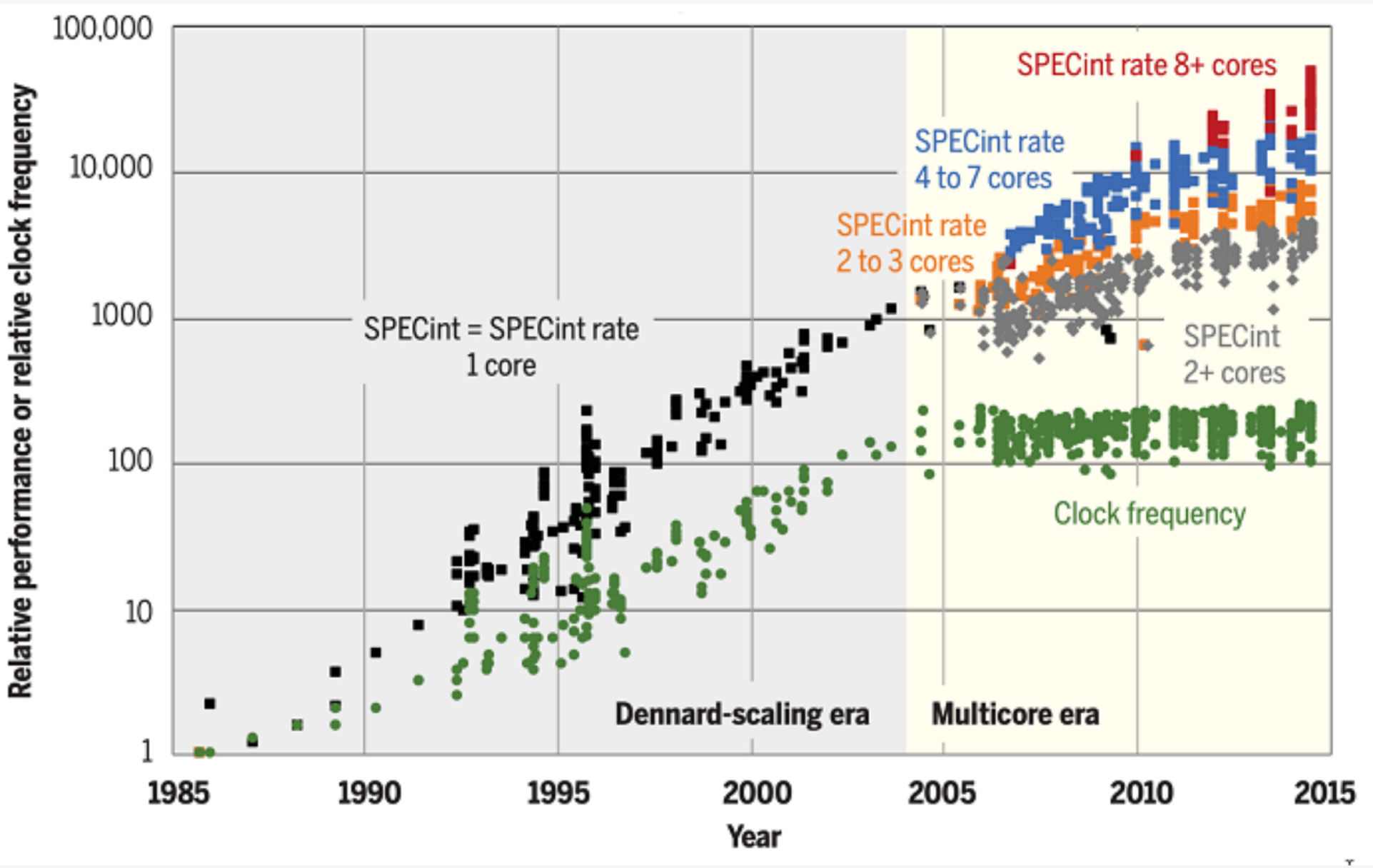
Every new generation doubled the clock frequency.

Broke down in ~2004

Leakage power was no longer negligible

Solution: multicores

- *Stop increasing the frequency, but increase parallelism*
  - **The number of processors (cores), plus**
  - **Internal parallelism (vector/SIMD) within a core**



SPECint (largely serial) performance, SPECint-rate (parallel) performance, and clock-frequency scaling for microprocessors from 1985 to 2015, normalized to the Intel 80386 DX microprocessor in 1985. ~ There's plenty of room at the Top: What will drive computer performance after Moore's law? Charles E. Leiserson et al, Science, (5 June 2020)

