parallel

concurrent

Amdahl's Law

thread of control

processors versus **processes**

non-deterministic

fork-join parallelism.

Parallel and concurrent programming
6. Parallel Performance

protection data race synchroniz**ation**

divide-and-conquer algorithms

THREAD

SAFETY correctness

Michelle Kuttel

MUTUAL EXCLUSION

locks

readers-writers problem

liveness

DEADLOCK starvation

High Performance Computing

producer-consumer problem

EXECUTORS *thread pools* **timing**

Dining philosphoers problem

Demonstrating parallel performance

To justify their existence, parallel algorithms must be **both**:

- **correct** *and*

- **efficient**

To demonstrate **correctness**, you **validate** an algorithm against the serial version to show that it produces the same results across a range of input.

To demonstrate **efficiency** (speedup), you **benchmark** an algorithm against the serial version, to show that it is (ultimately) faster.

Speedup

Speedup with *P* processes =

$\dfrac{\text{time to solve problem size n with 1 thread}}{\text{to solve problem size n with } P \text{ threads}}$
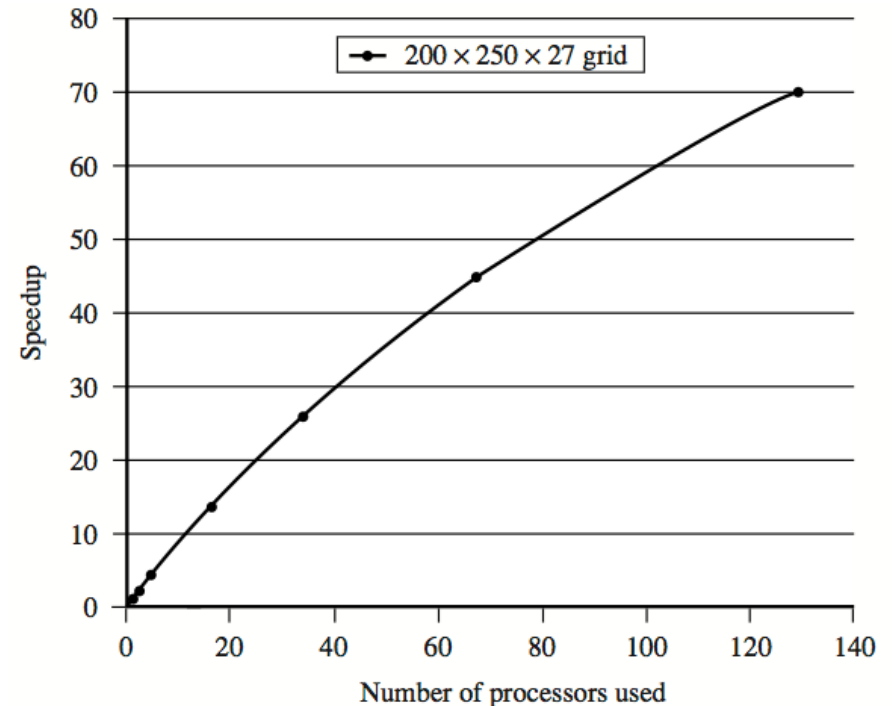
$$= T_1/T_P$$

Figure 1.1 Performance of the MM5 weather code.

Figure from "Parallel Programming in OpenMP, by Chandra et al.
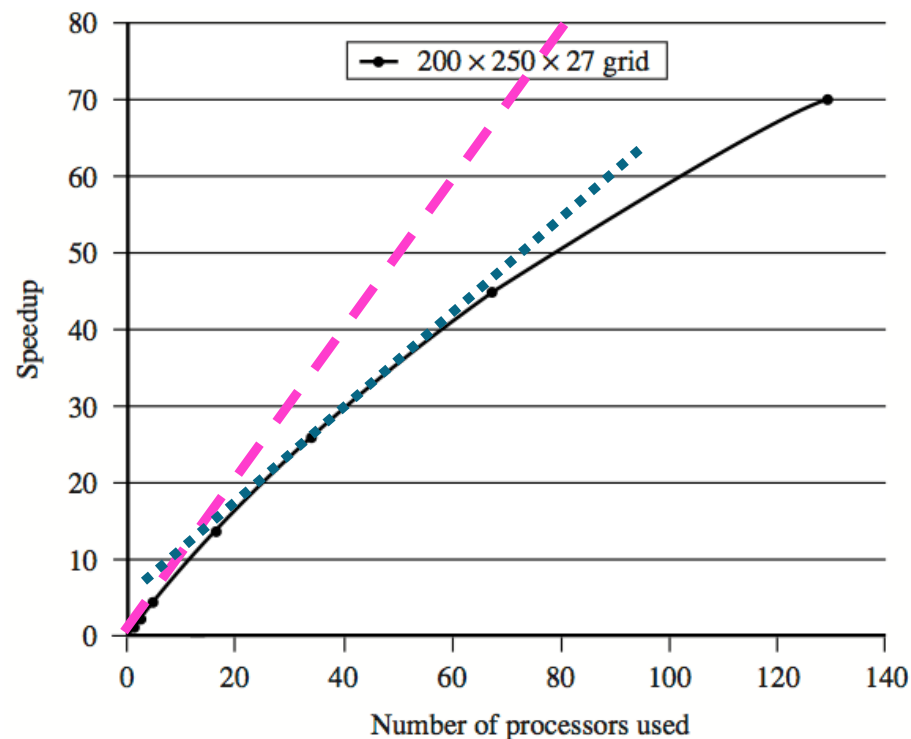
Speedup

Perfect linear speedup = P
- *Doubling **P** halves* running time
- The goal; **hard to get in practice**

Scalability is an easier to achieve goal.

