Lecture 19: Scaling

Amdahl’s law and Gustafson’s law

Current Assignments

- **Project 1: High Performance Linpack**
- You should have received an overleaf invitation.
- A link to the project description is on the website and posted in slack.
- The project is due in **1 week**.
- You will need to schedule one meeting with Ryan per week to get help and so we can monitor progress (11:59pm on April 22nd).
- We only have time for 2 projects, so they are each worth 20% of your final grade.

Speedup and Scalability

- Speedup, Scalability, strong scaling, weak scaling
- Amdahl’s law
- Gustafson’s law

Performance expectation

- When using 1 processor, the sequential program runs for 100 seconds. When we use 10 processors, should the program run for 10 times faster?
  - ❖ This works only for *embarrassingly parallel computations* – parallel computations that can be divided into completely independent computations that can be executed simultaneously. There may have no interaction between separate processes; sometime the results need to be collected.
    - ❑ Embarrassingly parallel applications are the kind that can scale up to a very large number of processors. Examples: Monte Carlo analysis, numerical integration, 3D graphics rendering, and many more.
  - ❖ In other types of applications, the computation components interact and have dependencies, which prevents the applications from using a large number of processors.

Scalability

- Scalability of a program measures how many processors that the program can effectively use.

Speedup and Strong scaling

- Let T_1 be the execution time for a computation to run on 1 processor and T_p be the execution time for the computation (with the same input – same problem) to run on P processors.

$$speedup(P) = \frac{T_1}{T_P}$$

  - Factor by which the use of P processors speeds up execution time relative to 1 processor for the same problem.
  - Since the problem size is fixed, this is referred to as “Strong scaling”.
  - Given a computation graph, what is the highest speedup that can be achieved?

Speedup

- $speedup(P) = \frac{T_1}{T_P}$
- Typically, $1 \leq speedup(P) \leq P$
- The speedup is ideal if $speedup(P) = P$
- Linear speedup: $speedup(P) = k \times P$ for some constant $0 < k < 1$

Efficiency

- The efficiency of an algorithm using P processors is

$$\text{Efficiency} = \text{speedup}(P) / P$$

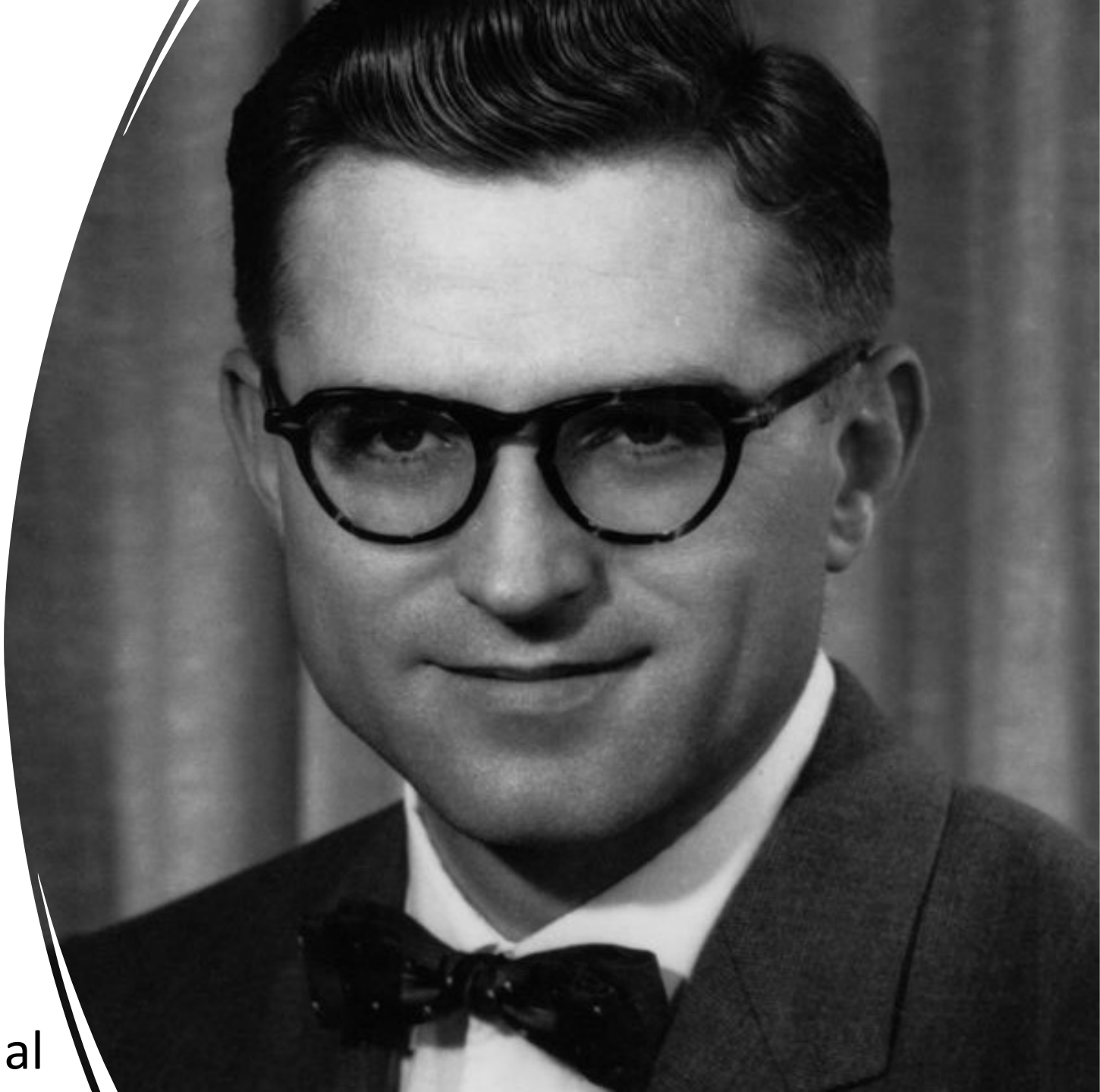
  - Efficiency estimates how well-utilized the processors are in running the parallel program.
  - Ideal speedup means Efficiency = 1 (100% efficiency).

