High Performance Computing in Julia

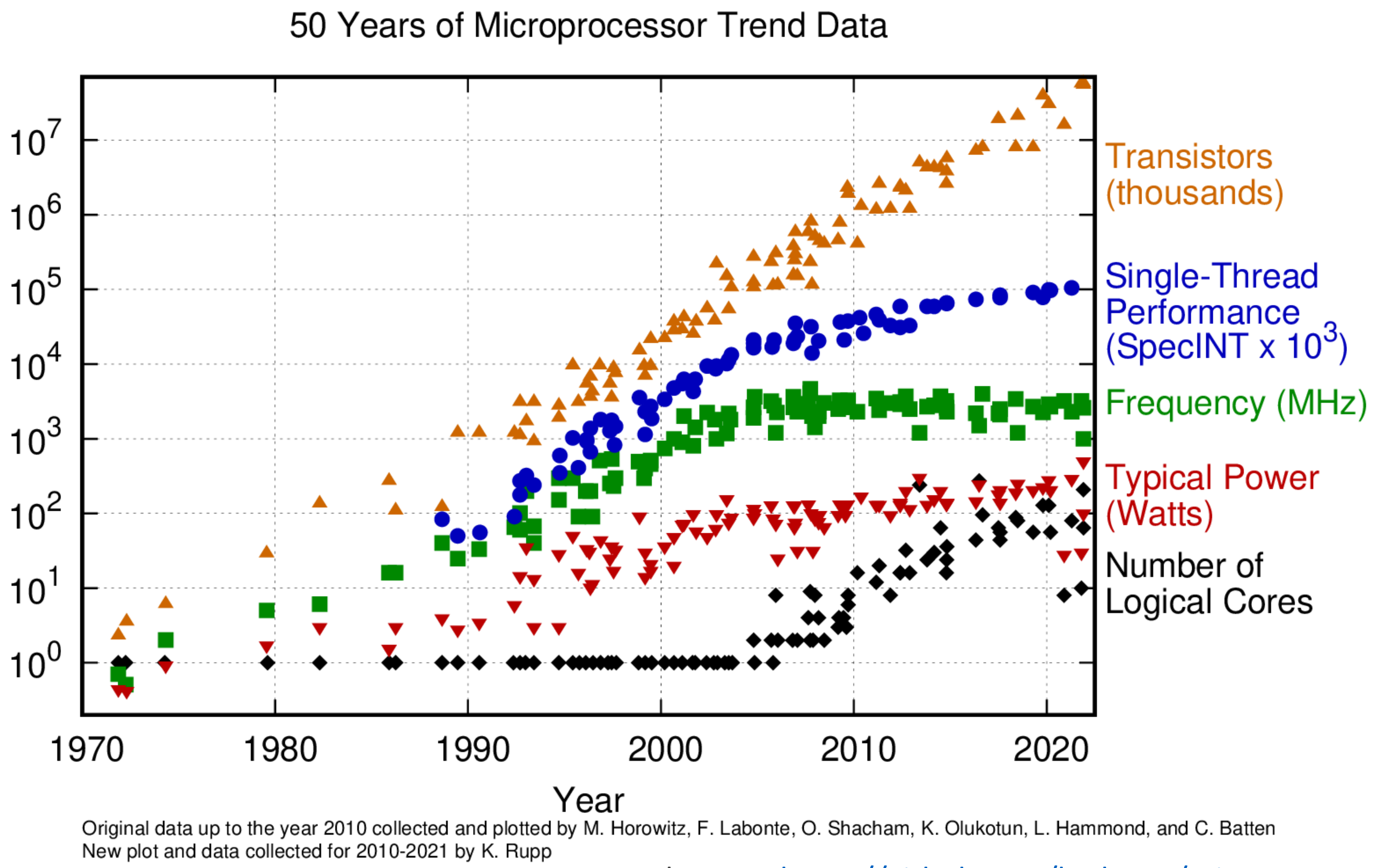
from the ground up.

