Performance Analysis

Department of Computer Science,

National University of Computer & Emerging Sciences,

Islamabad Campus

Performance?

- To **measure improvement** in **computer architectures**, it is **necessary** to **compare alternative designs**

- A **better system** has **better performance**, but ***what exactly is the performance?***

Performance?

Performance?

Model 3 Performance		Model Y Performance
 185" L x 73" W x 57" H		 187" L x 76" W x 64" H
3.2 s	0-60 MPH	3.5 s
162 mph	Top Speed	155 mph
299 mi	Range	280 mi
75 kWh	Battery	75 kWh
450 hp	Power	450 hp
15 cu ft	Cargo Space	66 cu ft
5	Seats	5 7 seat option available
250 kW max	Charging	250 kW max
$56,990	MSRP	$60,990

Performance Metrics – Sequential Systems

Performance?

- For the **computer systems** and **programs**:
  - one **main performance metric** is **Time**
  - Or just **wall-clock time**

Performance?

The **execution time** of a **program A** can be **split into:**

- **User CPU time:** **capturing** the **time** that the **CPU spends** for **executing A**
- **System CPU time**: **capturing** the **time** that the **CPU spends** for the **execution** of **routines** of the **operating system** **issued by A**
- **Waiting time**: **caused** by **waiting** for the **completion of I/O operations** and by the **execution** of **other programs** because of **time sharing**

Here we concentrate on **user CPU time**

Computer Performance

- **Measuring Computer Performance**
  - **Clock Speed**
  - **MIPS**
  - **FLOPS**
  - **Benchmark Tests**

- **Factors affecting Computer Performance**
  - Processor Speed
  - Data Bus width
  - Amount of cache
  - Faster interfaces
  - Amount of main memory

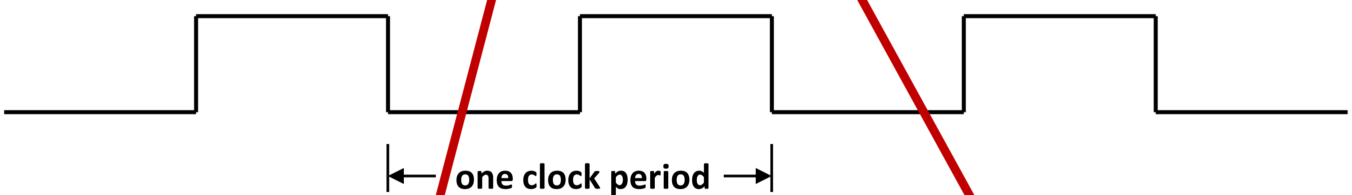
Measuring Performance

- Every **processor** has a **clock** which **ticks continuously** at a **regular rate**
- **Clock synchronises** all the **components**
- **Cycle time measured** in **GHz**
- **200 MHz (megahertz)** means the **clock ticks 200,000,000 times a second** (***Pentium1 -1995***)

Machine Clock Rate

- **Clock Rate (CR)** in **MHz**, **GHz**, etc. is **inverse of Clock Cycle (CC) time** (<u>clock period</u>)

$$CC = 1 / CR$$



10 nsec clock cycle → 100 MHz clock rate

5 nsec clock cycle → 200 MHz clock rate

2 nsec clock cycle → 500 MHz clock rate

1 nsec clock cycle → 1 GHz clock rate

500 psec clock cycle → 2 GHz clock rate

250 psec clock cycle → 4 GHz clock rate

200 psec clock cycle → 5 GHz clock rate

Measuring Performance

- **Clock Speed**
  - Generally the **faster** the **clock speed** the **faster the processor** – **3.2 GHz** is faster than **1.2 GHz**

- **MIPS – Millions of Instructions per Second**
  - **Better comparison**
  - But **beware of false claims:**
    - Such as, **only using the simplest & fastest instructions** and **different processor families** (having different **ISA**).

- **Flops – Floating Point Operations per sec.**
  - **Best measure** as **FP operations are the same in every processor** and **provide best basis**

Units of High Performance Computing

Basic Unit

Kilo
Mega
Giga
Tera
Peta
Exa
Zeta

Speed

1 Kflop/s	10^3 Flop/second
1 Mflop/s	10^6 Flop/second
1 Gflop/s	10^9 Flop/second
1 Tflop/s	10^{12} Flop/second
1 Pflop/s	10^{15} Flop/second
1 Eflop/s	10^{18} Flop/second
1 Zflop/s	10^{21} Flop/second

Capacity

1 KB	10^3 Bytes
1 MB	10^6 Bytes
1 GB	10^9 Bytes
1 TB	10^{12} Bytes
1 PB	10^{15} Bytes
1 EB	10^{18} Bytes
1 ZB	10^{21} Bytes

Measuring Performance

- When we **measure** **performance** we usually mean **how fast the computer carries out instructions**

- The measure we use is **MIPS** (*Millions of Instructions Per Second*).

- **MIPS affected by**:
  - The **clock speed** of the processor
  - The **speed of the buses**
  - The **speed** of **memory access.**

MIPS

- A performance measure often used in practice to evaluate the performance of a computer system is the **MIPS rate** for a program A:

$$MIPS(A) = \frac{n_{instr}(A)}{T_{U\_CPU}(A) \cdot 10^6} \quad . \tag{1}$$

$n_{instr}(A)$: number of instructions of program A
$T_{U\_CPU}(A)$: user CPU time of program A

Example:

n_{instr}(A) = 4 Millions

T_{U_CPU}(A) = 0.05 seconds

4 / 0.05 = 80 Millions / 10^6 = **80 MIPS**

MIPS

- modification:

$$MIPS(A) = \frac{r_{cycle}}{CPI(A) \cdot 10^6} ,$$

  where $r_{cycle} = 1/t_{cycle}$ is the clock rate of the processor.
  $CPI(A)$: **C**lock cycles **P**er **I**nstruction: average number of CPU cycles used for instructions of program A
- Faster processors lead to larger MIPS rates than slower processors.