Ahmdal's Law
(IBM Systems Architect)

How much faster is our problem solved as more CPUs are added?

Ahmdal recognized that the code consists of parts that are inherently serial and parts that are parallelable. Any parallel computation is limited by the serial part.

Better than that we can estimate the fraction of time taken in the serial part and the parallel part to 1) decide if it's worth adding more CPUs and 2) identify bottlenecks caused by serial code.



Gene Ahmdal

Amdahl's Law (fixed size speedup, strong scaling)

- Given a program, let *f* be the fraction that must be sequential and *1-f* be the fraction that can be parallelized
- $T_P = f\ T_1 + \frac{(1-f)T_1}{P}\ where$

  T_1 is the time taken with one process (think CPU) (independent variable)

  P is the number of processes (independent variable)

  T_P is the time taken given P (independent variable)

  f is the dependent variable we can estimate.

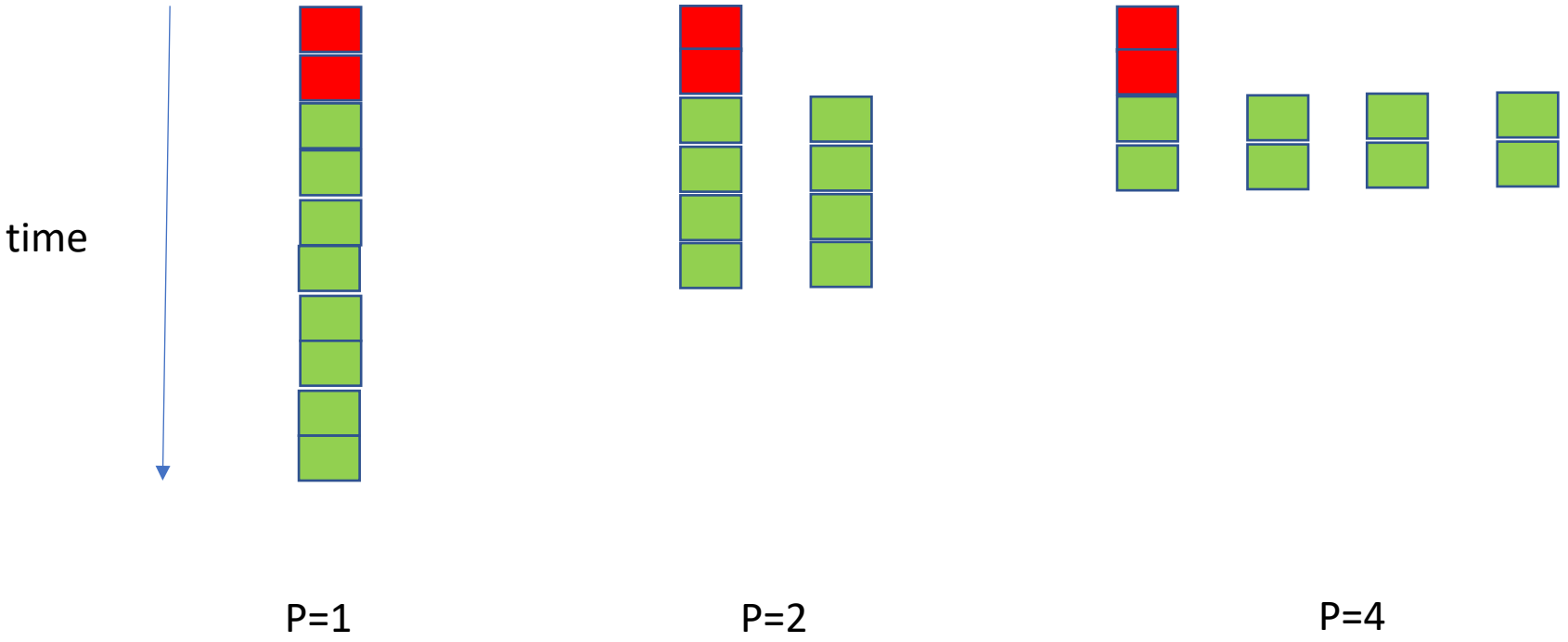
  We measure the independent variables.

Amdahl's Law (fixed size speedup, strong scaling)

- Given a program, let *f* be the fraction that must be sequential and *1-f* be the fraction that can be parallelized
- $T_P = f\ T_1 + \frac{(1-f)T_1}{P}$
- $Speedup(P) = \frac{T_1}{T_P} = \frac{T_1}{f\ T_1 + \frac{(1-f)T_1}{P}} = \frac{1}{f + (1-f)/P}$
- When $P \to \infty, Speedup(P) = \frac{1}{f}$
- Original paper: *Amdahl, Gene M. (1967). "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities" . AFIPS Conference Proceedings (30): 483–485.*

