# Технології графічного процесінгу

## (Масивно-паралельні обчислення на графічних прискорювачах
## …
## Massively Parallel Computing on Graphic Processing Units - GPUs)

# Lecture 2. Performance Metrics in Parallel and GPU Computing

Yuri G. Gordienko
(NTUU-KPI, 2021)

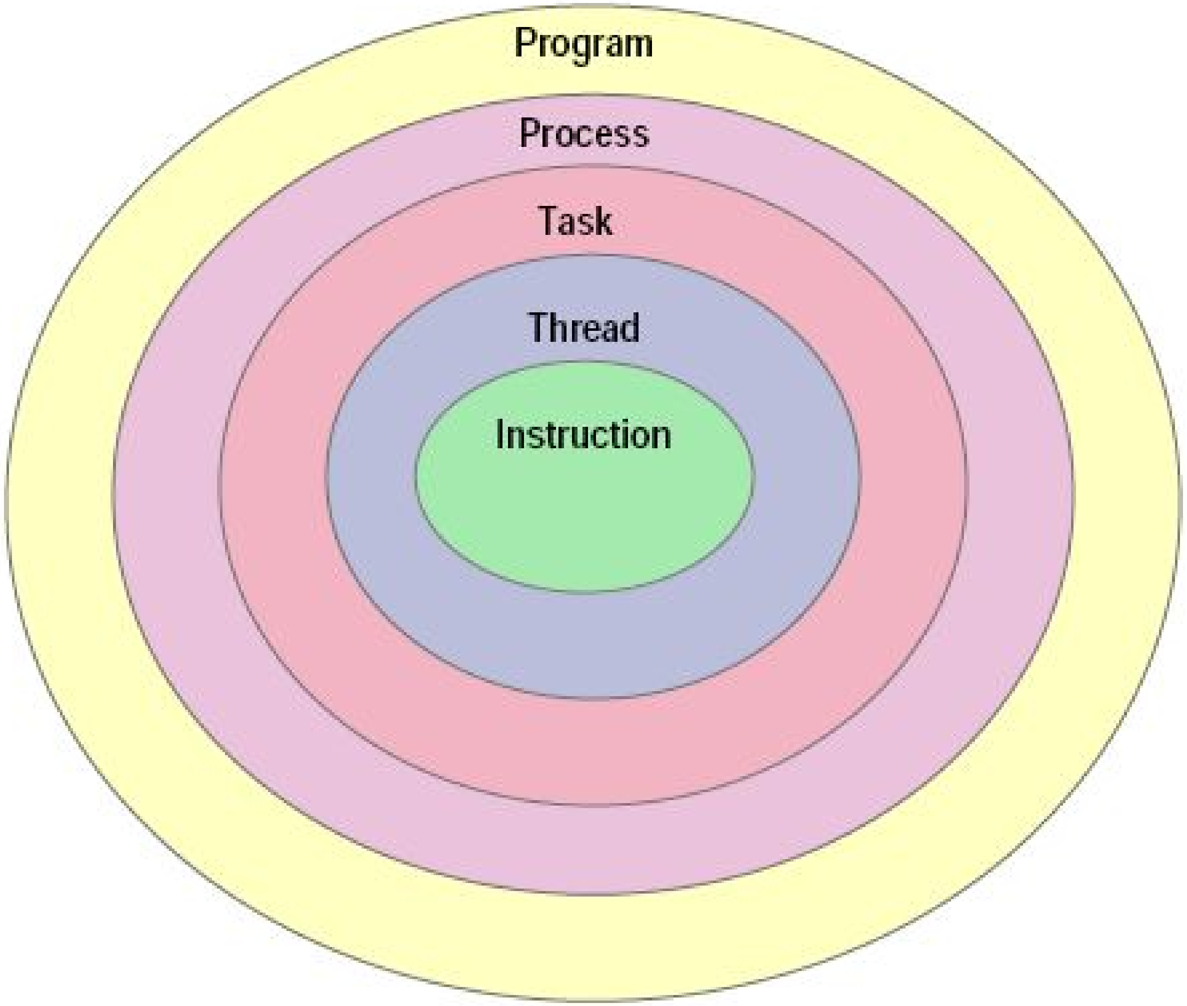
(on the basis of materials by M.Hammoud, M.F.Sakr, A.Simpson, H.Kim)

# Parallel Computing

The main aspects:

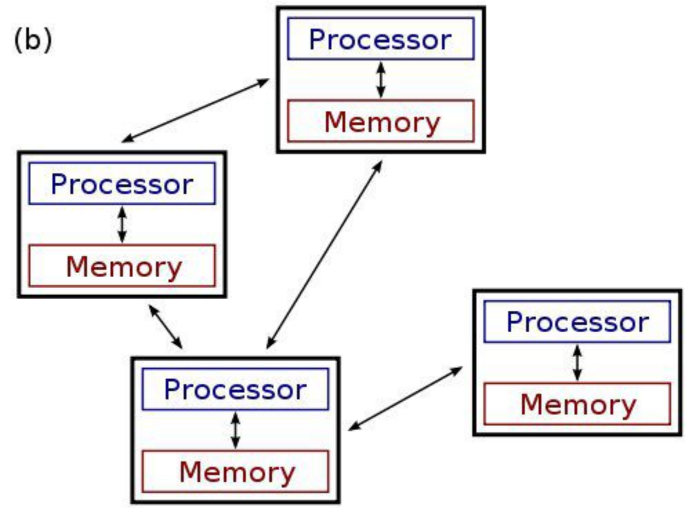
- Definition: what is Parallel Computing?
- Levels (heterogeneity, granularity)
- Classification of Systems
- Memory
- Programming Models

# Levels of Parallel Computing

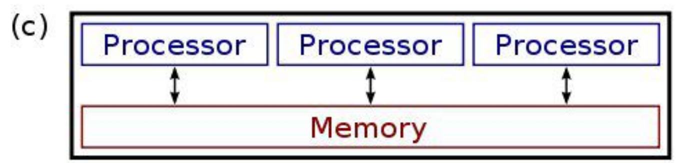


# Memory in Parallel Computing

- Distributed




- Shared

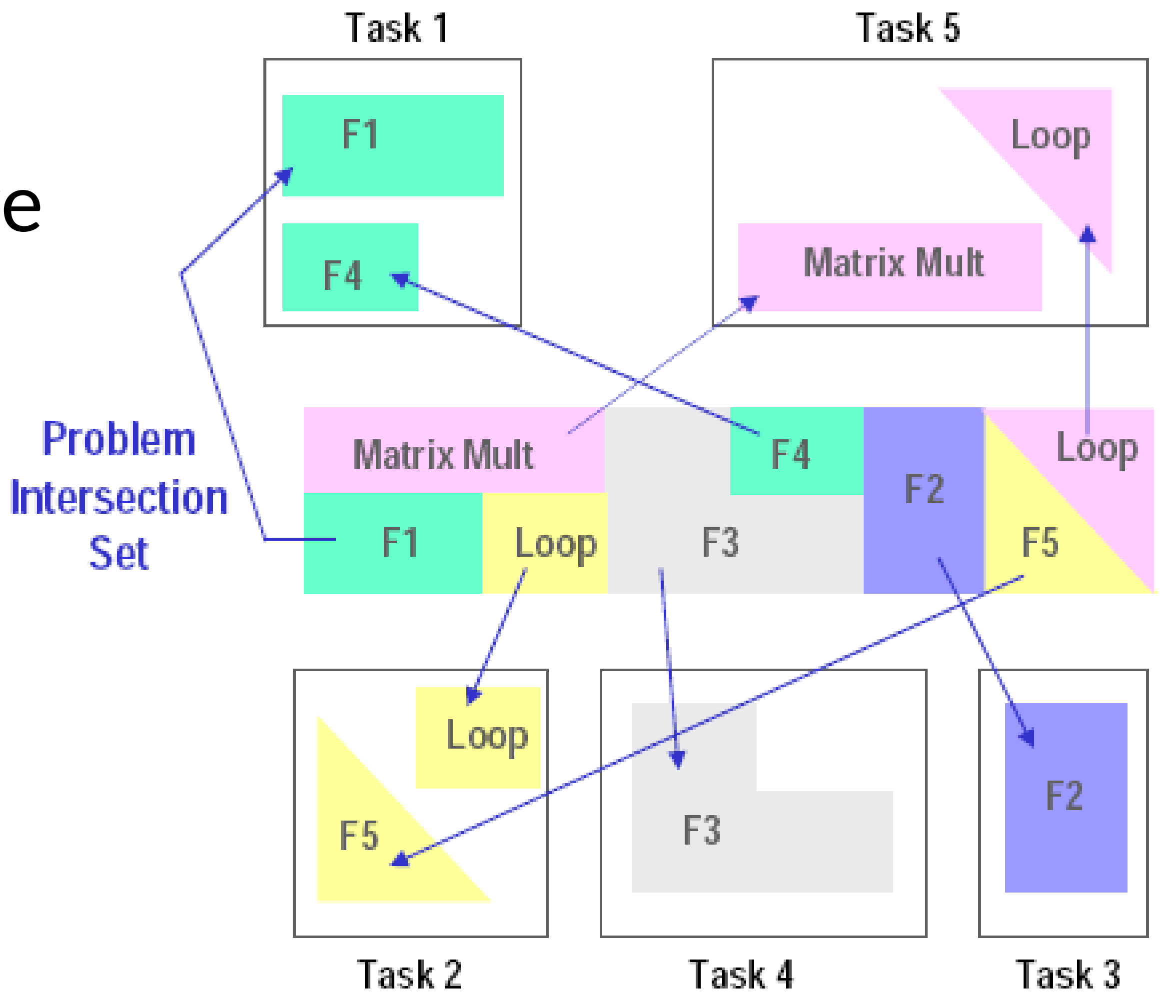



- Hybrid (Distributed-Shared)