Example:

r_{cycle} = 600 MHz *(Mega == 10^6)*

CPI(A) = 3

600 * 10^6 / 3 = 200 * 10^6 / 10^6 = **200 MIPS**

FLOPs

- For program with scientific computations, the MFLOPS rate (**M**illion **F**loating-point **O**perations **P**er **S**econd) is sometimes used. The MFLOPS rate of a program A is defined by

$$MFLOPS(A) = \frac{n_{flp\_op}(A)}{T_{U\_CPU}(A) \cdot 10^6} \qquad (2)$$

$n_{flp\_op}(A)$: number of floating-point operations executed by A.
$T_{U\_CPU}(A)$: user CPU time of program A

- The effective number of operations performed is used for MFLOPS: the MFLOPS rate provides a fair comparison of different program versions performing the same operations.

Example:

$n_{flp\_op}(A)$ = 90 Millions (floating-point operations)
$T_{U\_CPU}(A)$ = 3.5 seconds

$(90 * 10^{6)}/ (3.5 * 10^{6)}$ = **25.71** MFLOPS(A)

Benchmarks

Why Do Benchmarks?

- **How** we **evaluate differences?**
  - –**Different systems**
  - –**Changes to** a **single system**

- **Benchmarks represent large class** of **important programs**

Benchmarks

- **Microbenchmarks**
  - **Measure one performance dimension or aspect**
    - **Cache bandwidth**
    - **Memory bandwidth**
    - **Procedure call overhead**
    - **FP performance**
  - **Insight** into the **underlying performance factors**
  - **Not** a **good predictor** of overall **application performance**

- **Macrobenchmarks**
  - **Application execution time**
    - **Measures overall performance**, using **one application**
    - **Need application suite**

Popular Benchmark Suites

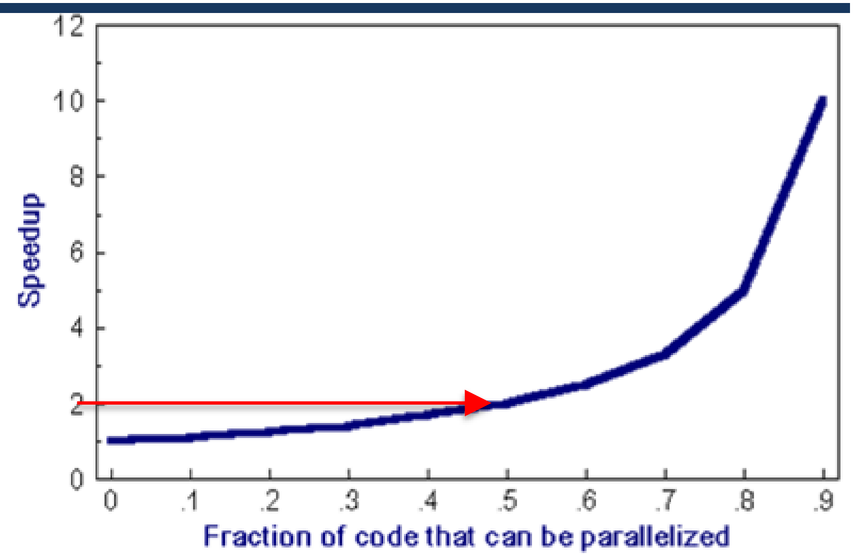
- **Desktop**
  - SPEC CPU2000 - CPU intensive, integer & floating-point applications
  - SPECviewperf, SPECapc - Graphics benchmarks
  - SysMark, Winstone, Winbench
- **Embedded**
  - EEMBC - Collection of kernels from 6 application areas
  - Dhrystone - Old synthetic benchmark
- **Servers**
  - SPECweb, SPECfs
  - TPC-C - Transaction processing system
  - TPC-H, TPC-R - Decision support system
  - TPC-W - Transactional web benchmark
- **Parallel Computers**
  - SPLASH - Scientific applications & kernels

Performance Metrics – Parallel Systems

Amdahl's Law & Speedup Factor

Amdahl's Law

- **Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:**




```
                    1
Max.speedup = --------
                1  - P
```

- If none of the code can be parallelized, P = 0 and the speedup = 1 (**no speedup**). If all of the code is parallelized, P = 1 and the **speedup is infinite (in theory)**.

- If 50% of the code can be parallelized, **maximum speedup = 2**, meaning the code will run twice as fast.

Amdahl's Law

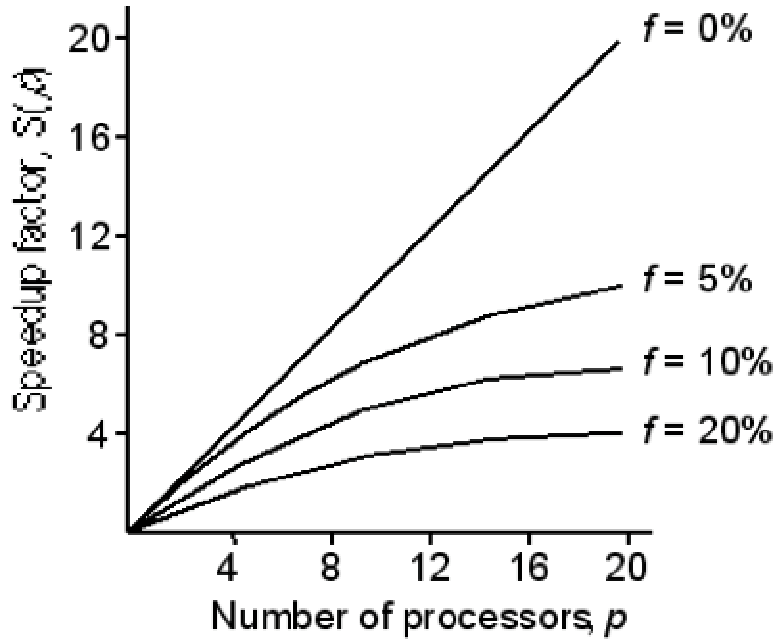
- It soon **becomes obvious** that **there are limits to the scalability of parallelism**

- For example, at P = .50, .90 and .99 (50%, 90% and 99% of the code is parallelizable)

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Maximum Speedup (Amdahl's Law)

Speedup against number of processors



Even with infinite number of processors, maximum speedup limited to 1/f .

Example: With only 5% of computation being serial, maximum speedup is 20, irrespective of number of processors.