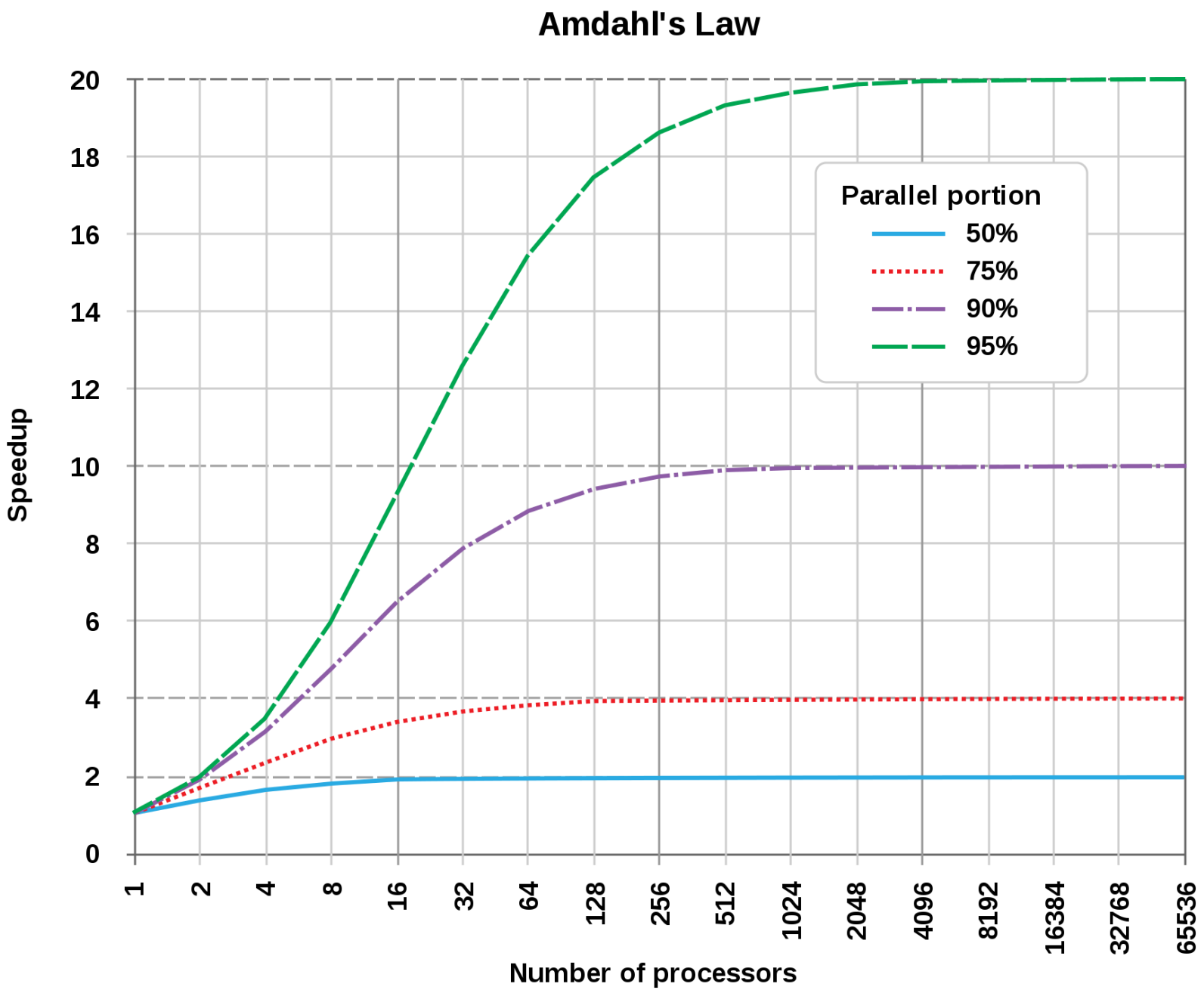
Amdahl's law

Amdahl's law: As P increases, the percentage of work in the parallel region reduces, performance is more and more dominated by the sequential region.



Implication of Amdahl's Law

- For strong scaling, the speedup is bounded by the percentage of sequential portion of the program, not by the number of processors!
- Strong scaling will be harc to achieve for many programs.



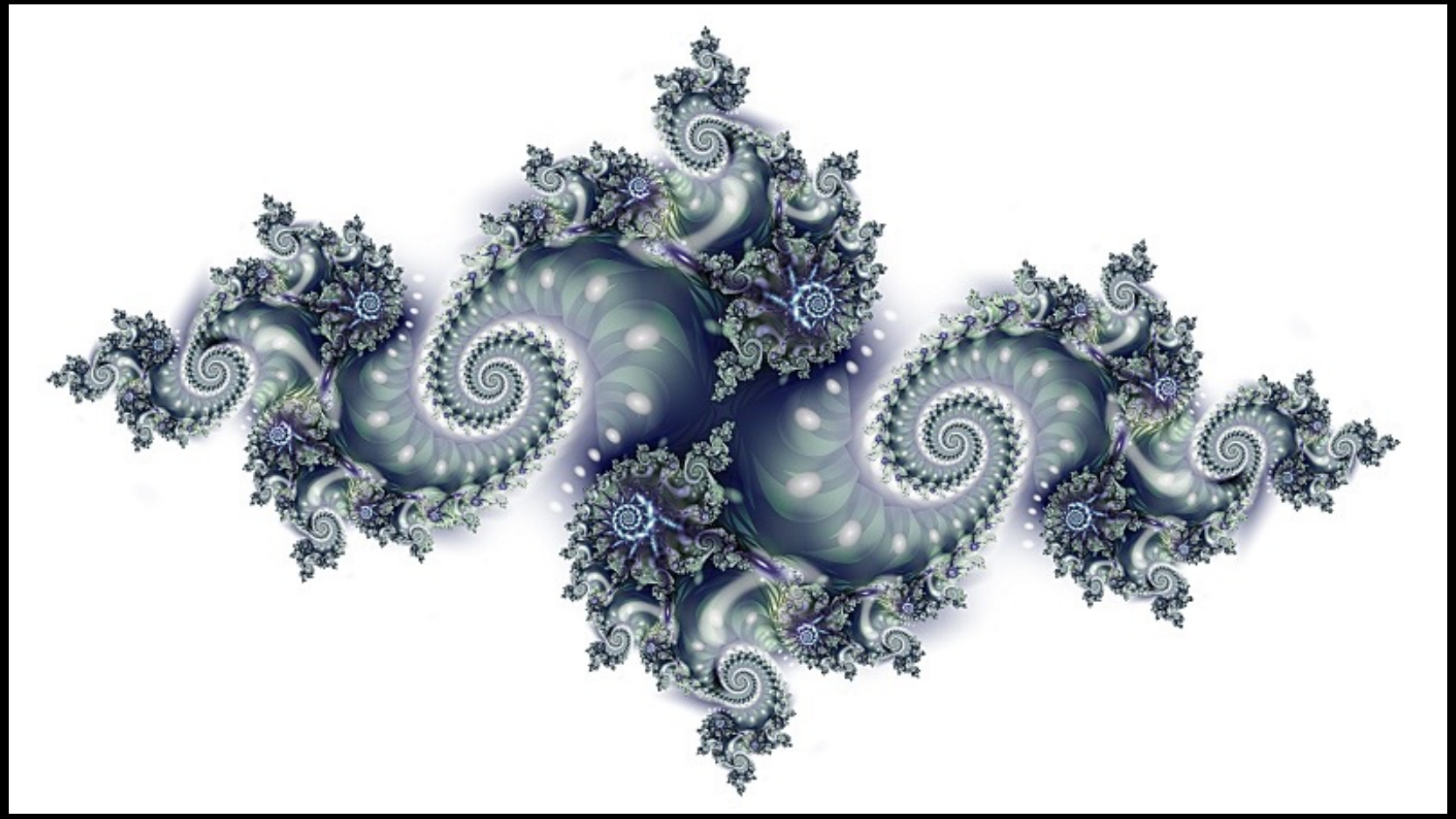
How do we use this in practice?