# Functional Decomposition

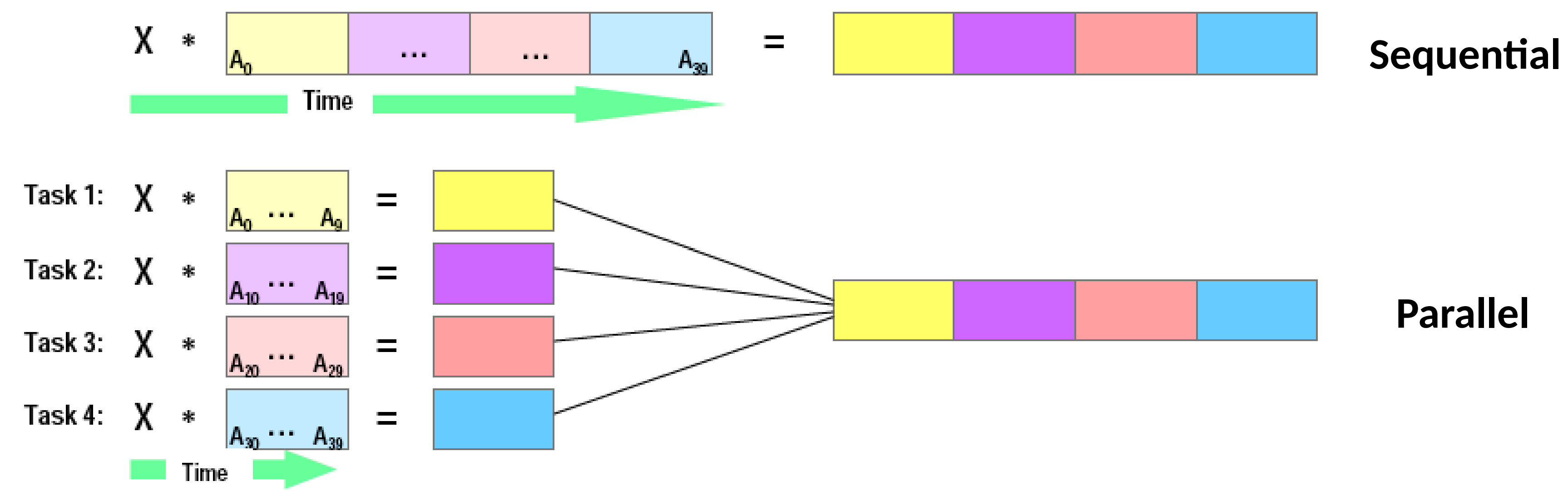
- Computations (not the data) are grouped

- Each task computes a part of the overall work.



# Domain (Data) Decomposition – GPU!

- The **data are divided** into portions
- Each portion is given to a task that performs the operation on it in in parallel



# Communication - Timing

Synchronous

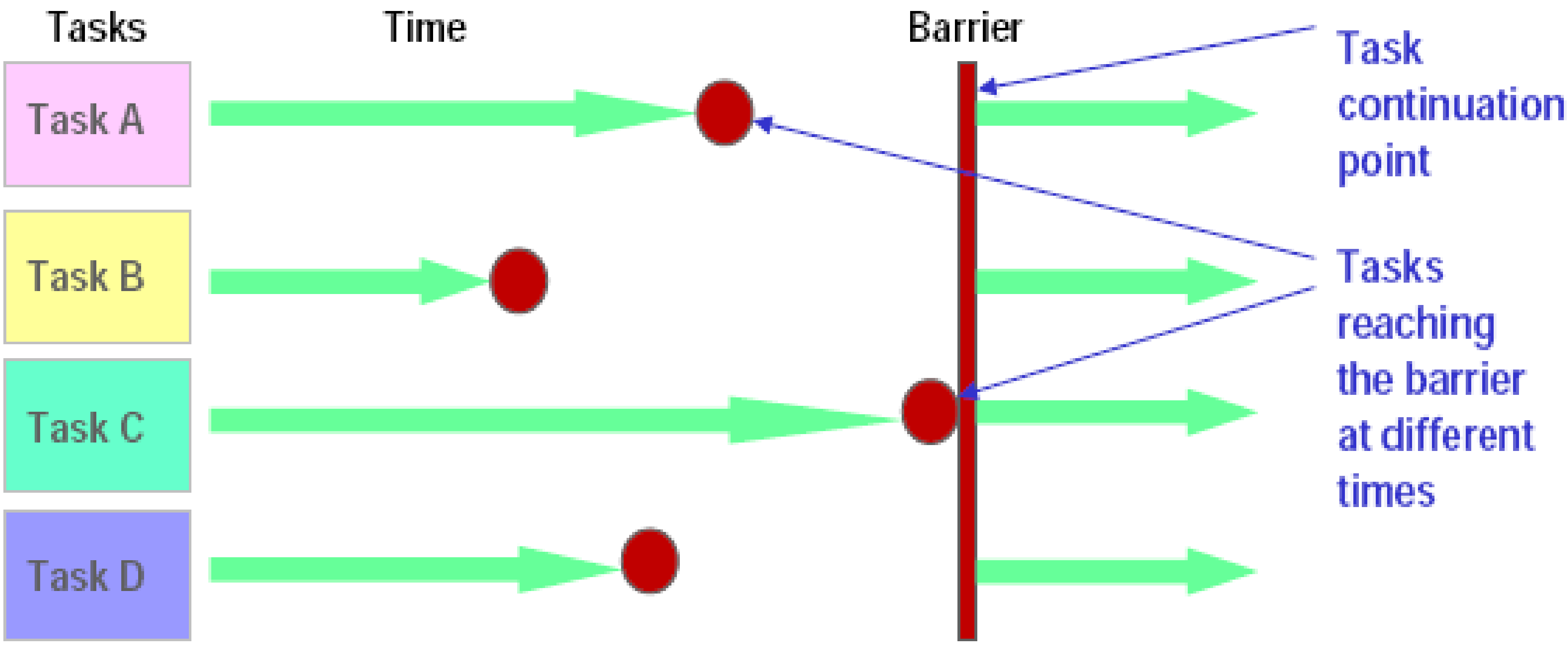
- “Handshaking" between tasks that are sharing data is needed; it could be done implicitly or explicitly
- **Blocking communications**: some work must be held until the communications are done

Asynchronous

- Tasks can communicate with data independently from the work they are doing
- **Non-blocking communications**

# Synchronization - Barrier

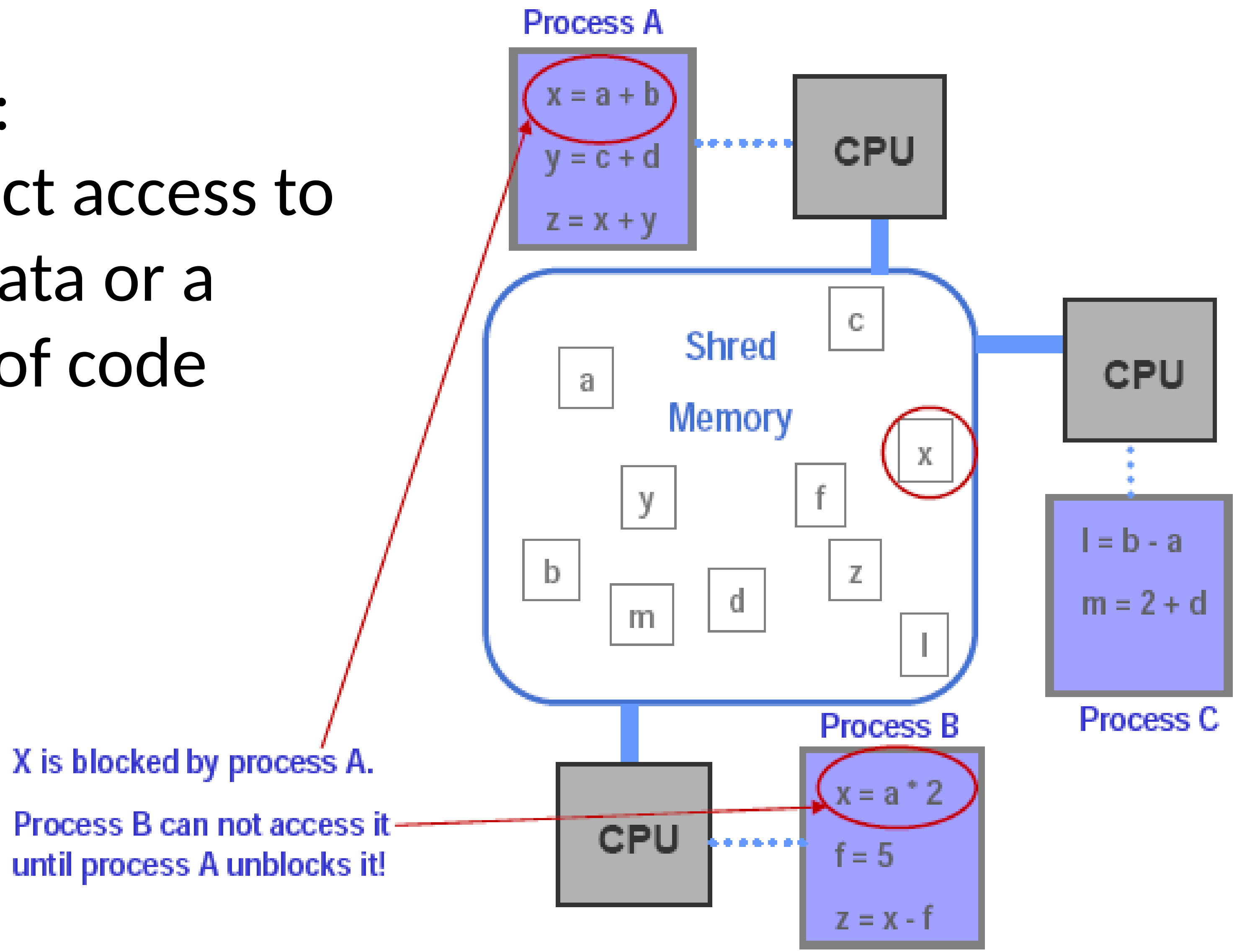
<u>Definition:</u> A point at which a task must stop, and can not proceed until all tasks are synchronized.