Introduction to Parallel Programming

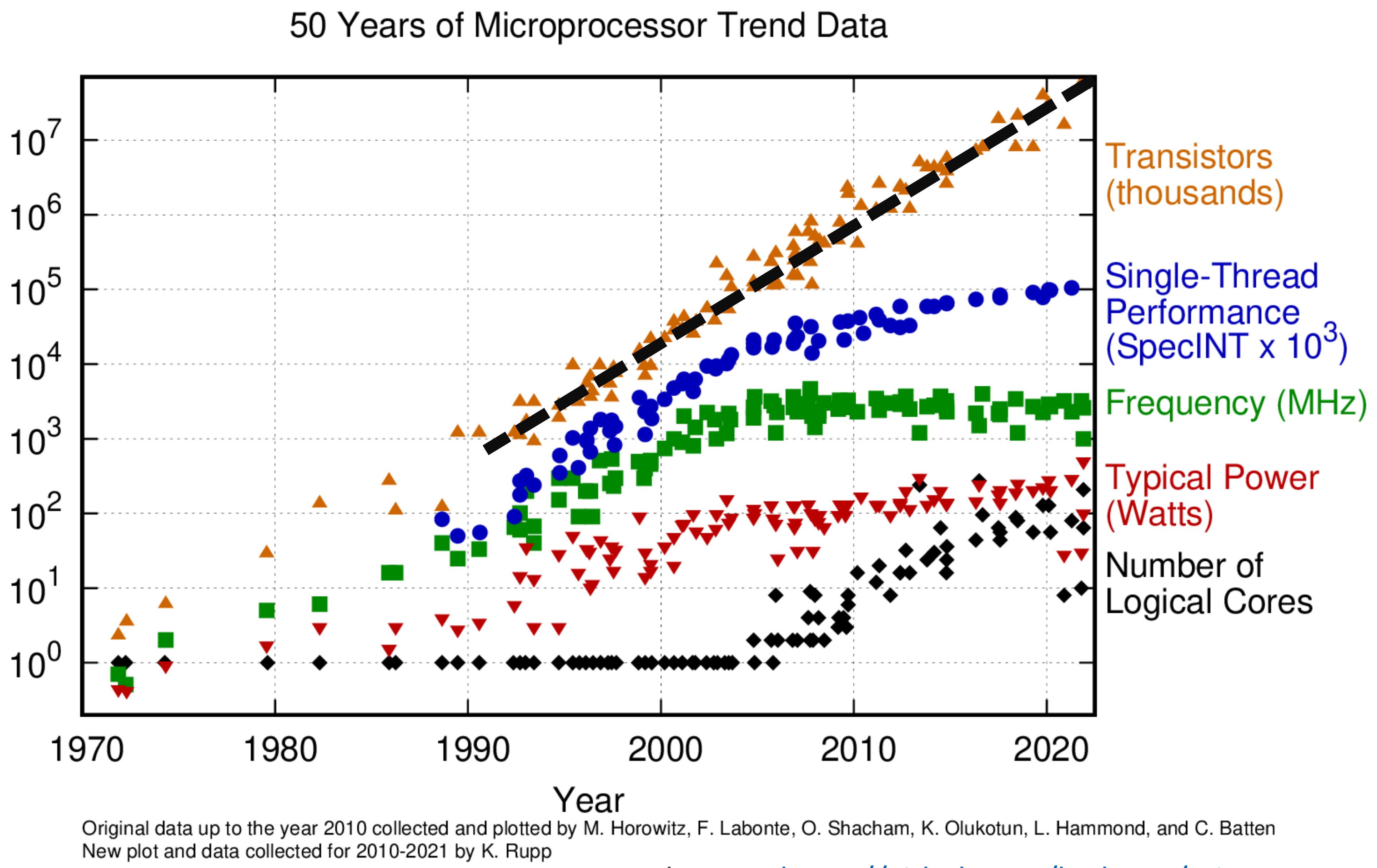
5/10

Microprocessor Trends – Moore’s Law

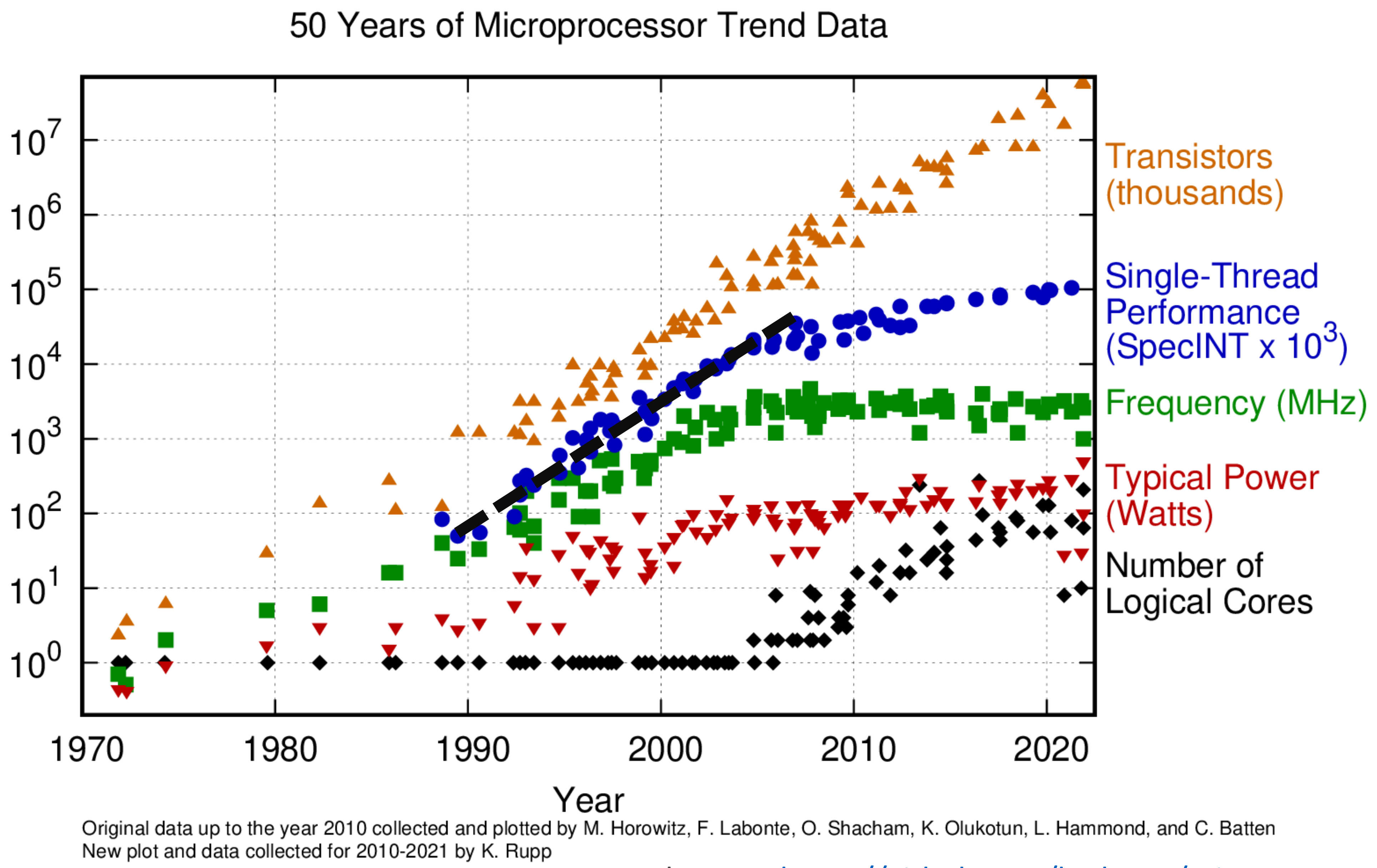
Moore’s Law is the **observation** that the number of transistors in a dense integrated circuit **doubles** almost every **two years**.



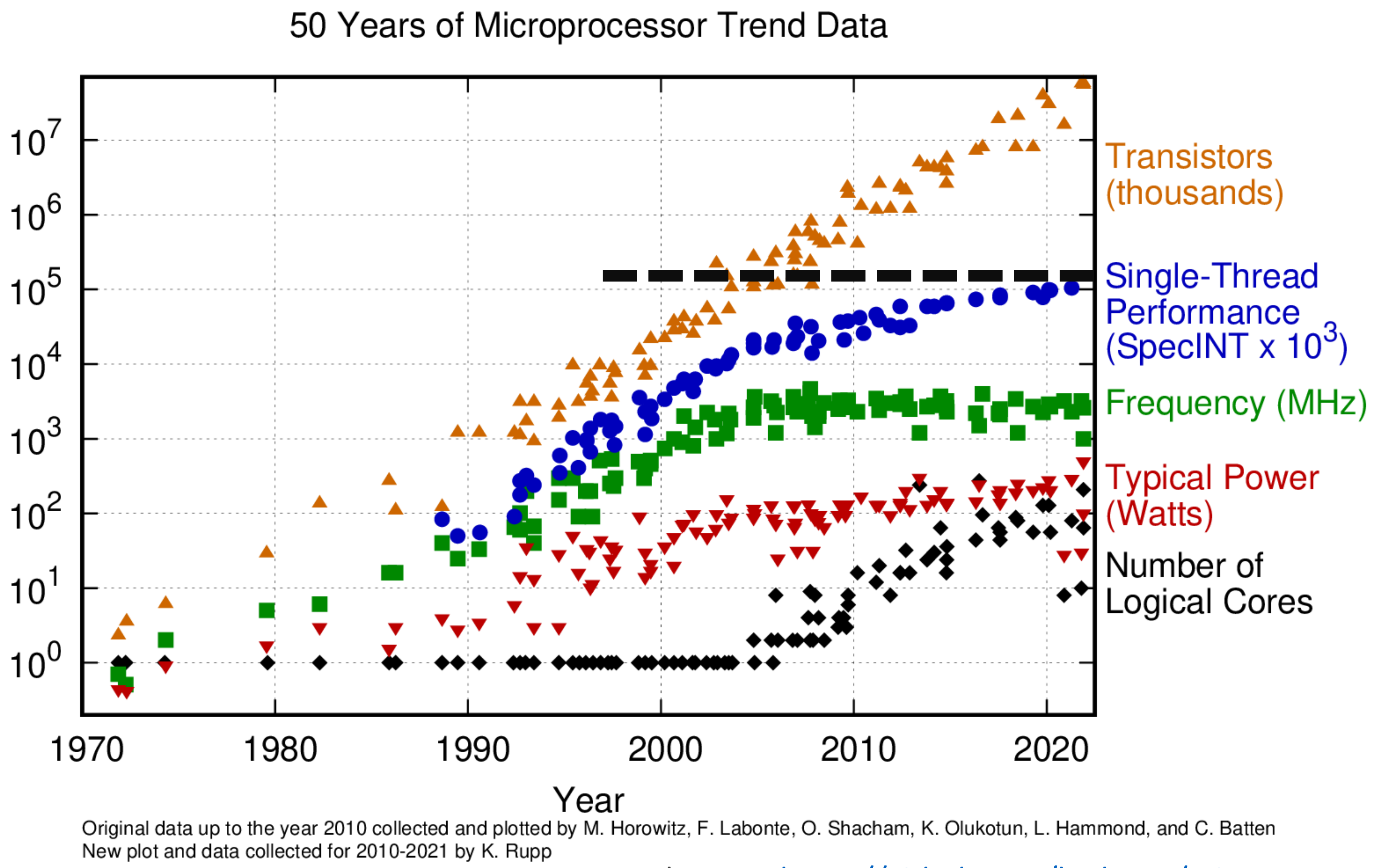
Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data



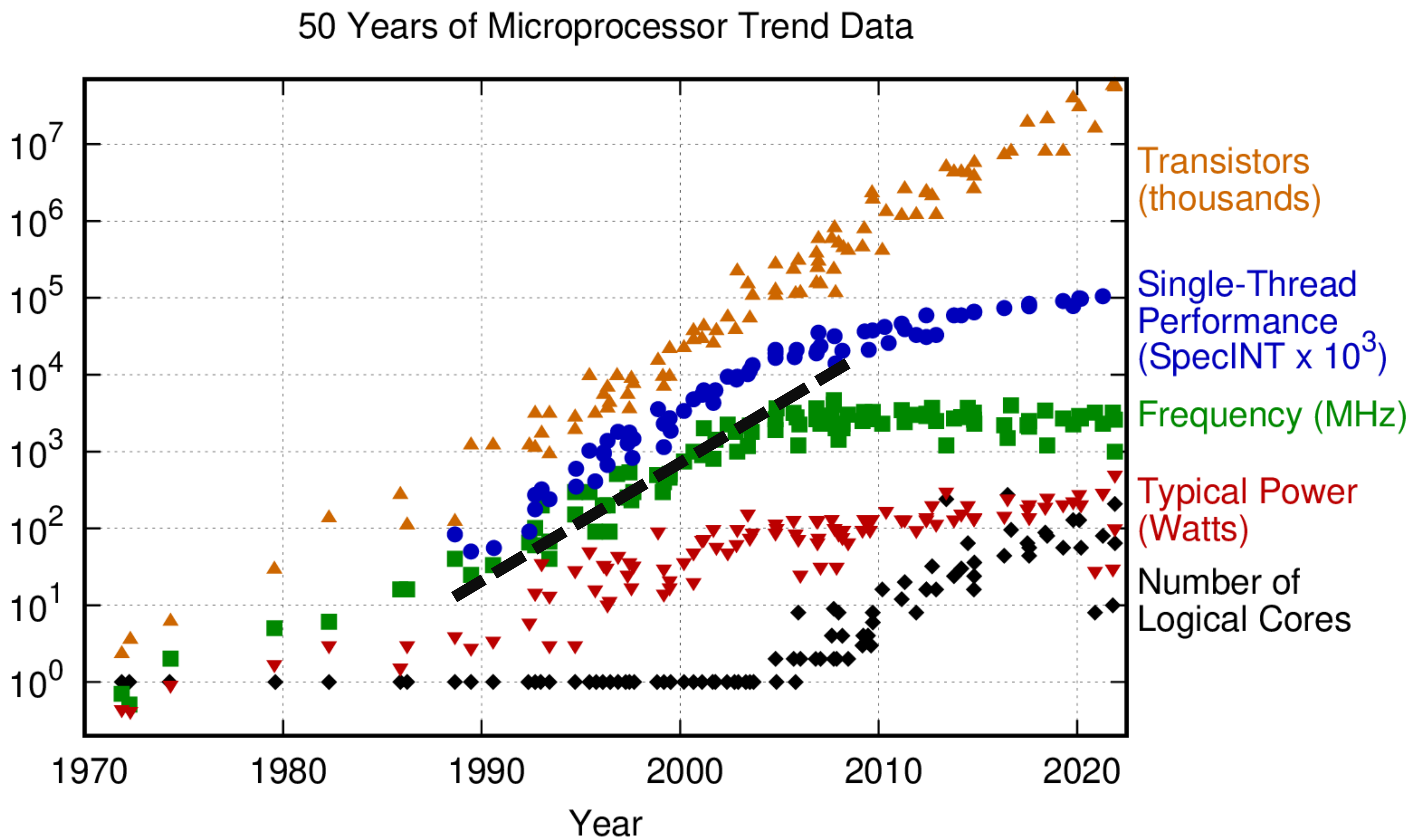
Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data



Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data

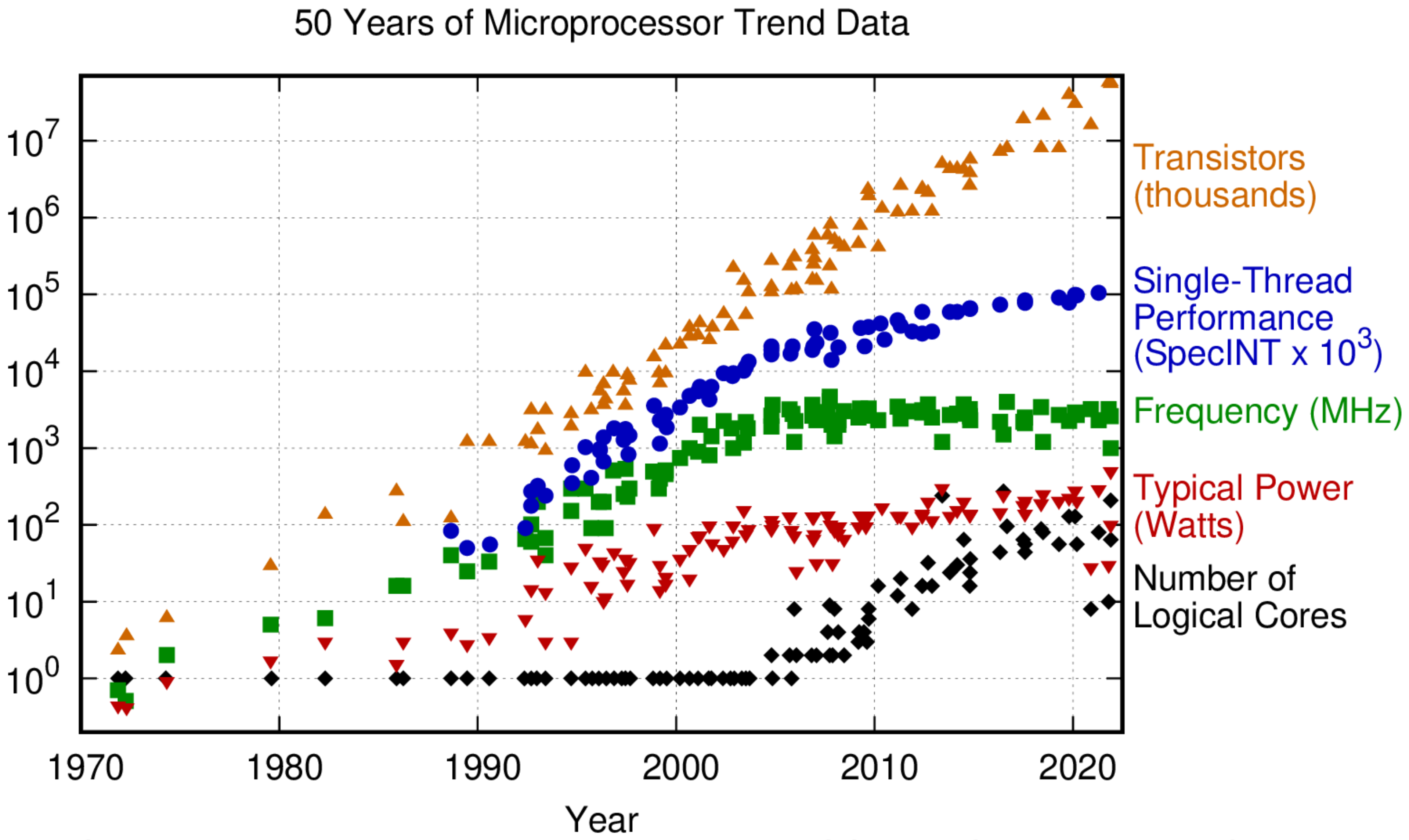


Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data



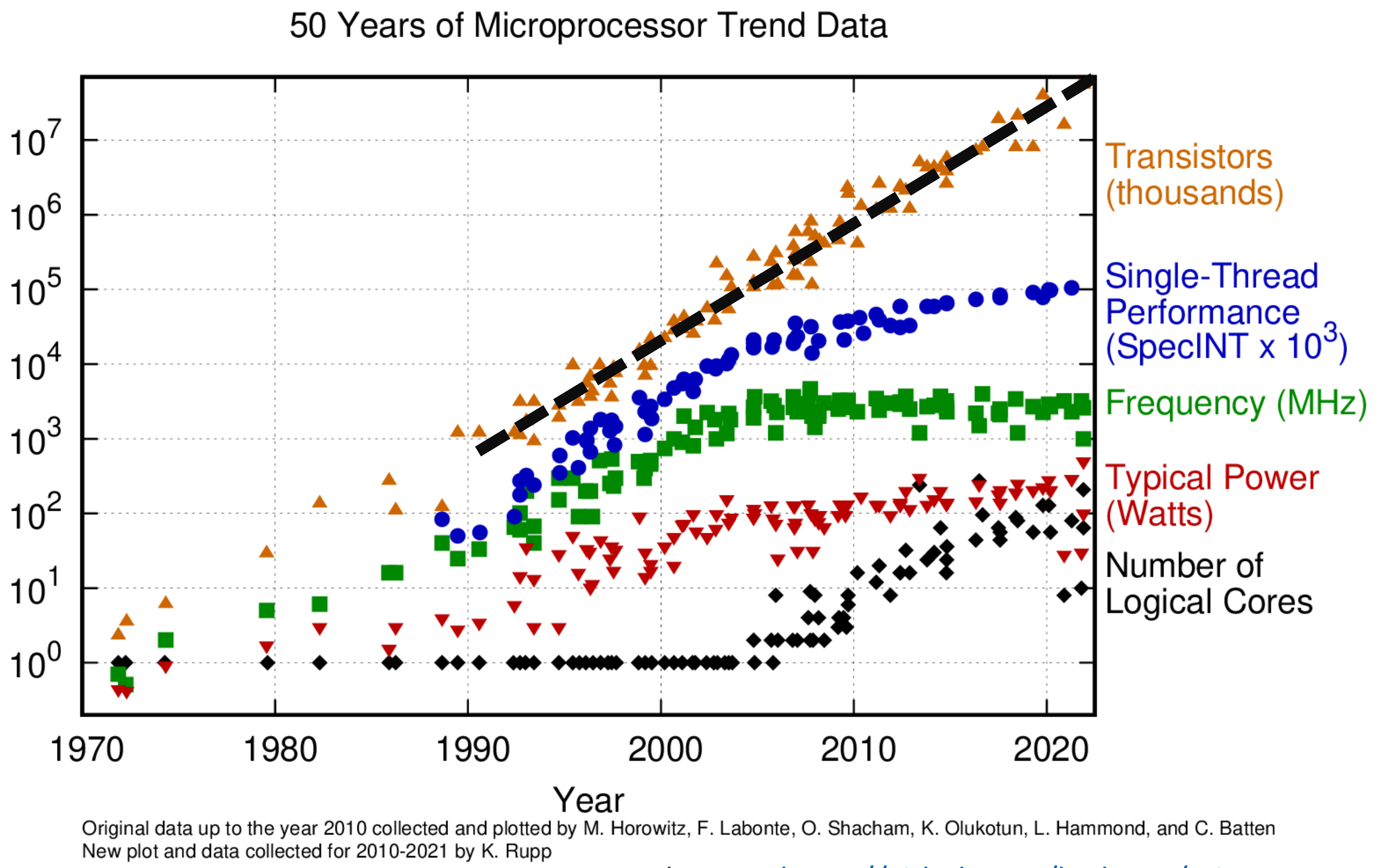
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data