- an algorithm is **scalable** if the speedup increases at **least linearly** with the problem size.
  - running time is inversely proportional to the number of processors used.

- Still hard to get linear scalability in practice;
  - most algorithms reach an optimal level of performance and then deteriorate

Performance of the MM5 weather code.

Theoretical analysis of expected speedup: Work and Span

Two key measures of run-time:

- **Work:** How long it would take 1 processor = T_1

  The serial time. e.g. for divide-and-conquer, just "sequentialize" the recursive forking

- **Span:** How long it would take infinity processors = T_∞
  - Example: $O(\log n)$ for summing an array with divide-and-conquer method
  - Also called "critical path length" or "computational depth"

Maximum possible speed-up is T_1 / T_∞
  - i.e. work divided by span

Parallel algorithms are about decreasing span without increasing work too much

Limits on T_P

Let T_P be the run-time with P processors.

- **Work Law:**

$$T_P >= T_1 / P$$

why?

- **Span Law:**

$$T_P >= T_\infty$$

why?

Limits on T_P

- Work Law:

$$T_P >= T_1 / P$$

each processor executes at most 1 instruction per unit time, and hence P processors can execute at most P instructions per unit time. Thus, to do all the work on P processors, it must take at least T_1/P time.

- Span Law:

$$T_P >= T_\infty$$

a finite number of processors cannot outperform an infinite number of processors, because the infinite-processor machine could just ignore all but P of its processors and mimic a P-processor machine exactly.

If we ignore memory-hierarchy issues

The DAG, or "cost graph"

Program execution can be seen as a DAG (directed acyclic graph)

- Nodes: Pieces of work
- Edges: Source must finish before destination starts

- A `fork` "ends a node" and makes two outgoing edges
  - New thread
  - Continuation of current thread

- A `join` "ends a node" and makes a node with two incoming edges
  - Node just ended
  - Last node of thread joined on
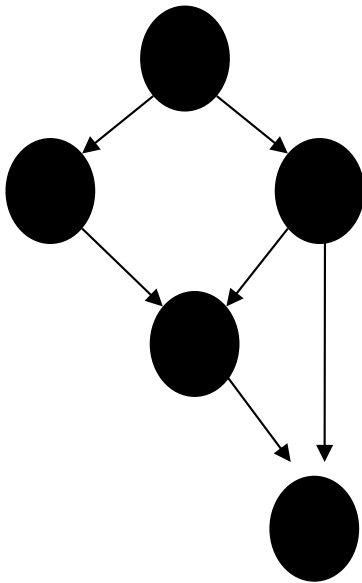
The DAG, or "cost graph"

- work – number of nodes, T_1
- span – length of the longest path, T_∞
  - critical path
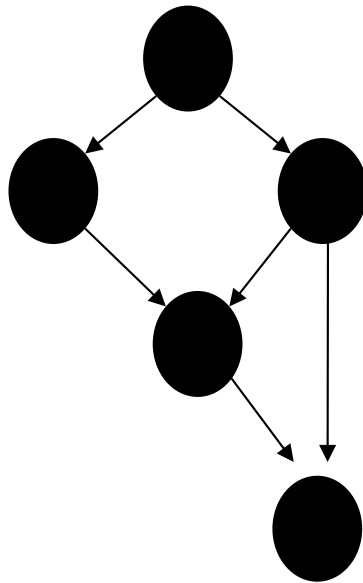
Checkpoint:
What is the span of this DAG?
What is the work?

Checkpoint

a**x**b **+** c**x**d

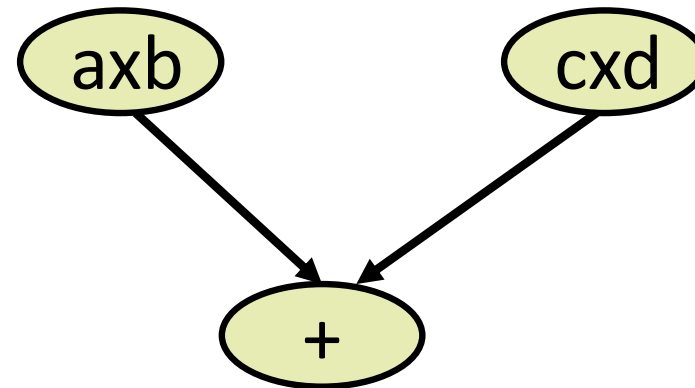Write a DAG to show the the *work*
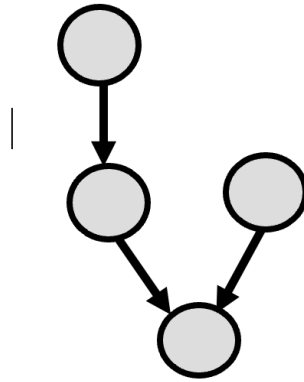and *span* of this expression.

Checkpoint

axb + cxd

- Write a DAG to show the the *work* and *span* of this expression

The set of instructions forms the vertices of the dag,
the graph edges indicate **dependences** between instructions.

