Performance Measures

By Nilesh Ghavate

Outline

- Performance Measures :
  - Speedup,
  - execution time,
  - efficiency,
  - cost,
  - scalability,
- Effect of granularity on performance,
- Scalability of Parallel Systems,
- Amdahl‘s Law, Gustavson‘s Law,
- Performance Bottlenecks.

Introduction

- Sequential algorithm -> execution time -> size of its input
- Parallel algorithm -> input size + no. of processors + IPC
  - Always there’s some loss in accuracy
- A parallel system is the combination of an algorithm and the parallel architecture on which it is implemented.

Sources of Overhead in Parallel Programs

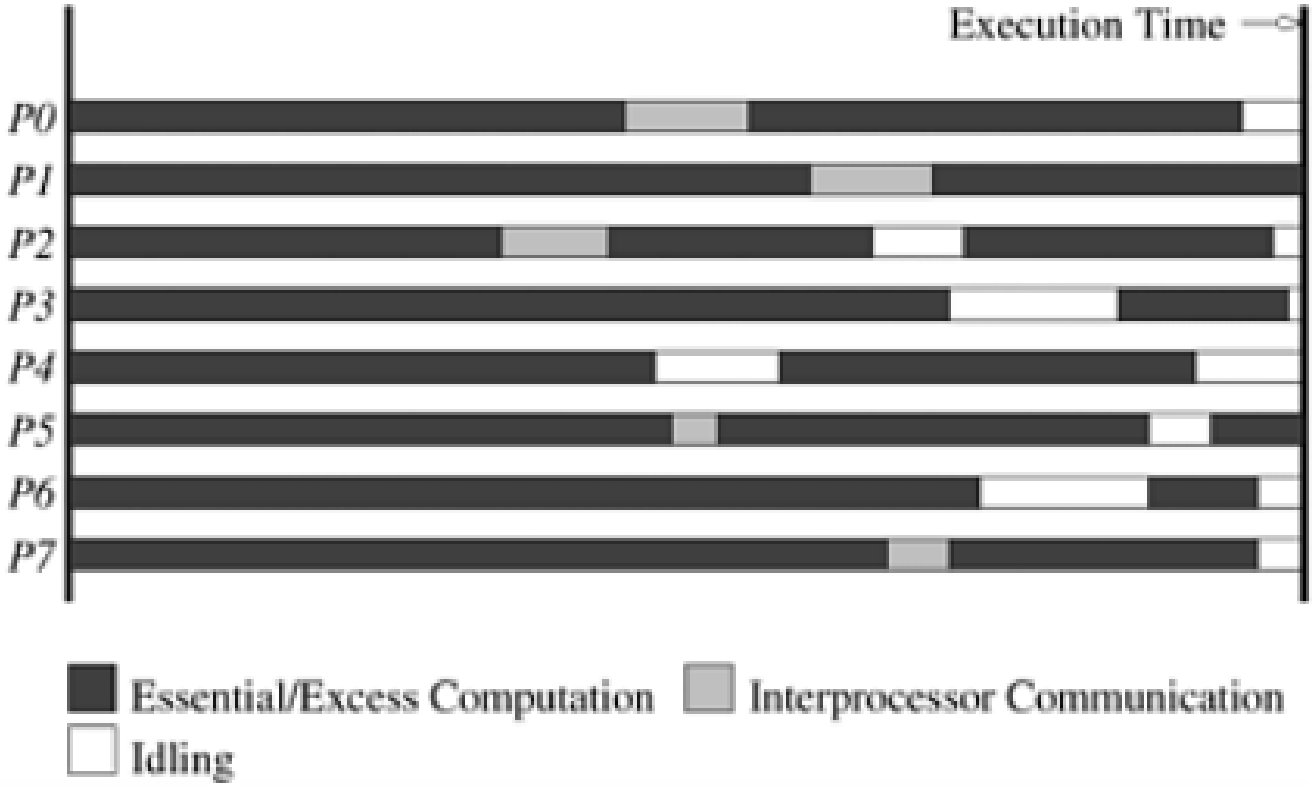


Fig: The execution profile of a hypothetical parallel program executing on eight processing elements

Sources of overhead

- 1) Interprocess Interaction (most significant)
- 2) Idling
  - arised due to improper load balance,
  - synchronization,
  - presence of serial components in a program.
- 3) Excess Computation
  - The difference in computation performed by the parallel program and the best serial program.

Performance Metrics

- Execution Time
- Total Parallel Overhead
- SpeedUp
- Efficiency
- Cost
- Scalability

Execution Time

- T_s : The serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer.