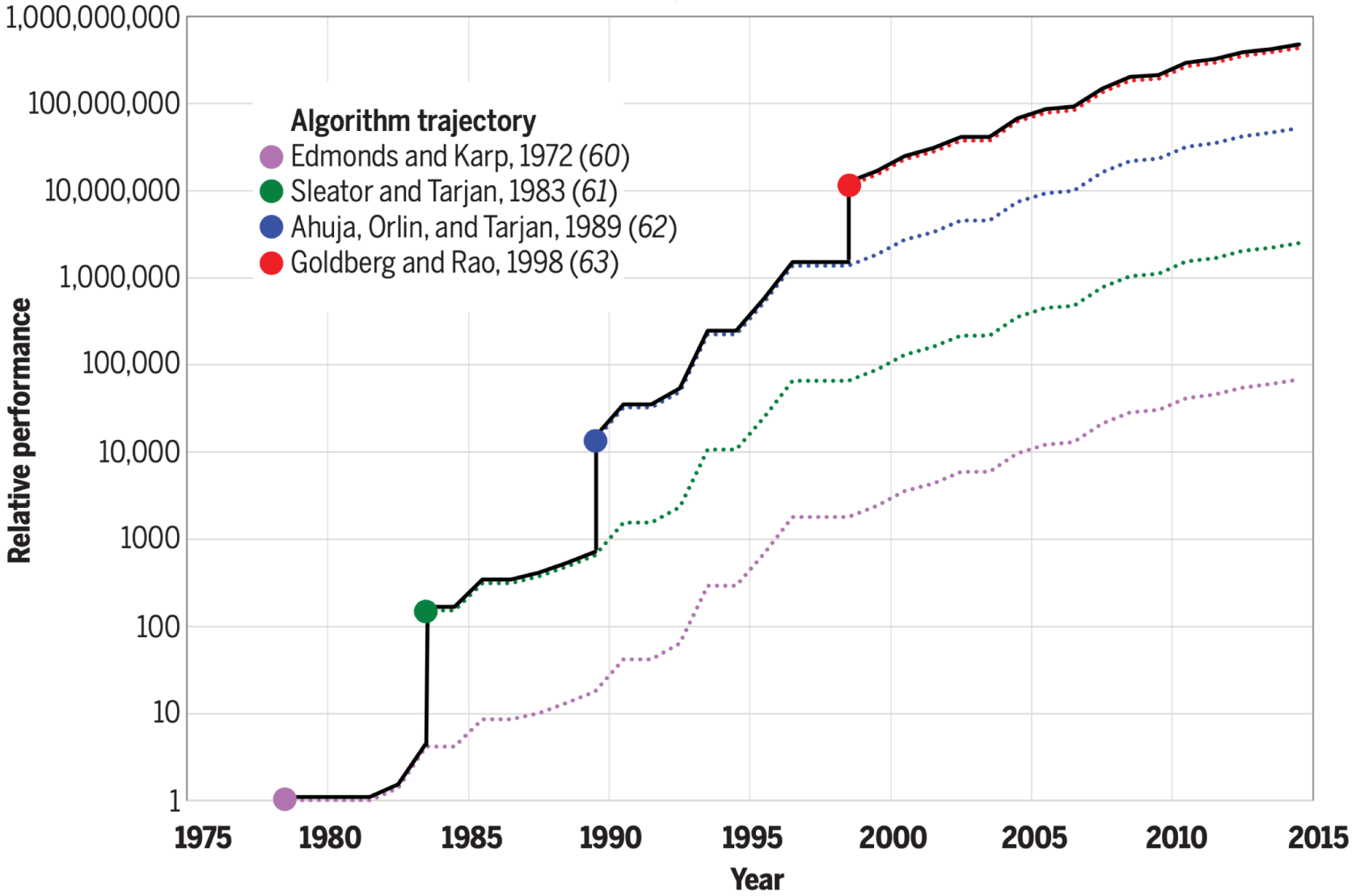


Fig. 1. Major algorithmic advances in solving the maximum-flow problem on a graph with $n = 10^{12}$ vertices and $m = n^{1.1}$ edges. The vertical axis shows how many problems (normalized to the year 1978) could theoretically be solved in a fixed time on the best microprocessor system available in that year. Each major algorithm is shown as a circle in the year of its invention, except the first, which was the best algorithm in 1978. The dotted trajectory shows how faster computers [as measured by SPECint scores in Stanford's CPU database (56)] make each algorithm faster over time. The solid black line shows the best algorithm using the best computer at any point in time. Each algorithm's performance is measured by its asymptotic complexity, as described in the Methods.

Parallel Machines

- *Shared Memory Multiprocessors:* A core is a full-fledged processor. Each core can access a single shared memory.
- *Multi-core chips with accelerators* (special co-processor that executes simple codes in parallel
  - most modern processors, laptops, tablets, and desktops. Exploiting both – processor and accelerator – for the *same program* is hard.
- *Distributed memory multi-computers* Each *node's* memory is private, nodes communicate via an *interconnection network*
- *Clouds/Data centers* Large scale, mostly independent tasks with infrequent communication. More scalable than DMMs.

This course focuses on the first one (SMM)

Dynamic Multi-threading

1. *Nested Parallelism (aka Task Parallelism)*, where a function call (or a block of code) is "*spawned*," allowing the caller and spawned function to run in parallel
2. *Loop Parallelism* where *iterations* of the loop can execute in parallel (they are independent).

Tasks/loop-iterations are executed by *threads* or *virtual processors*

Exactly when and where a thread executes is not decided by the programmer, but by a *run time system* (RTS). It coordinates, schedules and manages the parallel resources. Programmer does not worry about

- Data partitioning (shared memory) and
- Task scheduling.