# Synchronization - Locks/Semaphores

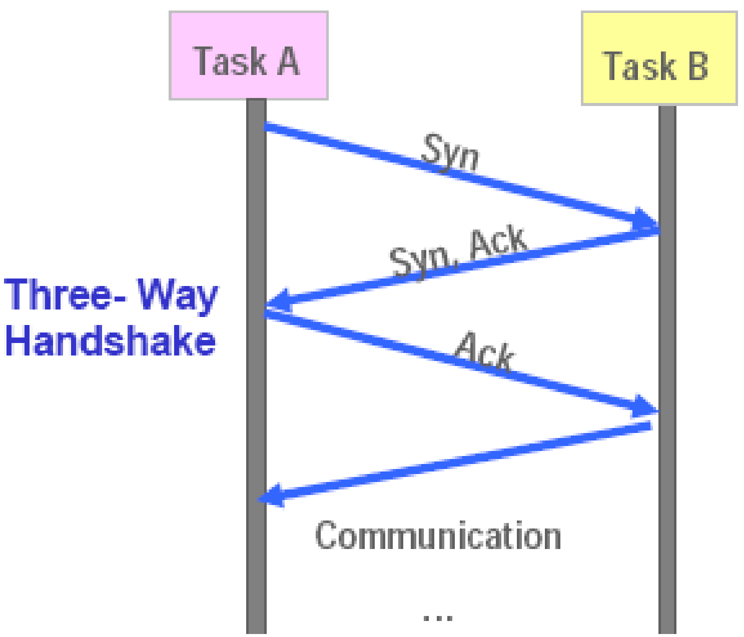
<u>Definition</u>:

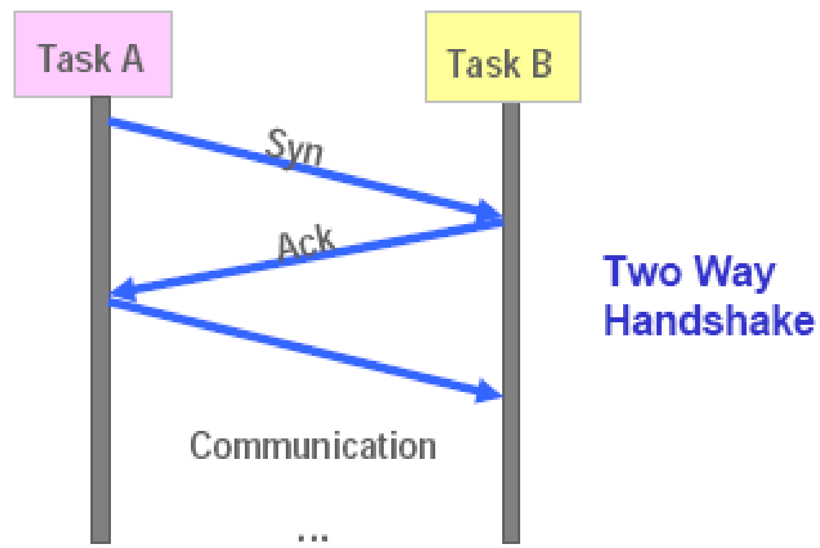
to protect access to global data or a section of code



# Synchronous Communication Oper-ns

<u>Definition:</u> Coordination is required between the task that is performing an operation and the other tasks performing the communication



# Load Balancing

Definition: to distribute work among all tasks so they are all kept busy all of the time

Ways to achieve:

- Adequate partitioning
- Dynamic work assignment
  - Scheduler/task-pool
  - Algorithm to detect and handle imbalances

Note: if barrier synchronization is used, then the slowest task determines the performance

# Granularity

<u>Definition:</u> computation/communication ratio

- <u>Fine-grain parallelism</u>: **few** computation events are done between communication events
    - High communication overhead
    - Small opportunity to enhance performance
- <u>Coarse-grain parallelism</u>: **many** computational events are done between communication events.
    - Large opportunity to enhance performance
    - Harder to do load balancing efficiently

# All these aspects affects:

All these aspects affects:

Performance
of Parallel and Distributed Computing

# Parallel Computing Performance - Metrics

# What Metrics are Used?

- Time

  is self-explanatory

- Speedup

- Efficiency

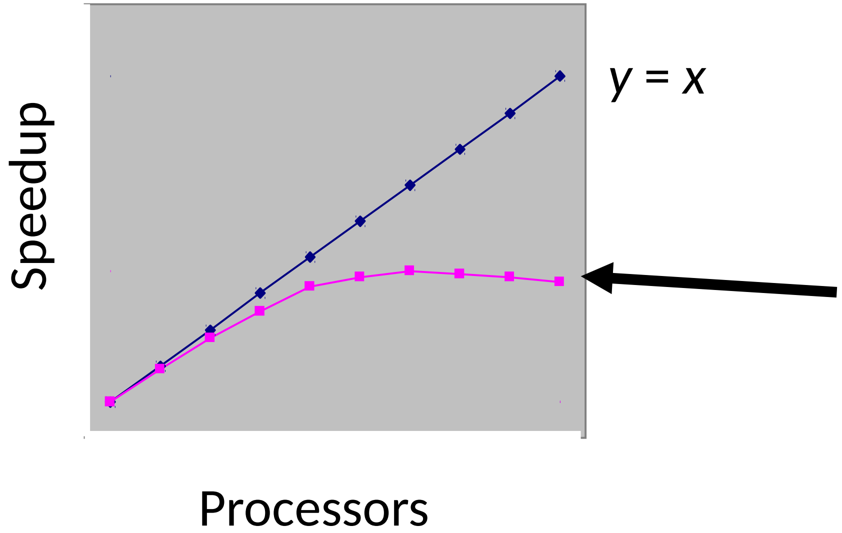
- Cost

# Speedup - Definition

Definition: the ratio $\Psi(n,p)$ between sequential execution time and parallel execution time (for data size $n$ and $p$ processors):

$$\text{Speedup} = \Psi(n,p) = \frac{\text{Sequential execution time}}{\text{Parallel execution time}}$$

Example: sequential program executes in 6 seconds and the parallel program executes in 2 seconds -> speedup is 3.




Speedup curves look like this

# Speedup - Notes

$$\Psi(n,p) = t_s/t_p$$

- In practice:
  - $t_s$ is the execution time on a single processor, using the fastest known sequential algorithm
  - $t_p$ is the execution time using $n$ parallel processors.
- In theory:
  - $t_s$ is the worst case running time for of the fastest known sequential algorithm for the problem
  - $t_p$ is the worst case running time of the parallel algorithm using $n$ processing units.

# Efficiency - Definition

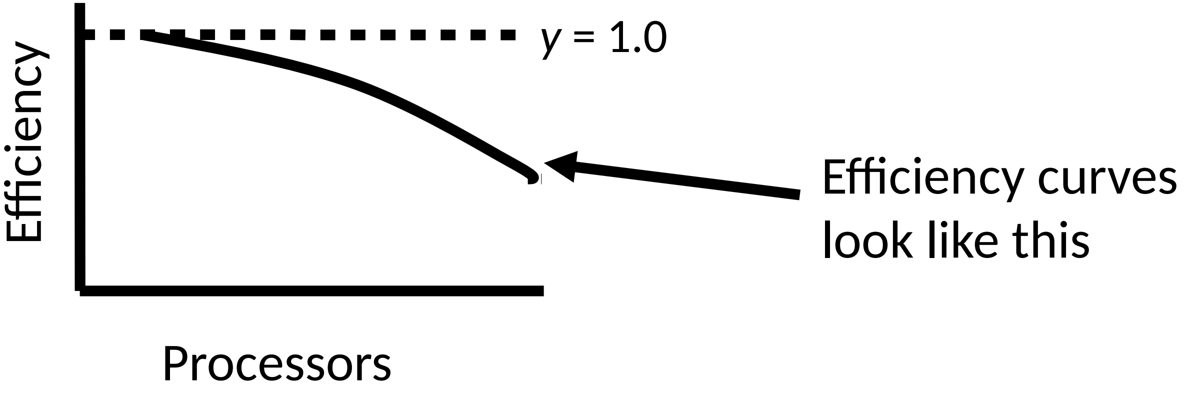
Definition: measure of processor utilization $\varepsilon(n,p)$ as the speedup divided by the number of processors $p$

$$\text{Efficiency} = \varepsilon(n,p) = \frac{\text{Speedup}}{\text{Processors}}$$

Example:

Program achieves speedup of 3 on 4 CPUs

Efficiency is 3 / 4 = 75%



# Efficiency - Notes

$$\text{Efficiency} = \varepsilon(n, p) = \frac{\text{Speedup}}{\text{Processors}}$$

- For algorithms for traditional problems (when superlinear speedup is not possible):

speedup ≤ processors

- Since speedup ≥ 0 and processors > 1, it follows from the above two equations that

$$\mathbf{0 \leq \varepsilon(\boldsymbol{n},\boldsymbol{p}) \leq 1}$$

- However, there are superlinear algorithms, when

speedup > processors

and for this case:

$$\mathbf{\varepsilon(\boldsymbol{n},\boldsymbol{p}) > 1}$$

# Cost - Definition

Cost = Parallel running time $\times$ processors

- "Cost" is a much overused word,
  the term "algorithm cost" is sometimes used for clarity.
- The cost of a parallel algorithm should be compared to the running time of a sequential algorithm.
  - Cost removes the advantage of parallelism by charging for each additional processor.
  - A parallel algorithm whose cost is growing "with similar rate" than the running time of an optimal sequential algorithm is called cost-optimal.

