Performance evaluation of parallel programs

Pacheco, Section 2.6

Gianluigi Zavattaro
Dip. di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna
gianluigi.zavattaro@unibo.it

Scalability

- How much faster can a given problem be solved with *p* workers instead of one?
- How much more work can be done with *p* workers instead of one?
- What impact for the communication requirements of the parallel application have on performance?
- What fraction of the resources is actually used productively for solving the problem?

Slide credit: prof. David Padua

Speedup

- Let us define:
  - p = Number of processors / cores
  - T_{serial} = Execution time of the serial program
  - $T_{parallel}(p)$ = Execution time of the parallel program with p processors / cores

Speedup

- Speedup $S(p)$

$$S(p) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(p)} \approx \frac{T_{\text{parallel}}(1)}{T_{\text{parallel}}(p)}$$

- In the ideal case, the parallel program requires $1/p$ the time of the sequential program
- $S(p) = p$ is the ideal case of linear speedup
  - Realistically, $S(p) \leq p$
  - Is it possible to observe $S(p) > p$?

Warning

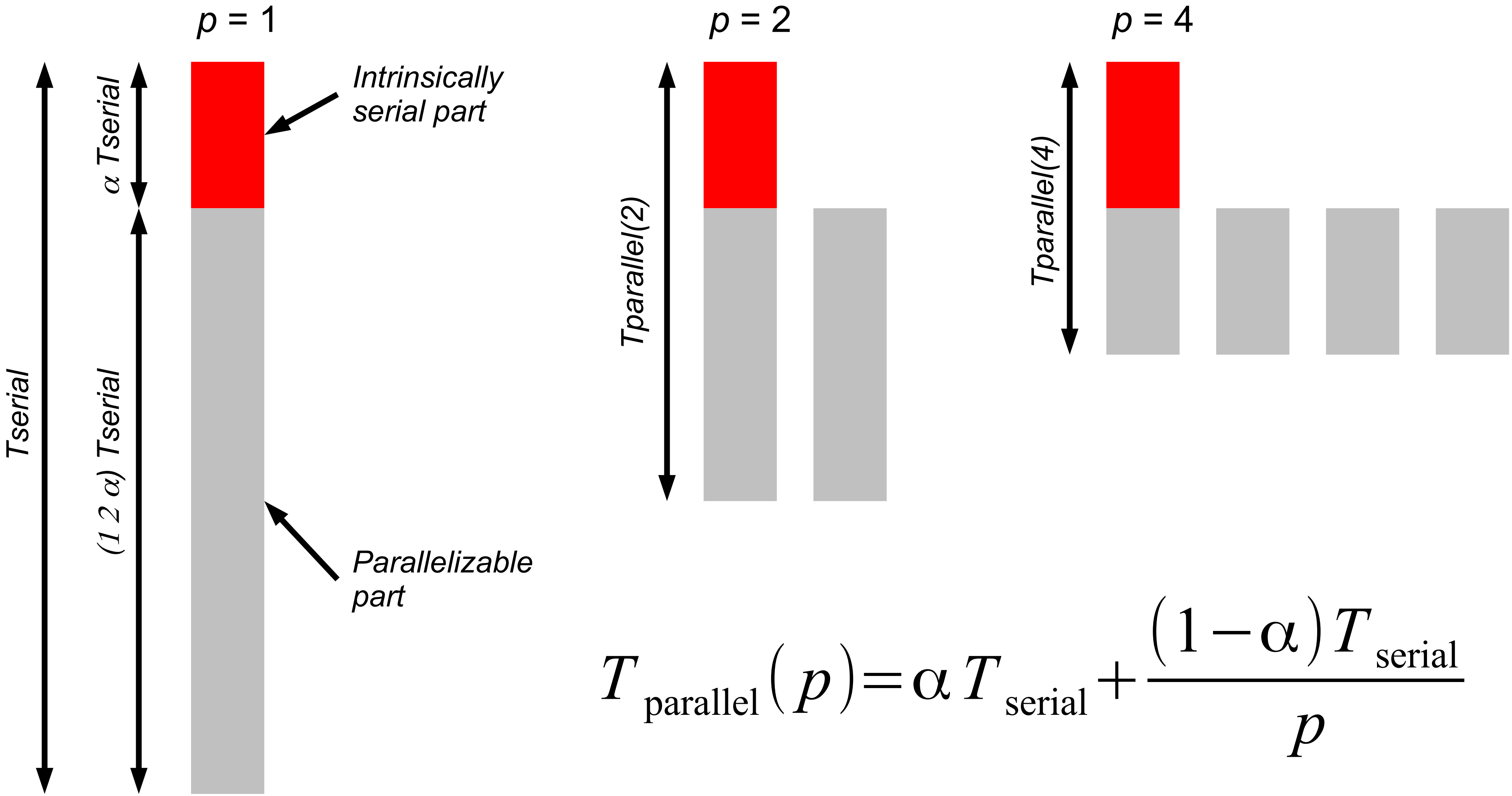
- <u>Never</u> use a serial program to compute T_{serial}
  - If you do that, you might see a spurious superlinear speedup that is not there
- Always use the parallel program with p = 1 processors