But programmer is responsible for correctness

Parallel constructs
4 keywords

Tasks

- Created/specified by keyword *spawn*
- Synchronized by keyword *sync*

Parallel loops

- Created by keyword *parallel* before loop.
- Local/private variables introduced with keyword *new*

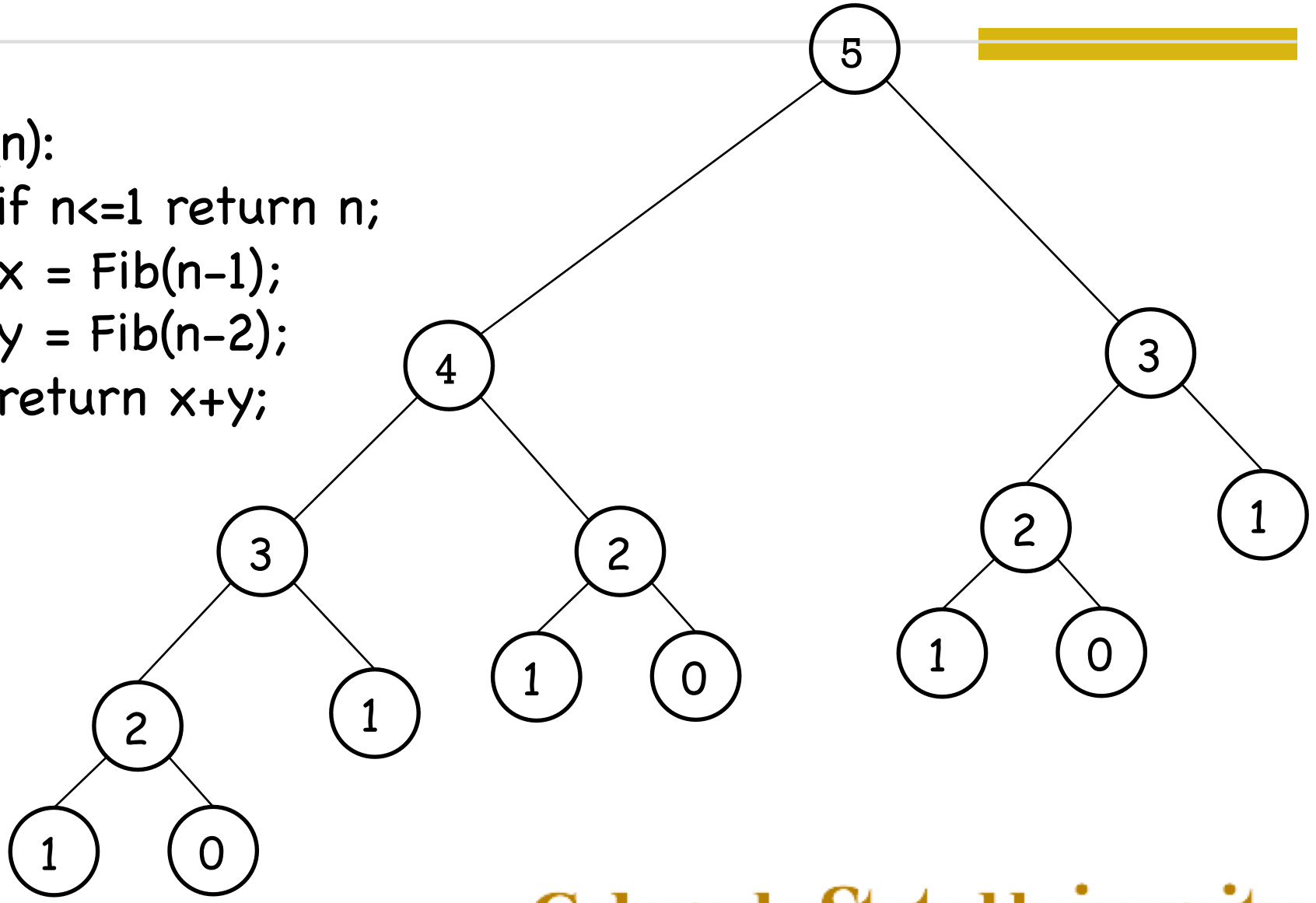
Key principle: removing all parallel constructs

- Doesn't change the program/algorithm

Models many modern parallel systems/languages
(OpenMP, Cilk/Intel TBB, OpenCL, etc.)

DMT: recursive Fibonacci

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



Complexity

Set up the recurrence, solve it (two recursive calls, plus constant work outside)

$$T(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + 1 & \text{otherwise} \end{cases}$$

Claim: the cost of Fib is (nearly) the same as Fib,

$$T(n) = \theta\big(\text{Fib}(n)\big) = \begin{cases} 0 & \text{if } n \leq 1 \\ \text{Fib}(n) - 1 & \text{otherwise} \end{cases}$$

- Proof by strong induction.