# Speedup, Cost, Efficiency

$$\text{Efficiency} = \frac{\text{Sequential running time}}{\text{Processors} \times \text{Parallel running time}}$$

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Processors}}$$

$$\text{Efficiency} = \frac{\text{Sequential running time}}{\text{Cost}}$$

# Parallel Computing Performance - Laws

# Parallel Computing Performance - Laws

Amdahl’s Law (1967): the principal limit of speedup in sequential-parallel code

Gustafson’s Law (1988): another way to evaluate the performance of a parallel program

Karp/Flatt Metric (1990): whether the principle barrier to the program speedup is the amount of inherently sequential code or parallel overhead

Isoefficiency (isogranularity) Metric: the scalability of a parallel algorithm executing on a parallel system

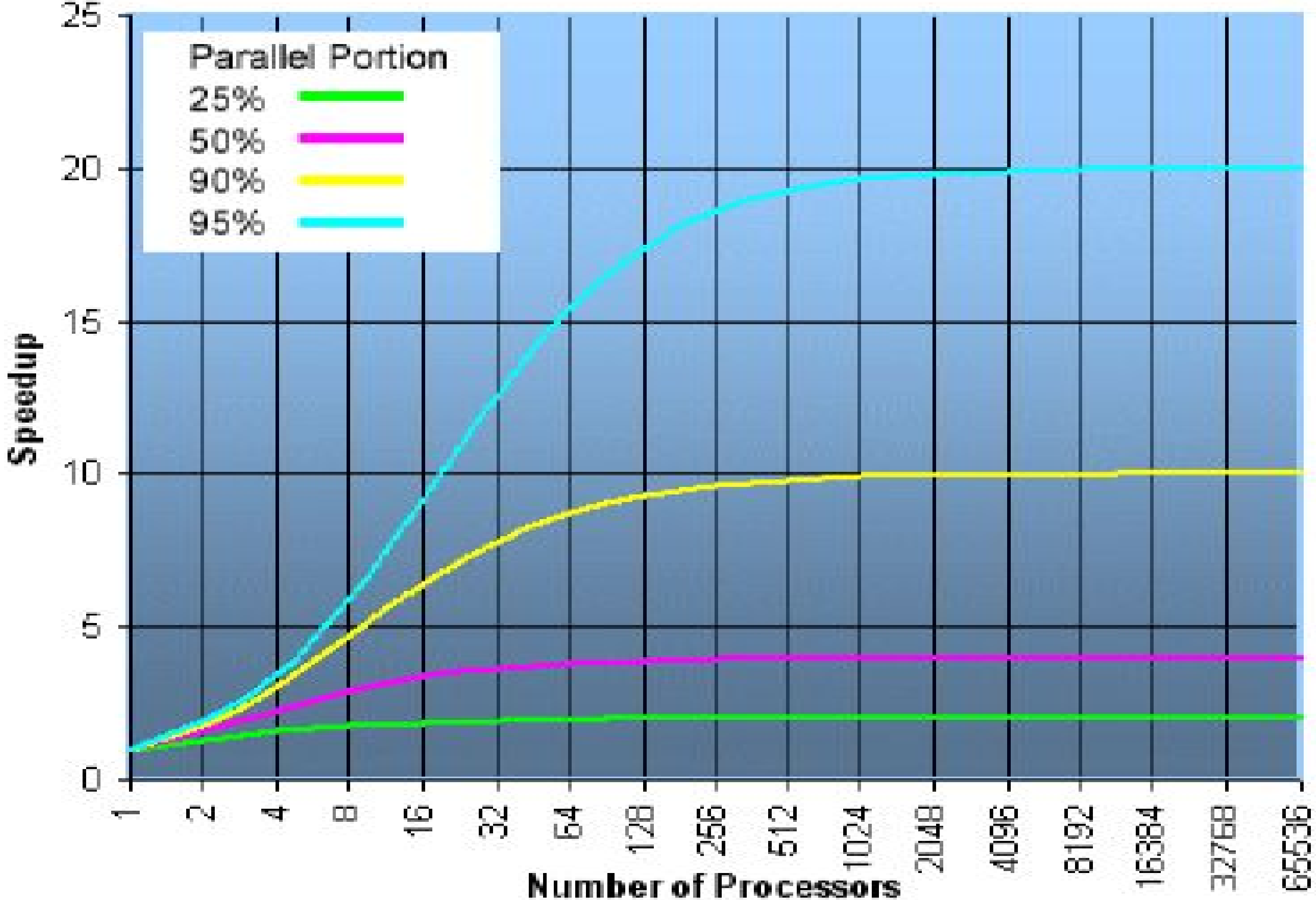
# Amdahl’s Law

# Amdahl's Law

- Suppose that the sequential execution of a program takes $T_1$ time units and the parallel execution on $p$ processors takes $T_p$ time units
- Suppose that out of the entire execution of the program, $s$ fraction of it is not parallelizable while $1-s$ fraction is parallelizable
- Then the speedup (Amdahl's formula):

$$\frac{T_1}{T_p} = \frac{T_1}{(T_1 \times s + T_1 \times \frac{1-s}{p})} = \frac{1}{s + \frac{1-s}{p}}$$

# Amdahl's Law: Illustration



# Amdahl's Law: An Example

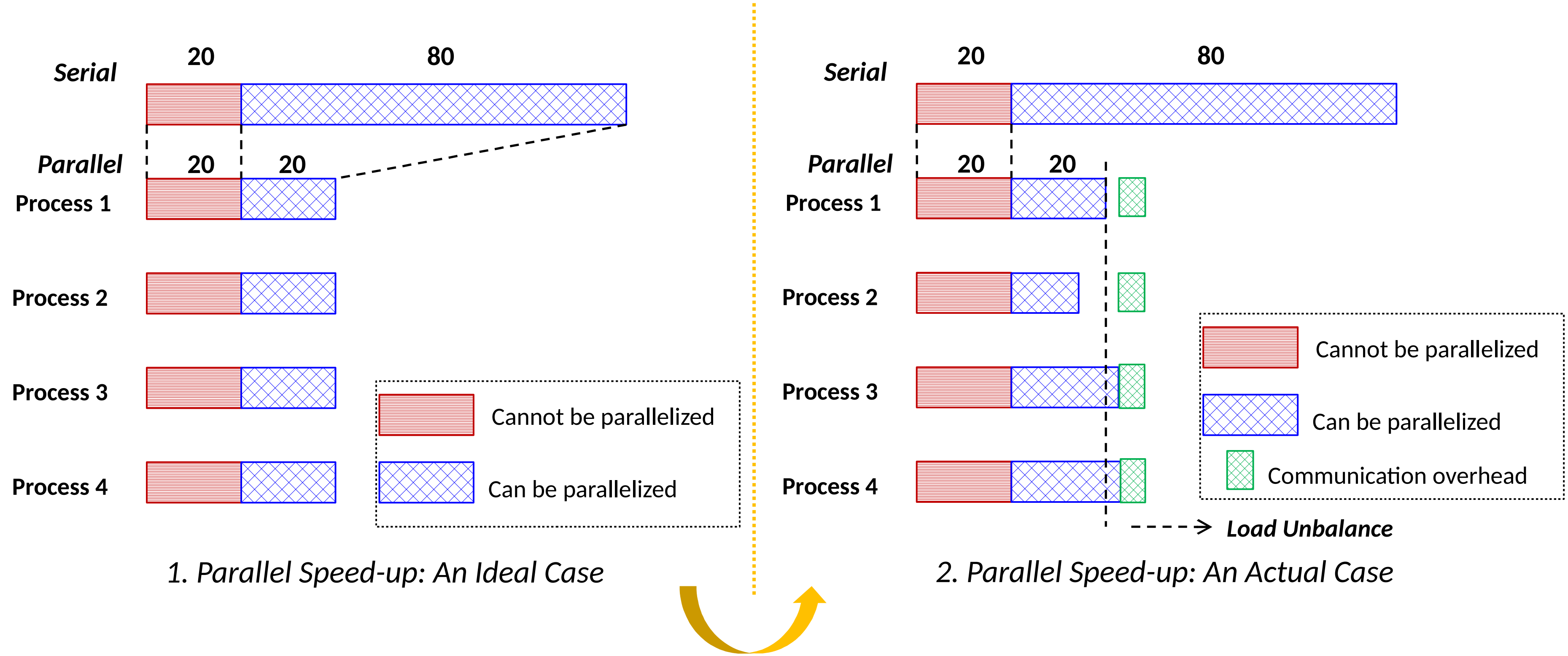
- Suppose that 80% of your program can be parallelized and that you use 4 processors to run your parallel version of the program
- The speedup you can get:

$$\frac{1}{s+\frac{1-s}{p}} = \frac{1}{0.2+\frac{0.8}{4}} = 2.5 \text{ times}$$

- Although you use 4 processors you cannot get a speedup more than 2.5 times!

# Amdahl’s Law: Real vs. Actual Cases

- Amdahl’s Law is too simple for real cases
- The communication overhead and workload imbalance among processes (in general) should be taken into account