- T_p : The parallel runtime is the time that elapses from the moment a parallel computation starts to the moment the last processing element finishes execution.
- $T_{total} = p * Tp$

Total Parallel Overhead

- It is the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element.

$$To = (p \times Tp) - Ts$$

Speedup

- Performance gain is achieved by parallelizing.
- **Definition of Speedup:**
- Speedup (S) = Time taken by the serial algorithm / Time taken by the parallel algorithm.

- Speedup (S) = Ts /Tp

Example

Adding n numbers using n processing elements

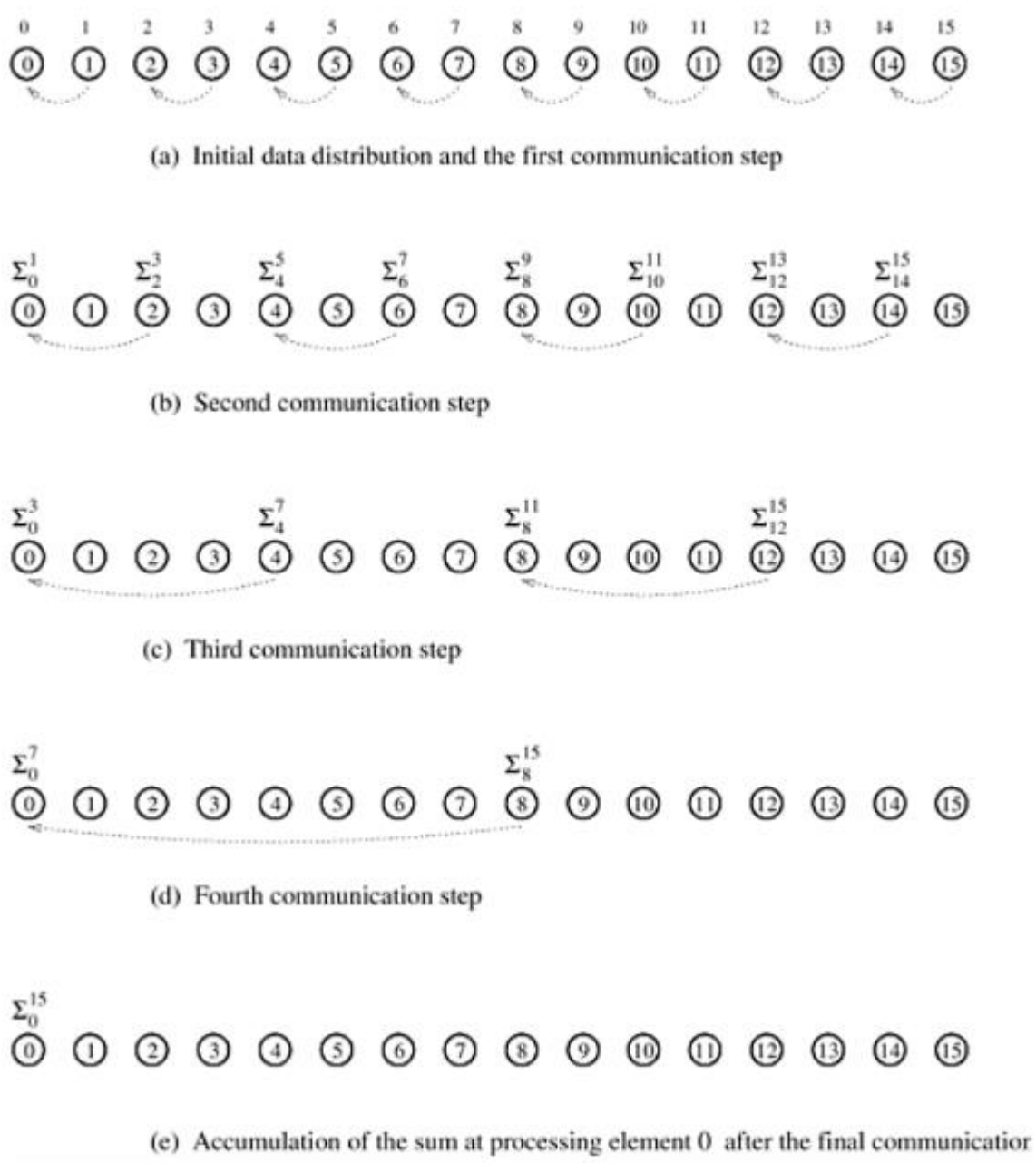
- Time for single processing unit : **O(n)**