- We say that an instruction x *precedes an instruction y if x must complete before y can begin.*
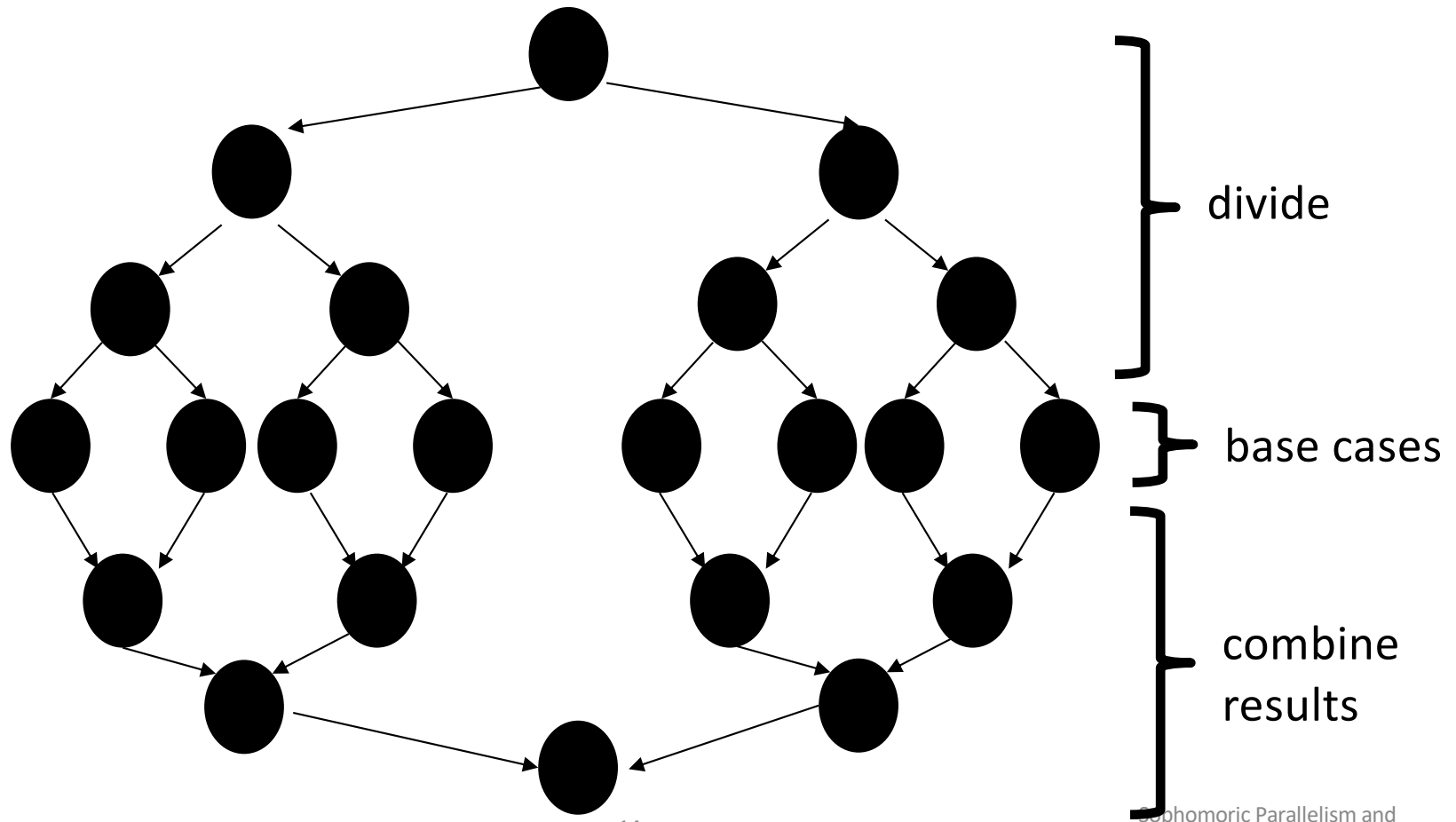
Which expression below corresponds to this cost graph (DAG)?