- Perform experiments to find the execution time for different process counts.
- This is called a scaling study
- From the measured times we can fit Ahmdal's law to find the value of f (the inherently serial time)

Install scipy and numpy into the plotting environment so we can do some parameter estimation

```
mfricke@hopper:~ $ module load miniconda3
mfricke@hopper:~ $ source activate plotting
(plotting) mfricke@hopper:~ $ conda install scipy numpy
```

We will use a Julia Fractal Generator as our example code



We will use a Julia Fractal Generator as our example code and OpenMP to parallelise the code for our experiments

https://github.com/gmfricke/Juliaset.git

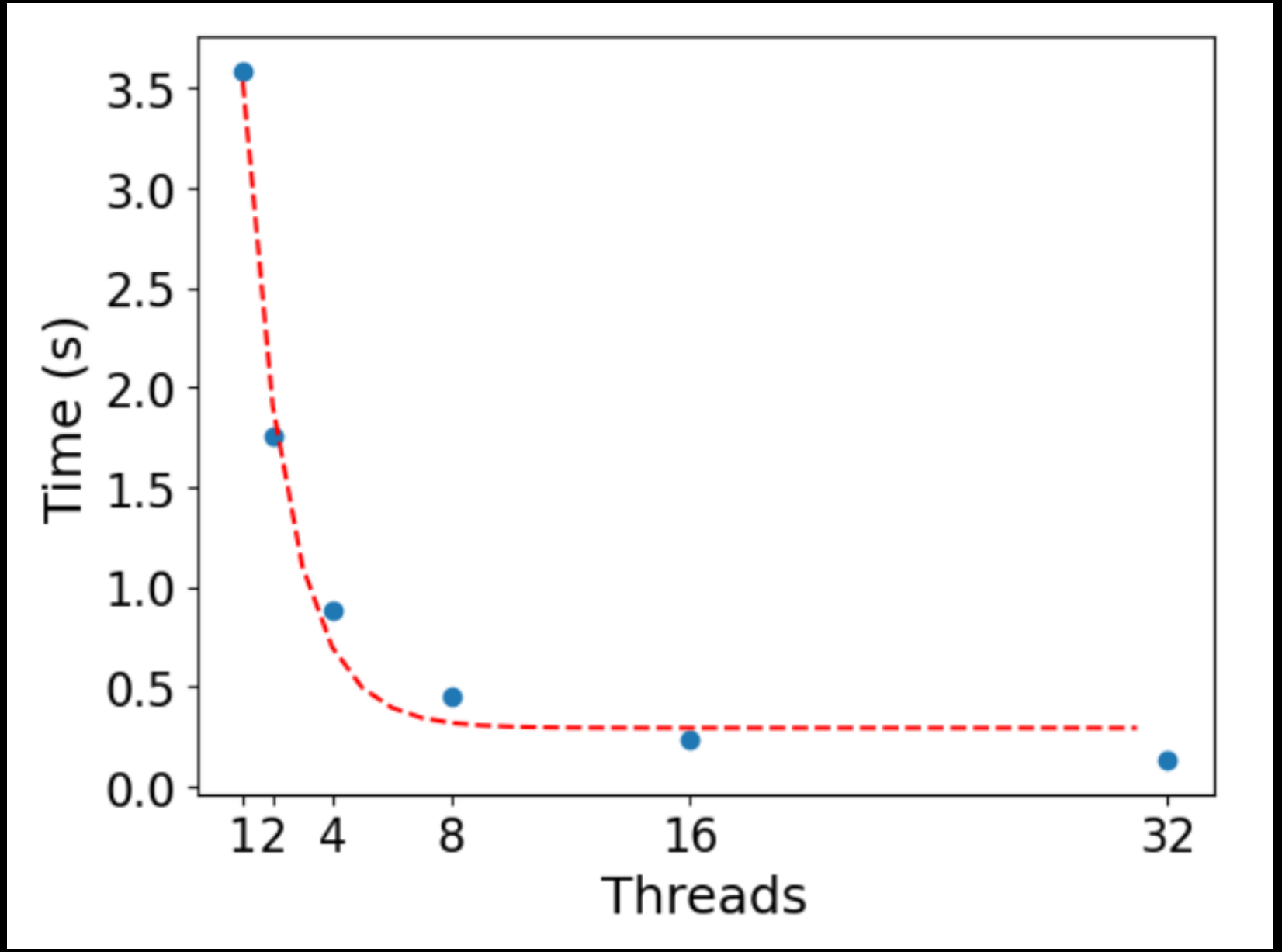
We will use a Julia Fractal Generator as our example code and OpenMP to parallelise the code for our experiments

```
gcc -fopenmp juliaset.c -o juliaset
```

We want to perform a parameter sweep over the number of threads

```
cat juliaset_strong.slurm
#!/bin/bash
#SBATCH --job-name strong_julia
#SBATCH --partition general
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 32
#SBATCH --time 5:00
#SBATCH --output=julia_strong_%a.out
#SBATCH --array 1-7
#SBATCH --mail-user mfricke@unm.edu
OUTPUT_PATH=juliaset_strong_$SLURM_ARRAY_TASK_ID.tga
SCALE_FACTOR=$((2 ** ($SLURM_ARRAY_TASK_ID-1)))
export OMP_NUM_THREADS=$SCALE_FACTOR
FRACTAL_HEIGHT=10000
FRACTAL_WIDTH=2000
echo Scale Factor: $SCALE_FACTOR
echo Number of Threads: $OMP_NUM_THREADS
echo FRACTAL_DIMS: $FRACTAL_HEIGHT x $FRACTAL_WIDTH
./juliaset $FRACTAL_HEIGHT $FRACTAL_WIDTH $OUTPUT_PATH
```