16 processor & 16 numbers
Inter process communication : 15

Example
Adding n numbers using n processing elements

- We can perform this operation in log n steps by propagating partial Sums up a logical binary tree of processing elements

- Time for parallel processing unit : **O(log n)**



16 processor & 16 numbers
Inter process communication : 15

Computing the globalsum of 16 partial sums using 16 processing elements

Example
Adding n numbers using n processing elements

- Speedup : O(n) / O(log n)

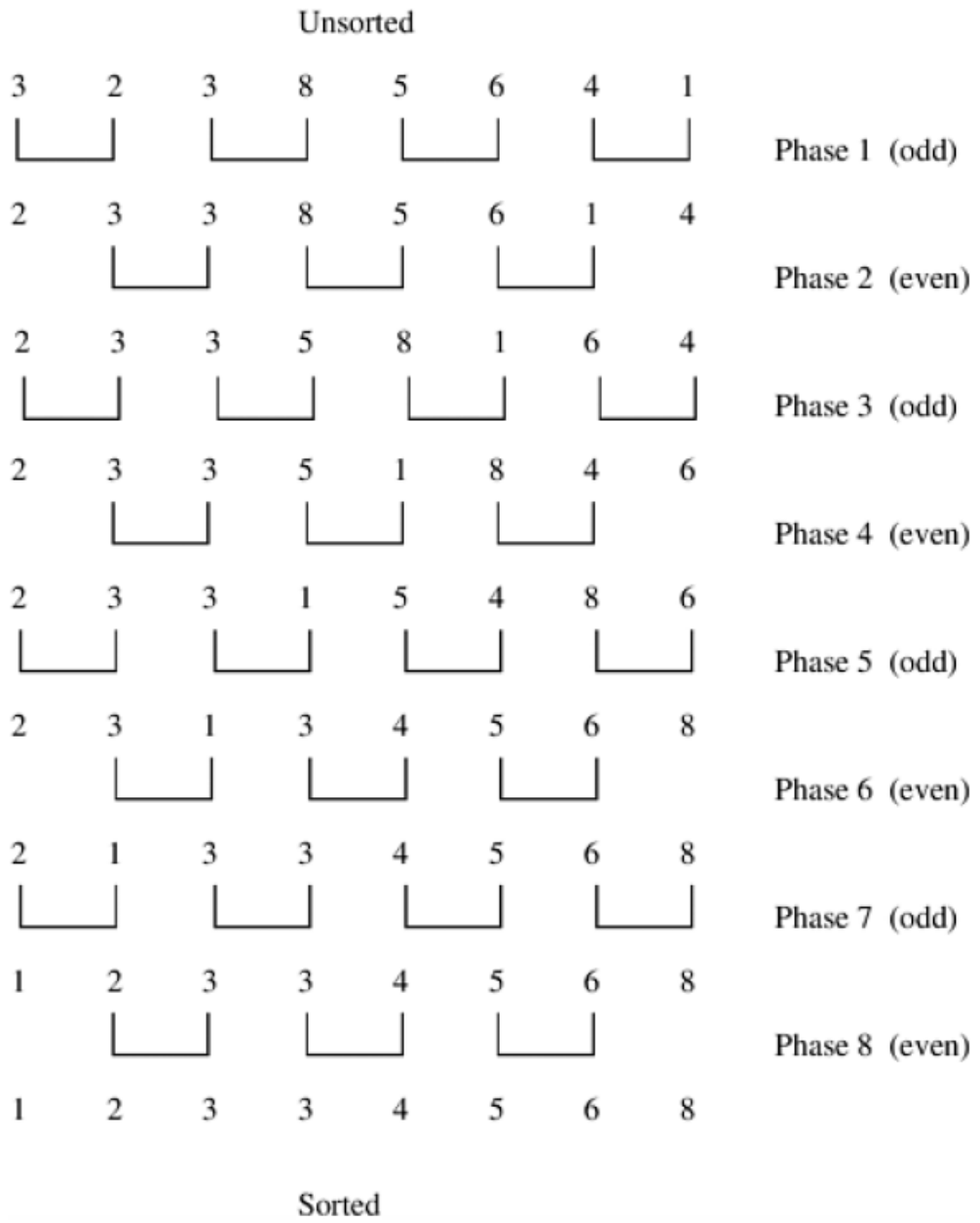
Example : Sorting elements

- **Serial Bubble Sort:** The given bubble sort takes **150 seconds** to sort **10^5** records.

- **Serial Quicksort:** An alternative sorting algorithm, quicksort, can sort the same records in **30 seconds**, which is significantly faster and represents the **best serial algorithm** for this case.