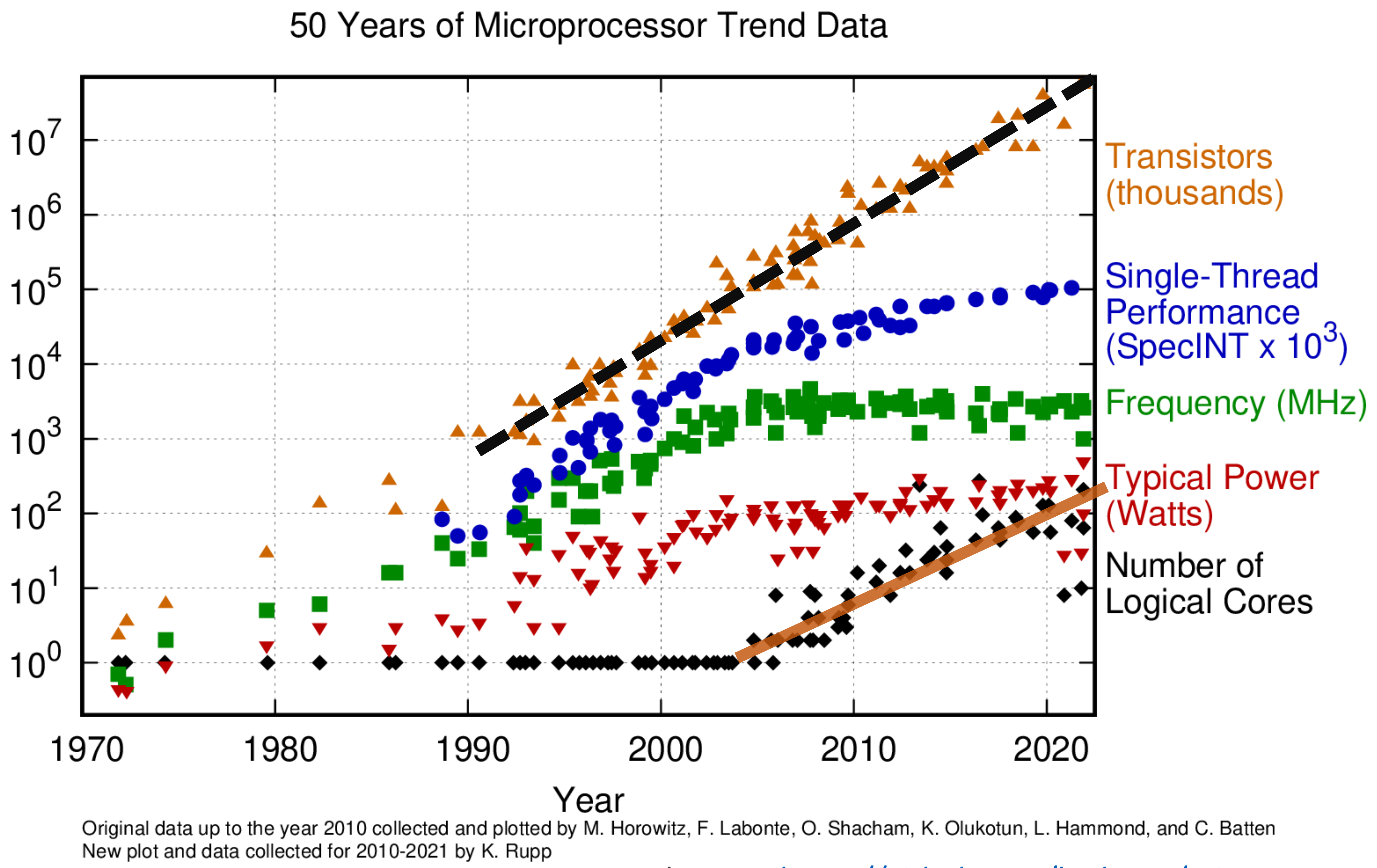


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data

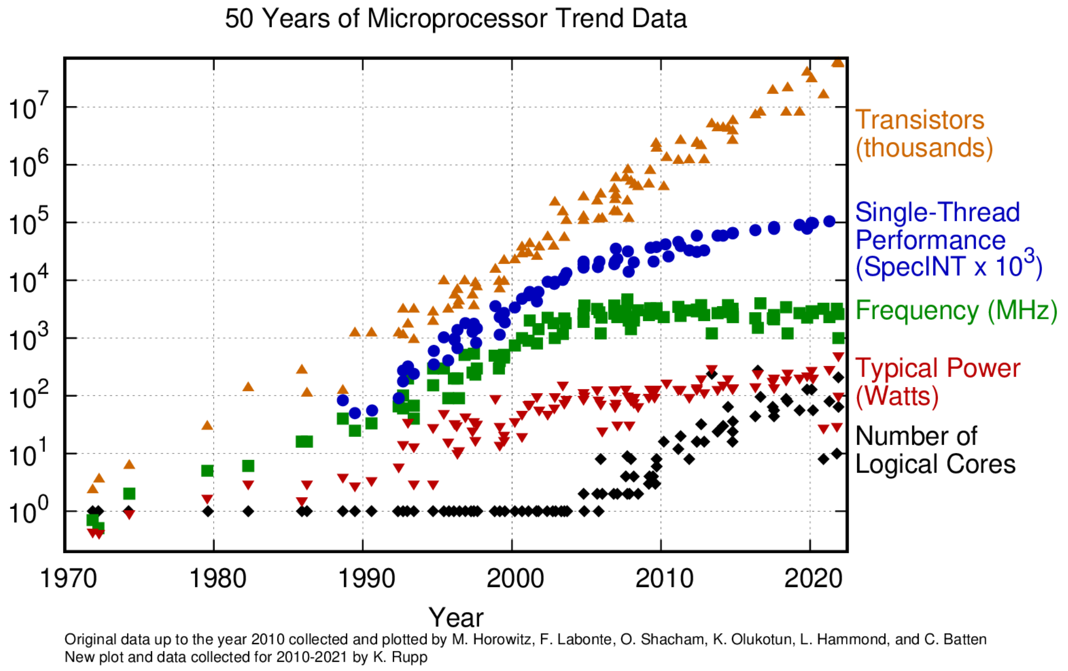


Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data



Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data

Why do we need parallel programming?



- Processors are **not** getting **faster**
- Number of physical processors available is **growing**
- We need to adapt our code to work with multiple processors

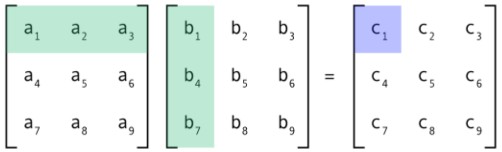
Karl Rupp - https://github.com/karlrupp/microprocessor-trend-data

What tasks benefit from parallelisation?

What tasks benefit from parallelisation?

No Speedup

Massive Speedup

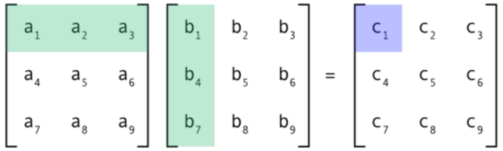


Matrix Multiplication

What tasks benefit from parallelisation?

No Speedup

Massive Speedup

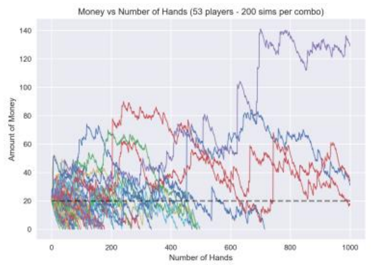


Matrix Multiplication

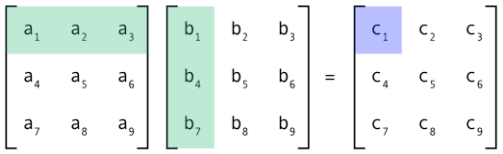
What tasks benefit from parallelisation?

No Speedup

Massive Speedup

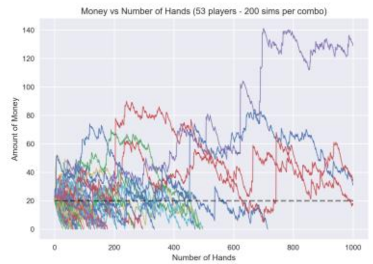


Monte-Carlo Simulation



Matrix Multiplication

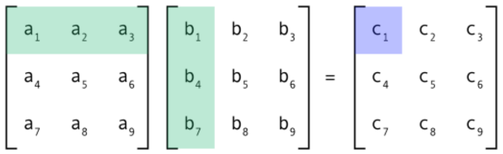
What tasks benefit from parallelisation?



Monte-Carlo Simulation

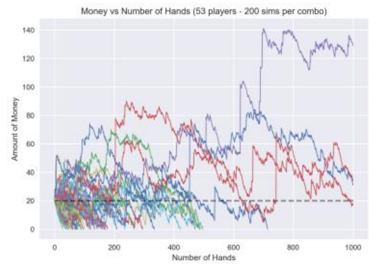
No Speedup

Massive Speedup



Matrix Multiplication

What tasks benefit from parallelisation?



Monte-Carlo Simulation

No Speedup

Massive Speedup

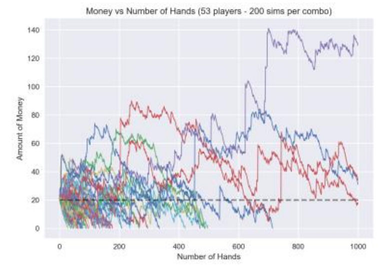
$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Matrix Multiplication

$$\frac{d^2x}{dt^2} = \frac{dv}{dt} = {F(x)}/{m}$$

Numerical Differential Equations

What tasks benefit from parallelisation?



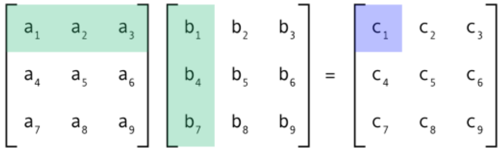
Monte-Carlo Simulation

$$\frac{d^2x}{dt^2} = \frac{dv}{dt} = F(x)/_m$$

Numerical Differential Equations

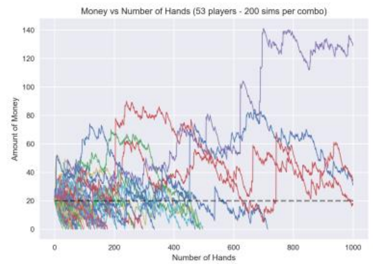
No Speedup

Massive Speedup



Matrix Multiplication

What tasks benefit from parallelisation?