F = serial fraction

E.g., 1/0.05 (5% serial) = 20 speedup (maximum)

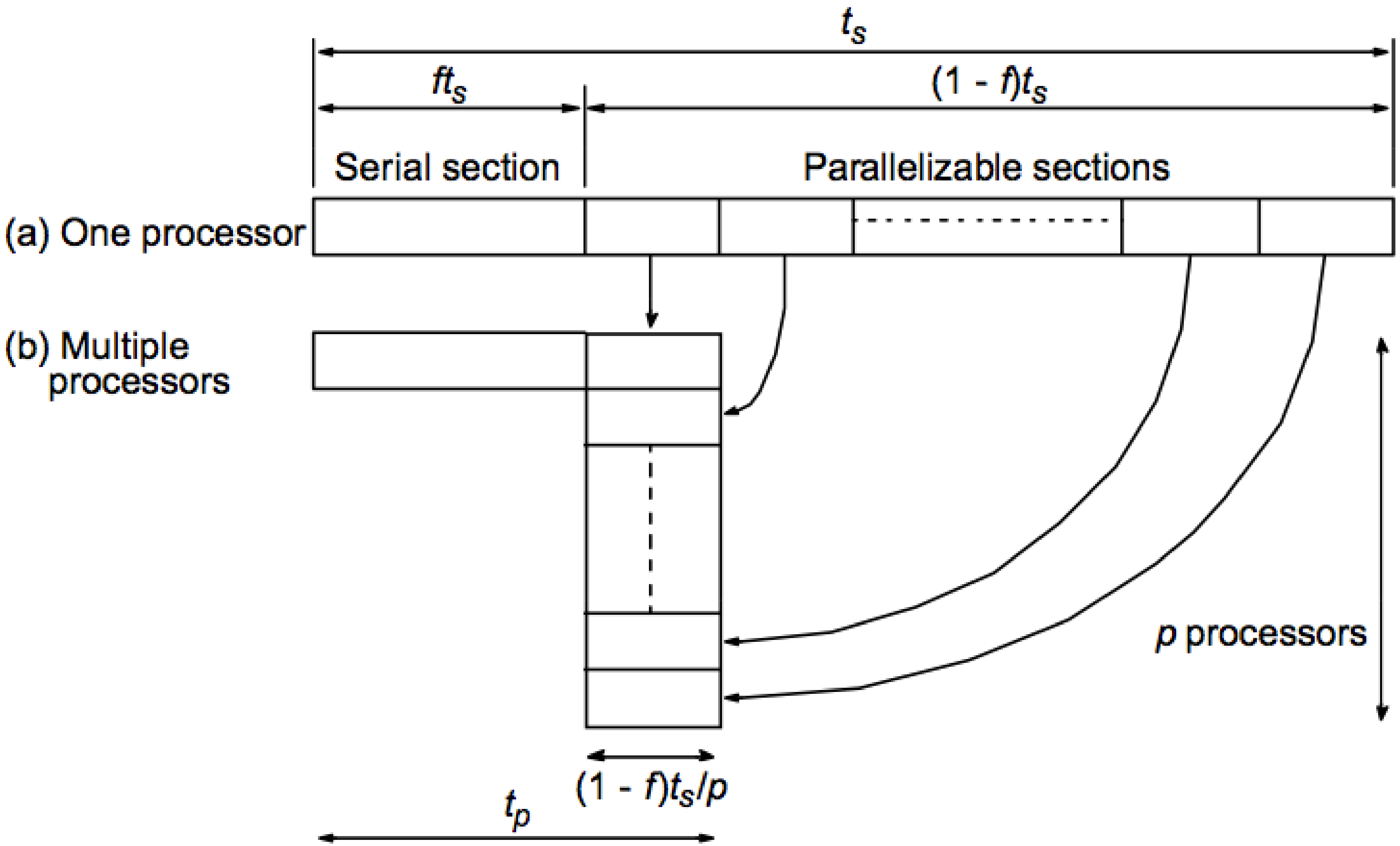
Maximum Speedup (Amdahl's Law)

Maximum speedup is usually ***p*** with *p* processors (***linear speedup***).

Possible to **get** super-linear speedup (greater than *p*) but usually a specific reason such as:

- **Extra memory** in multiprocessor system
- **Nondeterministic algorithm**

Maximum Speedup (Amdahl's Law)



Speedup

$$S(p) = \frac{\text{Execution time using one processor (best sequential algorithm)}}{\text{Execution time using a multiprocessor with } p \text{ processors}} = \frac{t_s}{t_p}$$

where t_s is execution time on a single processor and t_p is execution time on a multiprocessor.

- ***S*(*p*)** gives **increase in speed** by using multiprocessor
- Use **best sequential algorithm** with single processor system **instead of parallel program run with 1 processor** for **t_s**. Underlying algorithm for parallel implementation might be (and is usually) different.

Speedup

Speedup can also be used in terms of **computational steps**:

$$S(p) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with } p \text{ processors}}$$

Speedup

Speedup factor is given by:

$$S(p) = \frac{t_s}{f t_s + (1 - f)t_s / p} = \frac{p}{1 + (p - 1)f}$$

Here ***f*** is the part of the code that is serial:

e.g. if **f==1** (all the code is serial, then the **speedup** will be **1** **no matter how may processors are used**

Speedup (with *N* CPUs or Machines)

- Introducing the **number of processors** **performing** the **parallel fraction of work**, the relationship can be modelled by:

```
                          1
speedup    =    ------------
                  fS +    fP
                         -----
                          Proc
```