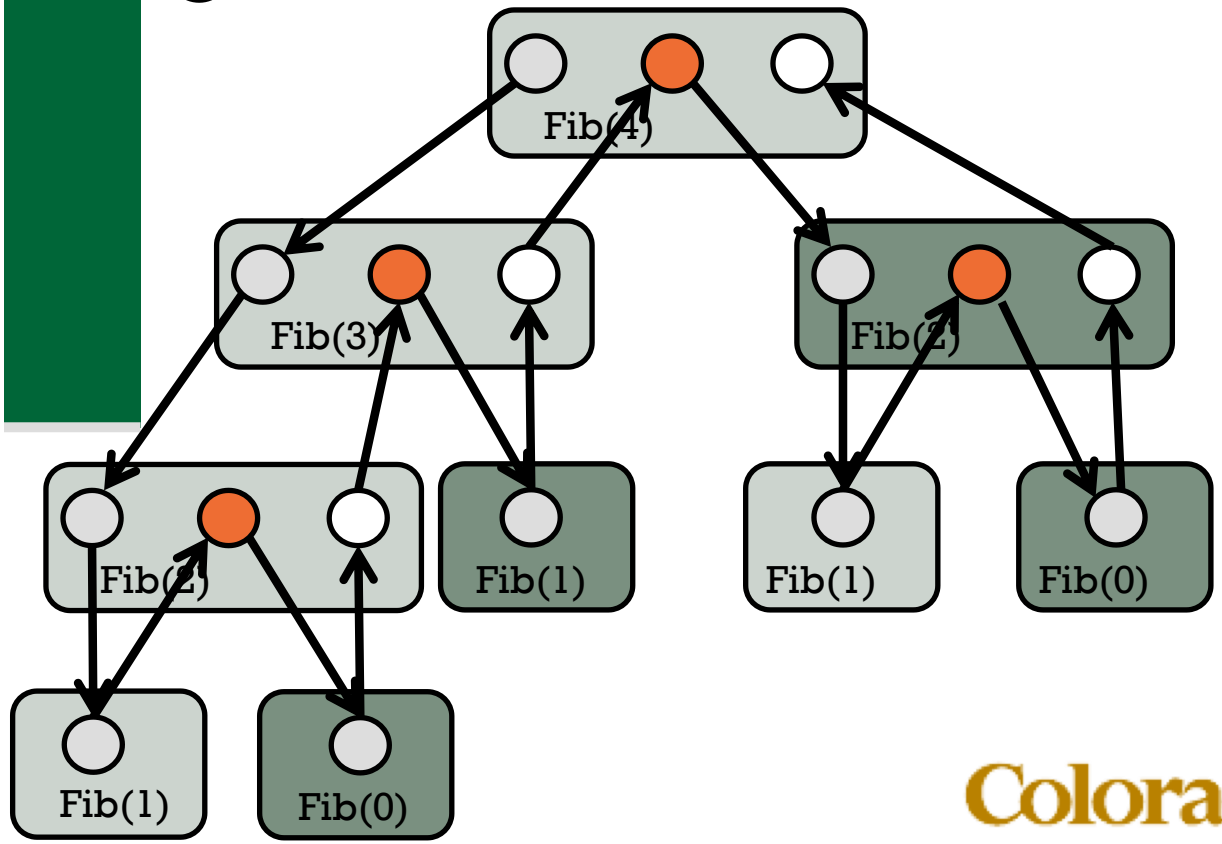
Stronger claim (about the value of Fib itself)

$$\text{Fib}(n) = \theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$

More advanced (CS 420)

