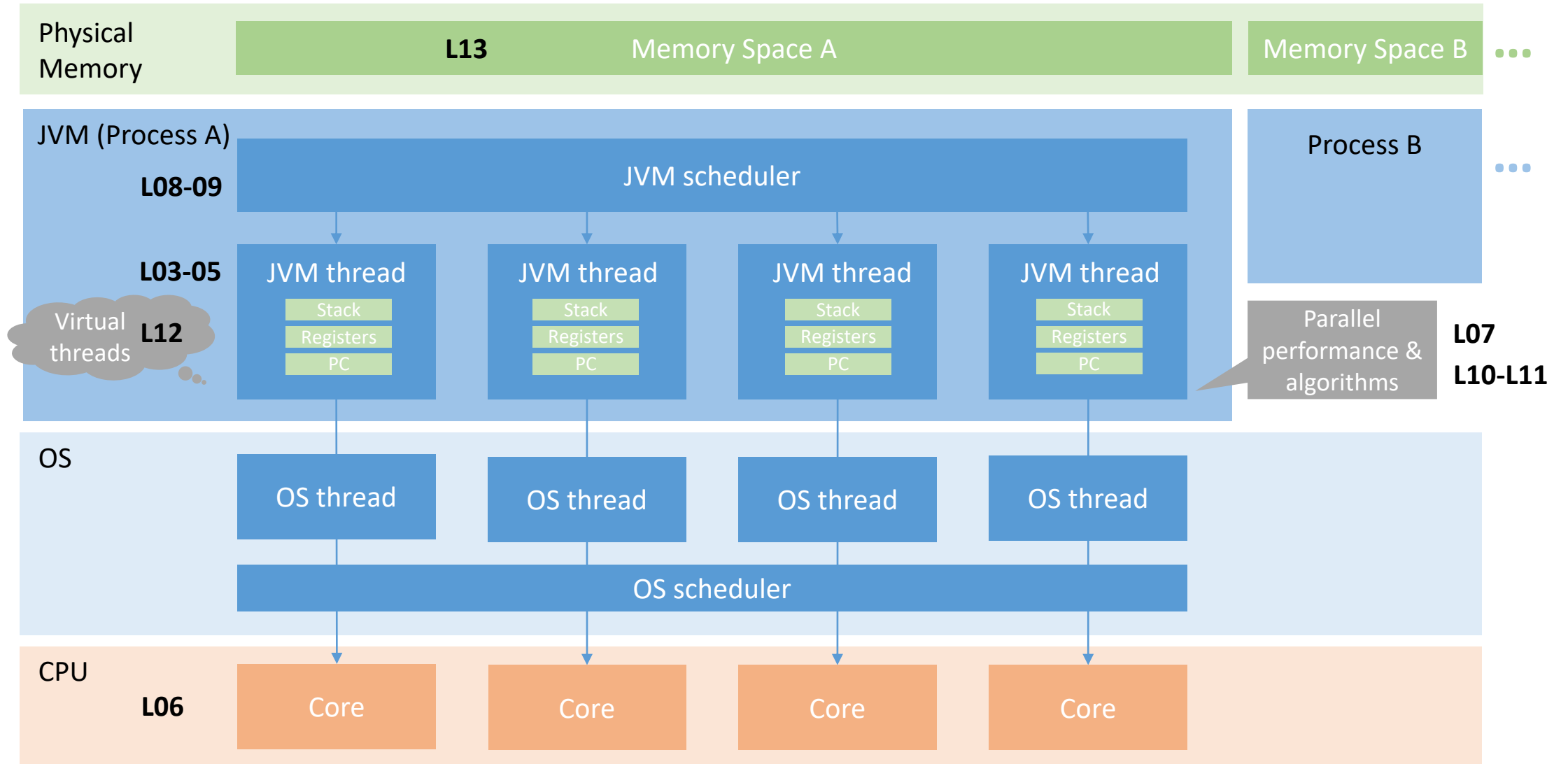


Parallel Programming

Basic Concepts in Parallelism

Big Picture (Part I)



Expressing Parallelism

- Work partitioning
 - Split up work of a single program into **parallel tasks**
- Can be done:
 - Explicitly / Manually (**task/thread parallelism**)
 - User explicitly expresses tasks/threads
 - Implicit parallelism:
 - Done automatically by the system (e.g., in **data parallelism**)
 - User expresses an operation and the system does the rest

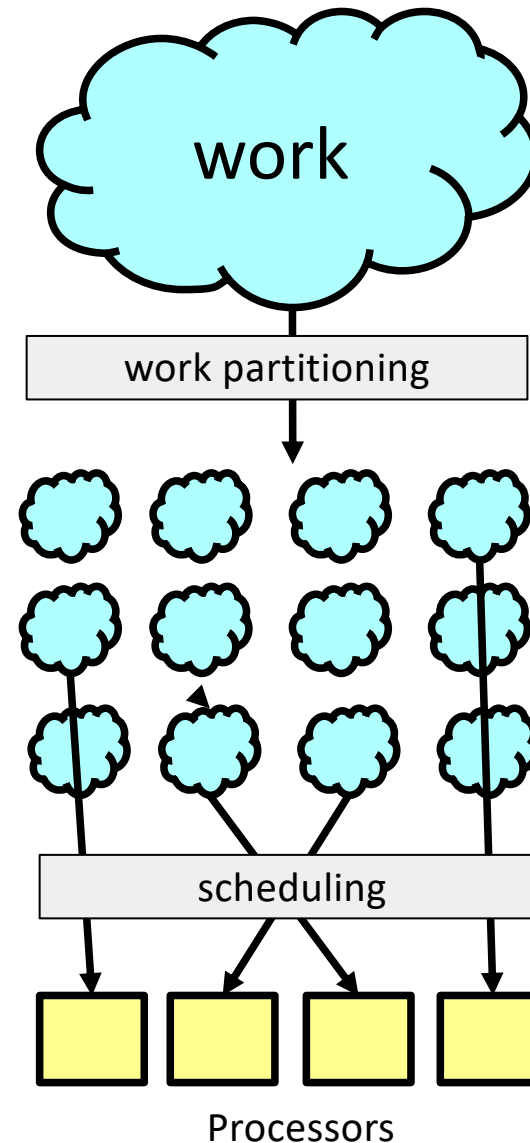
Work Partitioning & Scheduling

- **work partitioning**

- **split up** work into **parallel tasks/threads**
- (done by user)
- A task is a unit of work
- also called: **task/thread decomposition**