- where **fP** = parallel fraction,

  Proc = number of processors and

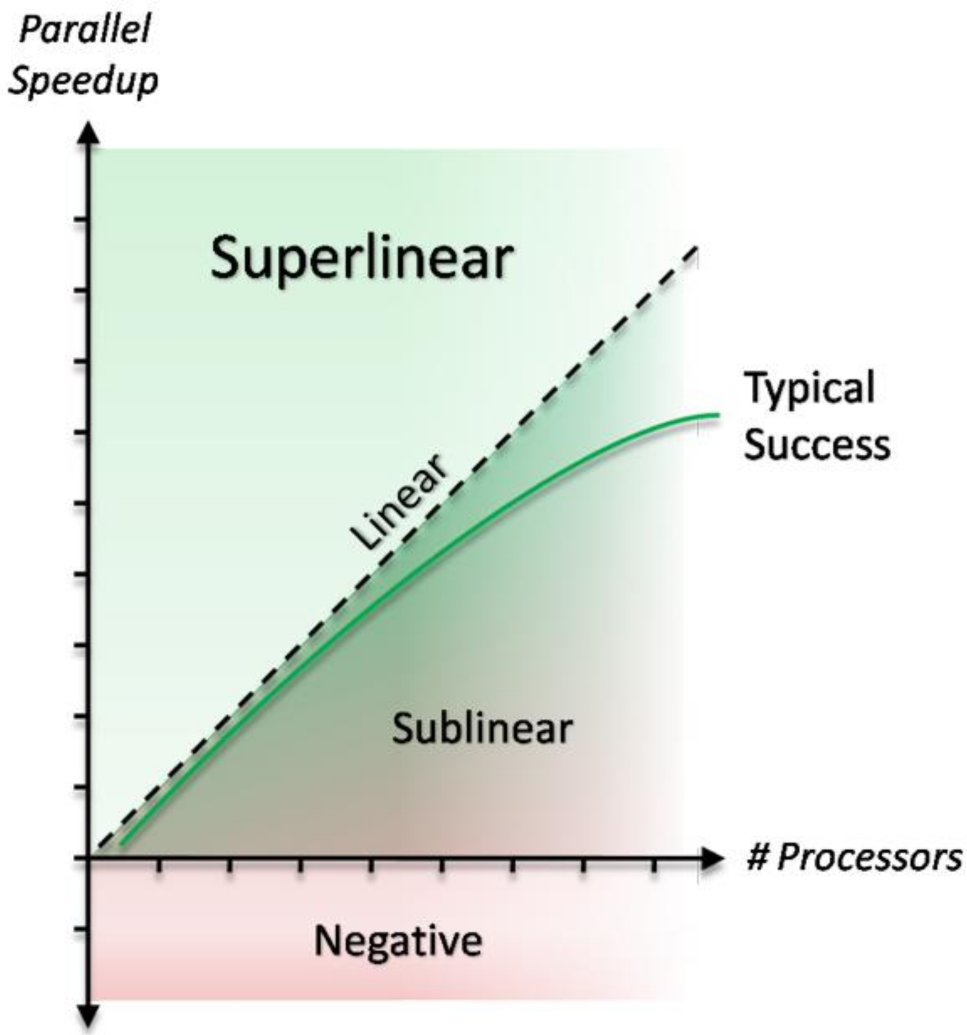
  fS = serial fraction

Linear and Superlinear Speedup

- **Linear speedup**
  - **Speedup** of **N**, for **N processors**
  - **Parallel** program is **perfectly scalable**
  - **Rarely achieved** in practice

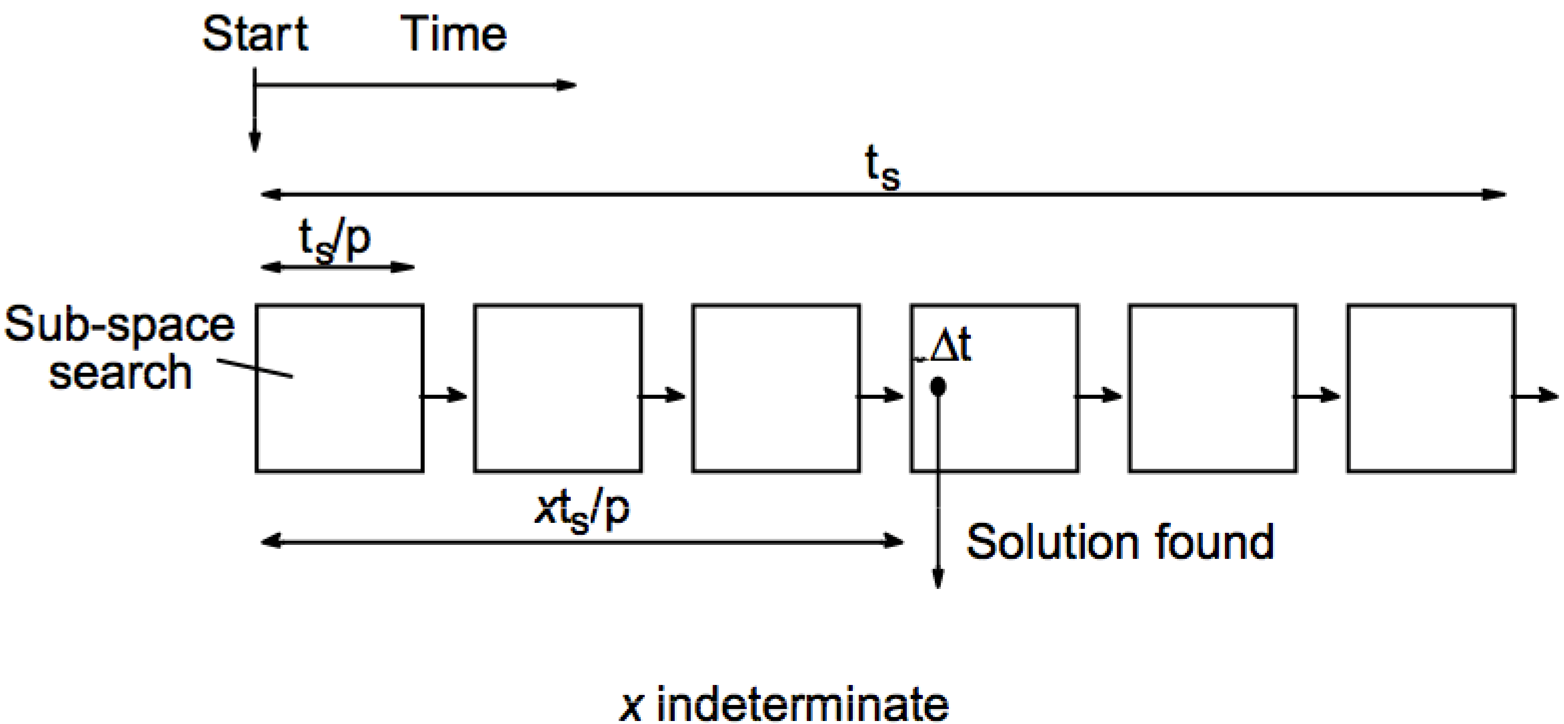
- **Superlinear Speedup**
  - **Speedup** of **>N**, for **N processors**
    - **Theoretically not possible**
    - **How** is this **achievable** on **real machines**?
      - **Think** about **physical resources (cache, memory etc)** of **N processors**