*1. Parallel Speed-up: An Ideal Case*

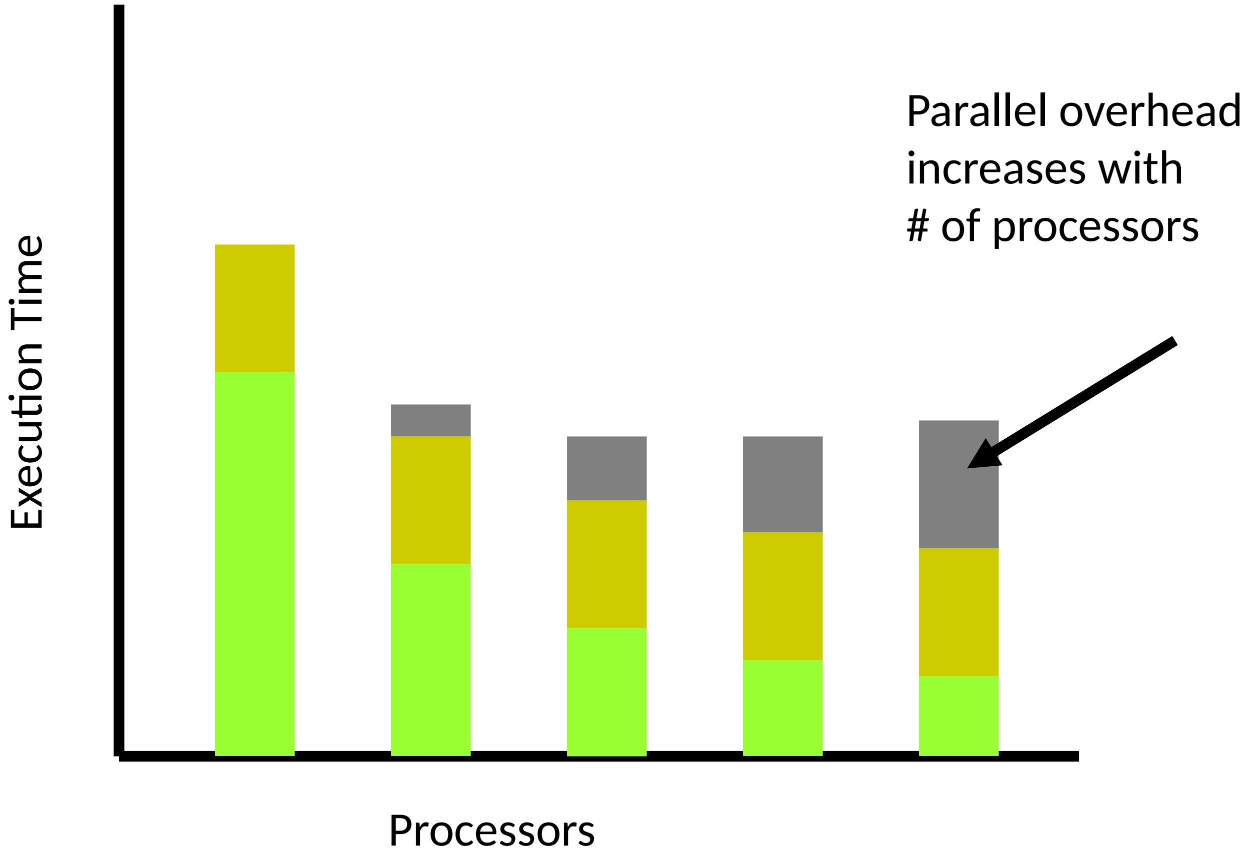
*2. Parallel Speed-up: An Actual Case*

# Amdahl’s Law Is Too Optimistic

Amdahl’s Law ignores parallel processing overheads:

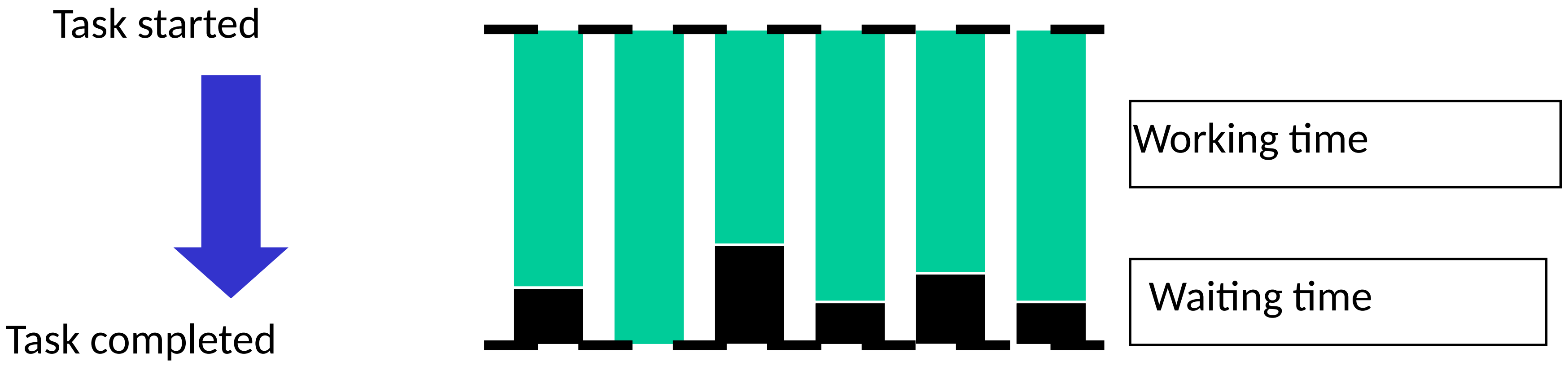
- The time for creating and terminating threads
- Parallel processing overhead is usually an increasing function of the number of processors
- Communication expenses

# Graph with Parallel Overhead Added

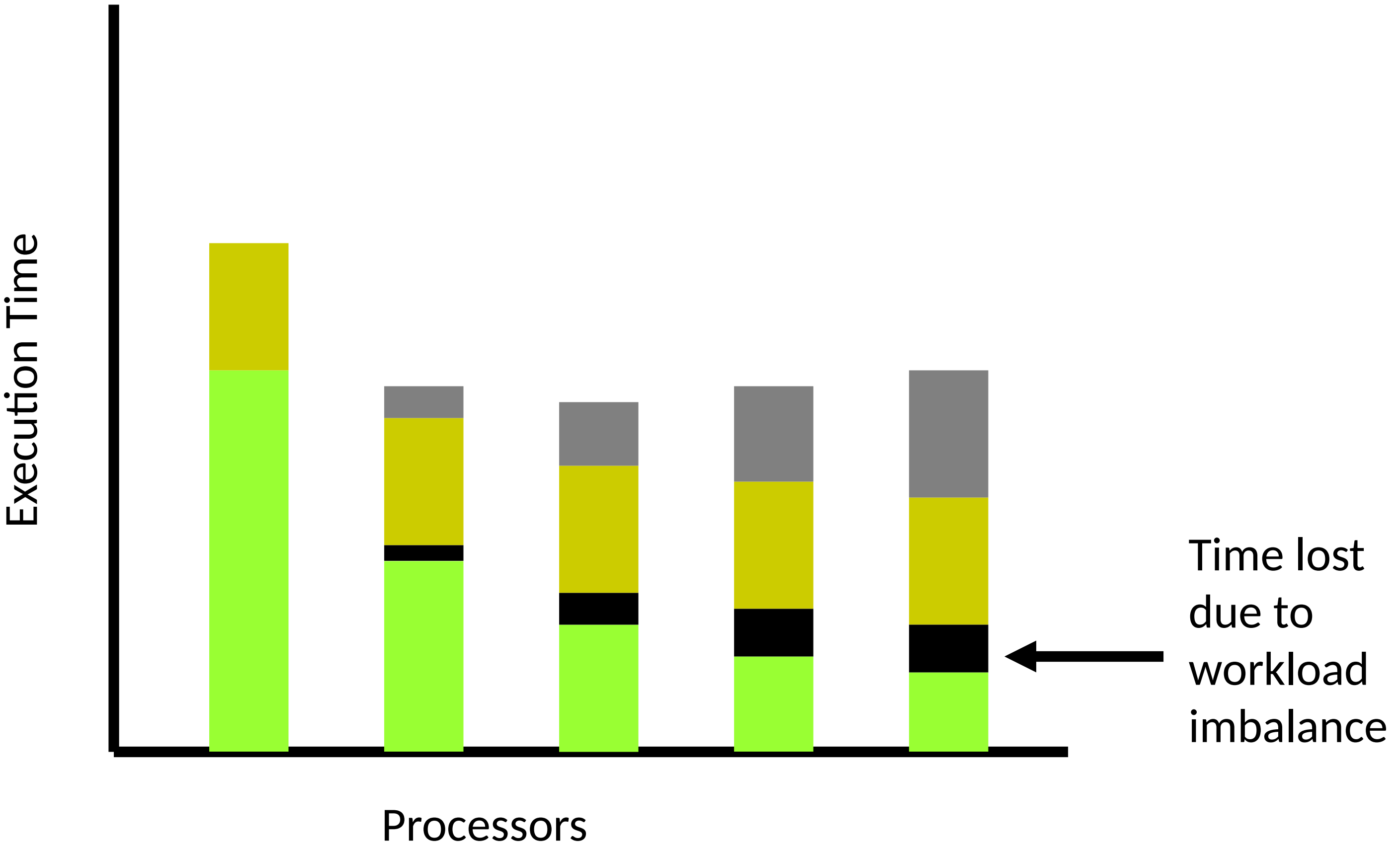


# Other Optimistic Assumptions

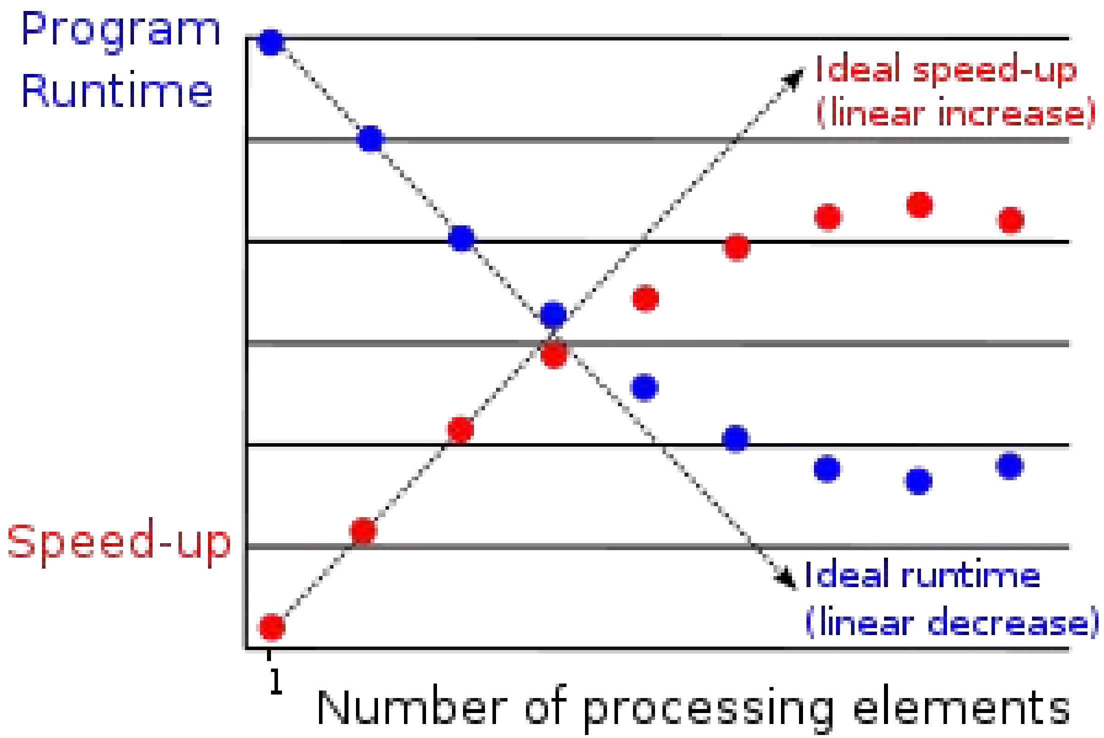
- Amdahl's Law assumes that the computation divides evenly among the processors
- In reality, the amount of work does not divide evenly among the processors
- Processor waiting time is another form of overhead