A. $x^2 + y^2$
B. $2x^2 + y$
C. $2x^2 + y^2$
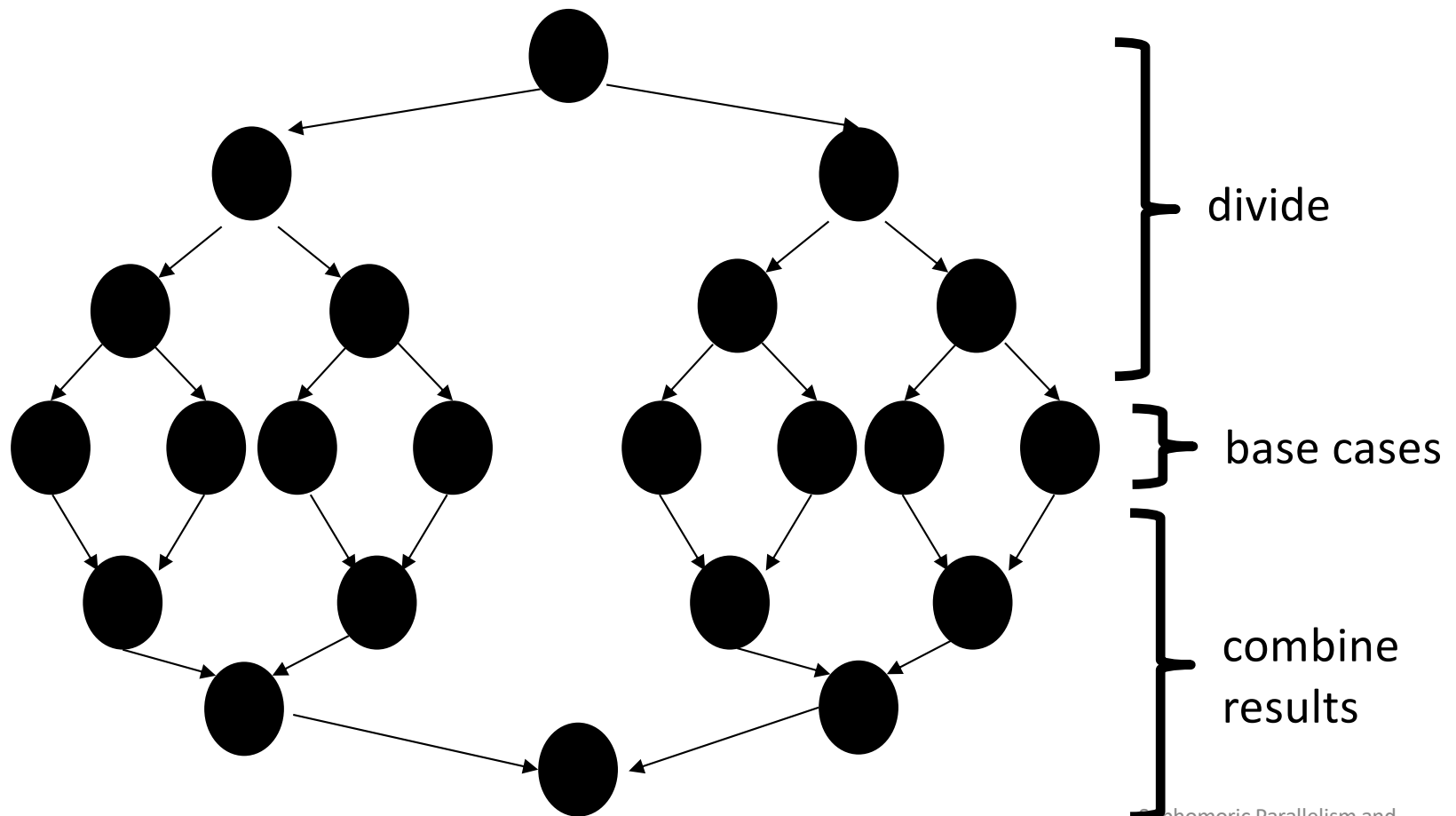D. $2x + y + z$
E. C and D

Divide-and-conquer parallel algorithms

- **fork** and **join** are very flexible, but divide-and-conquer maps and reductions use them in a very basic way:
    - DAG is tree on top of an upside-down tree

divide

base cases

combine results

Divide-and-conquer parallel algorithms

- What is the work and span of this DAG?

divide

base cases

combine results

Sophomoric Parallelism and
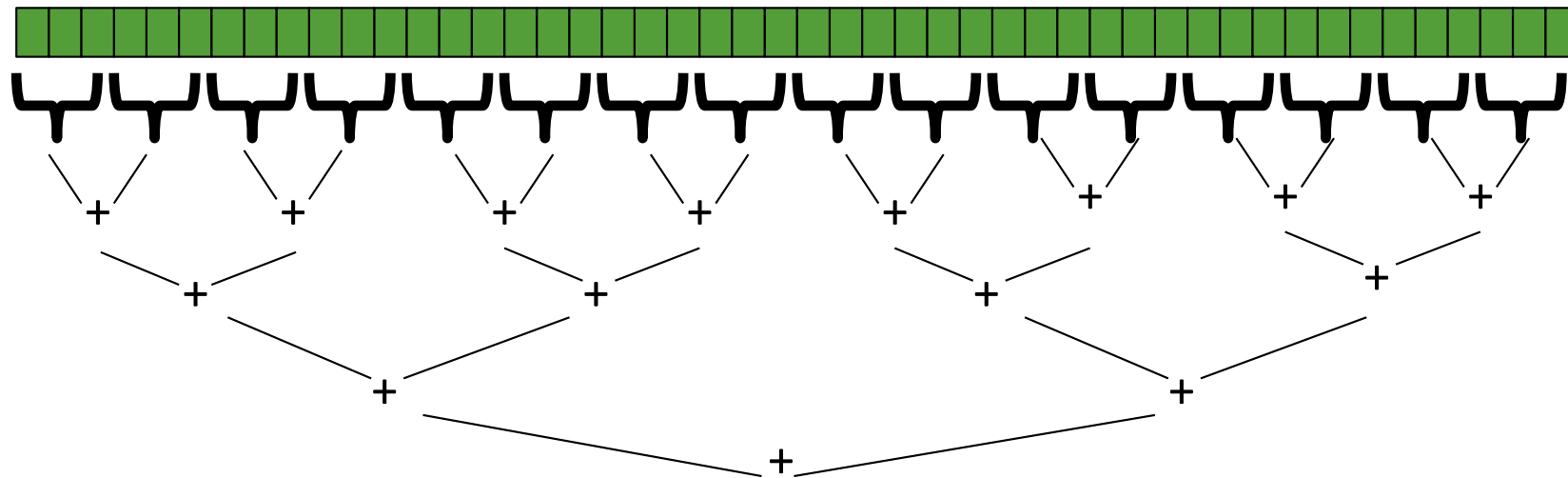Concurrency, Lecture 2

Divide-and-conquer parallel algorithms

Work is $O(n)$

Span is $O(\log n)$

Maximum speedup is $n / \log n$ (grows exponentially)

- Anything that can use results from two halves and merge them in $O(1)$ time has the same property...