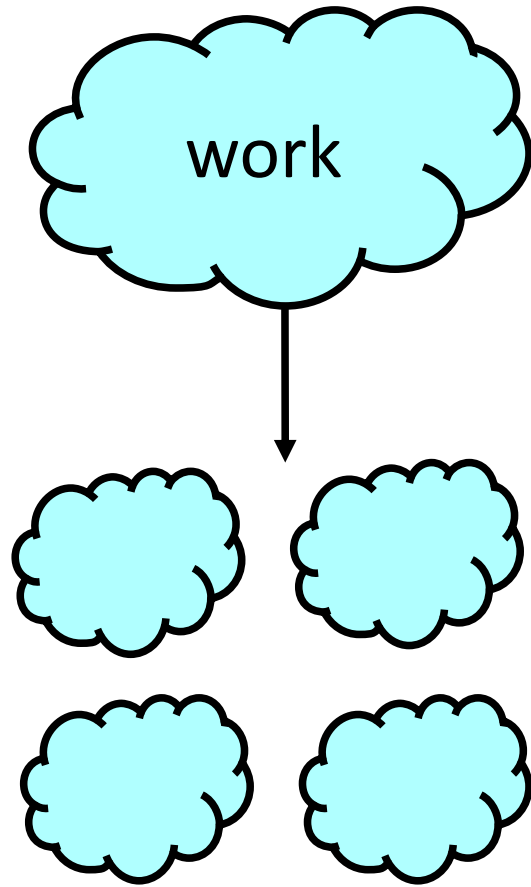
- **scheduling**

- assign tasks to processors
- (typically done by the system)
- goal: full utilization
(no processor is ever idle)

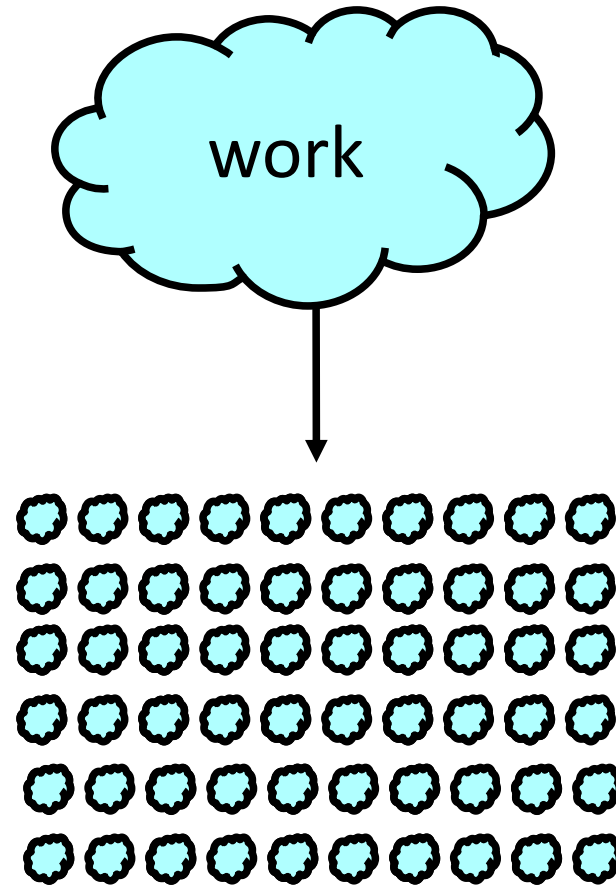


of chunks
should be larger
than the # of
processors

Task/Thread Granularity



Coarse granularity



Fine granularity

Coarse vs Fine granularity

- **Fine granularity:**

- more portable

- (can be executed in machines with more processors)

- better for scheduling

- but: if scheduling overhead is comparable to a single task → overhead dominates

Task granularity guidelines

- As small as possible
- but, significantly bigger than scheduling overhead
 - system designers strive to make overheads small

Scalability

An overloaded concept: e.g., how well a system reacts to increased load, for example, clients in a server

In parallel programming:

- speedup when we increase processors
- what will happen if processors $\rightarrow \infty$
- a program scales linearly \rightarrow linear speedup

Parallel Performance

Sequential execution time: T_1

Execution time T_p on p CPUs

- $T_p = T_1 / p$ (perfection)
- $T_p > T_1 / p$ (performance loss, what normally happens)
- $T_p < T_1 / p$ (sorcery!)

(parallel) Speedup

(parallel) speedup S_p on p CPUs:

$$S_p = T_1 / T_p$$

- $S_p = p \rightarrow$ linear speedup (perfection)
- $S_p < p \rightarrow$ sub-linear speedup (performance loss)
- $S_p > p \rightarrow$ super-linear speedup (sorcery!)
- **Efficiency: S_p / p**

Absolute versus Relative Speed-up

Relative speedup (Efficiency):

relative improvement from using P execution units.