Super-linear Speedup



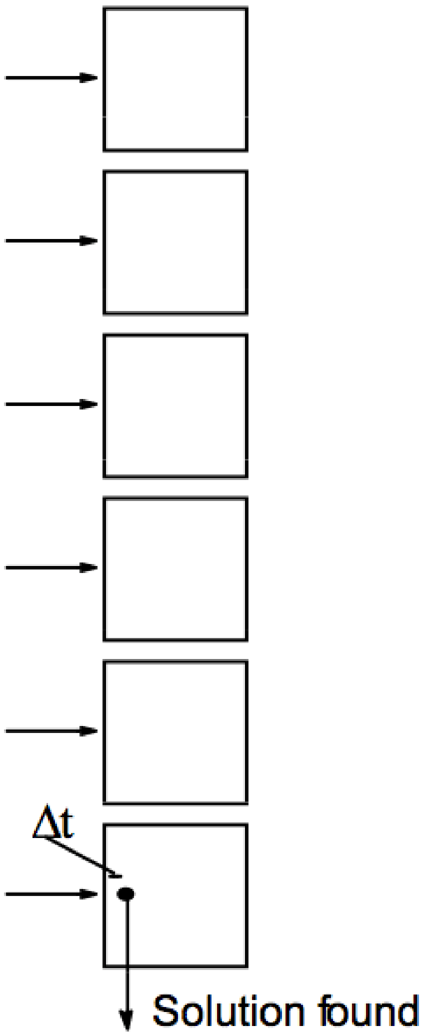
Super-linear Speedup Example - Searching

(a) Searching each sub-space sequentially



Super-linear Speedup Example - Searching

(b) Searching each sub-space in parallel



Efficiency

- **Efficiency** is the **ability** to **avoid wasting materials**, **energy**, **efforts**, **money**, and **time** in **doing something** or in **producing** a desired **result**

- The **ability** to **do things well**, **successfully**, and **without waste**

Efficiency

- **Efficiency:**

$$E = \frac{\text{Speedup}}{\text{Number of Processors}}$$

Speedups and Efficiencies of parallel program on different problem sizes

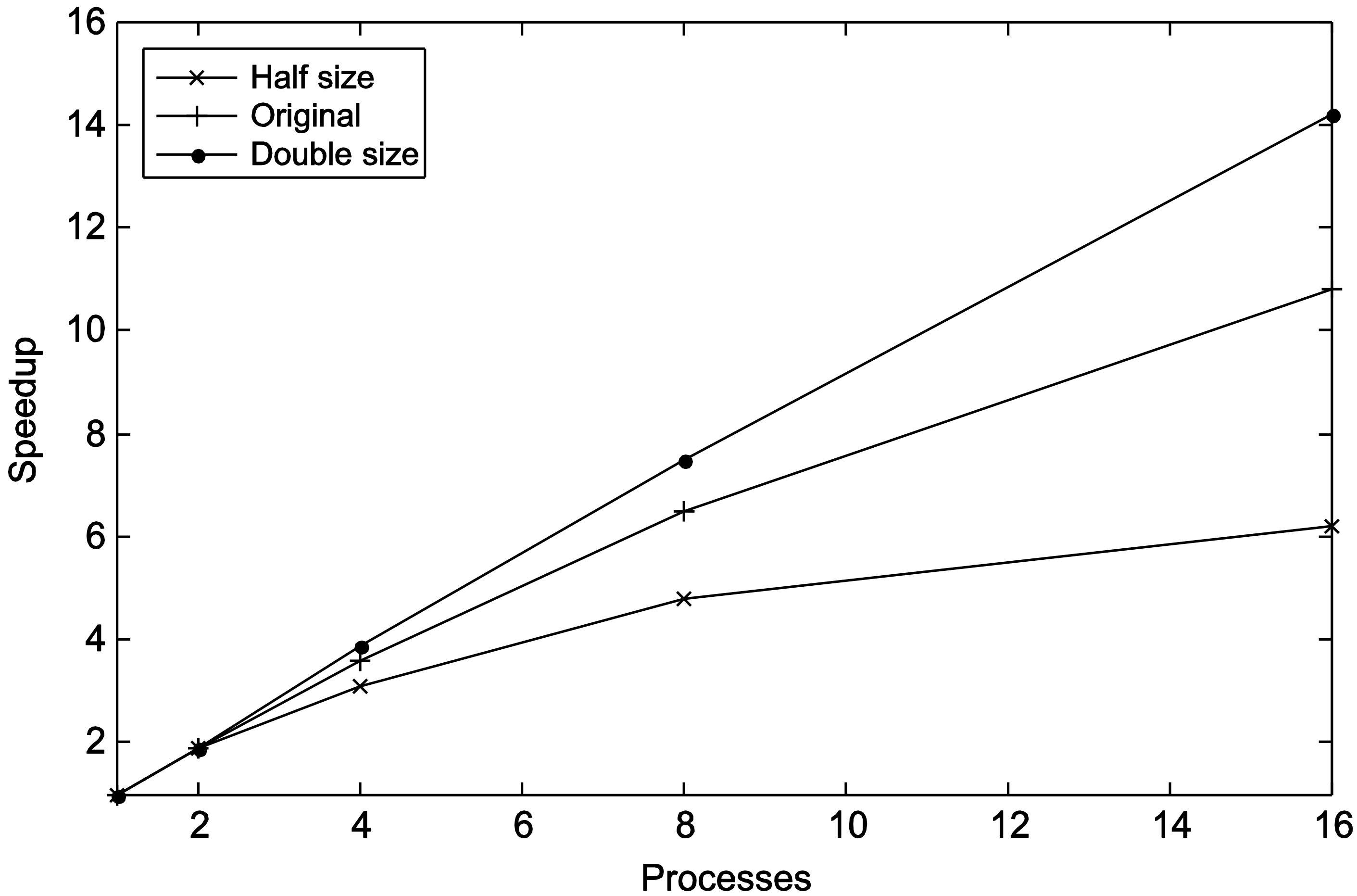
Machine size (Processors)