Performance with Java's Fork/Join framework

We have:

- Work Law:

$$T_P >= T_1 / P$$

- Span Law:

$$T_P >= T_\infty$$

I must pay attention to this when doing speed up for A1

The fork-join framework **guarantees** expected time

$$T_P = O(T_1 / P) + O(T_\infty)$$

- First term dominates for small **P**, second for large **P**

expected because internally the framework uses randomness, so the bound can be violated from "bad luck" but such "bad luck" is exponentially unlikely, so it simply will not occur in practice

Framework Speedup

The fork-join framework guarantees

$$T_P = (T_1 / P) + O(T_\infty)$$

• So, no implementation of your algorithm can beat $O(T_\infty)$ by more than a constant factor, and

• no implementation of your algorithm on P processors can beat (T_1 / P) (ignoring memory-hierarchy issues).

The framework on average gets within a constant factor of the best you can do, assuming the user (you) did his/her job.

So: You can focus on your algorithm, data structures, and cut-offs, rather than number of processors and scheduling

Examples

The fork-join framework guarantees

$$T_P = O((T_1 / P) + T_\infty)$$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$

  - So expect (ignoring overheads): $T_P = O(n/P + \log n)$

- Suppose instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$

  - So expect (ignoring overheads): $T_P = O(n^2/P + n)$

Extra work introduced:

Parallelization overhead is the time required to coordinate parallel tasks, as opposed to doing useful work. e.g.

- starting threads
- stopping threads
- synchronization

A parallel "Law" you need to know: Amdahl's law (1967!)

*For over a decade prophets have voiced the contention that the organization of **a single computer has reached its limits** and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit co-operative solution [...]*

***Demonstration is made of the continued validity of the single processor approach** and of the **weaknesses of the multiple processor approach** in terms of application to **real problems** and their attendant irregularities. […]*

*The nature of this overhead (in parallelism) appears to be sequential so that it is unlikely to be amenable to parallel processing techniques. **Overhead alone** would then **place an upper limit** on throughput of **five to seven times the sequential processing rate**, even if the housekeeping were done in a separate processor...At any point in time it is difficult to foresee how the previous bottlenecks in a sequential computer will be effectively overcome.*

Gene Amdahl*, Published in:
AFIPS '67 (Spring) Proceedings of the April 18-20, **1967**, spring joint computer conference
Pages 483-485

* the designer of IBM 360 series of mainframe architecture

Amdahl's Law Explained

Amdahl's Law is a mathematical theorem demonstrating the **diminishing returns** of adding **more processors**.

Typically some parts of programs parallelize well…
- e.g. maps/reductions over arrays

(call this the **parallel fraction**, *p)*

…and some parts are inherently sequential
- e.g. reading a linked list, getting input, reading from a file, doing computations where each needs the previous step, etc.

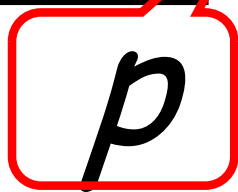(call this the **serial fraction**, **s**=*1-p*)

Amdahl's Law

Speedup= $$\dfrac{1}{1-p+\dfrac{p}{n}}$$

Amdahl's Law

Speedup= $$\cfrac{1}{1-p+\cfrac{p}{n}}$$

Parallel fraction

Amdahl's Law

Sequential
fraction

Parallel
fraction

$$\text{Speedup} = \cfrac{1}{1-p+\cfrac{p}{n}}$$

Amdahl's law[*]

In terms of **work** and **span**, this means that the serial portion restricts the minimum span thus

$$\mathbf{T}_\infty > (1-p)\, T_1$$

so the **maximum possible speedup** is

$$T_1 / \mathbf{T}_\infty < 1/(1-p)$$

Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

Example

$$\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}$$

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = \frac{1}{1 - 0.6 + \frac{0.6}{10}}$$

Speedup= 2.17

Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = \cfrac{1}{1 - 0.8 + \cfrac{0.8}{10}}$$