- **Parallel Odd-Even Sort:** The parallel version of bubble sort (odd-even sort) takes **40 seconds** using **4 processing elements**.

Figure 9.13. Sorting *n* = 8 elements, using the odd-even transposition sort algorithm. During each phase, *n* = 8 elements are compared.



Example : Sorting elements

- **Misleading Speedup Calculation:**
- If you compare the parallel odd-even sort to the serial bubble sort:
  - Speedup =150/40=**3.75.**
  - This comparison is invalid because bubble sort is **not the best serial algorithm** for this task.
- **Correct Speedup Calculation:**
- Speedup must be measured against the **best serial algorithm** (quicksort in this case):
  - Speedup =30/40=**0.75.**

Example : Sorting elements

- Speedup should always be measured relative to the **best serial algorithm** to reflect true performance gains.

- The parallel odd-even sort is less efficient than the best serial quicksort, achieving a **speedup < 1**.

- This highlights the importance of choosing optimal algorithms for both serial and parallel implementations.

Important points

- **Theoretical maximum speedup = P** (number of processing elements).
- **Maximum Speedup Condition:** Each processing element must work perfectly and take no more than **T_s/p time**. Assuming NO Overhead (partition , communication, synchronization overhead)
  - Serial running time = ts
  - Parallel running time = ts /p.
- **Speedup**

  S(n) = ts /(ts /p) =p (ideal linear speedup)

- **Why Speedup > P is Unrealistic:**
- Implies each processing element works faster than T_s/p.
- Suggests the problem could be solved in less than T_s time using a single processing element.
- This contradicts the definition of T_s as the runtime of the **best sequential algorithm**.

Important points

- Speedup greater than p violates theoretical bounds and is **not possible in practice**.
- Speedup S≤ P in realistic scenarios.

- **What is Superlinearity** ?
- **Superlinearity** refers to a situation in parallel computing where the **speedup** achieved by parallelizing a problem is **greater than the number of processing elements** used, or greater than the speedup predicted by a simple linear scaling model.

- **Initial Conditions for a Single Processor (Cache Hit 80%)**:
- A **single processor** solves a problem of size W.
- Cache hit ratio: **80%**, meaning 80% of memory accesses hit the cache, and the remaining 20% access main memory (DRAM).
- **Cache latency**: 2 ns and **DRAM latency**: 100 ns.
- **Effective memory access time (*Teff*)**:
- Calculation: 2×0.8+100×0.2=21.62 ns.

$$\text{Execution Rate (FLOPS)} = \frac{1}{T_{eff}} \times \text{FLOPS per access}$$

$$\text{Execution Rate (FLOPS)} = \frac{1}{21.6 \times 10^{-9}} = 46.3 \times 10^{6}\,\text{FLOPS} = 46.3\,\text{MFLOPS}$$

The **processing rate** is 46.3 MFLOPS (million floating-point operations per second) because each memory access corresponds to one FLOP.

- **Parallel Execution with Two Processors (Problem Size W/2 on Each Processor)**:
- Now, the problem is split into two processors, each working on **half of the problem size** W/2.
- Since the problem size is smaller, **cache hit ratio improves** due to the reduced data working set:
  - **Cache hit ratio**: 90%, which is higher than the single processor case (80%).
  - 8% of remaining data comes from **local DRAM** and 2% from **remote DRAM**.
  - **Remote DRAM latency**: 400 ns (due to communication between processors).

- **Effective memory access time(*Teff*):**
- Calculation: 2×0.9 + 100×0.08 + 400×0.02 =17.82 ns.

$$\text{Execution Rate (FLOPS)} = \frac{1}{17.8 \times 10^{-9}} = 56.18 \times 10^{6}\,\text{FLOPS} = 56.18\,\text{MFLOPS}$$

- The **execution rate** for each processor increases to 56.18 MFLOPS, leading to a **total execution rate** of **112.36 MFLOPS** (for both processors combined).

- **Superlinear Speedup:**
- 112.36 / 46.3 = 2.43 , which is greater than 1 (executing rate not time /speed)

Or

- 21.6 *10^9 / 17.8 * 10^9 = 1.213 (Speed), which is greater than 1

- **Cause of superlinear speedup:**
- The **improved cache hit ratio** from reducing the problem size per processor leads to better memory access times, which accelerates computation beyond the expected linear speedup.

Efficiency

- **Efficiency** in High-Performance Computing (HPC) is a measure of how effectively the computational resources (such as processors, memory, etc.) are utilized to solve a given problem.
- It helps to quantify the performance of a parallel system relative to the ideal performance.