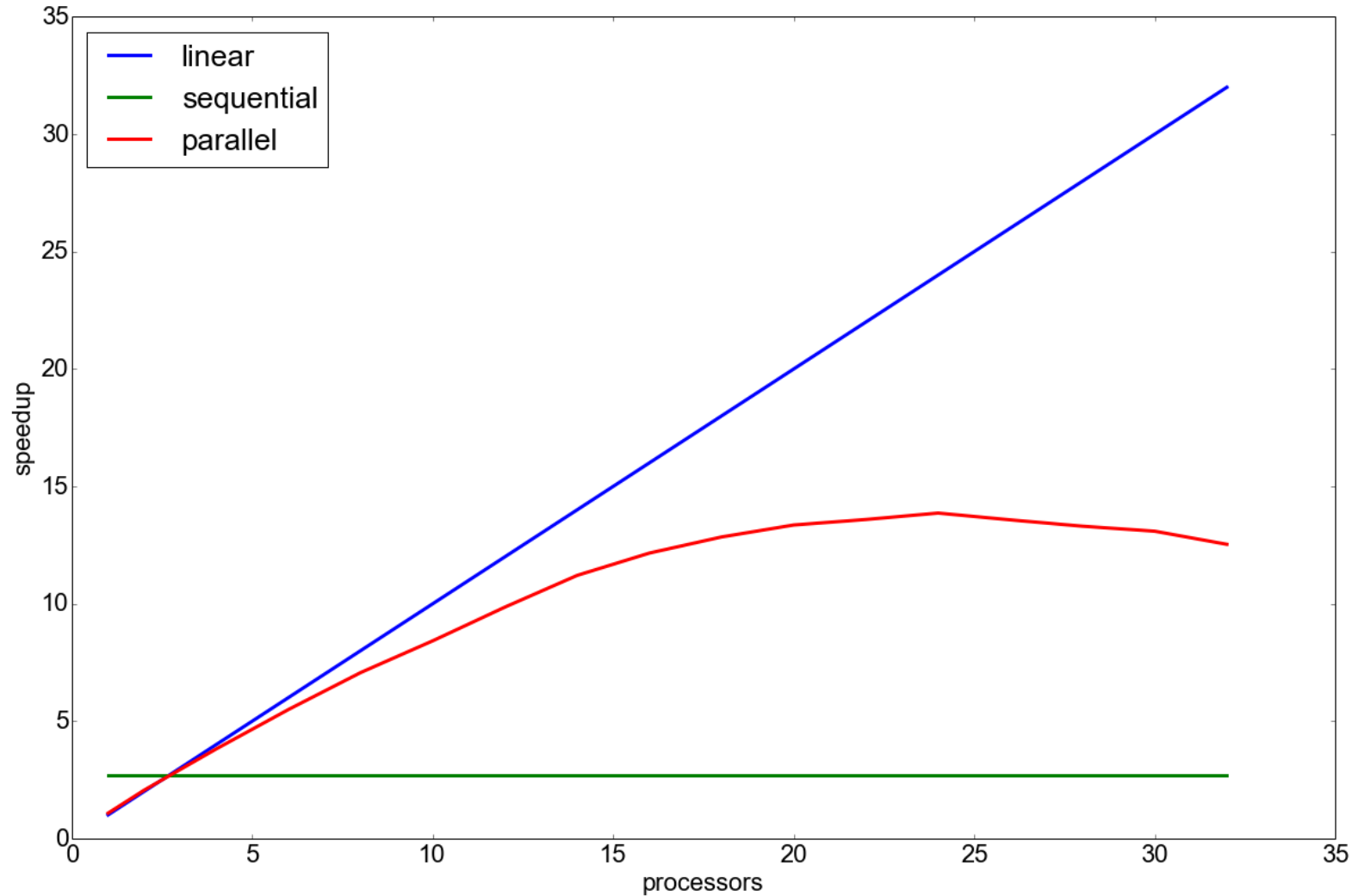
(Baseline: serialization of the parallel algorithm).

Sometimes there is a better serial algorithm that does not parallelize well.

In these cases it is fairer to use that algorithm for T_1 (absolute speedup).

Using an unnecessarily poor baseline artificially inflates speedup and efficiency.

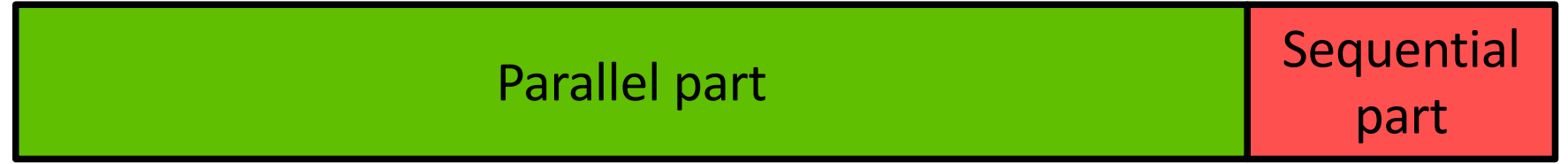
(parallel) speedup graph example



why $S_p < p$?

- Programs may not contain enough parallelism
 - e.g., some parts of program might be sequential
- overheads introduced by parallelization
 - typically associated with synchronization
- architectural limitations
 - e.g., memory contention

Question:

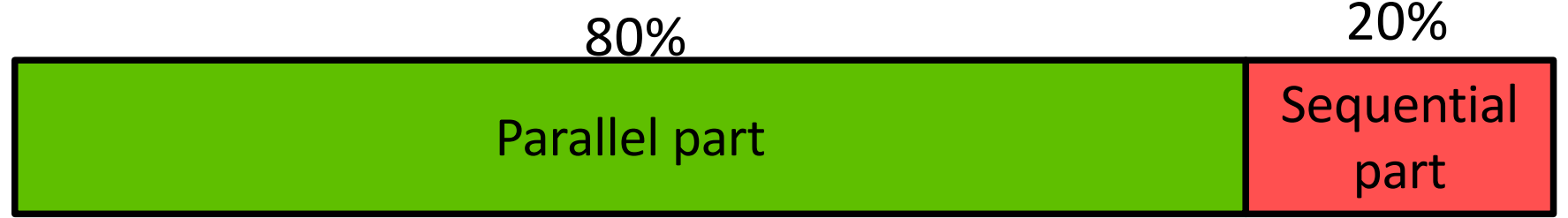


Parallel program:

- sequential part: 20%
- parallel part: 80% (assume it scales linearly)
- $T_1 = 10$

What is T_8 ? What is the speedup S_8 ?

Answer:



- $T_1 = 10$

- $T_8 = 3$

- $S_8 = T_1/T_8 = 10/3 = 3.33$

Amdahl's Law

...the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude.

— Gene Amdahl

Amdahl's Law – Ingredients

Execution time T_1 of a program falls into two categories:

- Time spent doing non-parallelizable serial work
- Time spent doing parallelizable work

Call these W_{ser} and W_{par} respectively

Amdahl's Law – Ingredients

Given P workers available to do parallelizable work, the times for sequential execution and parallel execution are:

$$T_1 = W_{ser} + W_{par}$$

And this gives a bound on speed-up:

$$T_p \geq W_{ser} + \frac{W_{par}}{P}$$

Amdahl's Law

Plugging these relations into the definition of speedup yields Amdahl's Law:

$$S_p \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{p}}$$

Amdahl's Law - Corollary

$$S_p \leq \frac{W_{ser} + W_{par}}{W_{ser} + \frac{W_{par}}{P}}$$

If f is the non-parallelizable serial fractions of the total work, then the following equalities hold:

$$\begin{aligned} W_{ser} &= fT_1, \\ W_{par} &= (1 - f)T_1 \end{aligned}$$

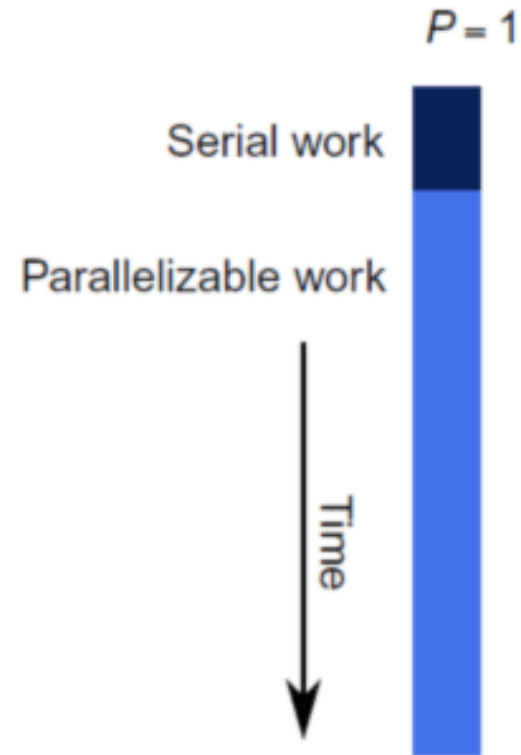
which gives:

$$S_p \leq \frac{1}{f + \frac{1-f}{P}}$$

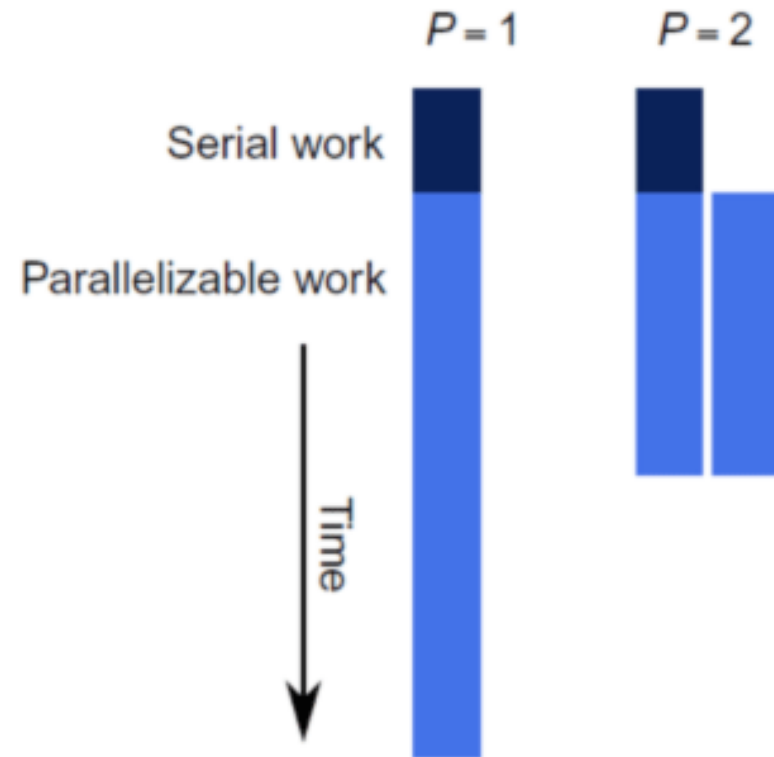
What happens if we have infinite workers?