Monte-Carlo Simulation

$$\frac{d^2x}{dt^2} = \frac{dv}{dt} = F(x)/_m$$

Numerical Differential Equations

No Speedup

Massive Speedup

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

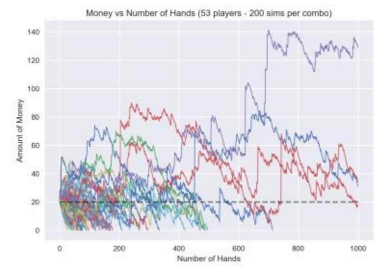
Matrix Multiplication

Game Simulation

What tasks benefit from parallelisation?

$$\frac{d^2x}{dt^2} = \frac{dv}{dt} = F(x)/_m$$

Numerical Differential Equations



Monte-Carlo Simulation

No Speedup

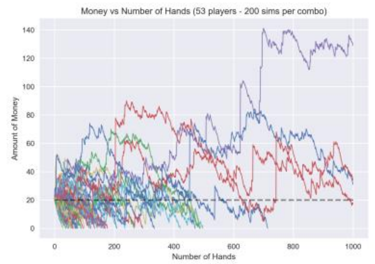
Massive Speedup

Game Simulation

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Matrix Multiplication

What tasks benefit from parallelisation?



Monte-Carlo Simulation

$$\frac{d^2x}{dt^2} = \frac{dv}{dt} = {F(x)}/{m}$$

Numerical Differential Equations

No Speedup

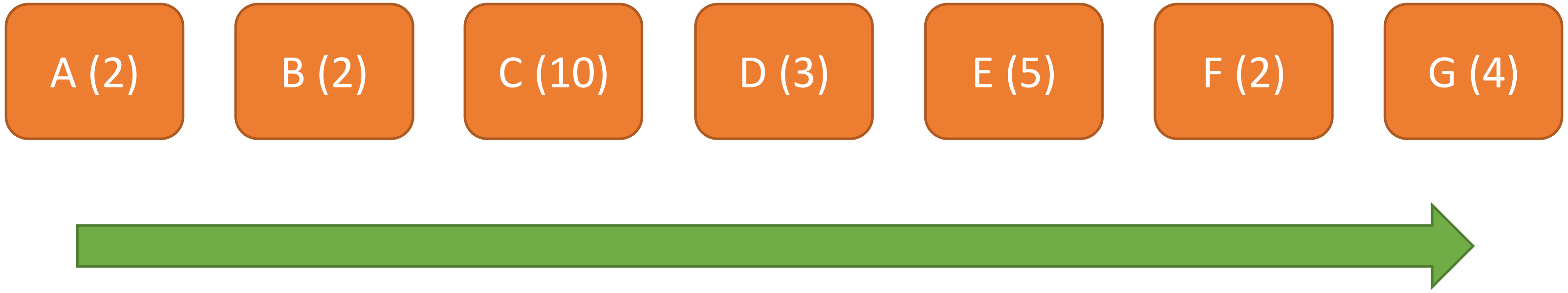
Massive Speedup

Game Simulation

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

Matrix Multiplication

Task Dependency Graphs



Tasks Processed Sequentially
(Total Time: 28)

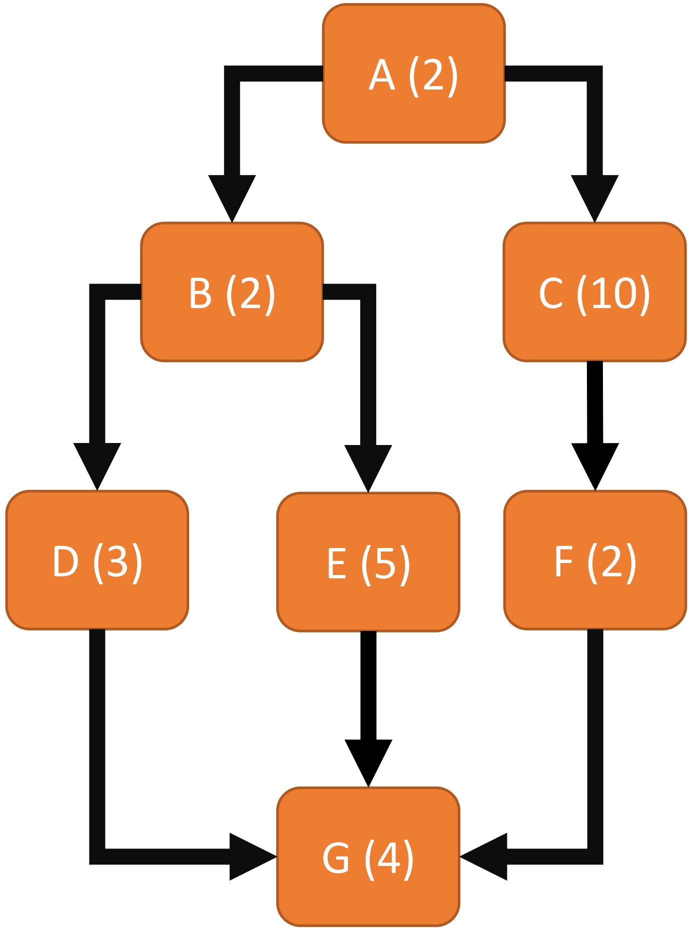
Task Dependency Graphs

A (2) B (2) C (10) D (3) E (5) F (2) G (4)

1. **B** and **C** depend on **A**
2. **D** and **E** depend on **B**
3. **F** depends on **C**
4. **G** depends on **D**, **E** and **F**

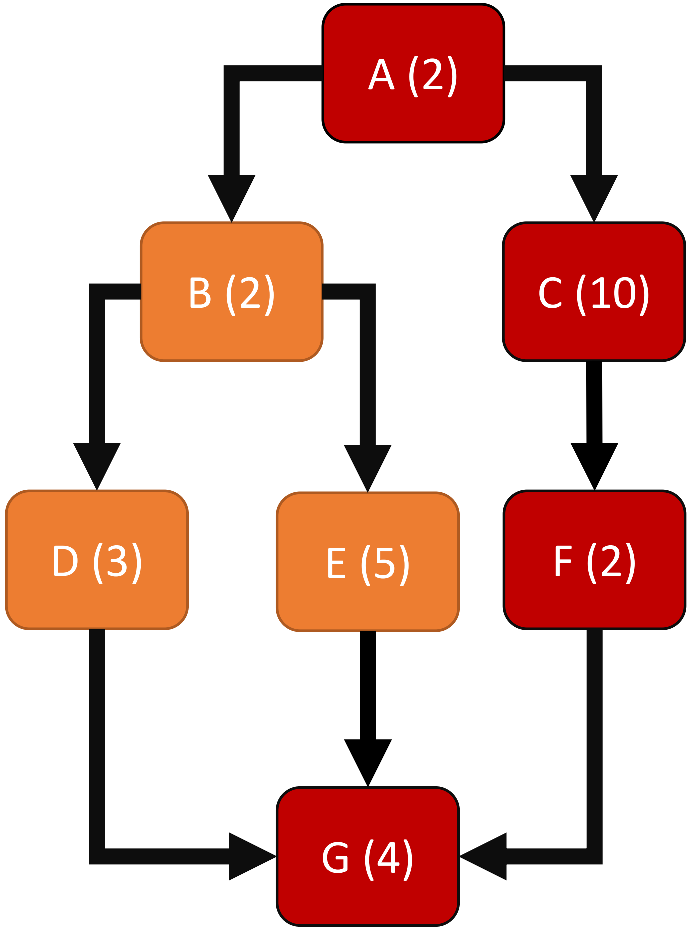
Task Dependency Graphs

1. **B** and **C** depend on **A**
2. **D** and **E** depend on **B**
3. **F** depends on **C**
4. **G** depends on **D**, **E** and **F**



Task Dependency Graphs

1. **B** and **C** depend on **A**
2. **D** and **E** depend on **B**
3. **F** depends on **C**
4. **G** depends on **D**, **E** and **F**



Critical Path
(Minimum Time: 18)