# Graph with Workload Imbalance Added



# Parallel Slowdown




A diagram of the program runtime (**shown in blue**) and program speed-up (**shown in red**) of a real-world program with sub-optimal parallelization. The dashed lines indicate optimal parallelization– linear increase in speedup and linear decrease in program runtime. Not: the runtime actually increases with more processors (and the speed-up likewise decreases) -> **this is parallel slowdown.**

# Types of Computing Problems

- **Embarrassingly parallel problem** - little or no effort is required to separate the problem into a number of parallel tasks. They are thus well suited to large, internet based distributed platforms (such as volunteer computing, like BOINC), and do not suffer from parallel slowdown. They require little or no communication of results between tasks, and are thus different from …

- **Distributed computing problems** - require communication between tasks, especially communication of intermediate results.

- **Inherently serial computing problems** - cannot be parallelized at all, and they are diametric opposite to embarrassingly parallel problems.

# More General Speedup Formula

$\psi(n,p)$ - speedup for problem of size *n* on *p* CPUs

$\sigma(n)$ - time in sequential portion of code for problem of size *n*

$\varphi(n)$ - time in parallel portion of code for problem of size *n*

**$\kappa(n,p)$ - parallel overheads**

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

# Amdahl's Law: Maximum Speedup

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

Assumes parallel work divides perfectly among available CPUs

This term is set to 0

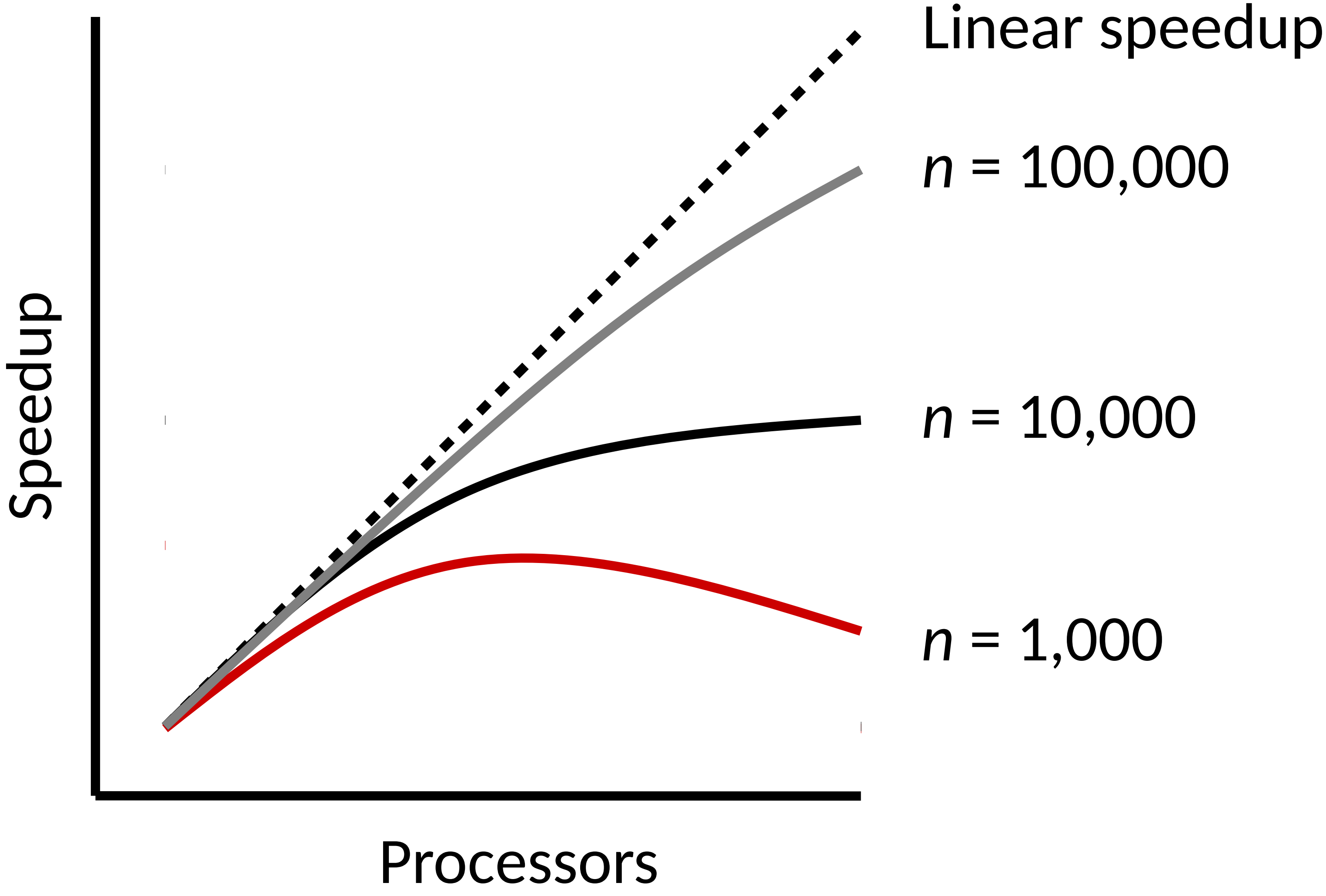
# The Amdahl Effect

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

**As $n \to \infty$ these terms dominate**

Speedup is an increasing function of problem size

# Illustration of the Amdahl Effect



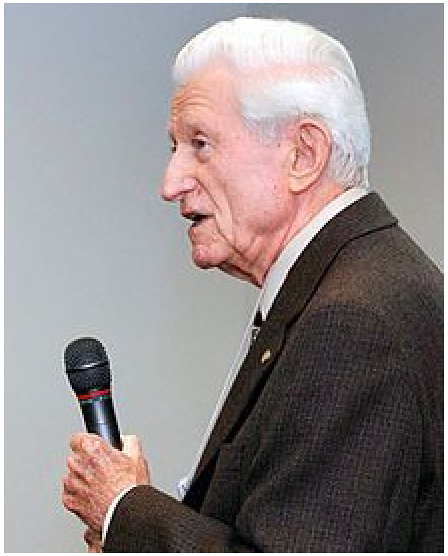
# Using Amdahl’s Law

- Program executes in 5 seconds
- Profile reveals 80% of time spent in some function, which we can execute in parallel
- What would be maximum speedup on 2 processors?

$$\psi \leq \frac{0.2 + 0.8}{0.2 + 0.8 / 2} = \frac{1}{0.6} \approx 1.67$$

- New execution time ≥ 5 sec / 1.67 = 3 seconds

# Gene Amdahl (1922-2015)



**On work in IBM**:

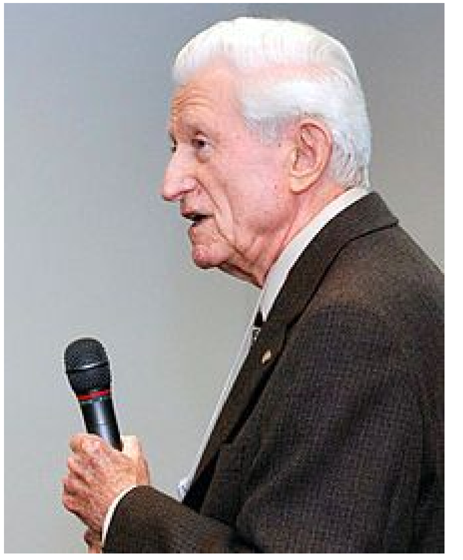
what I felt was that with that kind of an organization I'm not going to be in control of what I want to do any time in the future. It's going to be a much more bureaucratic structure. …

And I decided that **I didn't want to have that kind of life, basically**. … It was the way the structure was set up;

**I was going to be a peg-in-a-hole**.

# Gene Amdahl (1922-2015)

He left IBM again in September 1970, after his ideas for computer development were rejected, and **set up Amdahl Corporation** in Sunnyvale, California with aid from Fujitsu.



Competing with IBM in the mainframe market, the company manufactured **"plug-compatible" mainframes**.

In 1967 at the Spring Joint Computer Conference, Amdahl argued verbally and in three written pages, for performance limitations in any special feature or mode introduced to new machines
---> **Amdahl's law**.
**These arguments continue to this day.**

# Karp-Flatt Metric

# Karp-Flatt Metric: Example 1

- Suppose we benchmark a parallel program and get these speedup figures

| Processors | Speedup | Efficiency |
|---|---|---|
| 2 | 1.5 | 75% |
| 3 | 1.8 | 60% |
| 4 | 2 | 50% |

- Why is efficiency dropping?
- How much speedup could we expect on 8 processors?

# Deriving the Karp-Flatt Metric

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)}$$

- The denominator represents parallel execution time
- One processor does sequential code; others idle
- All processors incur overhead time
- **"Wasted time"** (when *p*-1 processors are idle):

$(p-1)\sigma(n) + p\kappa(n, p)$

- **"Experimentally determined serial fraction":**

"wasted time" divided by ($p$-1) times sequential time

# Karp-Flatt Metric – Comparison with Amdahl's Law – 1

Assume that:

- ***p*** - the number of processors in a system;
- ***T*(*p*)** - the total code execution time in a system with *p* processors;
- $T_s$ - the execution time of the serial part of the code;
- $T_p$ - the execution time of the **parallel** part of the code by one processor.