$$S_{\infty} \leq \frac{1}{f}$$

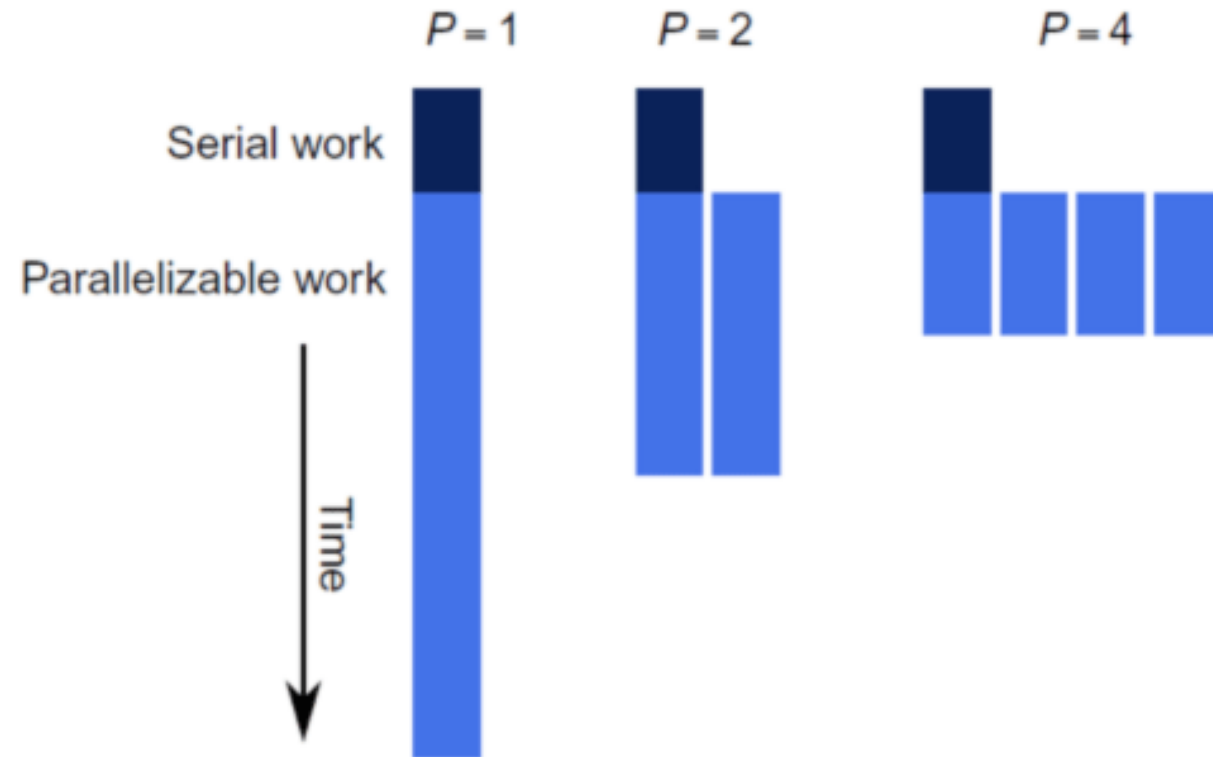
Amdahl's Law Illustrated



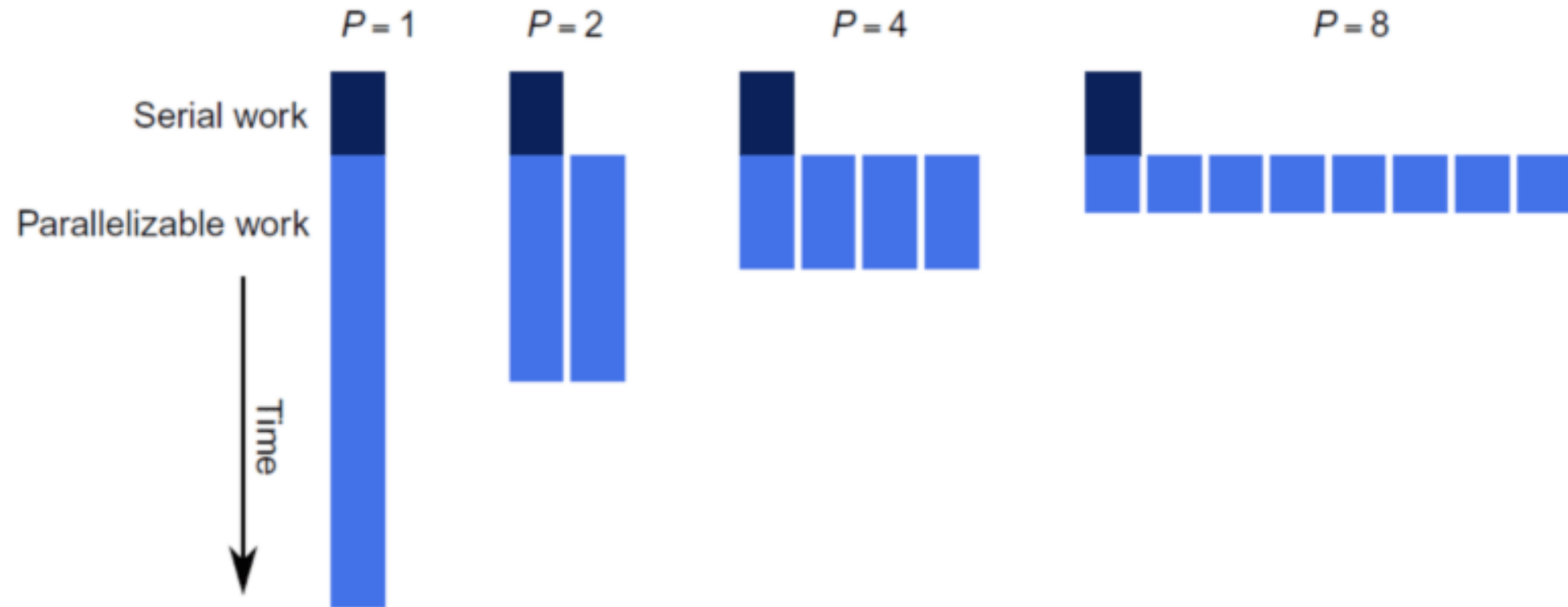
Amdahl's Law Illustrated



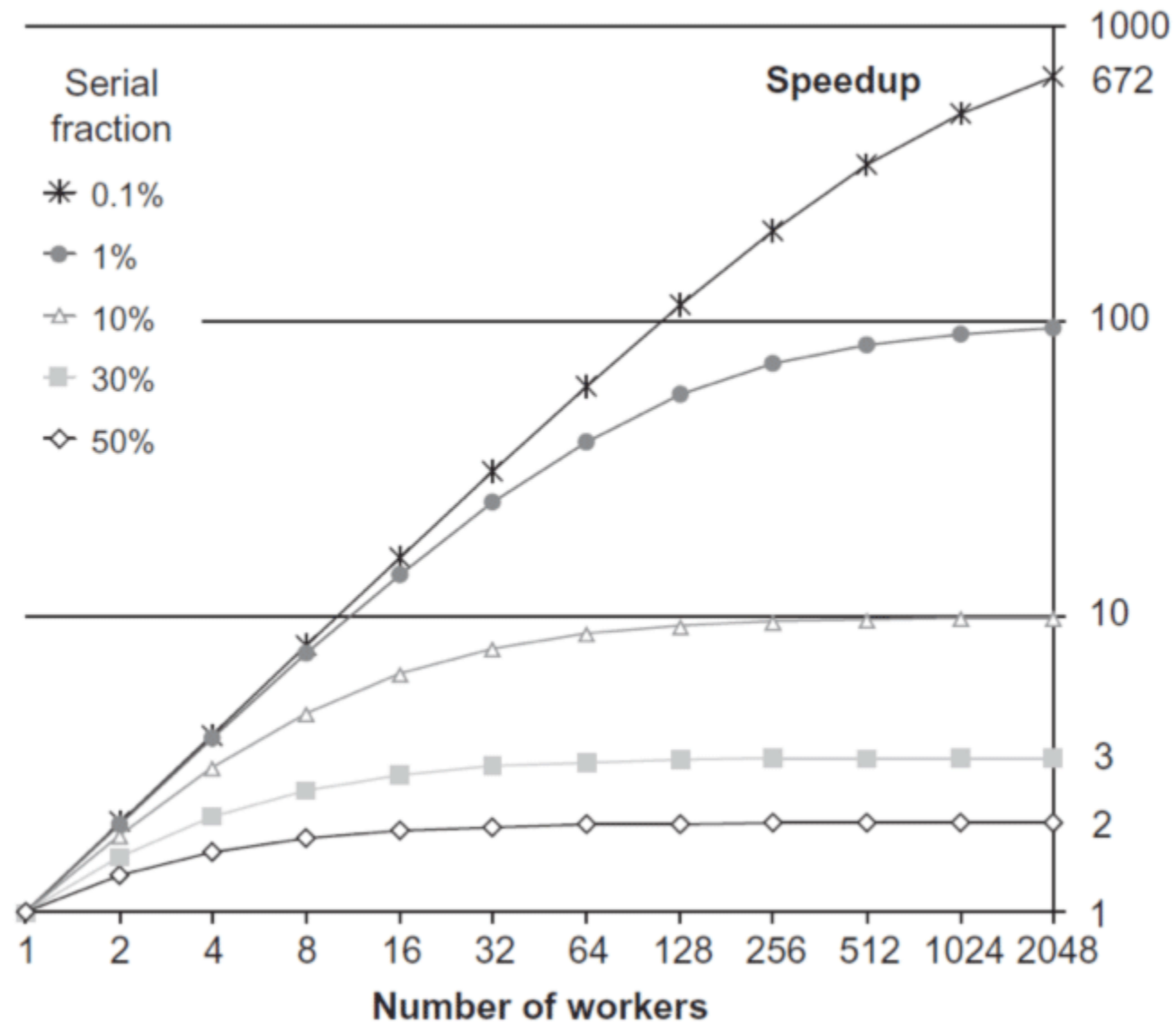
Amdahl's Law Illustrated



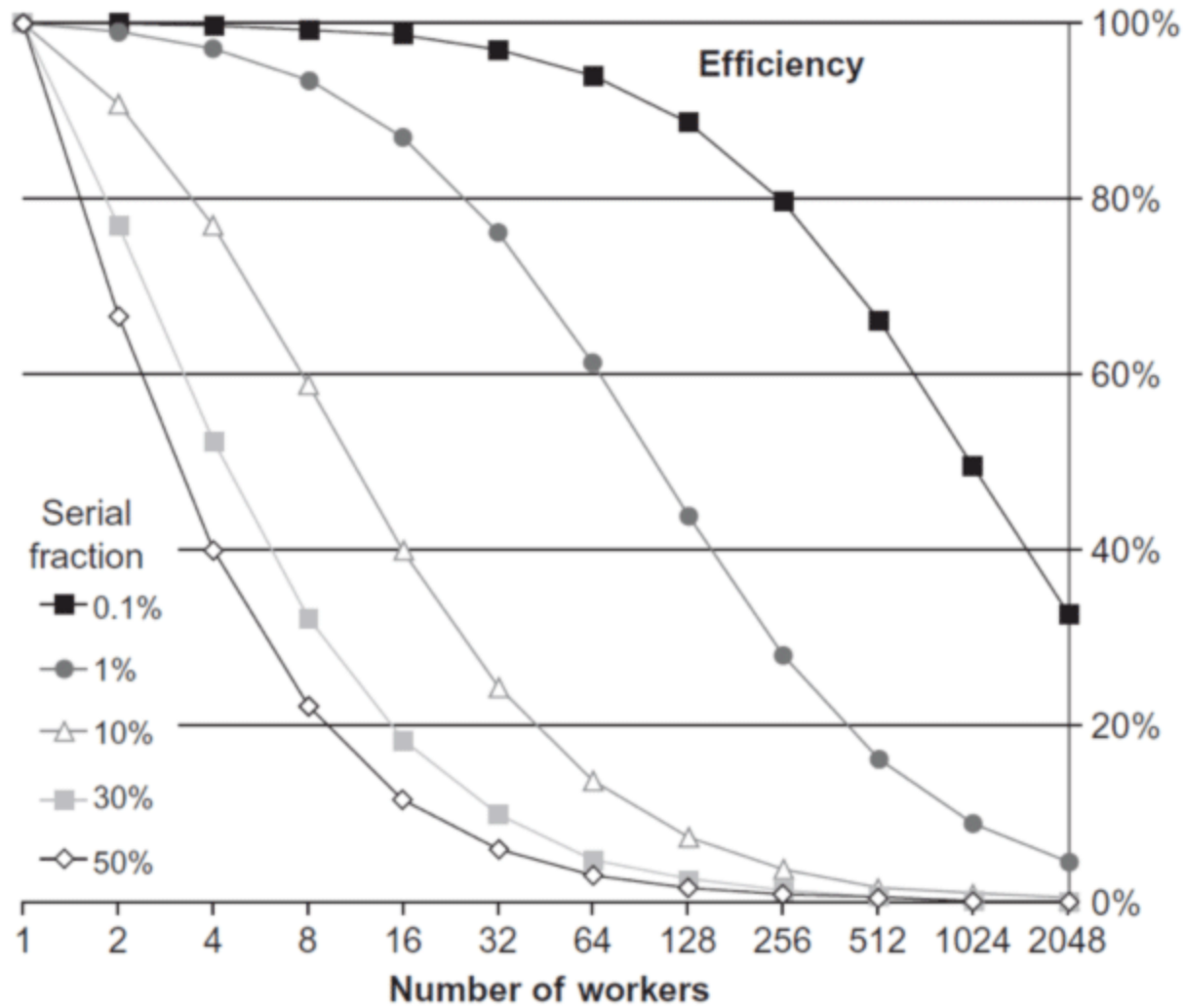
Amdahl's Law Illustrated



Speedup



Efficiency



Remarks about Amdahl's Law

- It concerns ***maximum speedup*** (Amdahl was an optimist (*or pessimist?*))
 - architectural constraints will make factors worse
- But his law is ***mostly bad news*** (as it puts a limit on scalability)
- takeaway: **all non-parallel parts of a program (no matter how small) can cause problems**
- Amdahl's law shows that efforts required to further reduce the fraction of the code that is sequential may pay off in large performance gains.
- Hardware that achieves even a small decrease in the percent of things executed sequentially may be considerably more efficient