Fib execution DAG

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

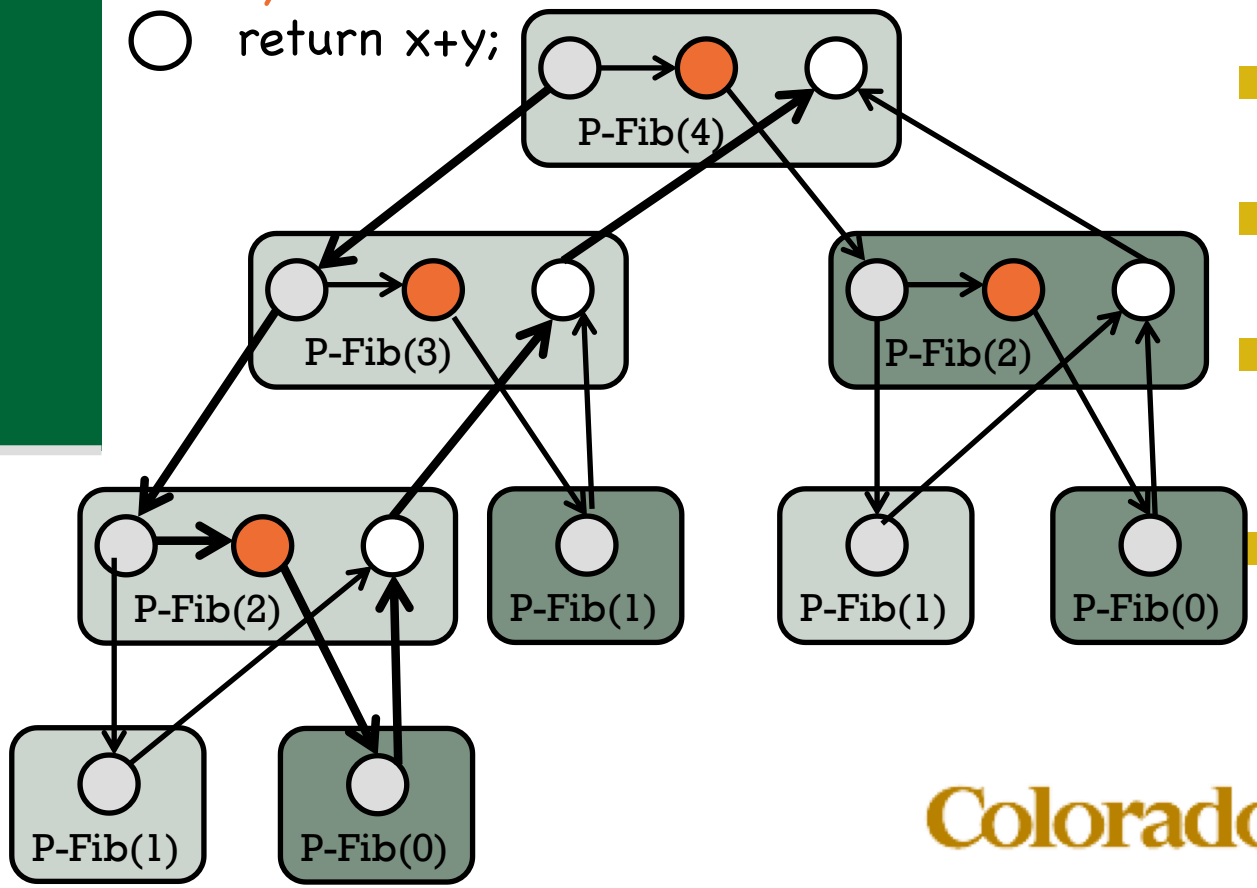


Ignore colors for now

- **Box:** *function call*
- **Circle** is a strand, a block/sequence of instructions (no control flow), aka basic block
- **Arrows:** indicate control flow, i.e., a call, a sequence, a return, etc.
- **Work:** total number of strands

Parallel Fib & its DAG

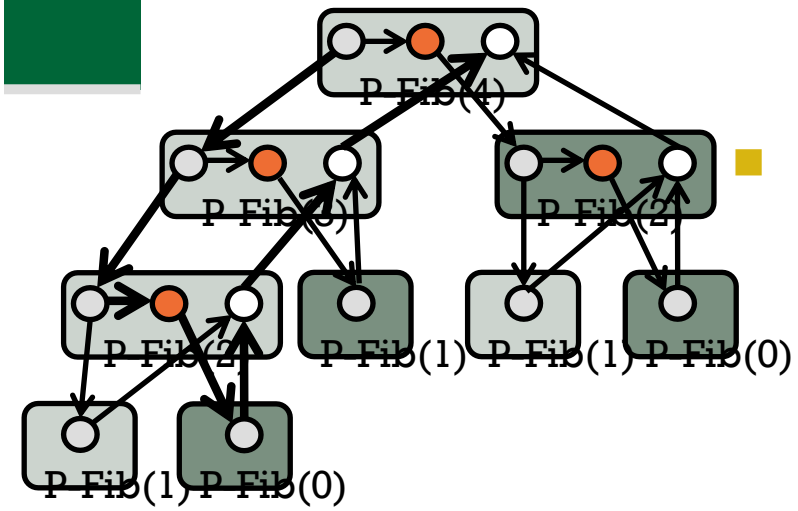
```
P-Fib(n):
    if n<=1 return n;
    x = spawn P-Fib(n-1);
    y = P-Fib(n-2);
    sync
    return x+y;
```



- Gray box: *spawned* (and executed by different thread)
- Dark box: executed by same thread
- Gray circles: code preceding *spawn*
- Orange circles: in parent
- White circles: (code after *sync*)
- *Bold edges*: critical path: *longest path through the dag*