Then:

$$T(p) = T_s + \frac{T_p}{p}$$

# Karp-Flatt Metric – Comparison with Amdahl's Law – 2

Assume that we have a system with 1 processor, i.e. ***p*** =1:

Then from:

$$T(p) = T_s + \frac{T_p}{p}$$

we get $T(1) = T_s + T_p$ and if we define serial fraction $e = \frac{T_s}{T(1)}$

then we can obtain

$$T(p) = eT(1) + \frac{T(1) - T_s}{p} = eT(1) + \frac{T(1) - eT(1)}{p} = T(1)\left(e + \frac{1-e}{p}\right)$$

As far as speedup $\psi = \frac{T(1)}{T(p)}$ we get $\frac{1}{\psi} = e + \frac{1-e}{p}$

# Karp-Flatt Metric – Comparison with Amdahl's Law – 3

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

- The experimentally determined serial fraction ***e*** is a function of speedup ***ψ*** and the number of processors ***p***
- We can use ***e*** to determine whether efficiency decreases are due to
  - Sequential component of computation
  - Increases in overhead

# Interpretation of *e*

- If ***e*** is **constant** as the number of processors ***p*** increases, then speedup ***ψ*** is **constrained by the sequential** component of the computation
- If ***e*** is **increasing** as the number of processors ***p*** increases, then speedup ***ψ*** is **constrained by the parallel** overhead, such as
  - Thread creation/termination time
  - Contention for shared data structures
  - Cache-related inefficiencies
- **Often a combination** of these two factors is observed

# Return to the Previous Example: Constant *e*

| Processors ($p$) | Speedup ($\psi$) | Efficiency | $e$ |
|---|---|---|---|
| 2 | 1.5 | 75% | **0.33** |
| 3 | 1.8 | 60% | **0.33** |
| 4 | 2.0 | 50% | **0.33** |

- In this case, serial fraction $e$ is **constant**, then speedup $\psi$ is **constrained by** the relatively large amount of time spent in **sequential code**

# Example 2: Compute $\pi$

| Processors ($\boldsymbol{p}$) | Speedup ($\boldsymbol{\psi}$) | Efficiency | $\boldsymbol{e}$ |
|---|---|---|---|
| 2 | 1.87 | 93% | 0.070 |
| 3 | 2.60 | 87% | 0.078 |
| 4 | 3.16 | 79% | 0.089 |

The benchmark data for a parallel program computing value of $\pi$.

Let's predict speedup on 6 processors:

- Assume that *e* can be extrapolated to be equal to 0.11.
- Then speedup would be ~3.871…

# Example 2: Speedup Prediction Formula

$$e = \frac{1/\psi - 1/p}{1 - 1/p}$$

$$\Rightarrow \psi = \frac{p}{e(p-1)+1}$$

# Example 3: Increase the Number of Processors

- Assume that we benchmarked a sequential program, which spends 85% of its time in functions that can be re-written for parallel execution.
- Then we re-write these functions for parallel execution and run the program on a 2-processor system.
- The parallel program achieves a speedup of 1.67 on 2 processors.

**Question:** if we run the program on a 4-processor system, what kind of speedup should we expect?

# Example 3:
# Prediction Based on **Amdahl's Law**

$$\psi \leq \frac{1}{0.15 + (1 - 0.15)/4}$$

$$\Rightarrow \psi \leq 2.76$$

# Example 3: Prediction Based on Karp-Flatt Metric

- From Karp-Flatt formula $e = \dfrac{1/\psi - 1/p}{1 - 1/p}$ for ***p*** = 2, ***ψ***=1.67, we get ***e*** = 0.1976
- We know that the sequential part of code is 0.15, then the rest part of ***e*** is (0.0476) is related with some parallel overheads
- Assume that the parallel overheads increase linearly with number of processors ***p>1***, then it will be 0.0476(***p***-1)=0.1428 when ***p*** = 4
- Then we can predict that for ***p*** = 4: ***e*** = 0.15+0.1428 =0.2928
- Finally, for ***p*** = 4, from the reverse Karp-Flatt formula we can estimate speedup ***ψ***=2.1294≈2.13

$$\psi = \frac{p}{e(p-1)+1}$$

# Superlinear Speedup

- According to our general speedup formula, the maximum speedup a program can achieve on $p$ processors is $p$
- ***Superlinear speedup*** is the situation where speedup is greater than the number of processors used

- It means the computational rate of the processors is faster when the parallel program is executing
- Superlinear speedup is usually caused, because:
  - the cache hit rate of the parallel program is higher
  - data input/output operation is much lower
  - some data can be obtained principally earlier in parallel than in sequential regimes

# Isoefficiency Metric

# Isoefficiency Metric - Definition

- *Parallel system* – a parallel program executing on a parallel computer
- *Scalability of a parallel system* - a measure of its ability to increase performance as number of processors increases
- A *scalable system* maintains efficiency as processors are added
- *Isoefficiency* - a way to measure scalability

# Isoefficiency - Notations

- *n* - data size
- *p* - number of processors
- $T(n,p)$ - execution time, using p processors
- $\Psi(n,p)$ - speedup
- $\sigma(n)$ - inherently sequential computations
- $\varphi(n)$ - potentially parallel computations
- $\kappa(n,p)$ - communication operations
- $\varepsilon(n,p)$ - efficiency

# Isoefficiency - Concepts

- $T_0(n,p)$ - the total wasting time spent by processes doing work not done by sequential algorithm.

$$T_0(n,p) = (p-1)\sigma(n) + p\kappa(n,p)$$

- We want the algorithm to maintain a constant level of efficiency as the data size *n* increases. Hence, $\varepsilon(n,p)$ is required to be a constant.
- Recall that $T(n,1)$ represents the sequential execution time.

# Isoefficiency Relation

The main steps to derivation:

- Begin with speedup formula
- Compute total amount of overhead
- Assume efficiency remains constant
- Determine relation between sequential execution time and overhead

# Deriving Isoefficiency Relation

Determine overhead

$$T_o(n,p) = (p-1)\sigma(n) + p\kappa(n,p)$$

Substitute overhead into speedup equation

$$\psi(n,p) \le \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)}$$

# Deriving Isoefficiency Relation

Determine overhead

$$T_o(n,p) = (p-1)\sigma(n) + p\kappa(n,p)$$

Substitute overhead into speedup equation

$$\psi(n,p) \le \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)}$$

# Deriving Isoefficiency Relation

Determine overhead

$$T_o(n,p) = (p-1)\sigma(n) + p\kappa(n,p)$$

Substitute overhead into speedup equation

$$\psi(n,p) \le \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)}$$

Substitute $T(n,1) = \sigma(n) + \varphi(n)$ also in it

# Deriving Isoefficiency Relation

Determine overhead

$$T_o(n,p) = (p-1)\sigma(n) + p\kappa(n,p)$$

Substitute overhead into speedup equation

$$\psi(n,p) \le \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)}$$

Substitute $T(n,1) = \sigma(n) + \varphi(n)$ also in it**, and get:**

$$T(n,1) \ge CT_0(n,p)$$

where

$$C = \frac{\varepsilon(n,p)}{1-\varepsilon(n,p)}$$

# Deriving Isoefficiency Relation

$$\psi(n,p) \le \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)} =$$

$$= \frac{p}{p} \times \frac{\sigma(n)+\phi(n)}{\sigma(n)+\phi(n)/p+\kappa(n,p)} =$$

$$= \frac{p\left(\sigma(n)+\phi(n)\right)}{p\sigma(n)+\phi(n)+p\kappa(n,p)} =$$

$$= \frac{p\left(\sigma(n)+\phi(n)\right)}{\sigma(n)+\phi(n)+(p-1)\sigma(n)+p\kappa(n,p)} =$$

$$= \frac{p\left(\sigma(n)+\phi(n)\right)}{\sigma(n)+\phi(n)+T_0(n,p)} = \frac{pT(n,1)}{T(n,1)+T_0(n,p)}$$

# Isoefficiency Relation Usage

- Used to determine **the range of processors *p*** for which a given level of **efficiency *ε(n,p)* can be maintained**
- The way **to maintain a given efficiency *ε(n,p)*** is to **increase the problem size *n*** when the **number of processors *p* increase**.
- The **maximum problem size *n*** we can solve is limited by the **amount of memory *M*** available
- The **memory size *M*** is a constant **multiple** of the **number of processors *p*** for most parallel systems