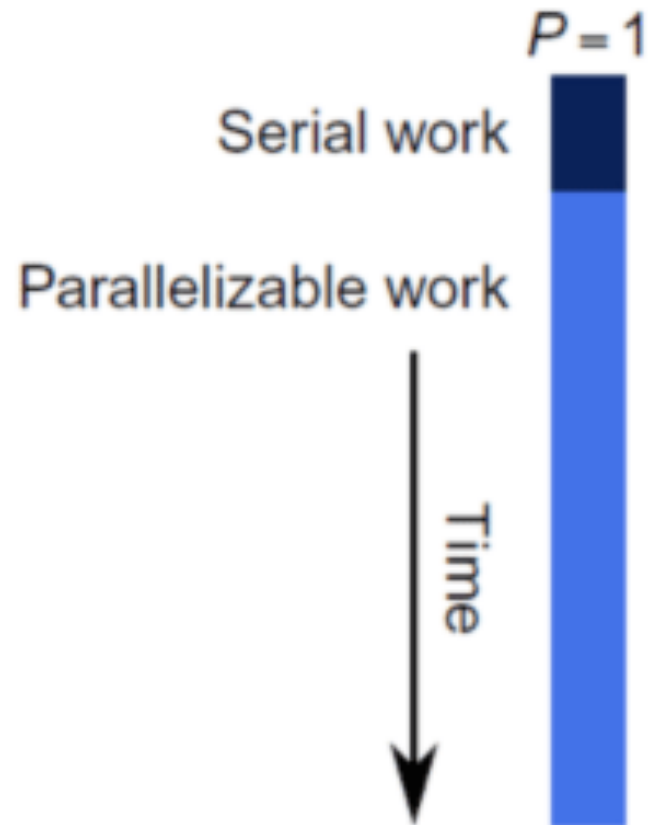
Gustafson's Law

- An alternative (optimistic) view to Amdahl's Law

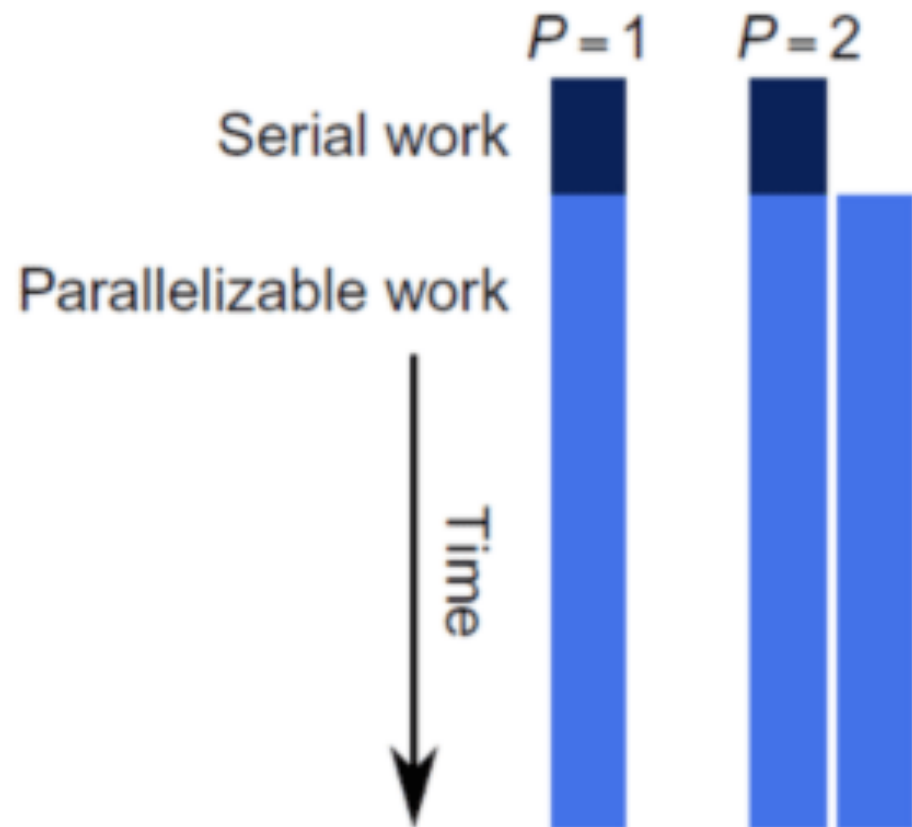
Observations:

- consider problem size
- run-time, not problem size, is constant
- more processors allows to solve larger problems in the same time
- parallel part of a program scales with the problem size