Speedup= 3.57

Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

$$\text{Speedup} = \frac{1}{1 - 0.9 + \frac{0.9}{10}}$$

Speedup=5.26

Example

- Ten processors
- 99% concurrent, 1% sequential
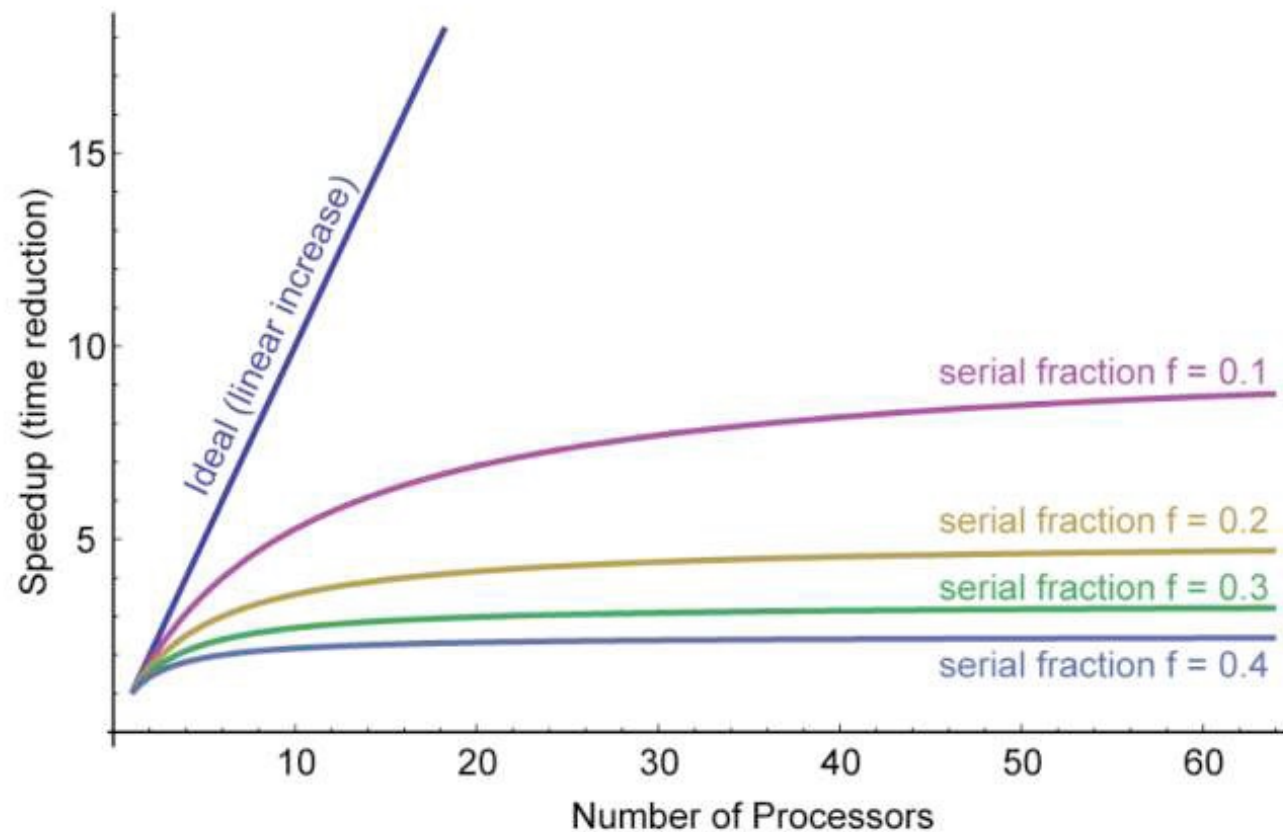- How close to 10-fold speedup?

Example

- Ten processors
- 99% concurrent, 1% sequential
- How close to 10-fold speedup?

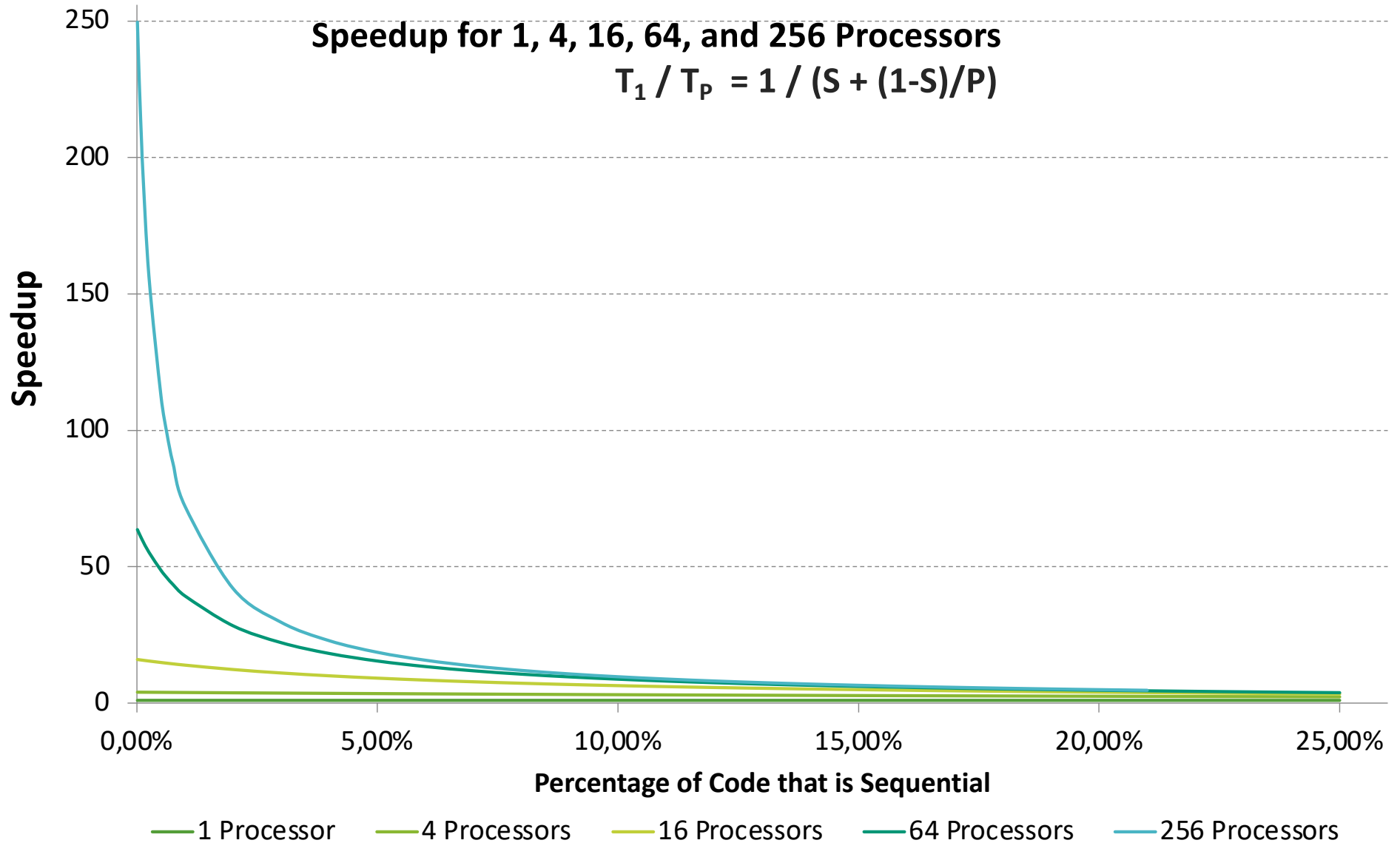$$\text{Speedup} = \cfrac{1}{1 - 0.99 + \cfrac{0.99}{10}}$$