Strong Scaling



```
# Strong scaling. Vary the number of threads for a fixed problem size
y1 = 3.58595
y=list(map(lambda x: y1/x, y)) # divide y1, the time with one thread by each of the other times

# plot input vs output
pyplot.scatter(x, y)

popt, _ = curve_fit(Ahmdal_objective, x, y)
## summarize the parameter values
a = popt
#speedup = 1 / (s + p / N)
print('y = 1/(%.5f + %.5f/N)' % (a, 1-a))
# define a sequence of inputs between the smallest and largest known inputs
x_line = numpy.arange(min(x), max(x), 1)
# calculate the output for the range
y_line = Ahmdal_objective(x_line, a)
# create a line plot for the mapping function
pyplot.plot(x_line, y_line, '--', color='red')


pyplot.xticks(x, size=16)
pyplot.yticks(size=16)
pyplot.xlabel("Threads", size=18)
pyplot.ylabel("Speedup", size=18)

pyplot.show()
```

```
# Ahmdal's law
def Ahmdal_objective(x, a):
 return 1/(a + (1-a)/x)
```

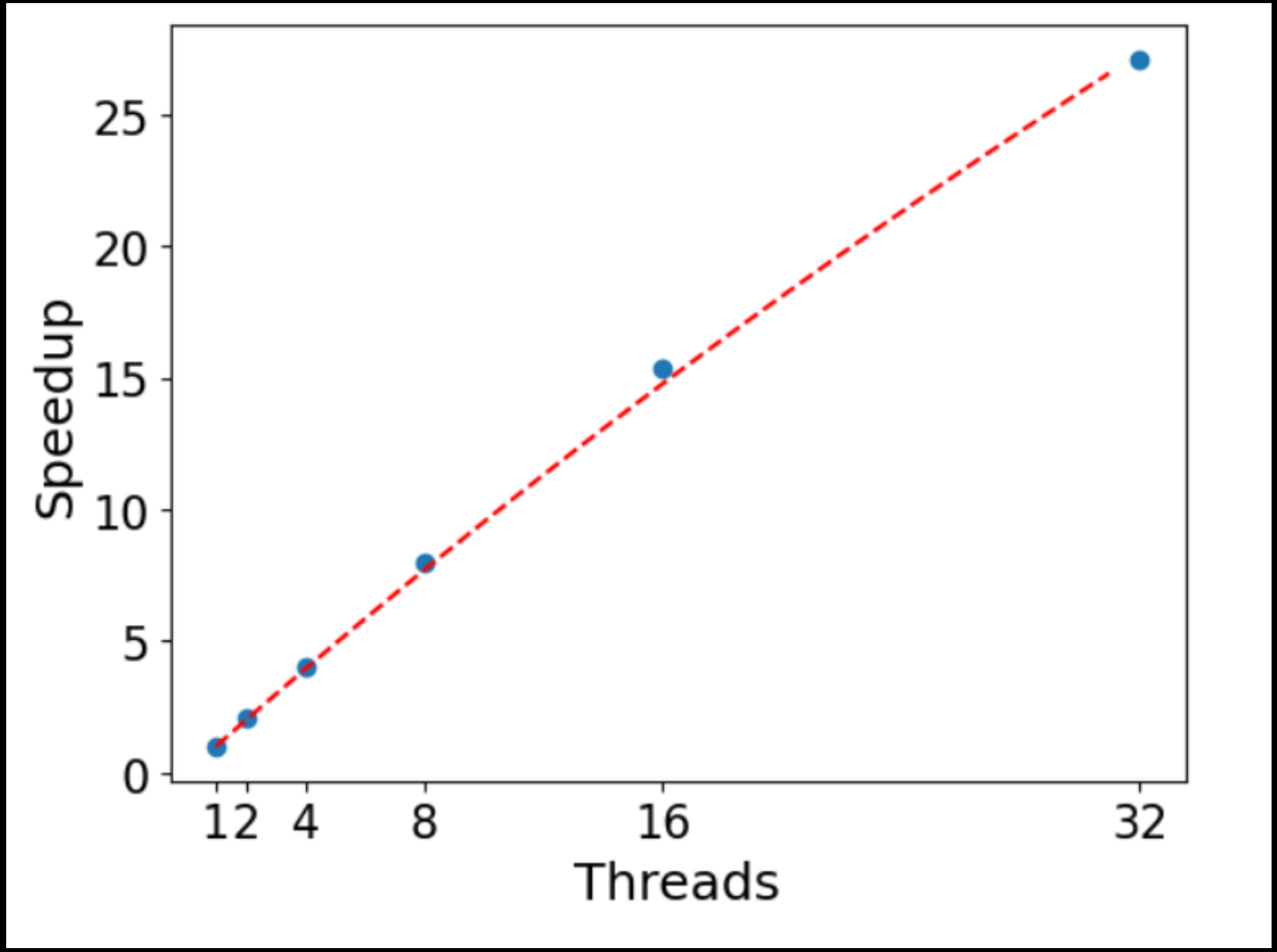
```
y = 1/(0.00558 + 0.99442/N)
```

Strong Scaling

Speedup = T_1/T_N

y = 1/(0.00558 + 0.99442/N)