Formula for Efficiency:

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Number of Processors (p)}}$$

Efficiency

- **Ideal Efficiency**:
- The ideal efficiency is **1** (or 100%), meaning there is **no overhead**, and each processor contributes equally to the computation.
- For this to happen, the workload must be perfectly divisible with no communication overhead between processors.

Efficiency

- **Real Efficiency**:
- In practice, efficiency is often less than 100% because of various factors such as:
  - Communication overhead
  - Synchronization overhead
  - Load imbalance
  - Memory contention

Example of Efficiency:

1. **Serial execution** (1 processor):
   - $T_S = 100$ seconds (time taken by 1 processor).
2. **Parallel execution** (4 processors):
   - $T_P = 30$ seconds (time taken by 4 processors).
   - Speedup $S = \frac{T_S}{T_P} = \frac{100}{30} = 3.33$.
   - Efficiency $E = \frac{S}{p} = \frac{3.33}{4} = 0.83$ or 83%.

This means that even though four processors are used, the system is operating at 83% efficiency. The **remaining 17%** is likely due to communication overhead, load imbalance, or other factors.

Cost

- **Cost of Solving a Problem on a Single Processor**: execution time of the fastest known sequential algorithm.
- **Cost of Solving a Problem on a Parallel System**:

Formula:

$$\text{Cost} = \text{Parallel Runtime} \times \text{Number of Processors (p)}$$

- This reflects the **total time** spent by all processing elements to solve the problem, and it measures the **work done by the system**.

Cost-Optimal Parallel System

- A cost-optimal parallel system is one where the cost of solving the problem on the parallel system **grows asymptotically** at the **same rate** as the sequential execution time (i.e., the fastest-known sequential algorithm).
- This means that, **for large problem sizes**, the parallel system **will scale efficiently** and the **increase in cost will be proportional** to the problem size.

Cost-Optimal Parallel System:

The equation for cost-optimality can be represented as:

$$\text{Cost (Parallel)} = T_P \times p$$

Efficiency of a Cost-Optimal Parallel System:

- The cost-optimal parallel system to maintain **Q(1)** efficiency means that the **efficiency of the parallel system remains constant** even as the number of processing elements (processors) increases. In other words, the system continues to achieve the same level of efficiency, regardless of how many processors are added.
- There is **no performance degradation** as more processors are added.

Example of a Q1 Efficiency:

Imagine a parallel system that can solve a problem using p processors in **parallel time** T_P. For a **cost-optimal system**:

- When **1 processor** is used, the serial execution time T_S might be, for example, 100 seconds.
- With **2 processors**, the parallel execution time T_P might drop to 50 seconds.
- With **4 processors**, the parallel execution time T_P might drop to 25 seconds, and so on.

In an ideal scenario:

- **Serial execution time**: $T_S = 100$ seconds.
- **Cost for 1 processor**: $T_S \times 1 = 100 \times 1 = 100$.
- **Cost for 2 processors**: $50 \times 2 = 100$.
- **Cost for 4 processors**: $25 \times 4 = 100$.

There are no significant overheads from communication or synchronization, allowing the system to maintain **high scalability** and **optimal performance**.

Scalability

- **Scalability** refers to the ability of a system to handle a growing amount of work or its potential to be enlarged to accommodate that growth.
- In the context of HPC, scalability is the system's capacity to maintain or improve performance as more computational resources (like processors or nodes) are added to solve larger or more complex problems.

Types of Scalability in HPC:

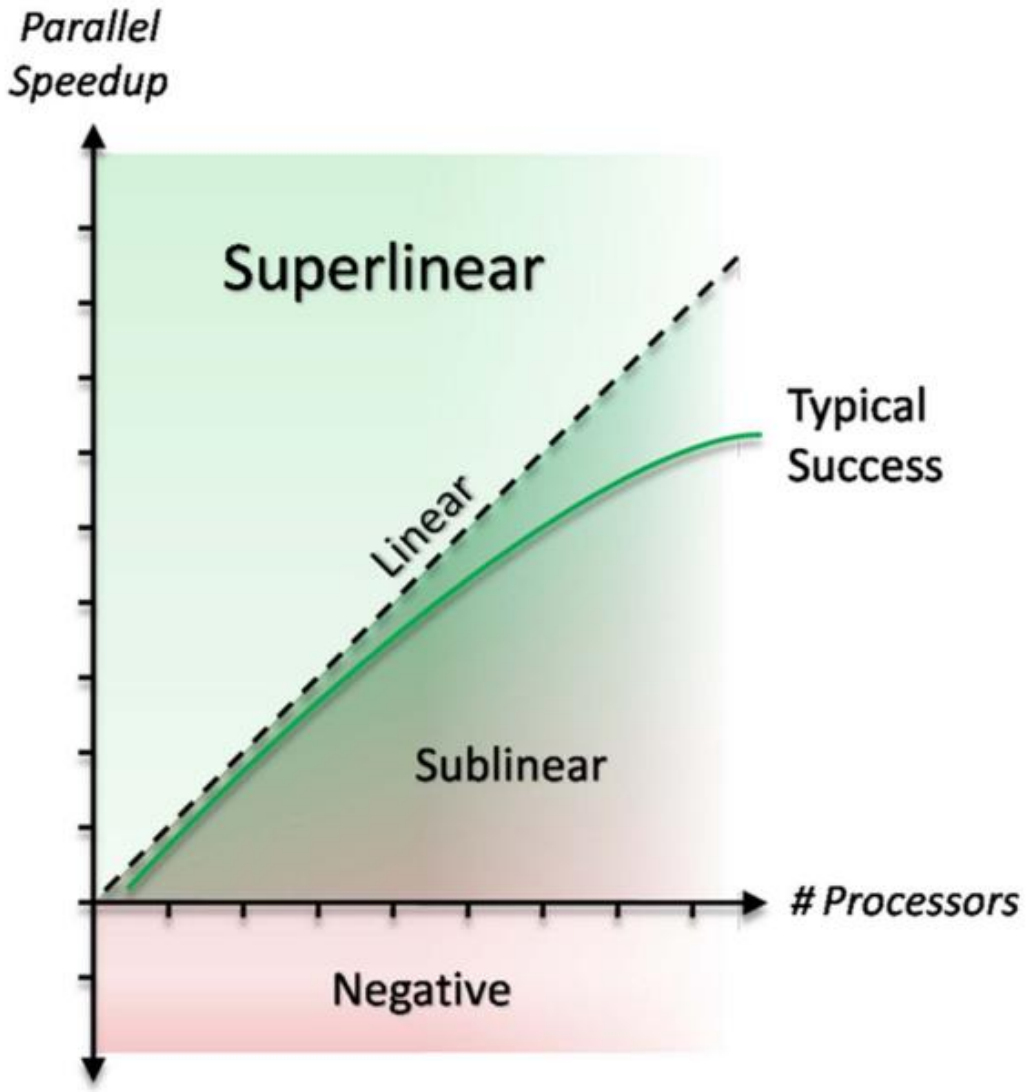
- **Strong Scalability**:
- **Definition**: Strong scalability measures how the parallel system performs as the **problem size remains constant**, but the number of processors (or computational resources) increases.
- **Goal**: The goal is to **reduce the execution time** of the fixed-size problem by distributing it across more processors.
- **Ideal Scenario**: In an ideal case, the execution time should decrease **linearly** as more processors are added.
- **Example**: Sorting a large dataset.
- **Scenario**: The problem size (dataset) remains constant, and you add more processors to speed up the sorting.

Types of Scalability in HPC:

- **Weak Scalability**:
- **Definition**: Weak scalability measures how the parallel system handles a **larger problem size** when the number of processors increases.
- **Goal**: The goal is to **solve a larger problem in the same amount of time by increasing both the problem size and the number of processors**.
- **Ideal Scenario**: In an ideal case, the problem size should scale proportionally with the number of processors, and the execution time should remain roughly constant as more resources are added.
- **Example**:Increasing the size of a matrix in a numerical simulation while adding more nodes to keep the execution time constant.

Types of Scalability in HPC:

- **Linear Scalability**:
- **Definition**: Linear scalability means that the execution time decreases **inversely proportional** to the number of processors.
- **Ideal Scenario**: If you double the number of processors, the time required to solve the problem is halved.
- **Example**: A perfect case where the system's speedup is exactly proportional to the number of processors.


Parallel
Speedup
Superlinear
Linear
Typical
Success
Sublinear
Processors
Negative

Types of Scalability in HPC:

- **Superlinear Scalability**:
- **Definition**: Superlinear scalability occurs when the system's performance increases **more than linearly** as the number of processors increases. This is unusual but can happen under certain conditions, such as:
  - **Reduced memory access time**
  - **Optimized memory access patterns**
- **Example**: A parallel system that exhibits faster-than-expected performance as more processors are added, often due to improved cache efficiency or better load balancing.