Speedup=9.17

Graphing Amdahl's Law

graphic from lecture slides: Defining Computer "Speed": An Unsolved Challenge, Dr. John L. Gustafson, Director Intel Labs, 30 Jan 2011
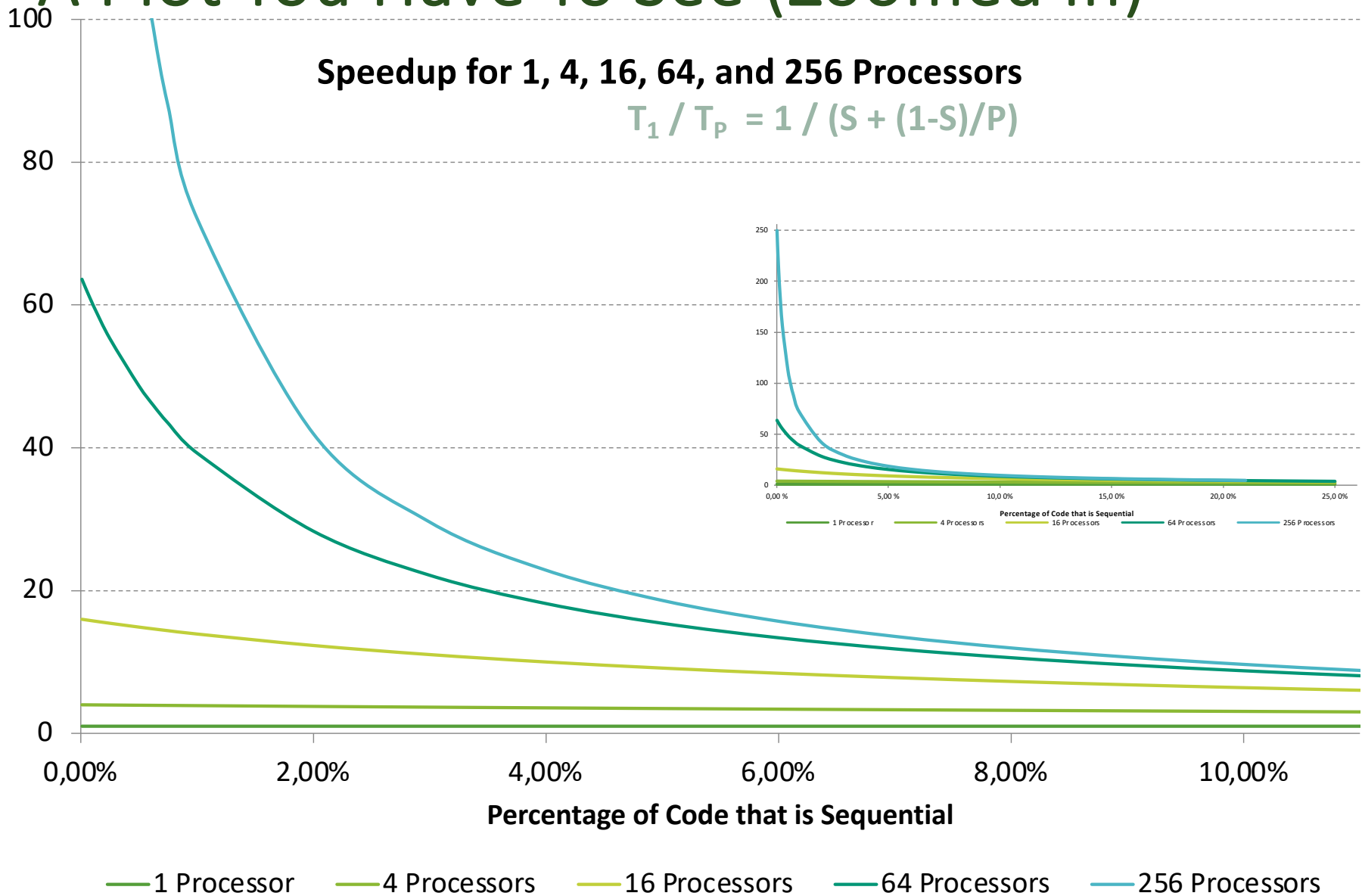
A Plot You Have To See

Speedup for 1, 4, 16, 64, and 256 Processors

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

Speedup vs. Percentage of Code that is Sequential

Legend: 1 Processor, 4 Processors, 16 Processors, 64 Processors, 256 Processors

A Plot You Have To See (Zoomed In)

Speedup for 1, 4, 16, 64, and 256 Processors

$$T_1 / T_P = 1 / (S + (1-S)/P)$$

Percentage of Code that is Sequential

— 1 Processor — 4 Processors — 16 Processors — 64 Processors — 256 Processors

Why such bad news

Suppose 33% of a program is sequential

- Then a billion processors won't give a speedup over 3

<mark>Moore's Law</mark>

In the "good old days" (1980-2005) <mark>12ish years was long enough to get 100x speedup</mark>

- Now suppose in 12 years, clock speed is the same but you get 256 processors instead of 1

- For 256 processors to get at least 100x speedup, we need

$$100 \leq 1 / (S + (1-S)/256)$$

Which means $S \leq .0061$ (i.e., 99.4% parallel)

The impact of Amdahl's law

Amdahl's Law indicated in 1967 that **parallel processing had no future** because it is not possible to speedup an application indefinitely by using more and more processors.