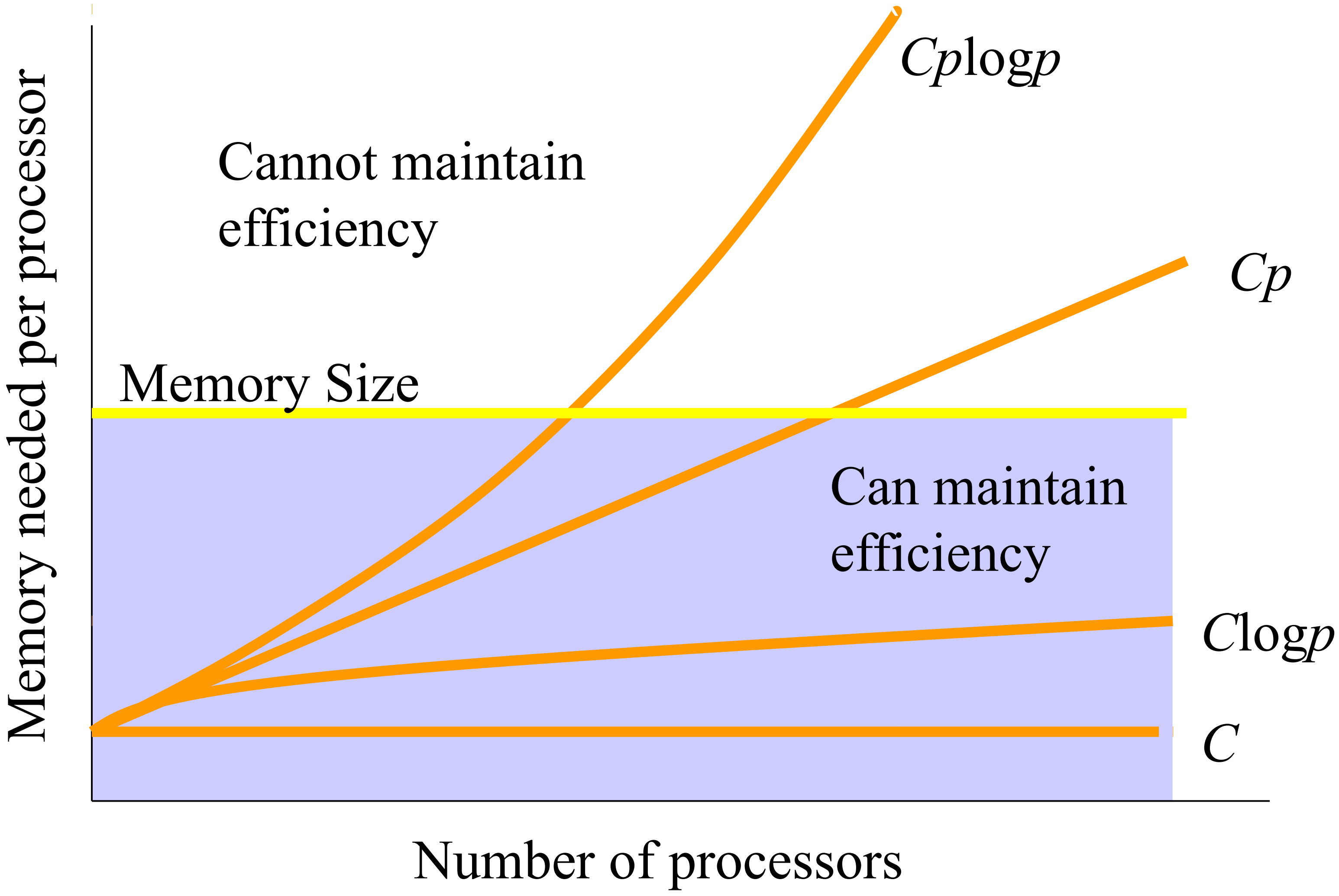
# The Scalability Function

- Suppose the isoefficiency relation can be transformed to $n \geq f(p)$, where $f$ is an **isoefficiency function**
- Let $M(n)$ is a memory required for problem of size $n$
- $M(f(p))/p$ characterizes how **memory** usage **per processor** must increase to maintain same efficiency
- $M(f(p))/p$ is called the **scalability function** [i.e., $scale(p) = M(f(p))/p$ ]

# Meaning of Scalability Function

- To maintain efficiency ***ε(n,p)*** when increasing ***p***, we must increase ***n***
- Maximum problem size ***n*** is limited by available memory ***M***, which increases linearly with ***p***: ***M*** ~ ***p***
- Scalability function ***scale(p)*** shows how memory usage per processor ***M(f(p))/p*** must grow to maintain efficiency ***ε(n,p)***
- If the scalability function ***scale(p)*** is a **constant** this means the parallel **system is perfectly scalable**

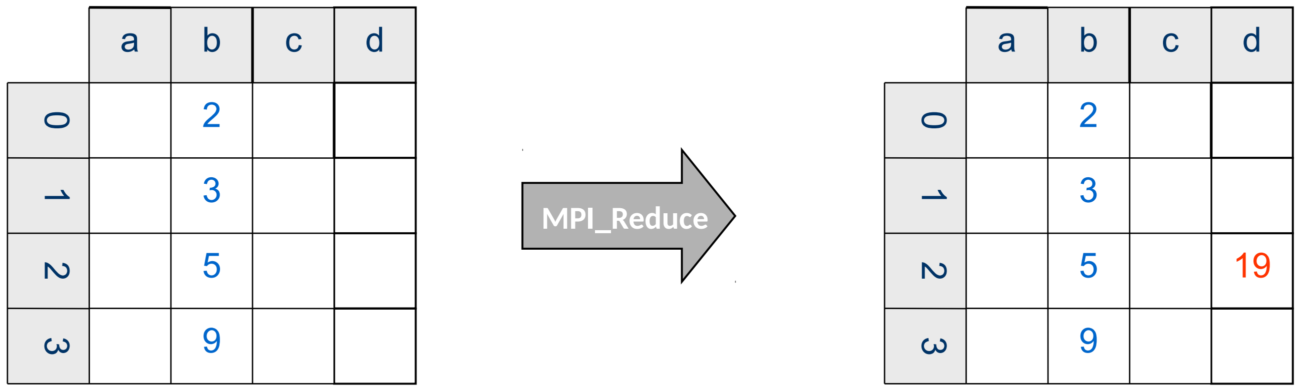
# Interpreting Scalability Function



# Examples

- Reduction task – collects the answers to all the sub-problems and combines them in some way to form the output.
- Floyd-Warshall Algorithm – the graph analysis algorithm for finding shortest paths in a weighted graph.
- Finite Difference Method - numerical methods for approximating the solutions to differential equations using finite difference equations to approximate derivatives

# Example 1: Reduction




- Sequential algorithm complexity $T(n,1) = \Theta(n)$
- Parallel algorithm
  - Computational complexity = $\Theta(n/p)$
  - Communication complexity = $\Theta(\log p)$
- Parallel overhead
  $T_0(n,p) = \Theta(p \log p)$

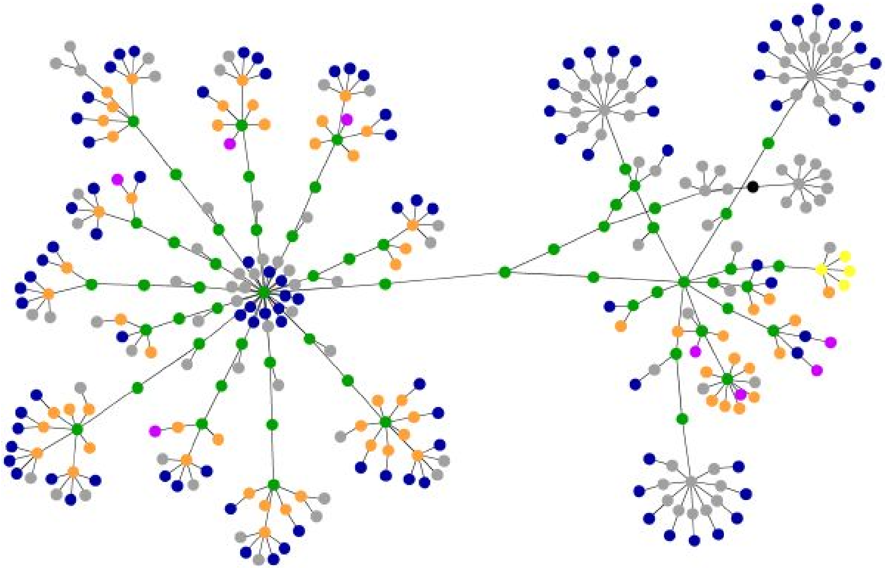
# Reduction (continued)

- Isoefficiency relation: $n \geq C\, p \log p$
- We ask: To maintain same level of efficiency ***ε(n,p)***, how must ***n*** increase when ***p*** increases?
- Since $M(n) \sim n$,

$$M(Cp\log p)/p \approx Cp\log p/p = C\log p$$

- The system has good scalability

# Example 2: Floyd-Warshall Algorithm



- Sequential time complexity: $\Theta(n^3)$
- Parallel computation time: $\Theta(n^3/p)$
- Parallel communication time: $\Theta(n^2 \log p)$
- Parallel overhead: $T_0(n,p) = \Theta(pn^2 \log p)$

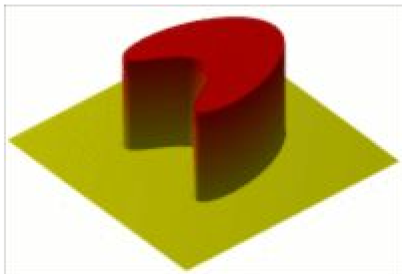
# Floyd-Warshall Algorithm (continued)

- Isoefficiency relation
  $n^3 \geq C(p\, n^2 \log p) \Rightarrow n \geq C\, p \log p$
- $M(n) = n^2$

$$M(Cp\log p)/p = C^2 p^2 \log^2 p / p = C^2 p \log^2 p$$

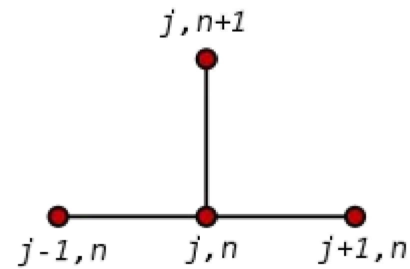
- The parallel system has poor scalability

# Example 3: Finite Difference Method



$$\frac{\partial u}{\partial t} - \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) = 0$$

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}.$$




- Sequential time complexity per iteration: $\Theta(n^2)$
- Parallel communication complexity per iteration: $\Theta(n/\sqrt{p})$
- Parallel overhead: $\Theta(n\sqrt{p})$

# Finite Difference Method (continued)

- Isoefficiency relation
  $n^2 \geq Cn\sqrt{p} \Rightarrow n \geq C\sqrt{p}$
- $M(n) = n^2$

$$M(C\sqrt{p})/p = C^2 p/p = C^2$$

- This algorithm is perfectly scalable

# Summary on Metrics

- Time
- Speedup
- Efficiency
- Cost
- Amdahl's Law: predict maximum speedup
- Karp-Flatt metric:
  - analyze parallel program performance
  - predict speedup with additional processors
- Isoefficiency metric: estimate scalability

# Guidelines

In order to organize parallel work <u>efficiently</u> developers need to follow these guidelines:

- Maximize the fraction of our program that can be parallelized
- Balance the workload of parallel processes
- Minimize the time spent for communication

# Contacts

Any course-related information (notifications, reports) from you:

send your message to my e-mail yuri.gordienko@gmail.com with the word **GPU2021** in the “Subject” field (if not, your message will be filtered out to Spam).