Types of Scalability in HPC:

- **Sublinear Scalability**:
- **Definition**: Sublinear scalability occurs when the system's performance increases at a rate **slower than linear** as processors are added. This indicates inefficiency, where adding more resources does not significantly improve performance.
- **Causes**:
  - **Communication overhead**
  - **Synchronization overhead**
  - **Memory contention**

Speedup Analysis

- Recall speedup definition: $\Psi(n,p) = t_s/t_p$
- A bound on the maximum speedup is given by

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

  - Inherently sequential computations are $\sigma(n)$
  - Potentially parallel computations are $\varphi(n)$
  - Communication operations are $\kappa(n,p)$

  - The "≤" bound above is due to the assumption in formula that the speedup of the parallel portion of computation will be exactly *p.*

  - Note $\kappa(n,p) = 0$ for SIMDs, since communication steps are usually included with computation steps.

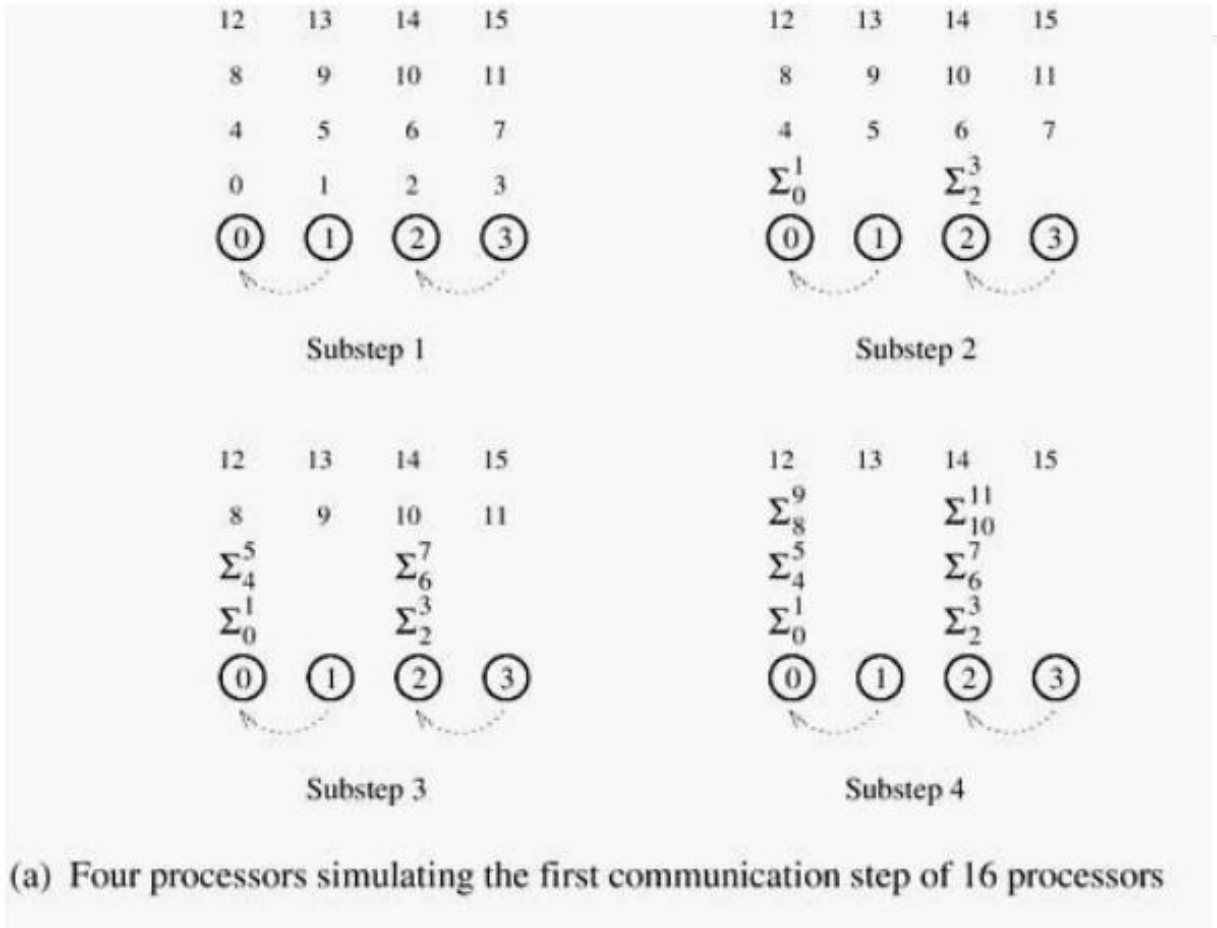
The Effect of Granularity on Performance

- **Initial Parallel Algorithm**:
  - The algorithm assumes one processing element per input, which is excessive in real-world scenarios.
  - Instead, we group multiple input elements per processing element, increasing the **granularity** of computation.
- **Scaling Down a Parallel System:**
  - Instead of having **n processing elements for n inputs**, we use **only p processing elements (p < n).**
  - Each processing element now handles n/p inputs.