Amdahl argued that for typical (Fortran) codes,

$$T_s = 40\%$$

- therefore cannot use more than **a small number** of processors efficiently!

Amdahl's Law

It is now accepted that Amdahl was probably quite correct from a historical point of view.

- In the late 1960's problem sizes were too small for parallel computing and, had it existed, it would not have been used efficiently.

However…

Flaws in Amdahl's law: Gustafson's Law

Amdahl's law has an assumption that may not hold true:

- the ratio of T_S to T_P is **not constant** for the same program
  - usually **varies with problem size**.
  - *Typically, the T_P grows faster than T_S.*

i.e. the computational problem grows in size through the growth of parallel components – *Amdahl effect*

1. Gustafson, John L. (1988). *Communications of the ACM*. 31 (5): 532–533. doi: 10.1145/42411.42415

Two kinds of scalability

strong scaling (Amdahl):
- fixed **problem size**

- defined as how the solution time varies with the number of processors for a fixed *total* problem size.
- E.g. same task, double number of processors

weak scaling (Gustafson)**:**
- Fixed fixed **problem size** *per processor*
- defined as how the solution time varies with the number of processors for a fixed problem size *per processor*.
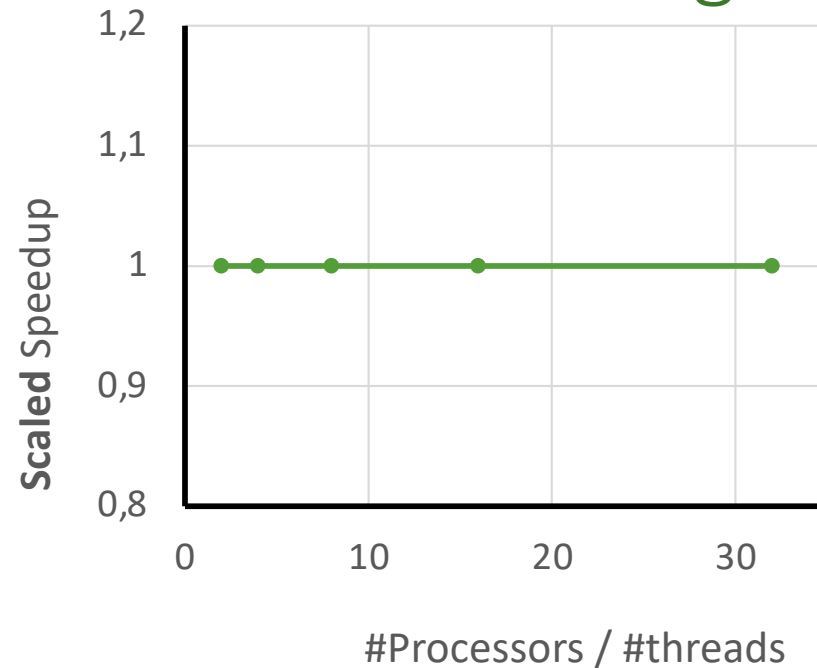- Double the task when you double processors

Strong scaling
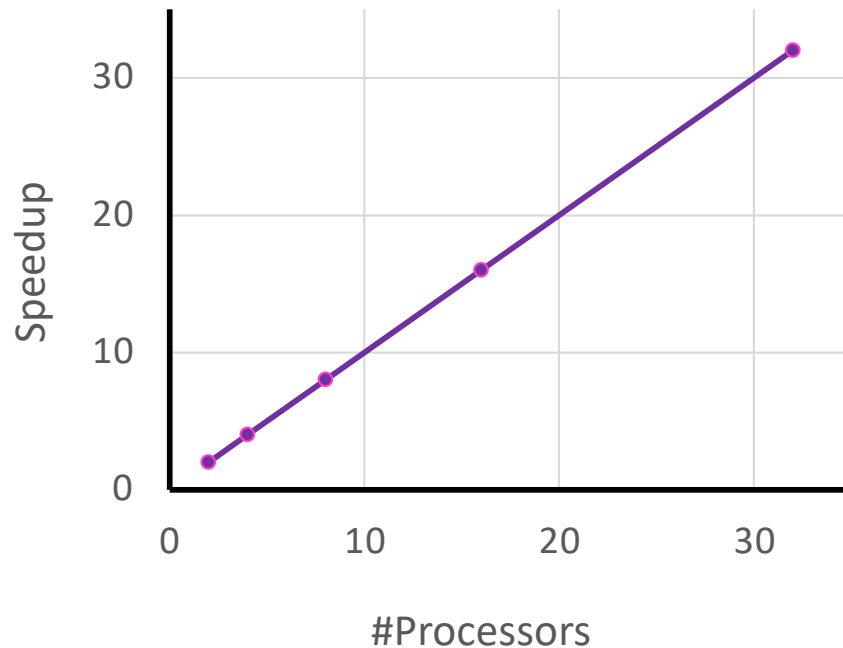
- Problem size stays same

Speedup with *P* threads =

$$\frac{\text{time to solve problem size n with 1 thread}}{\text{to solve problem size n with } P \text{ threads}}$$

Weak scaling

- Problem size doubles as thread size doubles

Scaled Speedup with *P* threads =

$$\frac{\text{time to solve problem size n with 1 thread}}{\text{to solve problem size n*P with } P \text{ threads}}$$