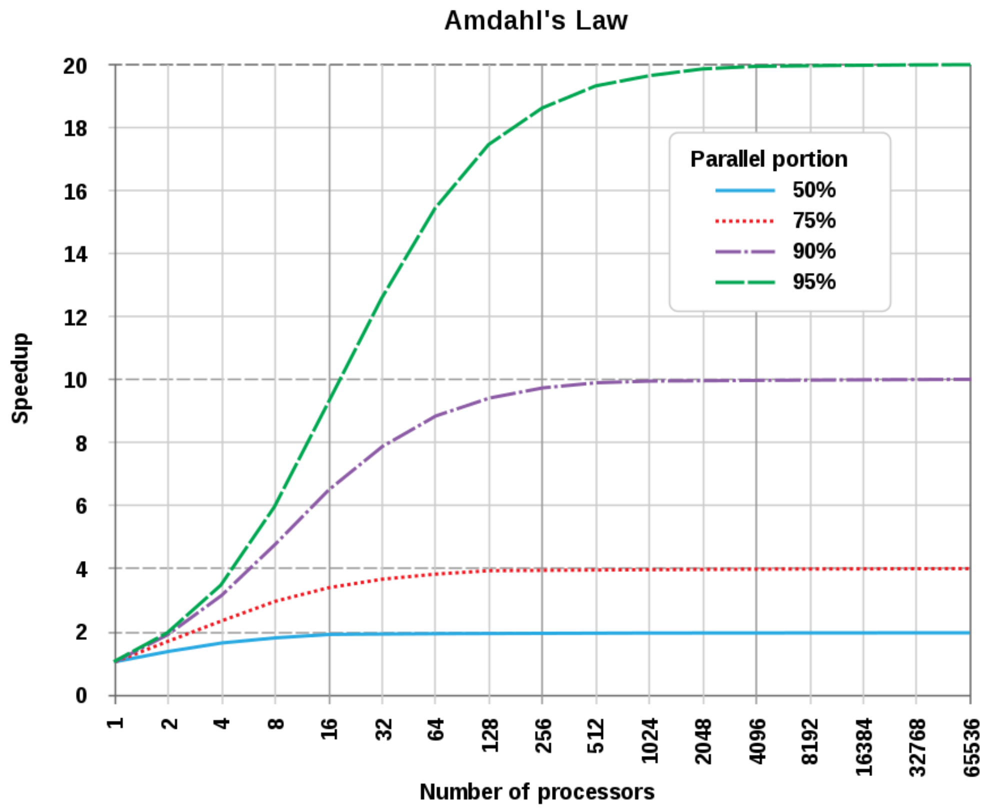
Maximum Speedup:
$\sim 1.55x$

Amdahl's Law

We can estimate the speedup of a **fixed-workload** task using Amdahl's Law, which states:

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

- S is the speedup of the task
- s is the speedup of the parallel portion
- p is the proportion of time that benefits from the improved resources of s.



Gustafson’s Law

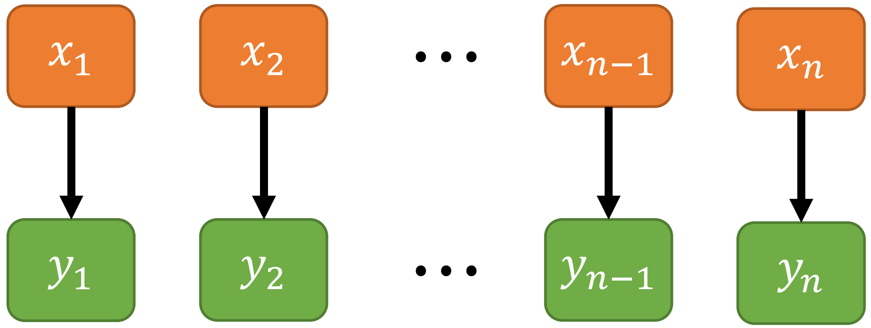
- Amdahl’s Law is only concerned with a **fixed problem size**
- Gustafson realised that we tend to **increase the problem size** when given more resources.
- Take a parallelised algorithm that takes T_p units of time on a parallel computer with N processors
- We know that some fraction $(1 - f)$ is dedicated to serial processing
- If executing on a serial computer, the total time would be
$$T_s = (1 - f)T_p + fNT_p$$
- The speedup of using the parallel computer is
$$S = \frac{T_S}{T_p} = (1 - f) + fN$$
- The efficiency of the speedup $e = {}^{S}/_{N} = {}^{(1-f)}/_{N} + f$

Practical Considerations

- Scheduling & synchronising tasks across multiple workers introduces some additional latency
- We call this **overhead**, which means that the problem sizes must be large enough to overcome this overhead.
- Amdahl and Gustafson do not take this communication cost into account
- The theoretical speedup is only worth using as a guideline, all implementations should be **benchmarked**

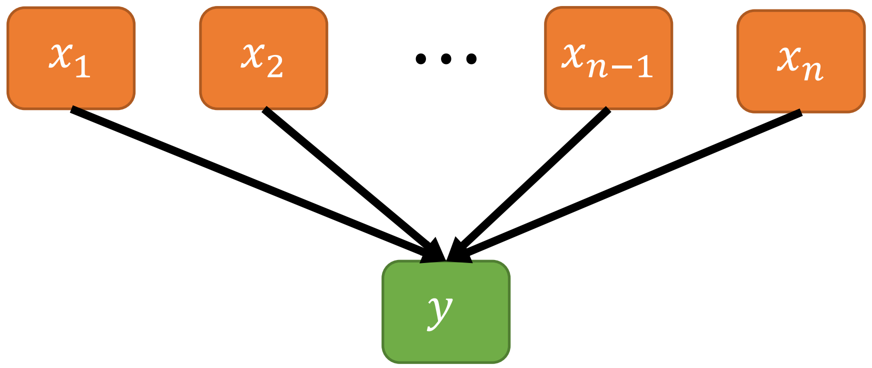
Dependency Graph: Map

- Each task is **independent** of one another, e.g:
$$y_i = f(x_i)$$
- All elements can be processed in **parallel**
- This is also known as an **embarrassingly parallel problem**
- Execution order is arbitrary



Dependency Graph: Reduce

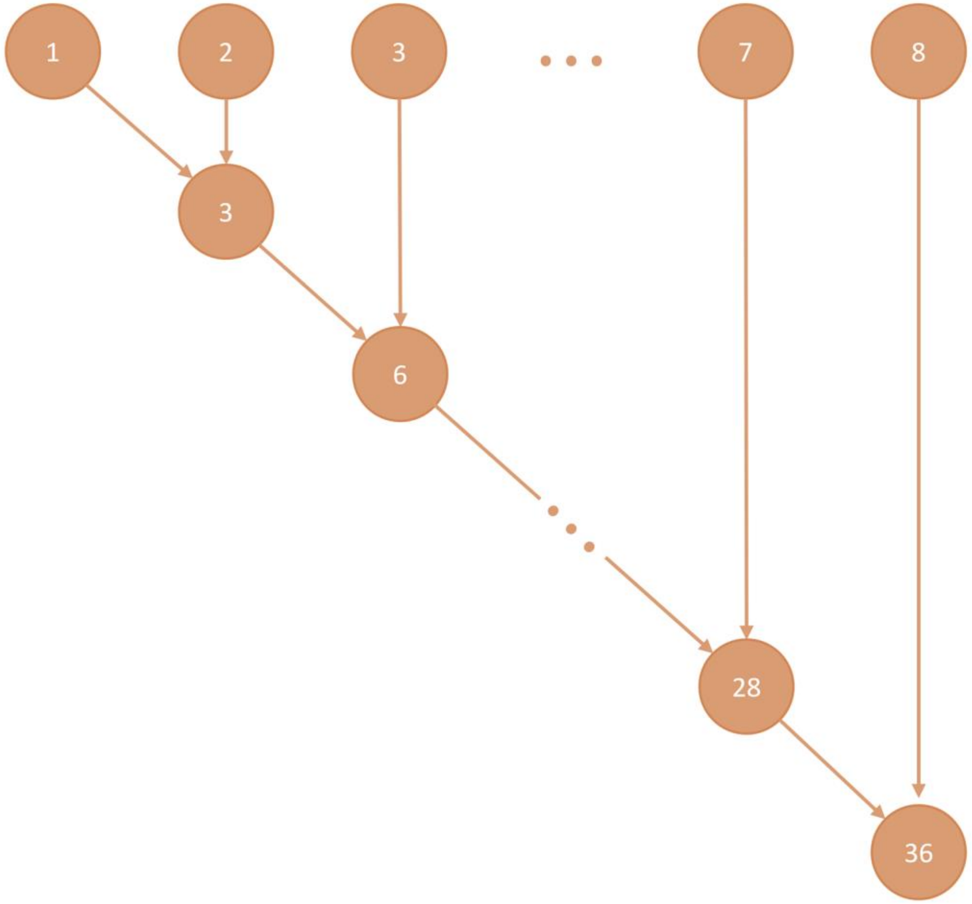
- Elements are **reduced** to a single value by some operator
- Usually deal with **binary operators** (those which take two arguments), e.g. $+, -$ etc
- Can parallelise by breaking up the reduction into stages and using **associativity** and **commutativity** of the operator



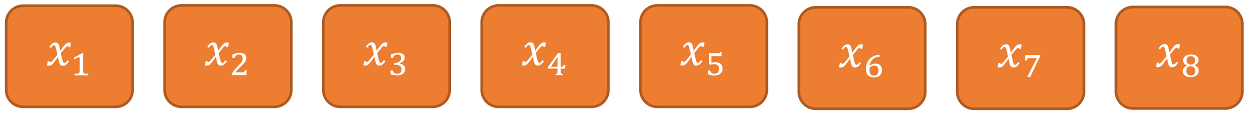
Summation

```
s = 0
for i in 1:8
    s += i
end
```