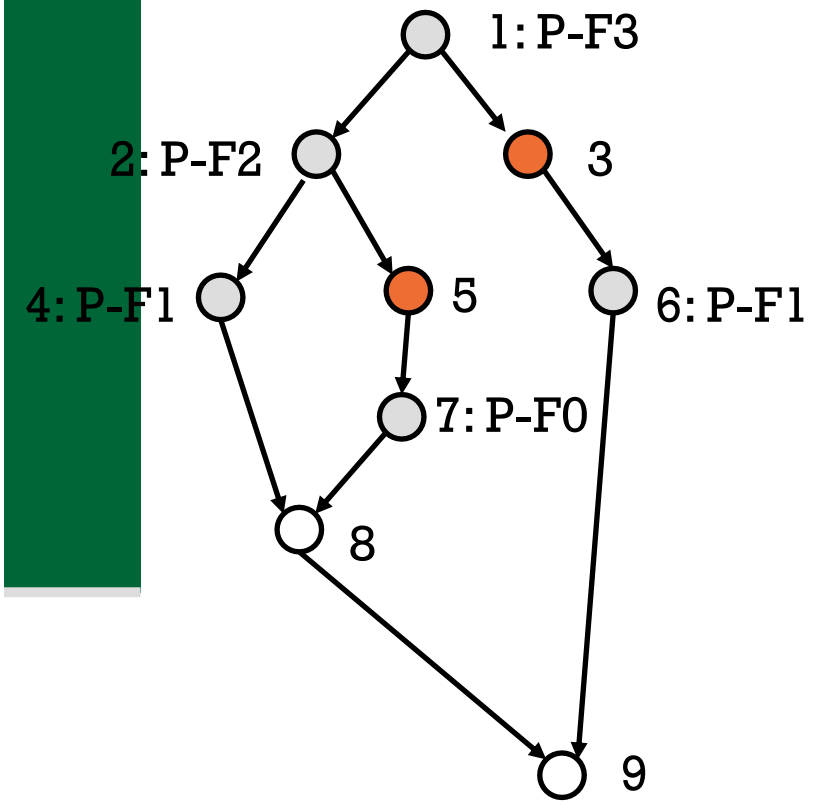
P-Fib & execution model

```
P-Fib(n):
    if n<=1 return n;
    x = spawn P-Fib(n-1);
    y = P-Fib(n-2);
    sync
    return x+y;
```



- To prevent an incorrect answer, *sync* before adding results
- Actually, *return* has implicit sync, so the *sync* here is not necessary
- Edge (u, v) is a dependence u must execute before v
- Strand with two successors: one is spawned
- Stand with two incoming edges must sync
- Strands on a path execute sequentially, otherwise they execute in parallel
- Spawn and call edges point downward, a horizontal edge indicates that the parent may keep computing while spawn executes, return edges point up
- Two Metrics
  - Work: total number of strands (17)
  - Span: number of nodes in critical path (8)

Impact of the schedule



Unfolded DAG for PF-3

Two processors

Schedule 1

```
P2        3  5  7
P1     1  2  4  6  8  9
----------------------
time  1  2  3  4  5  6
```

Schedule 2

```
P2        3  6
P1     1  2  4  5  7  8  9
-----------------------
time  1  2  3  4  5  6  7
```

- **Idle time**: number of empty slots (processor not busy) in schedule
- schedule 1: 3, schedule 2: 5

Performance Metrics

- *Work:* total time to execute the program sequentially. Assuming 1 time unit per strand, this is the number of nodes (circles) in the DAG.
- *Span:* max time to execute strands on any path in the DAG
  - number of nodes on the critical path of the DAG.

Intuitive interpretation:

- *Work* models *sequential* execution time,
- *Span* models *ideal parallel* execution time

Work and Span

- Two different cost metrics, we have $T(n, P)$
- Both can grow asymptotically. We are interested in how $T(n, P)$ grows/reduces as
  - n grows (usual asymptotic analysis)
  - P grows (scalability of the algorithm – can it effectively use parallelism)
- Work & span are particular cases:
  - $T(n, 1) = T_1(n) = T_1$ is the work
  - $T(n, \infty) = T_\infty(n) = T_\infty$ is the span

Performance (lower) bounds

Work Law

In one step, P processors can execute at most P strands. Total number of strands is the work, T_1 So,

$$T_P \geq \frac{T_1}{P}$$

Span Law

Arrange the DAG by "critical path," (also called the ASAP (as-soon-as-possible) schedule. Execute "layer by layer."

- Always possible with unbounded number of processors, so
- Number of time steps is T_∞ critical path length
- With only P processors, execution time cannot be any smaller than this, so

$$T_P \geq T_\infty$$

So, $T_P \geq \max(T_\infty, T_1)$

But what about the schedule

- We really have a more complicated function, $T(n, P, \text{schedule})$
- Many questions:
  - What is the best schedule?
  - How does it depend on the machine, the OS, the compiler?
  - Can we always find the best schedule?
  - How to analyze across the schedules?
- *There is good news*

Performance (upper) bounds

$$2\max(T_\infty, T_1) \geq T_P \geq \theta(\max(T_\infty, T_1))$$

$$T_P = \theta(\max(T_\infty, T_1))$$

In the ASAP schedule, let there be n_t nodes with critical path length t

- With only P processors, we *simulate* the ASAP schedule:
  - The ASAP time step t is split into blocks of size P and run sequentially
  - This takes $\left\lceil \frac{n_t}{P} \right\rceil$ actual time steps. Putting this together,

$$T_P = \sum_{t=1}^{T_\infty} \left\lceil \frac{n_t}{P} \right\rceil$$

$$\leq \sum_{t=1}^{T_\infty} \left(\frac{n_t}{P} + 1 \right)$$

$$= \sum_{t=1}^{T_\infty} \left(\frac{n_t}{P} + 1 \right) = \sum_{t=1}^{T_\infty} \frac{n_t}{P} + \sum_{t=1}^{T_\infty} 1 = \frac{1}{P} \sum_{t=1}^{T_\infty} n_t + T_\infty = \frac{T_1}{P} + T_\infty$$