Problem size	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

S: Speedup

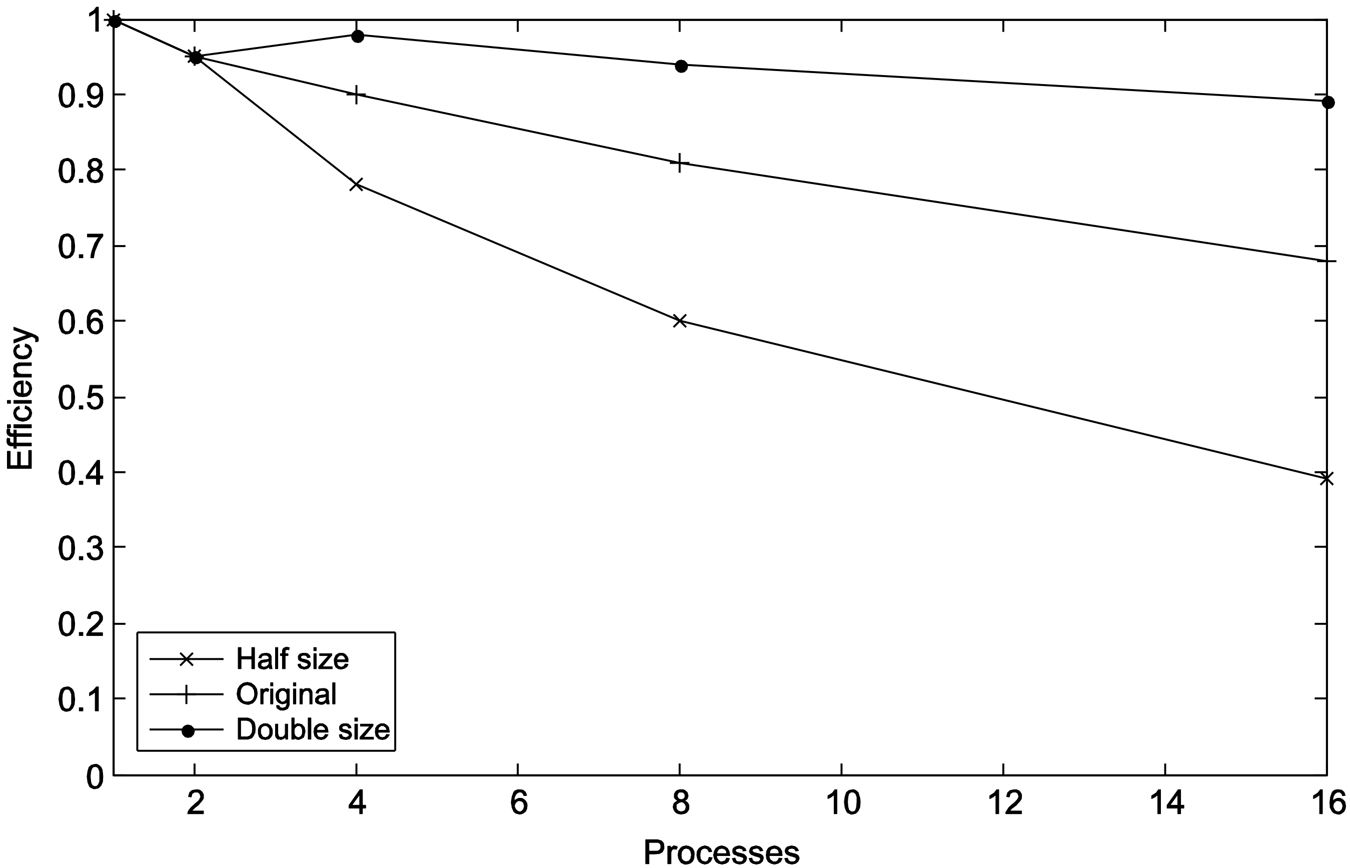
E: Efficiency

Speedup



Half size
Original
Double size
Speedup
Processes
16
14
12
10
8
6
4
2
2
4
6
8
10
12
14
16

Efficiency



Efficiency
Processes
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
2
4
6
8
10
12
14
16
Half size
Original
Double size

Gustafson's Law

Amdahl's law Sufficient?