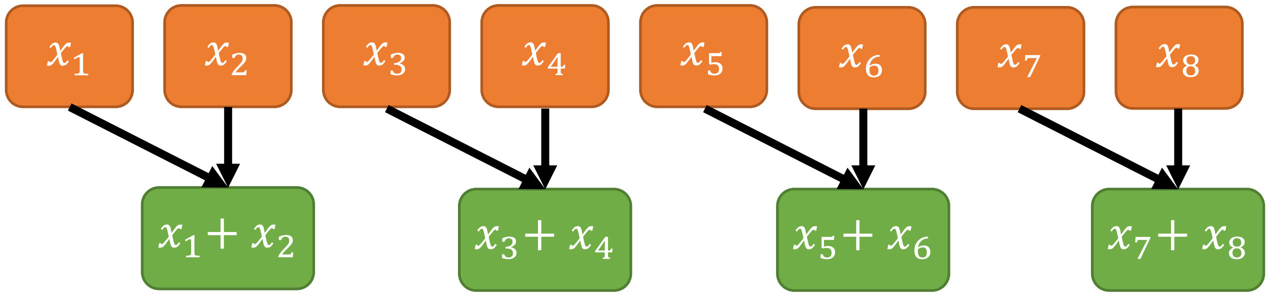
- Use of the single variable causes a long critical path.
- The next addition requires the previous addition to finish
- But addition is **associative**



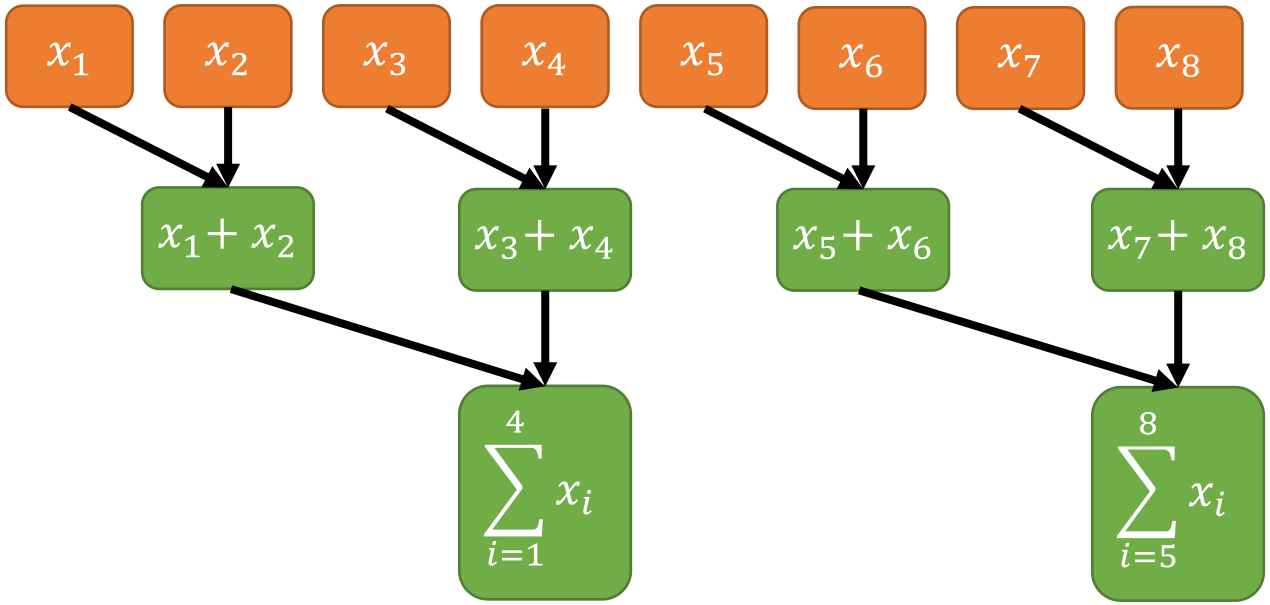
Parallel Summation



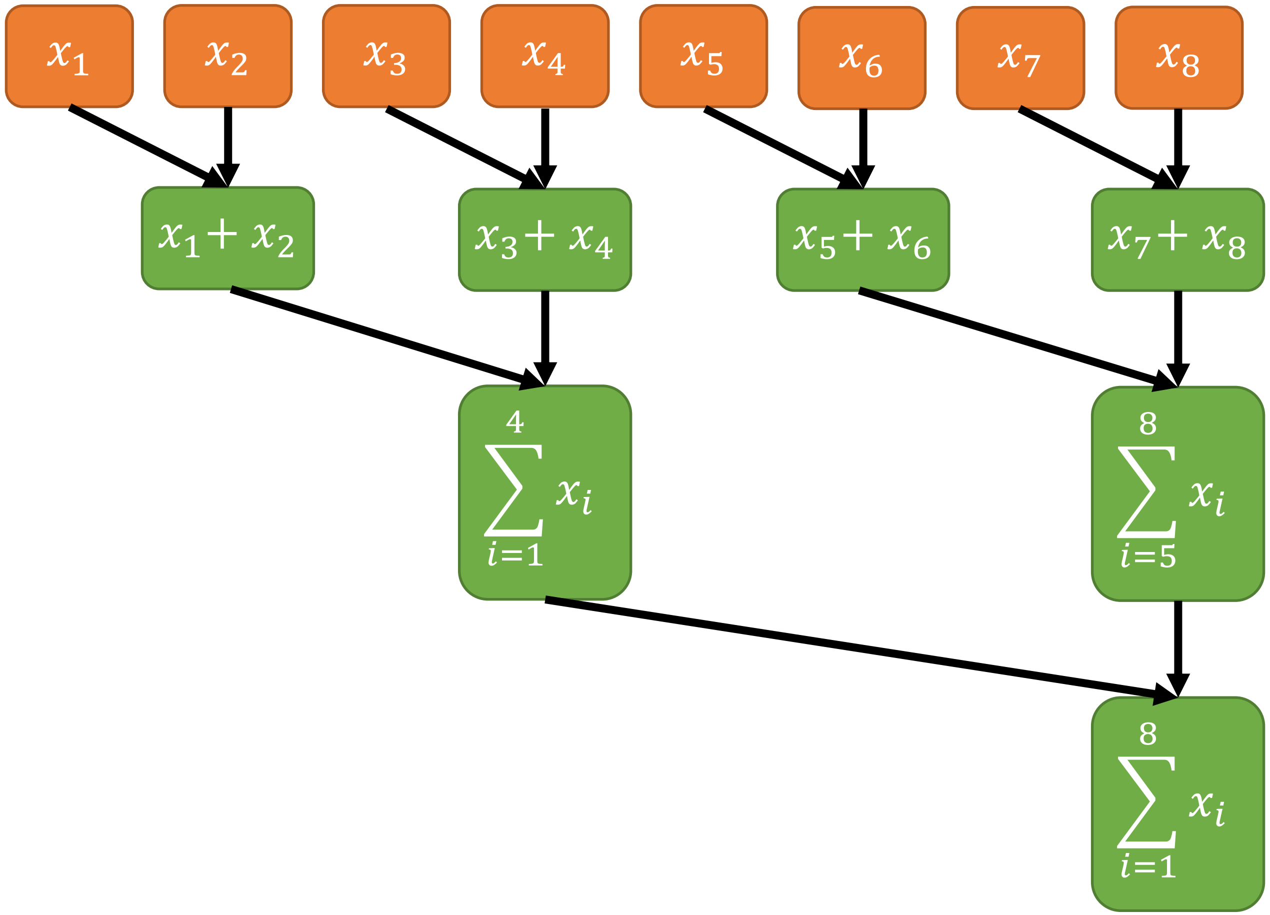
Parallel Summation



Parallel Summation



Parallel Summation



Parallel Mechanisms

Map

- Easy to parallelise as each operation is independent of the last
- Operations can be done in any order
- May require some **load balancing**
- Scheduling the work introduces some **overhead**

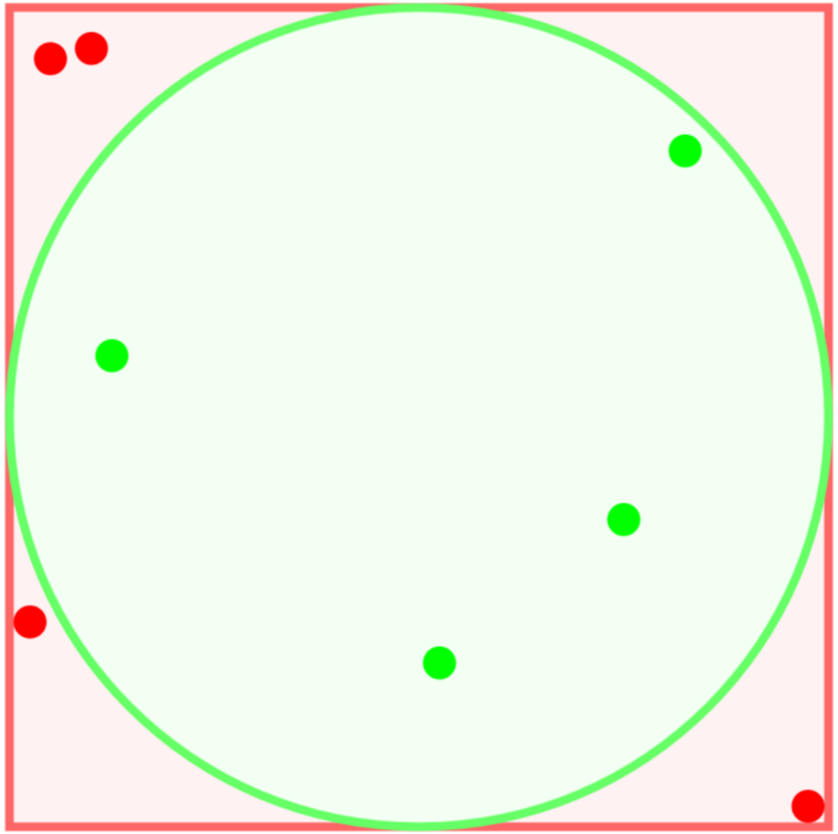
Reduction

- Order of operations depends on **associativity** of the operator
- Often requires additional memory to store intermediate results
- Scheduling introduces overhead