Practical considerations

- We now have just two independent parameters, n and P
- Both grow asymptotically large, one grows much faster than the other $n \gg P$
  - So treat P as a "slow-growing" constant
  - So, n is the important parameter

A fast sequential algorithm is the first and foremost priority: reduce the work

- The seek among all such algorithms, those that can be executed fast (low T_∞)
- What class (*remember our function clubs*) grows slowly?
- Poly-logarithmic functions

Practical measures: parallelism (//ism) & speedup

- *Speedup:* How much faster does it run on processors, *compared to a single one*.

$$S_P \equiv \frac{T_1}{T_P}$$

- *Parallelism:* How much faster does the ideal parallel run on, *compared to sequential*

$$\Pi_P \equiv \frac{T_1}{T_\infty}$$

- *Speedup bounds:*
  - *Linear Speedup:* Speedup that grows linearly with P, $S_P = kP$, for $0 < k \leq 1$
  - *Ideal Speedup:* linear speedup, with $k = 1$ no idle time, all processors busy all the time
- When $P > \Pi$ there will be idle time and hence non-ideal speedup

Analyzing (blocks of) code

- Compose two functions, either in sequence or in parallel.
  - Blocks executed in parallel (e.g., in a spawn-sync block) they are in parallel
  - Otherwise they execute in the usual sequential manner
  - Also beware of conditional blocks of code
- Work is always added
- Span is added when in series
- Combined with max when in parallel

Analyzing code

- Code may be composed either in sequence or in parallel.
  - Blocks executed in parallel (e.g., in a spawn-sync block)
  - Or the usual (sequential) manner
- Work is always added

$$T_1(A;B) = T_1(A) + T_1(B)$$

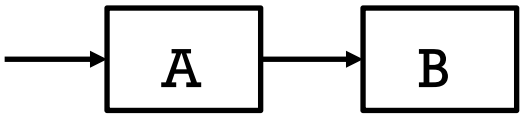
$$T_1(\text{spawn } A; B \text{ sync}) = T_1(A) + T_1(B)$$

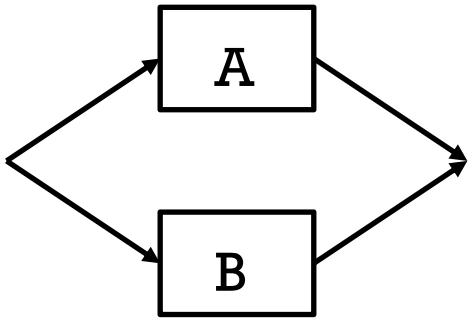
- Span is
  - Added when in series

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

  - Combined with max when in parallel

$$T_\infty(\text{spawn } A; B \text{ sync}) = \max\left(T_\infty(A), T_\infty(B)\right)$$





Parallelizing the $\theta(n)$ work Fib

- Now we have dependent computations
- How to break the dependences?

```
EfficientFib(n):
if (n<2) return 1;
X[0] = 1;
X[1] = 1;
for (i=2; i≤n; i++)
  X[i%2] = X[(i-1)%2] +  X[i%2];
return X[n%2];
```

Take a detour

1. *First parallelize reductions*
2. *Then parallelize scans*
3. *Show that Fib is similar to a scan*

First parallelize reductions

Add up the elements of an array

$$x = \sum_{j=1}^{n} A[j]$$

```
Reduce (A):
X=0;
for (j=1; j≤n; j+)
   x = x + A[j];
```

- Sequential implementation

(linear)

- This is also the lower bound (need to read at least n inputs)
- For parallel algorithms, this (just the) *work*
- How to parallelize?
  - How to break the dependence of one result (iteration) on the previous one?
  - Divide and conquer to the rescue

D&C Reductions

- Work: remains linear

$$T_1(n) = 2T_1\left(\frac{n}{2}\right) + 1$$

  - Just changed the order in which elements are added up (in a *tree* rather than *left to right*
- Can we parallelize it?
  - Easy: spawn one call
- Span: becomes $\theta(\lg n)$

$$T_\infty(n) = T_\infty\left(\frac{n}{2}\right) + 1$$

```
D&CReduce(lo, hi, A):
if (lo==hi) return A[lo];
else
  mid = (hi+lo)/2;

  spawn
  x1 = D&CReduce(lo, mid, A);
  x2 = D&CReduce(mid+1, hi, A);
  sync
  return (x1+x2);
```

Victory!! On to newer challenges

Scan (prefix sum/max)

Add up the elements of an array A

$$X[i] = \sum_{j=1}^{n} A[j]$$

- But return all partial answers

- Efficient sequential algorithm:

$$X[i] = \begin{cases} A[i] & \text{if } i = 1 \\ X[i-1] + A[i] & \text{if } i > 1 \end{cases}$$

- Work: $\theta(n)$

```
SeqScan(A):
  X[0] = A[0];
  for (j=1; j≤n; j++);
      X[j] =  X[j-1] + A[j];
```

Recursive (D&C) Scan

Same idea: divide into halves; (recursively) compute prefix sums, but now add the *last answer of first half* to each element of the second half.