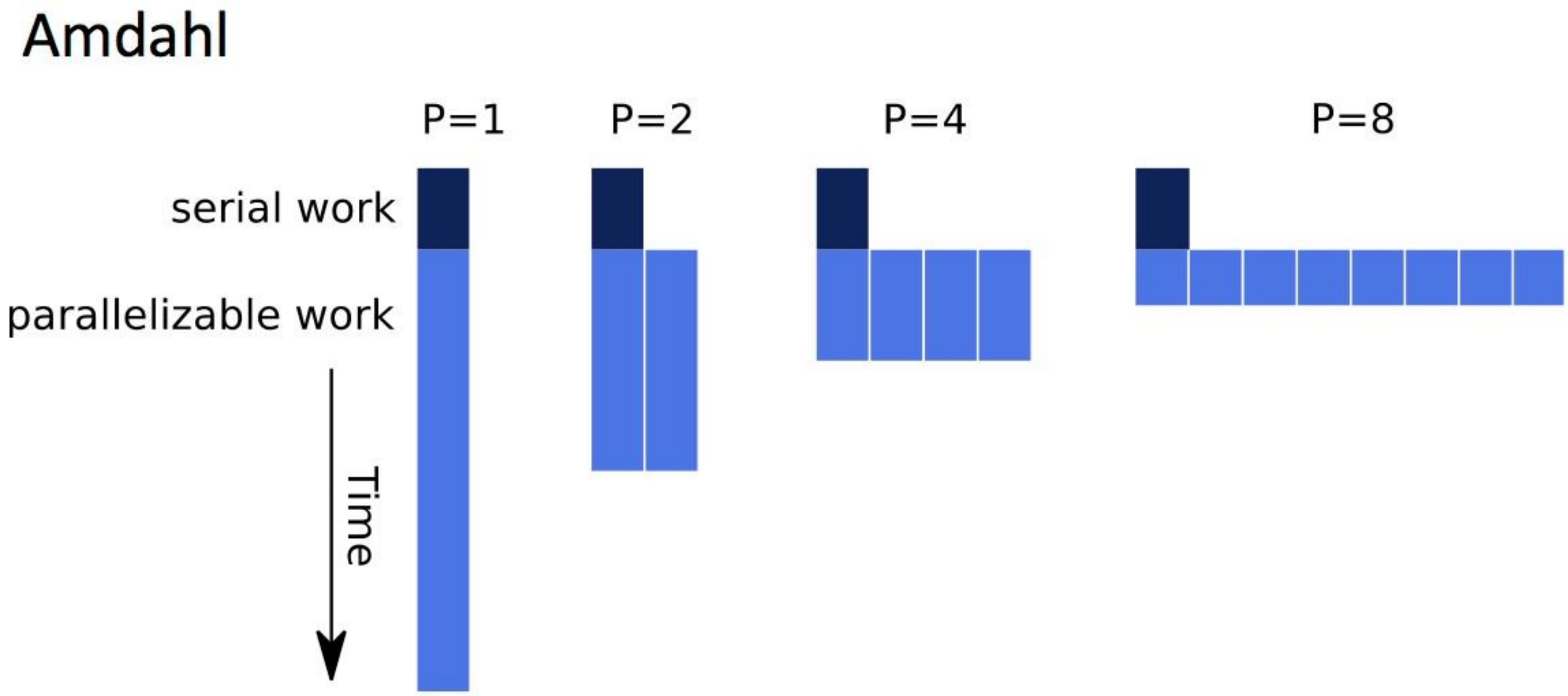
- **Amdahl's law works** on a ***fixed*** **problem size**
  - **Shows how execution time decreases as number of processors increases**
  - **Limits maximum speedup achievable**
  - **So, does it mean large parallel machines are not useful?**
  - **Ignores performance overhead (e.g. communication, load imbalance)**

- **Gustafson's Law says** that **increase of problem size** for **large machines can retain scalability** with **respect to the number of processors**

Gustafson’s Law

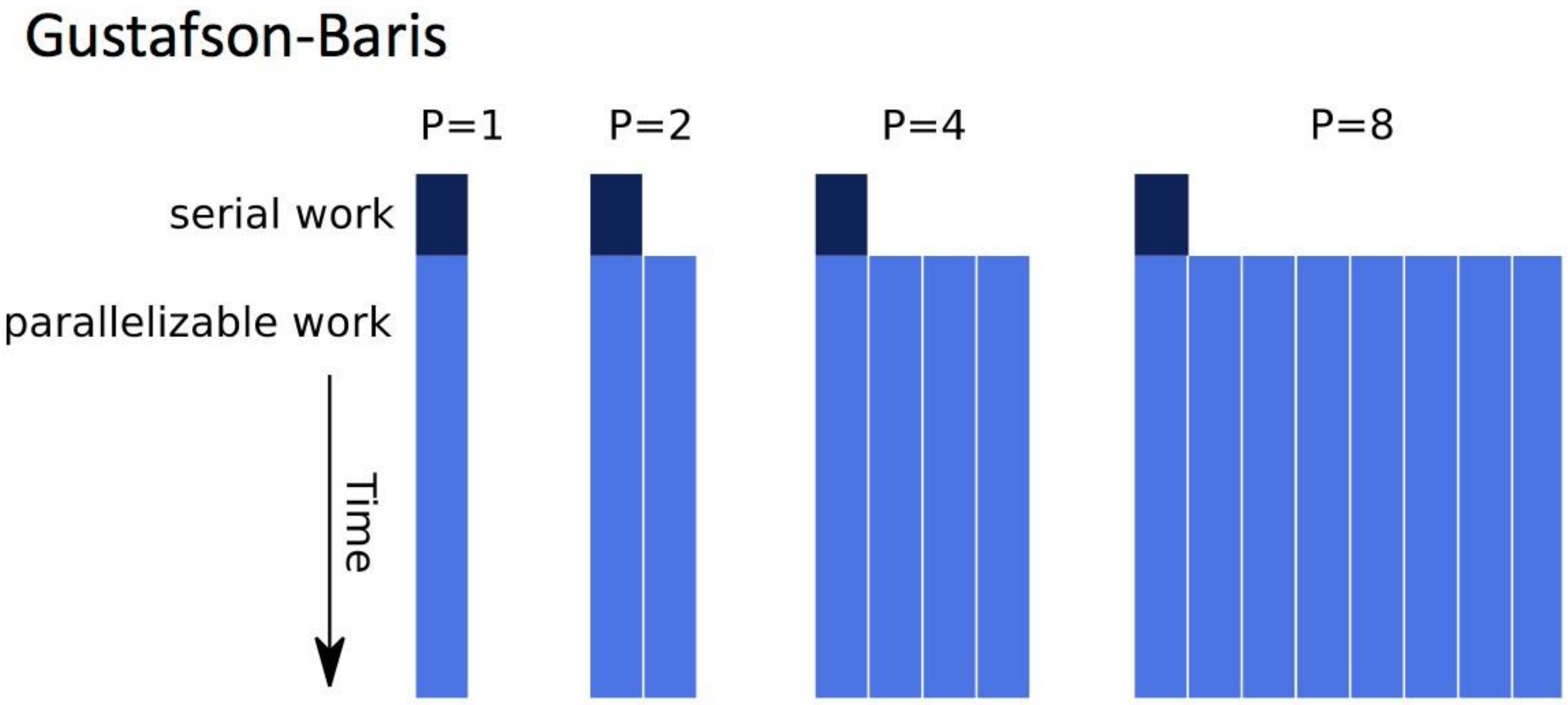
- **Time-constrained scaling (i.e., we have fixed-time to do performance analysis or execution)**
- **Example: a user wants more accurate results within a time limit**
- **Execution time** is **fixed** as **system scales**