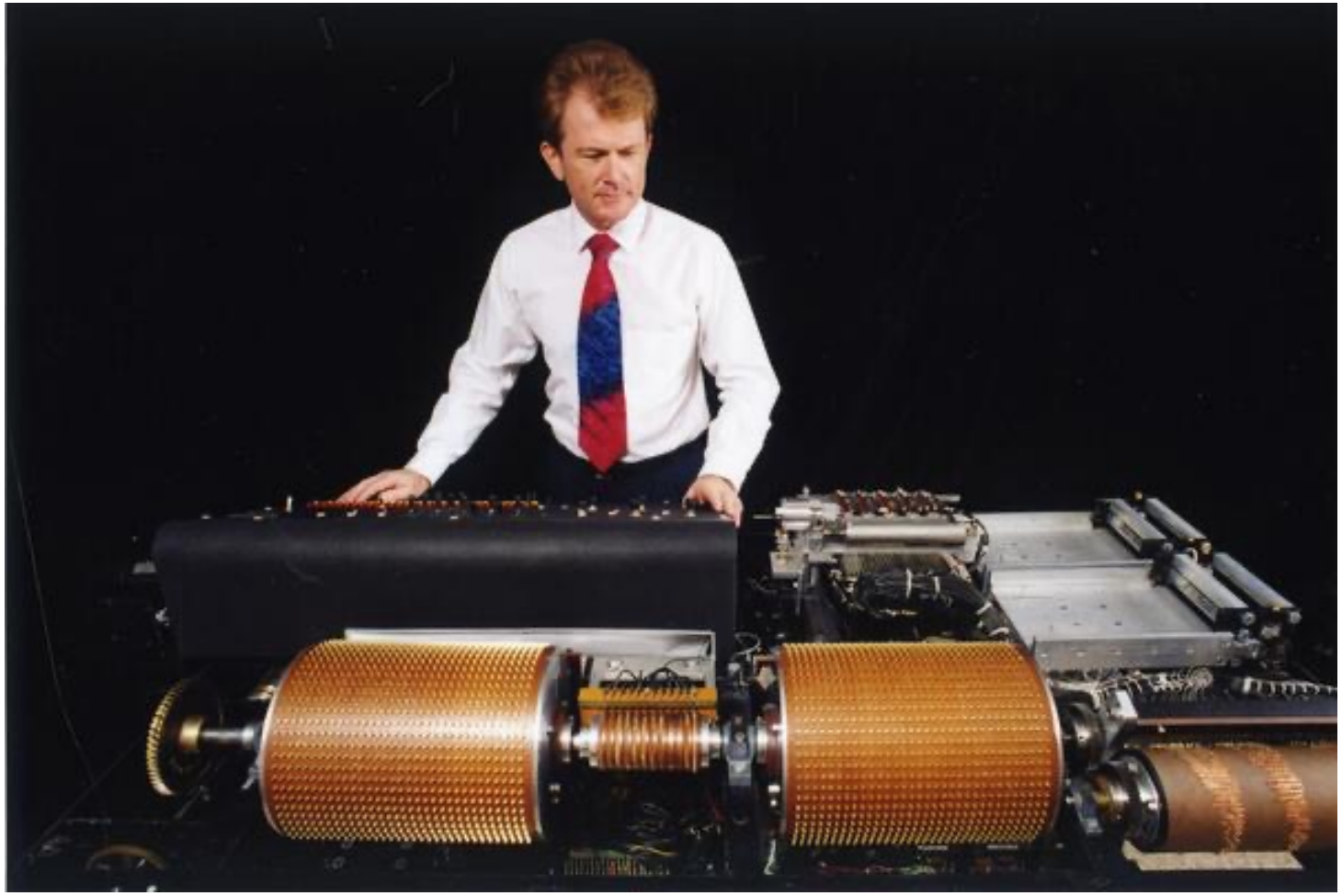
f is only 0.006



Gustafson's Law (scaled speedup, weak scaling)

- While Ahmdal's law lets us describe the limits of parallel speedup Gustafson's law tells us how we can benefit from more CPUs when solving larger problems.

John Gustafson
AMD, Intel



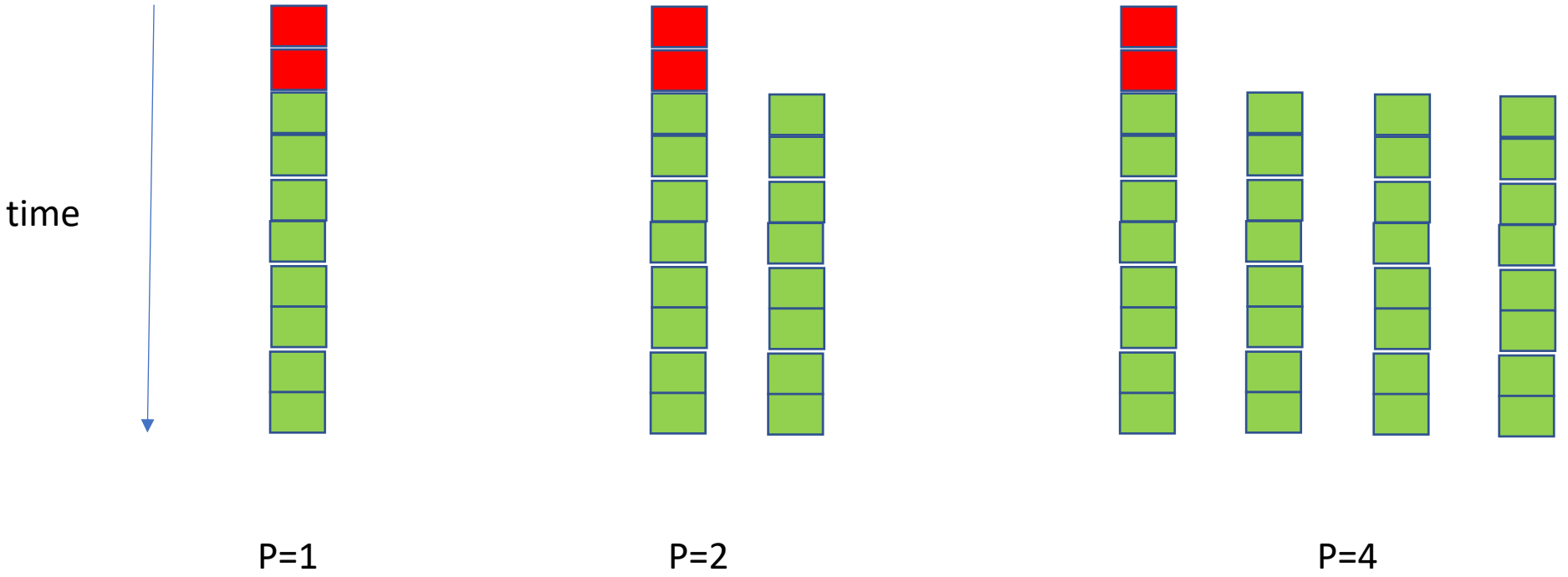
The **Atanasoff–Berry computer** (**ABC**) was the first automatic electronic digital computer (1941) [replica]

Gustafson's Law (scaled speedup, weak scaling)

- Large scale parallel/distributed systems are expected to allow for solving problem faster or larger problems.
  - Amdahl's Law indicates that there is a limit on how faster it can go.
  - How about bigger problems? This is what Gustafson's Law sheds lights on!
- In Amdahl's law, as the number of processors increases, the amount of work in each node decreases (more processors sharing the parallel part).
- In Gustafson's law, as the number of processors increases, the amount of work in each node remains the same (doing more work collectively).