- Work: now $\theta(n \lg n)$ (master theorem again). *Extra work*
- Can it be parallelized?
  - What about the for loop
  - Again use D&C + *spawn sync*
  - Span: $\theta(\lg^2 n)$

```
Scan(lo, hi):
  if (lo==hi) X[i] = A[i];
  return;
  mid = (hi+lo)/2;
  spawn
  X[lo:mid] = Scan(lo, mid);
  X[mid+1:hi] = Scan(mid+1, hi);
  sync
  //Update second half (for loop)
  X[mid+1:hi] = X[mid]+X[mid+1:hi];
return;
```

Improving the Span

The for loop is killing us. All iterations independent, but we are doing it with $\lg n$ span

- Inherent limitation of *spawn sync*
- Need a new construct *parallel loops*
- Span is the max span of any loop body ($\theta(1)$ here)
  - Overall span: $\theta(\lg^2 n)$

```
Scan(lo, hi):
    if (lo==hi) X[i] = A[i];
    return;
    mid = (hi+lo)/2;
    spawn
    X[lo:mid] = Scan(lo, mid);
    X[mid+1:hi] = Scan(mid+1, hi);
    sync
    //Update second half (for loop)
    parallel
    X[mid+1:hi] = X[mid]+X[mid+1:hi];
return;
```

Parallel Loops

- The *parallel* keyword before a for loop indicates that all the iterations of the for loop can execute in parallel.
- Legal only if loop iterations are *independent*, i.e., an iteration does not use values computed in previous iterations.

```
for i in 0 to n-1:    C[i] = A[i]+B[i]
```

can be made parallel

```
parallel for i in 0 to n-1:    C[i] = foo(A[i], B[i])
```

- Example of *illegal parallelization*

```
for i in 0 to n-1:    A[i] = foo(A[i-1], B[i])
```

cannot be made parallel. Iteration i uses a value computed by iteration i-1, so *must be executed before* iteration i. Introduces *data race*

Is our parallel scan worth it?

- Extra work:
  - $\theta(n \lg n)$ vs $\theta(n)$
- But span is $\theta(\lg n)$ so it's a *fast parallel algorithm*
- Two laws: work law the span law. Both n and P grow asymptotically, but$n \gg P$. So,

$$\frac{n \lg n}{P} \gg \lg n \text{ i.e.,}$$

$$\frac{T_1}{P} \gg T_\infty$$

- Work law dominates. *Unless the work can be brought down to $\theta(n)$ our parallel scan will be too slow*

Work optimal parallel scan

Chunking algorithm:

- Split input into P chunks, of size $\frac{n}{P}$ each, one per processor.
  - Each processor sequentially reduces its chunk
  - All processors cooperate to scan these P results
  - Finally, each processor independently, compute the scan of it's section, starting with the scan of it predecessor
- Double the work, but this is acceptable, e.g., 100 processors will go 50x faster than the sequential algorithm
  - *Linear (not ideal) speedup*

Getting to lg n span

- See https://en.wikipedia.org/wiki/Prefix_sum

Tree based (work out on excel spreadsheet)

- One pass up the tree to compute a reduction (and save all partial sums contributing to that)
- Second pass down the tree to *update/repair* the elements with the prefix results of *everything to the left of* the node

- Work is $\theta(n)$, span is $\theta(\lg n)$

Victory (finally)!!

Back to Fibonacci

- Generalize to computing an array of *all* the Fib numbers up to n
- Lower bound:
  - Is it $\theta(n)$ (that's the size of the output)?
  - No, processors may write outputs in parallel (as in scan)
- Recall memo-Fib: in each iteration, update one value using the previous and pre-previous

```
MemoFib(n):
  F[0] = F[1] = 1;
  for (i=2; j≤n; j++);
    F[i] = F[i-1] + F[i-2];
    //and just for kicks
    F[i-1] = F[i-1]; // useless copy
```

Key idea: reduction (as a verb)

- How to *reduce* the Fib to a scan/reduction
- Like in the memory-efficient version, copy
  - New to previous
  - Previous to pre-previous
- Use matrix notation:

$$\begin{pmatrix} F_i \\ F_{i-1} \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} F_{i-1} \\ F_{i-2} \end{pmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} F_{i-2} \\ F_{i-3} \end{pmatrix}$$

$$= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

Reduce Fib to reduction/scan

- Define a constant 2×2 matrix: $M = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

$$\begin{pmatrix} F_i \\ F_{i-1} \end{pmatrix} = M^{i-1} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

i.e., multiply the $i-1^{\text{th}}$ power of M with the vector $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$

- Computing Fib *reduces to a reduction*
- If all numbers are needed, do a scan
- Also note: all this is for *pedagogic purposes only*

Scan in practice

- Scan is a representative for problem that seem "inherently sequential"
  - The inherent sequentiality can be broken if there is an associative binary operator
  - Kogge & Stone 1973 (*recursive convolution, IIR filters*)
  - Hardware:
    - Fast multipliers
    - Fast adders
    - Double the work is acceptable