Non-parallelizable portions

- Suppose that a fraction α of the total execution time of the serial program can not be parallelized
  - E.g., due to:
    - Algorithmic limitations (data dependencies)
    - Bottlenecks (e.g., shared resources)
    - Startup overhead
    - Communication costs
- Suppose that the remaining fraction (1 - α) can be fully parallelized
- Then, we have:

$$T_{\text{parallel}}(p) = \alpha T_{\text{serial}} + \frac{(1-\alpha) T_{\text{serial}}}{p}$$

Example



$$T_{\text{parallel}}(p) = \alpha\, T_{\text{serial}} + \frac{(1-\alpha)\, T_{\text{serial}}}{p}$$

Example

- Suppose that a program has $T_{\text{serial}} = 20\text{s}$
- Assume that 10% of the time is spent in a serial portion of the program
- Therefore, the execution time of a parallel version with *p* processors is

$$T_{\text{parallel}}(p) = 0.1\,T_{\text{serial}} + \frac{0.9\,T_{\text{serial}}}{p} = 2 + \frac{18}{p}$$

Example (cont.)

- The speedup is

$$S(p) = \frac{T_{\text{serial}}}{0.1 \times T_{\text{serial}} + \frac{0.9 \times T_{\text{serial}}}{p}} = \frac{20}{2 + \frac{18}{p}}$$

- What is the maximum speedup that can be achieved when p $\rightarrow +\infty$?

Amdahl's Law

- What is the maximum speedup?

$$S(p) = \frac{T_{\text{serial}}}{T_{\text{parallel}}(p)}$$

$$= \frac{T_{\text{serial}}}{\alpha T_{\text{serial}} + \frac{(1-\alpha)T_{\text{serial}}}{p}}$$

$$= \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Gene Myron Amdahl (1922-2015)

Amdahl's Law

Amdahl's Law

- From

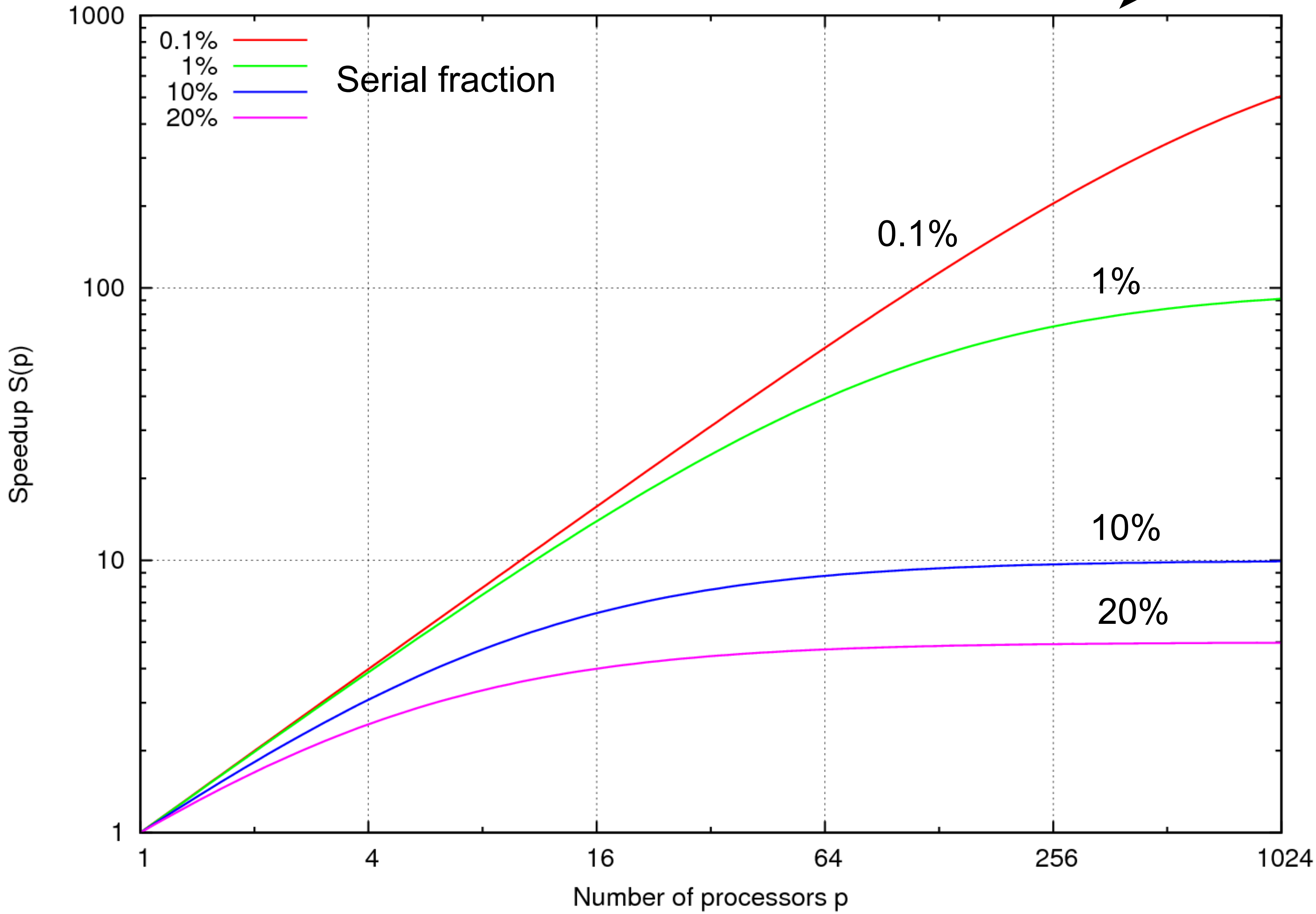
$$S(p) = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