Gustafson’s law

Gustafson’s law: As P increases, the total work on each process remains the same. So the total work increases with P.



Gustafson's Law (scaled speedup, weak scaling)

- The work on each processor is 1 (f is the fraction for sequential program, (1-f) is the fraction for parallel program.
- With P processor (with the same $T_P = 1$), the total amount of useful work is $f + (1 - f)P$. Thus, $T_1 = f + (1 - f)P$.
- Thus, speedup(P) = $f + (1 - f)P$.

No of PEs	Strong scaling speedup (Amdalh's law, f = 10%)	Weak scaling speedup (Gustafson's law, f = 10%)
2	1.82	1.9
4	3.07	3.7
8	4.71	7.3
16	6.40	14.5
100	9.90	90.1

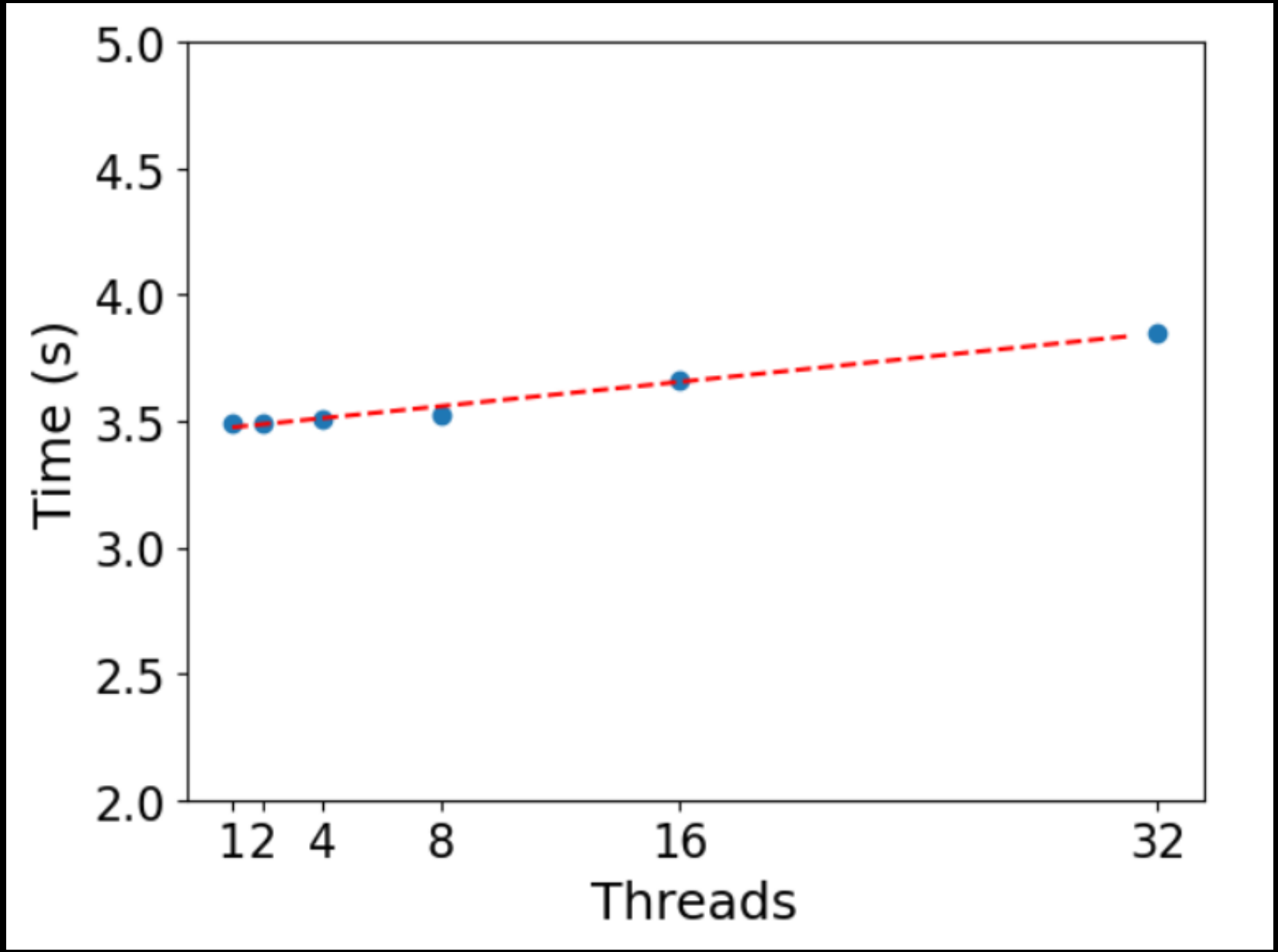
Implication of Gustafson's law

- For weak scaling, speedup(P) = $f + (1 - f)P$
  - Speedup is now proportional to P.
- Scalability is much better when the problem size can increase.
  - Many application can use more computing power to solve larger problems
    - ❖ Weather prediction, large deep learning models.
- *Gustafson, John L. (May 1988). "Reevaluating Amdahl's Law". Communications of the ACM. **31** (5): 532–3.*

We want to perform a parameter sweep over the number of threads

```
cat juliaset_weak.slurm
#!/bin/bash
#SBATCH --job-name weak_julia
#SBATCH --partition general
#SBATCH --ntasks 1
#SBATCH --cpus-per-task 32
#SBATCH --time 5:00
#SBATCH --output=julia_weak_%a.out
#SBATCH --array 1-7
#SBATCH --mail-user mfricke@unm.edu
OUTPUT_PATH=juliaset_weak_$SLURM_ARRAY_TASK_ID.tga
SCALE_FACTOR=$((2 ** ($SLURM_ARRAY_TASK_ID-1)))
export OMP_NUM_THREADS=$SCALE_FACTOR
FRACTAL_HEIGHT=$((10000*$SCALE_FACTOR))
FRACTAL_WIDTH=2000
echo Scale Factor: $SCALE_FACTOR
echo Number of Threads: $OMP_NUM_THREADS
echo FRACTAL_DIMS: $FRACTAL_HEIGHT x $FRACTAL_WIDTH
./juliaset $FRACTAL_HEIGHT $FRACTAL_WIDTH $OUTPUT_PATH
```