Amdahl versus Gustafson's Law



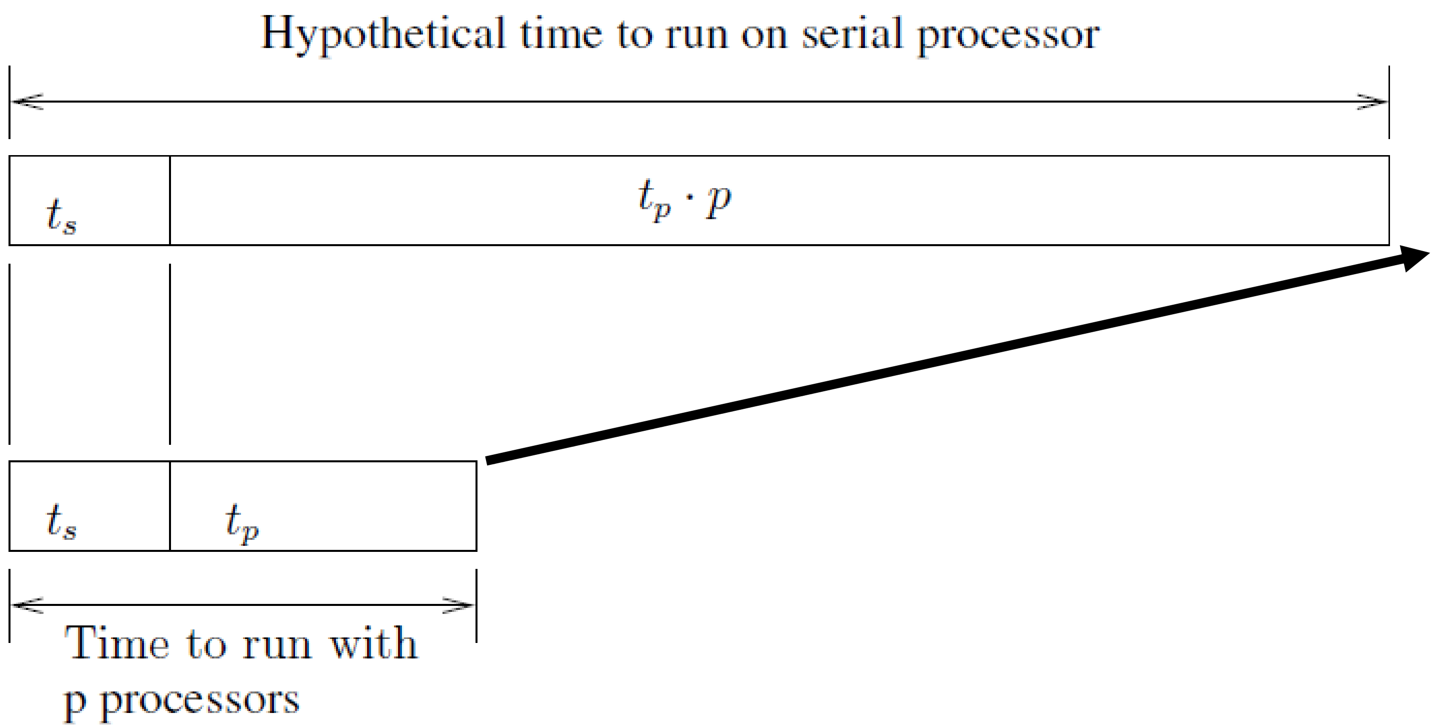
Credits: University of Oregon

Amdahl versus Gustafson's Law



Credits: University of Oregon

Gustafson's Law



- **P processors**, with **increased** number of **processors** the **problem size will <u>also be increased</u>**
  - **<u>Importantly</u> parallel part will be increased**

Gustafson's Law

$$\text{Speedup} = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

$$\text{Scaled speedup} = \frac{\text{Hypothetical time to solve problem on sequential computer}}{\text{Actual parallel execution time}}$$

$$S(p) = \frac{t_s + t_p p}{t_s + t_p} = \frac{t_s}{t_s + t_p} + \frac{t_p}{t_s + t_p} p$$

$$= s + (1 - s)p = s + p - ps = p + (1 - p)s$$

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

where, **S(p)** <u>**Scaled Speedup**</u>, using **P processors**

s → **fraction** of **program** that is **serial** (<u>**cannot be parallelized**</u>)

Gustafson's Law- Example

- An application running on 10 processors spends 3% of its time in serial code. What is the scaled speedup of the application?

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

S(p) = 10 + (1 - 10) * (0.03) = 10 – 0.27 = 9.73 (Scaled Speedup)

Speedup Using Amdahl's Law ?

$$S(p) = \frac{t_s}{f t_s + (1 - f)t_s/p} = \frac{p}{1 + (p - 1)f}$$

Gustafson's Law- Example

Speedup Using Amdahl's Law

$$S(p) = \frac{t_s}{f t_s + (1 - f)t_s/p}$$

S(p) = 1 / ((0.03)+ (0.97)/10

= 1 / 0.03 + 0.097

= 1 / 0.127

= 7.874

OR $$\frac{p}{1 + (p - 1)f}$$

S(p) = 10 / 1 + (10-1)* (0.03)

= 1 / 1 + 0.027

= 1 / 1.027

= 7.874

Summary Gustafson’s Law

- **Derived** by **fixing** the **parallel execution time** (**Amdahl fixes** the <u>**problem size**</u> -> ***fixed serial execution time***)

- For **many practical situations**, **Gustafson’s law makes more sense**
  - Have a **bigger computer**, solve a **bigger problem**

Scalability

- In general, a **problem** is ***scalable*** if it **can handle ever increasing problem sizes**

- If we **increase** the **number of <u>processes/threads</u>** and keep the **efficiency fixed** **<u>without increasing problem</u> size**, the problem is ***strongly scalable***.

- If we **keep** the **efficiency fixed** by **<u>increasing the problem size</u>** at the same rate as we **increase** the **number of processes/threads**, the problem is ***weakly scalable***.

Any Questions