we get an asymptotic speedup $1 / \alpha$ when p grows to infinity

If a fraction α of the total execution time is spent on a serial portion of the program, then the maximum achievable speedup is $1/\alpha$

Speedup

Note: log-log scale



Scaling Efficiency

- *Objective:*
  - Evaluate the impact of Amdahl's law on your parallel program
  - Quantify the effect on the execution time for each processor/core that is added
- *Solution*: measure *Strong Scaling*
  - Increase the number of processors *p* keeping the *total* problem size fixed
  - The *total* amount of work remains constant, while the amount of work for each processor decreases as *p* increases
  - How to quantify the impact of each added processor/core?
    - Divide the speedup for the number of processors/cores
  - Goal: understand how much the total execution time is reduced by adding more processors

Strong Scaling Efficiency

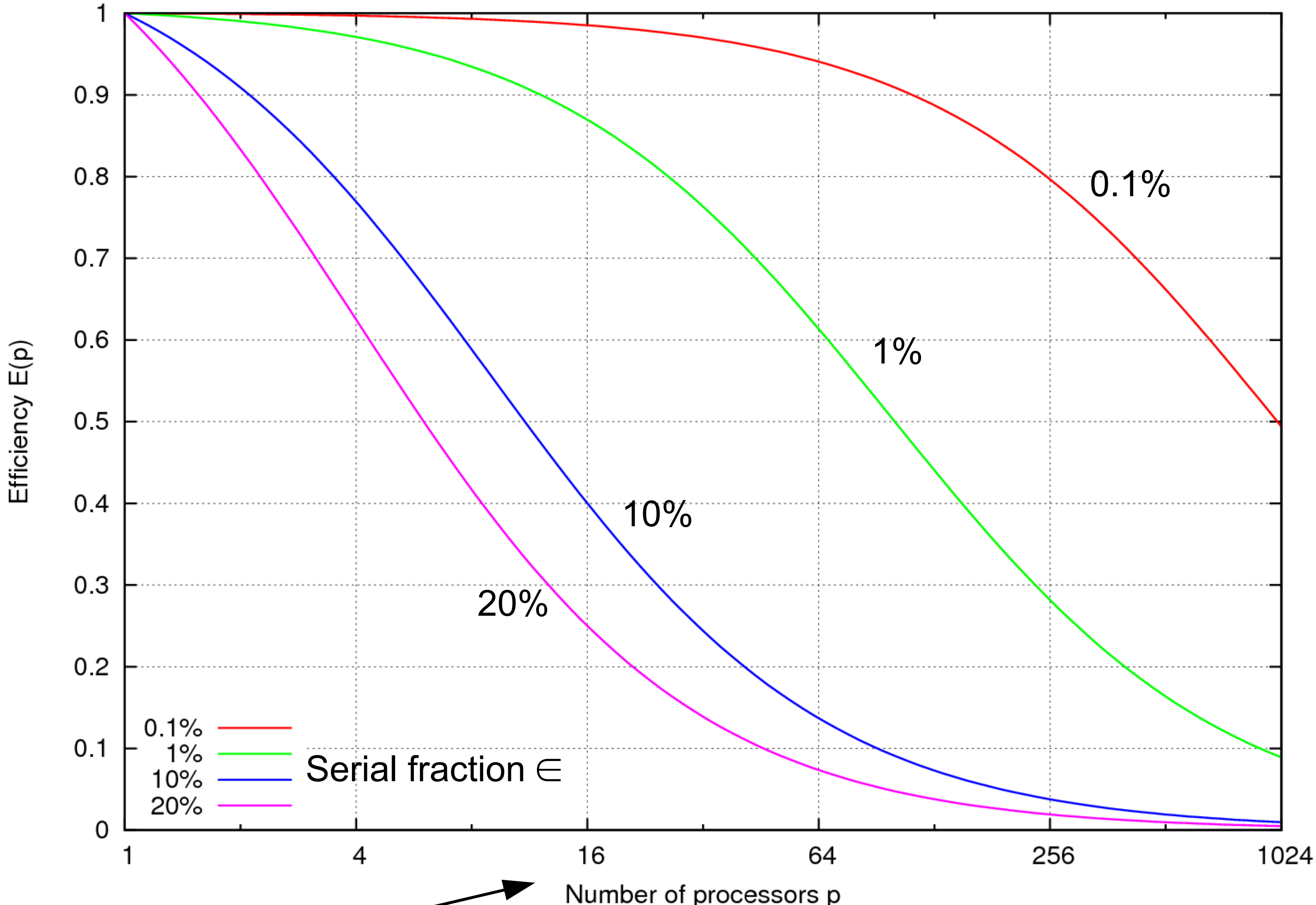
- $E(p)$ = Strong Scaling Efficiency

$$E(p) = \frac{S(p)}{p} = \frac{T_{\text{parallel}}(1)}{p \times T_{\text{parallel}}(p)}$$

where

- $T_{\text{parallel}}(p)$ = Execution time of the parallel program with p processors / cores