Weak Scaling



```
#s=[10000, 20000, 40000, 80000, 160000, 32000]
x=[1,2,4,8,16,32]
y=[3.49008, 3.49439, 3.50964, 3.52632, 3.65801, 3.85159]
y1=3.49008
#y=list(map(lambda z: y1*z, y))
x=[1,2,4,8,16,32]

# Speedup scaled by the number of threads
y=[x[0]*y1/3.49008, x[1]*y1/3.49439, x[2]*y1/3.50964, x[3]*y1/3.52632, x[4]*y1/3.65801, x[5]*y1/3.85159]


# plot input vs output
pyplot.scatter(x, y)

popt, _ = curve_fit(Gustafson_objective, x, y)
# summarize the parameter values
a = popt
print('y = %.5f + %.5f*N' % (a, 1-a))
# define a sequence of inputs between the smallest and largest known inputs
x_line = numpy.arange(min(x), max(x), 1)
# calculate the output for the range
y_line = Gustafson_objective(x_line, a)
# create a line plot for the mapping function
pyplot.plot(x_line, y_line, '--', color='red')

pyplot.xticks(x, size=16)
pyplot.yticks(size=16)
pyplot.xlabel("Threads", size=18)
pyplot.ylabel("Scaled Speedup", size=18)

pyplot.show()
```

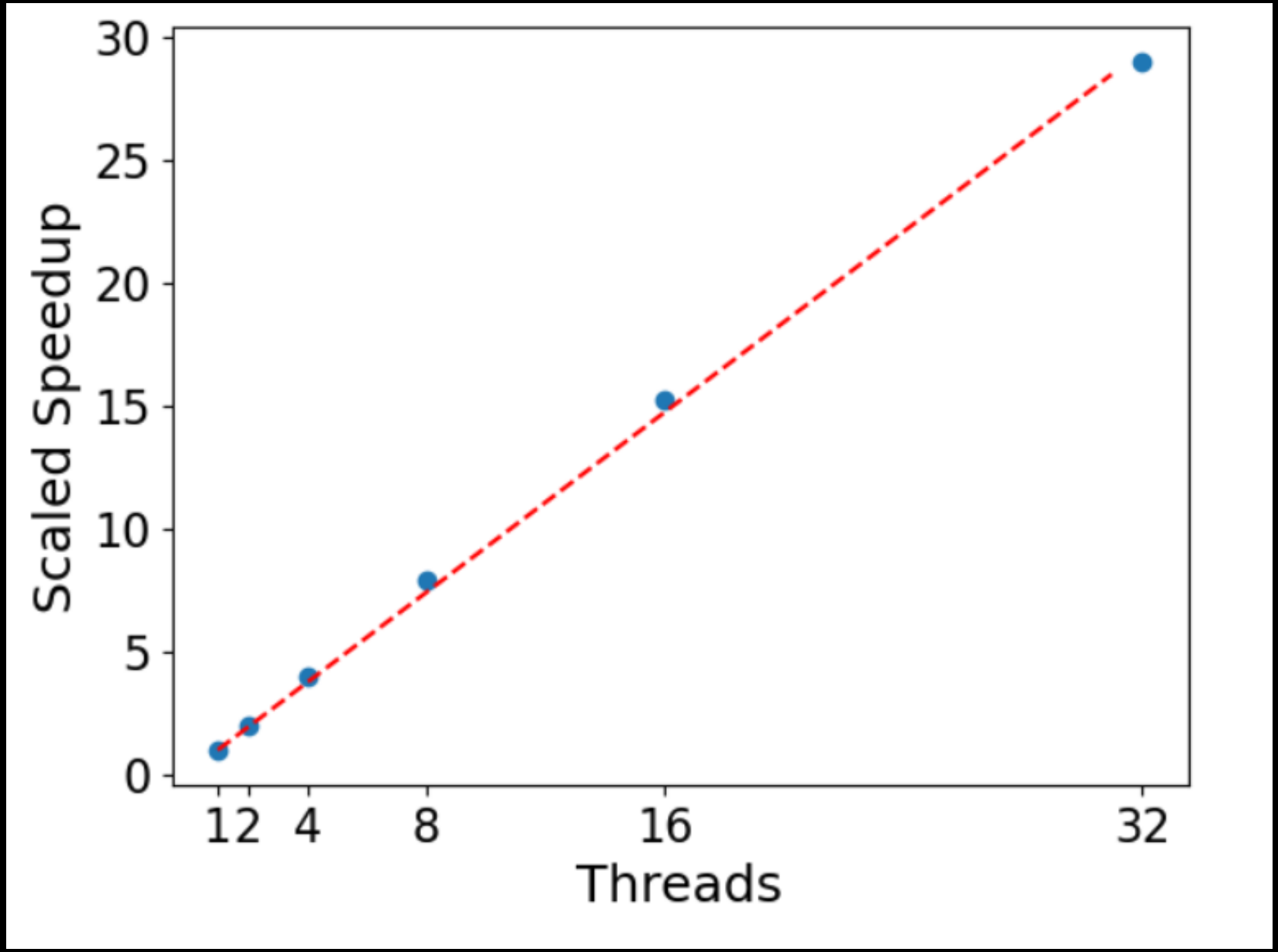
```
# Gustafferson's law
def Gustafson_objective(x, a):
 return a + (1-a)*x
```

Weak Scaling

Scaled Speedup = NT_1/T_N

y = 0.08415 + 0.91585*N)

f is only 0.08



What’s the point?

- These laws allow us to determine how software will benefit, or not, from adding more resources.
- They allow us to detect bottlenecks in our code that could be addressed.
- We can plan what resources to provide instead of just hoping that more will be faster.