Monte-Carlo Simulation (Estimating π)

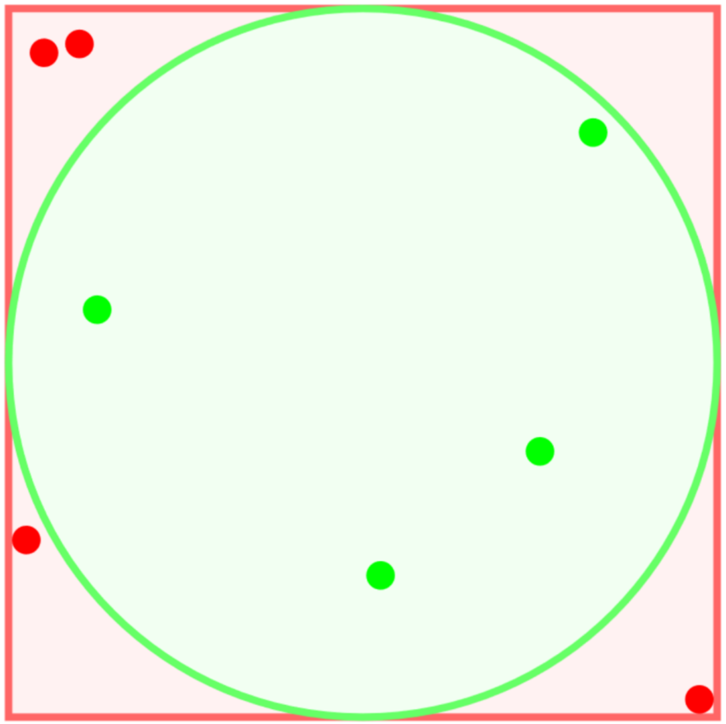
- Throw darts uniformly randomly in a square box, with a circular board.
- If dart goes inside the green it is a **hit**, otherwise a **miss**
- Estimate π via

$$\pi \approx 4\frac{n_c}{n}$$

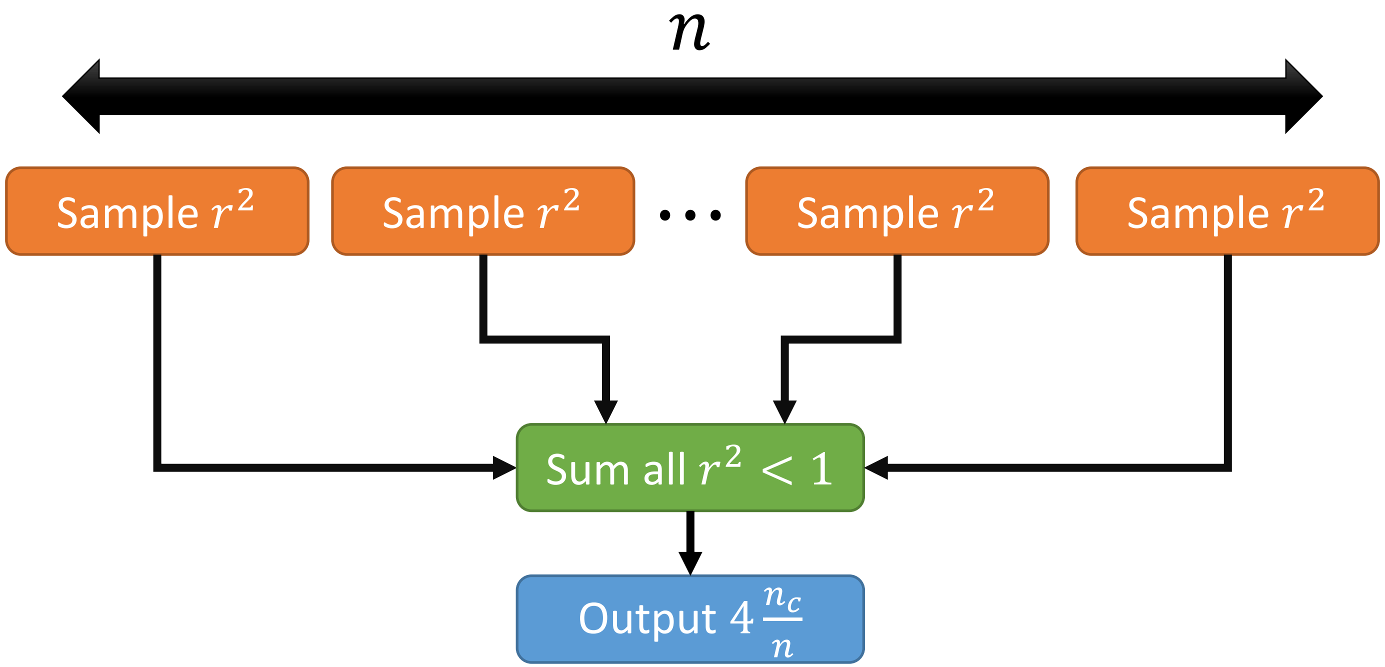


Monte-Carlo Simulation (Estimating π)

```julia
function est_pi_mc_serial(n)
    n_c = zero(typeof(n))
    for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_c += 1
        end
    end
    return 4 * n_c / n
end
```



Monte-Carlo Simulation (Estimating π)



Live Demonstration

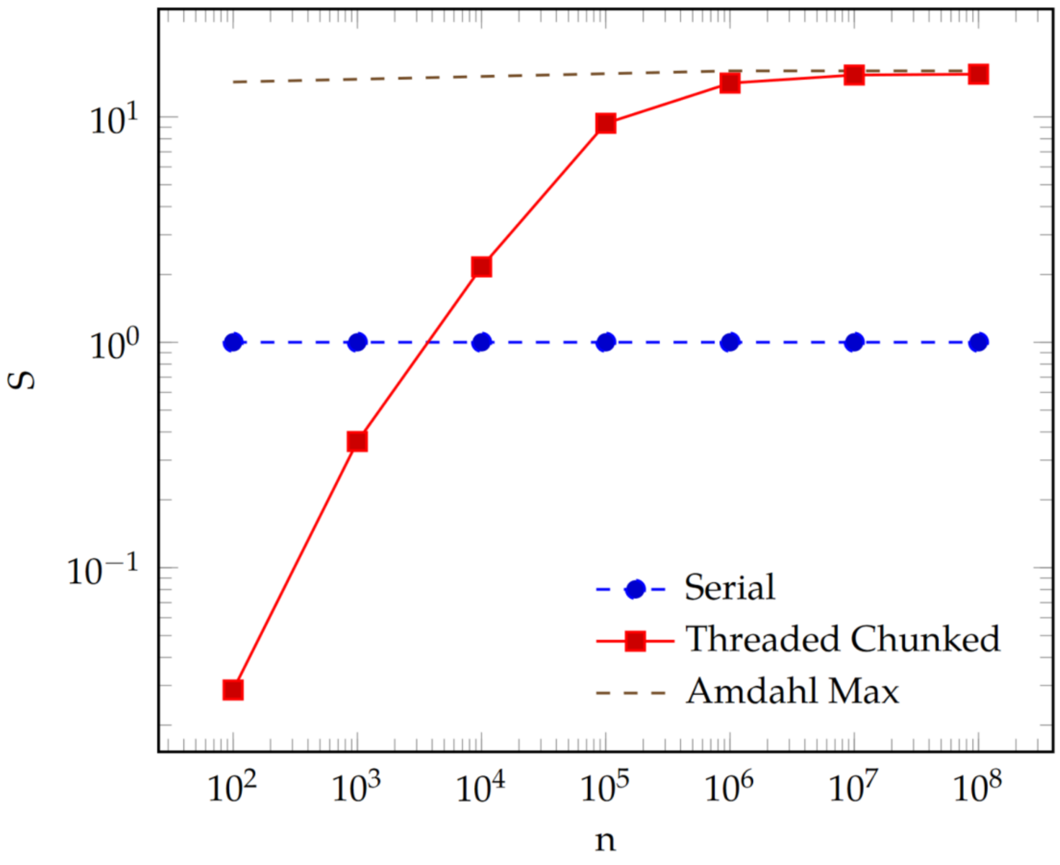
Monte-Carlo Simulation (Estimating π)

```
function est_pi_mc_threaded(n)
    n_c = zero(typeof(n))
    Threads.@threads for _ in 1:n
        # Choose random numbers between -1 and +1 for x and y
        x = rand() * 2 - 1
        y = rand() * 2 - 1
        # Work out the distance from origin using Pythagoras
        r2 = x*x+y*y
        # Count point if it is inside the circle (r^2=1)
        if r2 <= 1
            n_c += 1
        end
    end

    return 4 * n_c / n
end
```

- Can use `@threads` to perform each element of the loop in **parallel** (using multiple cores)
- However, this introduces a **bug**, which makes the result under-estimate π
- This type of bug is called a **race condition**.

Monte-Carlo Simulation (Estimating π)



Workshop – Thursday 02/02/2023

Assignment: Multithreading

Released Wednesday 01/02/2023

Bring your laptops!