Strong Scaling Efficiency and Amdahl's Law

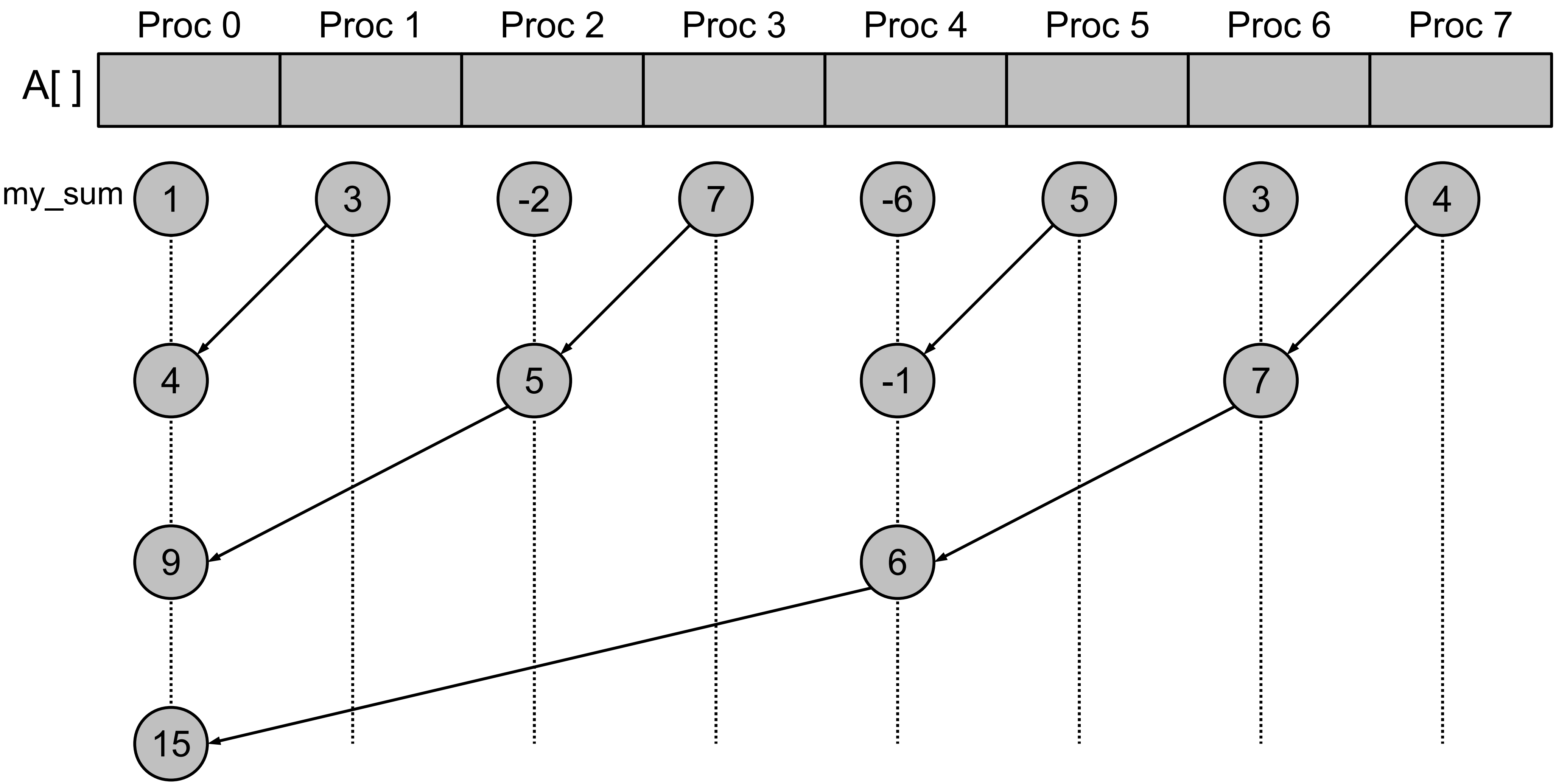


Note: logarithmic *x* scale

Negative result on strong scaling efficiency

- The efficiency always tends to zero because the speedup is limited by the constant $1/\alpha$
- But in many cases (remember the initial discussion about the need for more computational power) we want to use parallelism to do more computational work
  - If we consider increasing work, i.e., bigger input, the speedup could grow indefinitely
  - An example: vector sum reduction recalled in the next slide

Vector sum reduction



- In principle, with *n* processors we sum *n* values in $O(\log_2 n)$ time

Speedup with increasing work

- Observations:
  - Summing *n* values requires *O(n)* work (*n* constant operations)
  - To increase the work by a factor *p*, we consider *p* x *n* values
  - Speedup: $$Speedup(p) = \frac{T_{seq}(p)}{T_{par}(p)} = \frac{O(p \times n)}{O(\log(p \times n))}$$

  hence $$\lim_{p \to +\infty} Speedup(p) = +\infty$$
- This means:
  - if we increase both work and number of processors / cores, the speedup can increase (in principle) unboundedly
  - In these cases, according to Amdahl's terminology, we have that the serial fraction α decreases when the work increases

Weak Scaling Efficiency

- An alternative measure that considers increasing problem size is *Weak Scaling:*
  - Increase the number of processors *p* keeping the per-processor work fixed