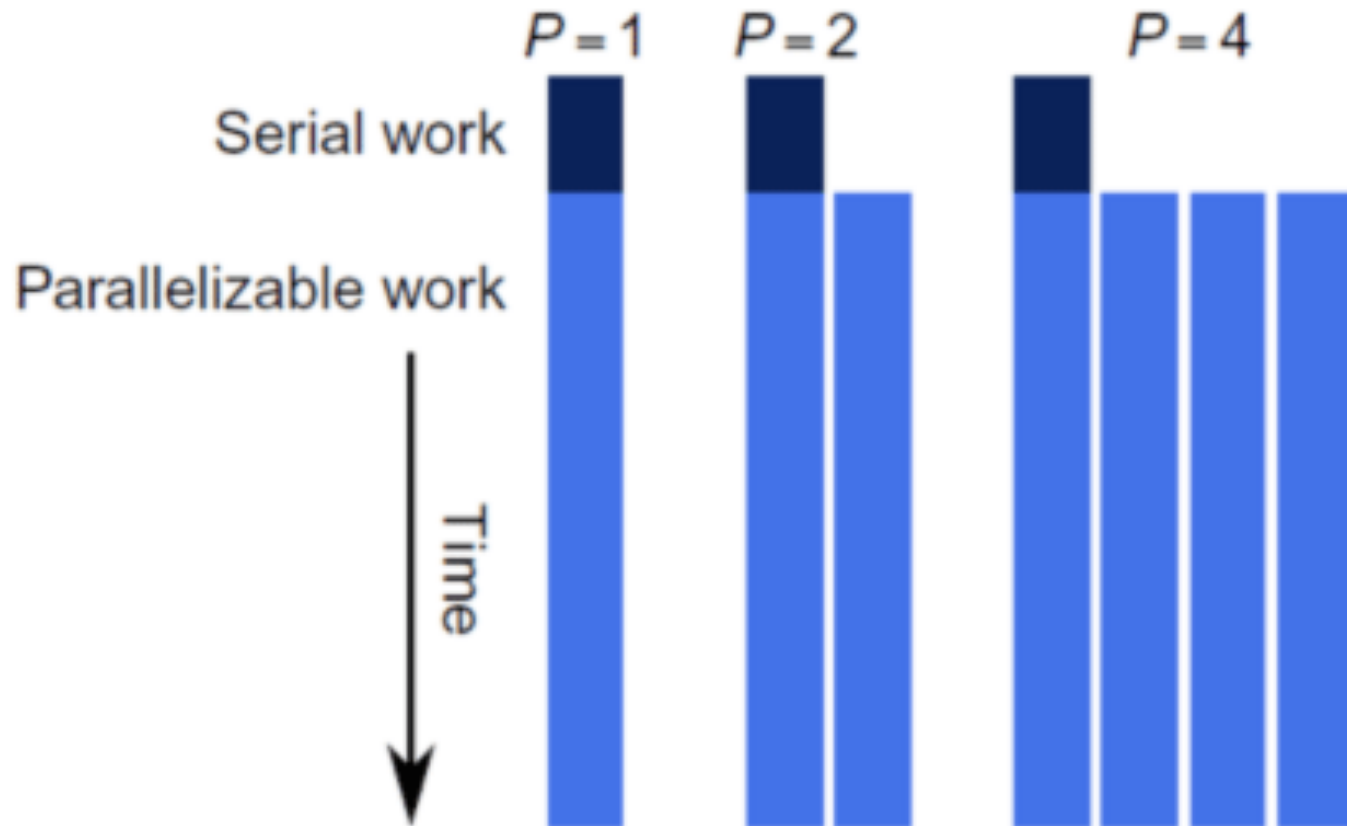
Gustafson's Law



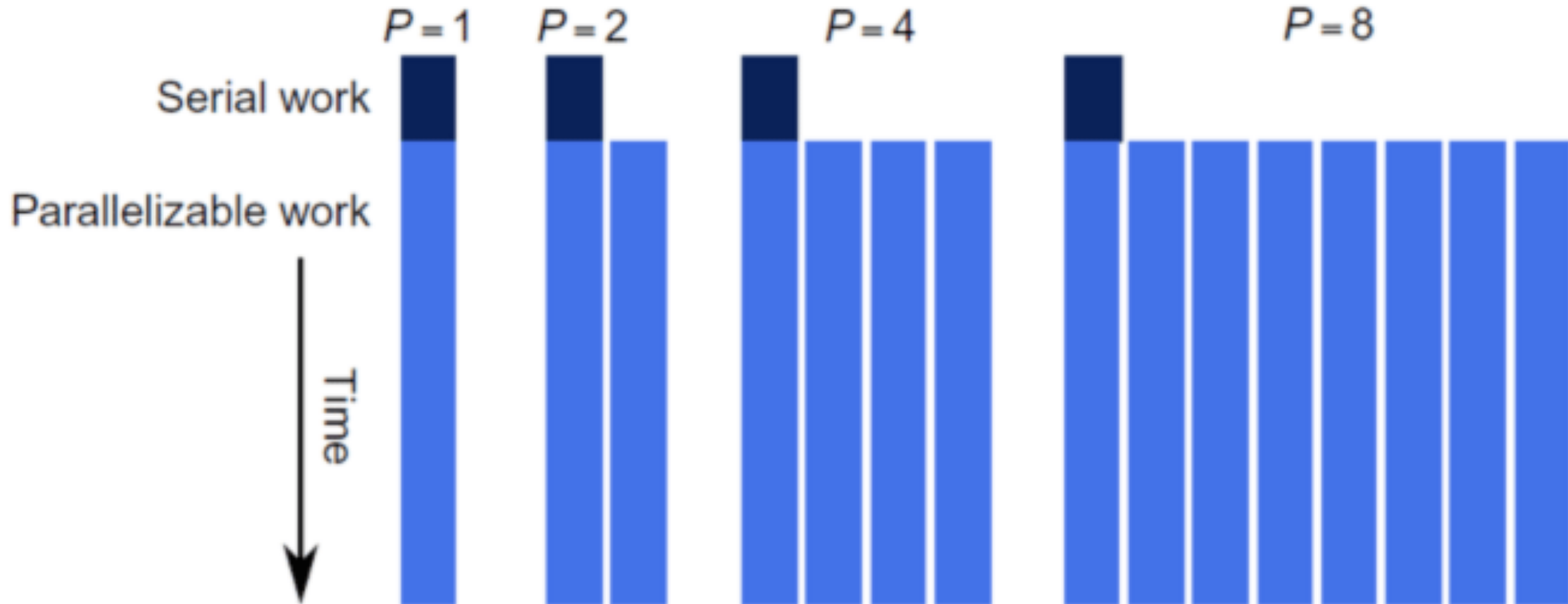
Gustafson's Law



Gustafson's Law



Gustafson's Law



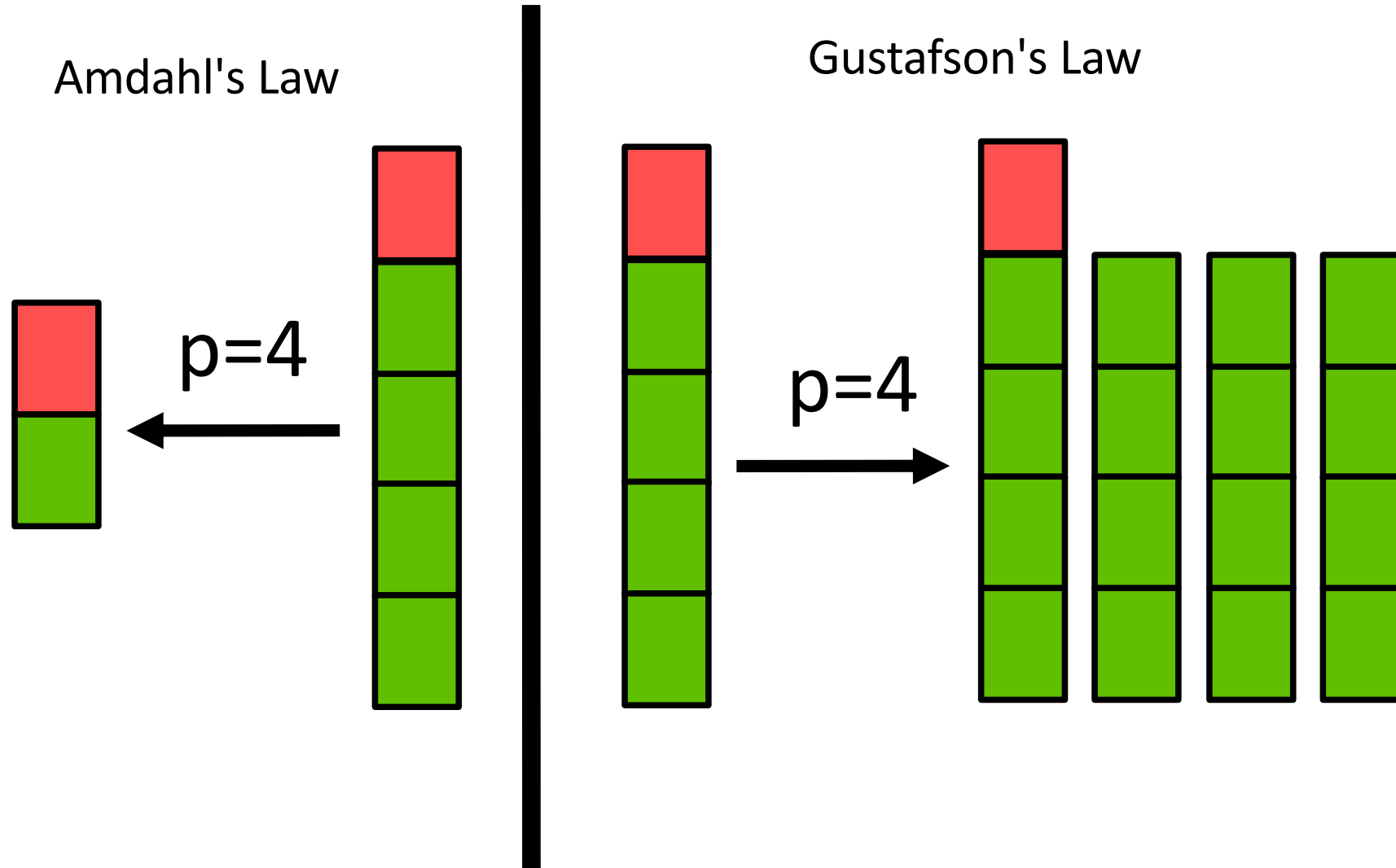
Gustafson's Law

- f : sequential part (no speedup)

$$W = p(1 - f)T_{wall} + fT_{wall}$$

$$\begin{aligned} S_p &= f + p(1 - f) \\ &= p - f(p - 1) \end{aligned}$$

Amdahl's vs Gustafson's Law



Summary

- Parallel speedup
- Amdahl's and Gustafson's law
- Parallelism: task/thread granularity