  - The total amount of work grows as *p* increases
  - Goal: solve larger problems within the same amount of time
- *W*(*p*) = Weak Scaling Efficiency

$$W(p) = \frac{T_1}{T_p}$$

where

  - T_1 = time required to complete 1 work unit with 1 processor
  - T_p = time required to complete *p* work units with *p* processors

Example

- See omp-matmul.c
  - demo-strong-scaling.sh computes the times required for strong scaling
  - demo-weak-scaling.sh computes the times required for weak scaling
- Important note
  - For a given n, the amount of "work" required to compute the product of two n x n matrices is ~ n^3
  - To double the total work, do not double n!
    - The amount of work required to compute the product of two $(2n \; Y \, 2n)$ matrices is ~ $8n^3$, eight times the n x n case
  - To double the work, we must use matrices of size $(\sqrt[3]{2}n \times \sqrt[3]{2}n)$
  - In general, with p processes we use matrices of size $(\sqrt[3]{p}n \times \sqrt[3]{p}n)$

Taking times

- To compute speedup and efficiencies one needs to take the program wall clock time, excluding the input/output time

The actual computation is done here

```
#include <omp.h>
...
double start, finish;
/* read input data; this should not be measured */
start = omp_get_wtime();
/* actual code that we want to time */
finish = omp_get_wtime();
printf("Elapsed time = %e seconds\n", finish - start);

/* write output data; this should not be measured */
```