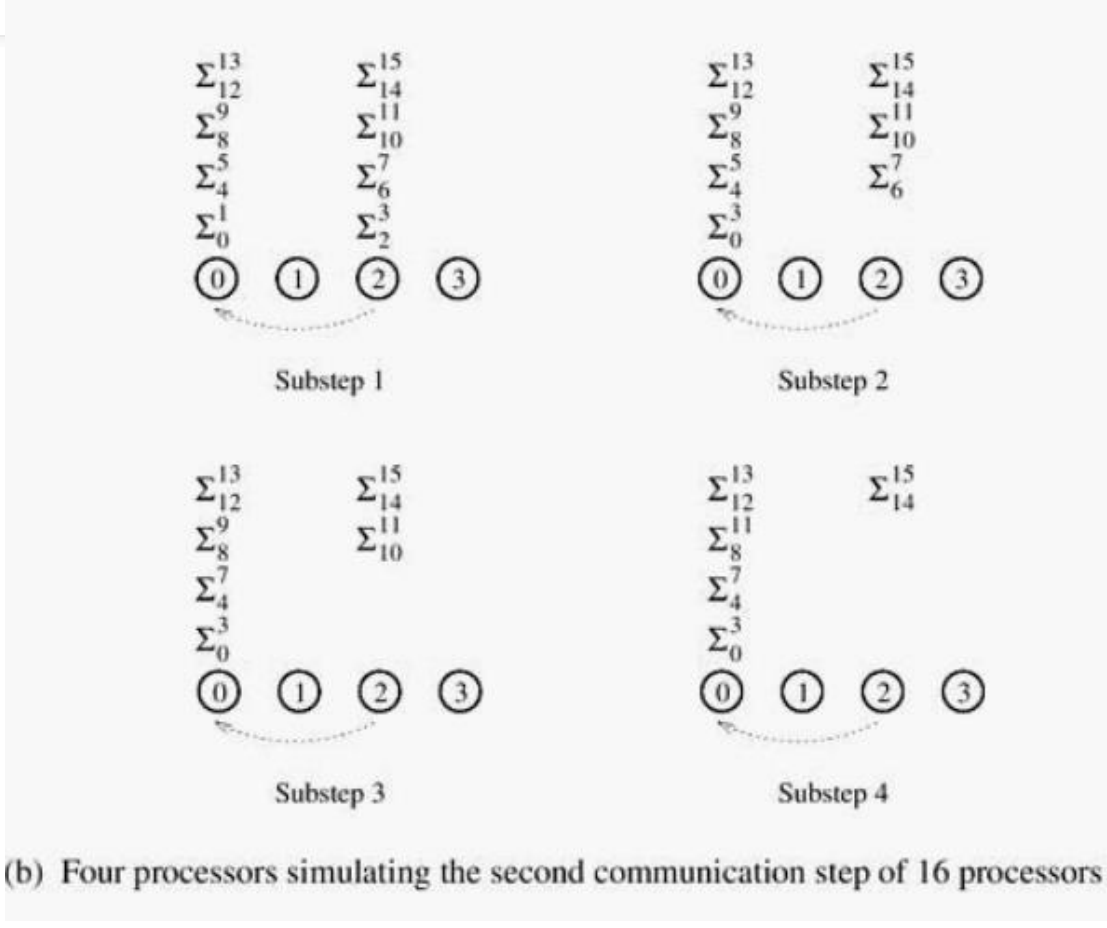
The Effect of Granularity on Performance

- **Impact on Computation & Communication:**
  - Each **processing element does more work**, increasing its computational load by a factor of **n/p** i.e (1* n/p).
  - Communication time decreases by the factor of n/p.
  - Computation increases by the factor of n/p.
  - So the total **parallel runtime** also increases by at most a factor of **n/p**, meaning the system does not become inefficient.
- **Cost-Optimality**:
  - If the original system with **n parallel processing elements** was cost-optimal, then only scaled-down system remains cost-optimal.
  - However, if the original system was **not cost-optimal**, increasing computational granularity might not fix the issue.

Illustrative Example – Adding n Numbers with p processor

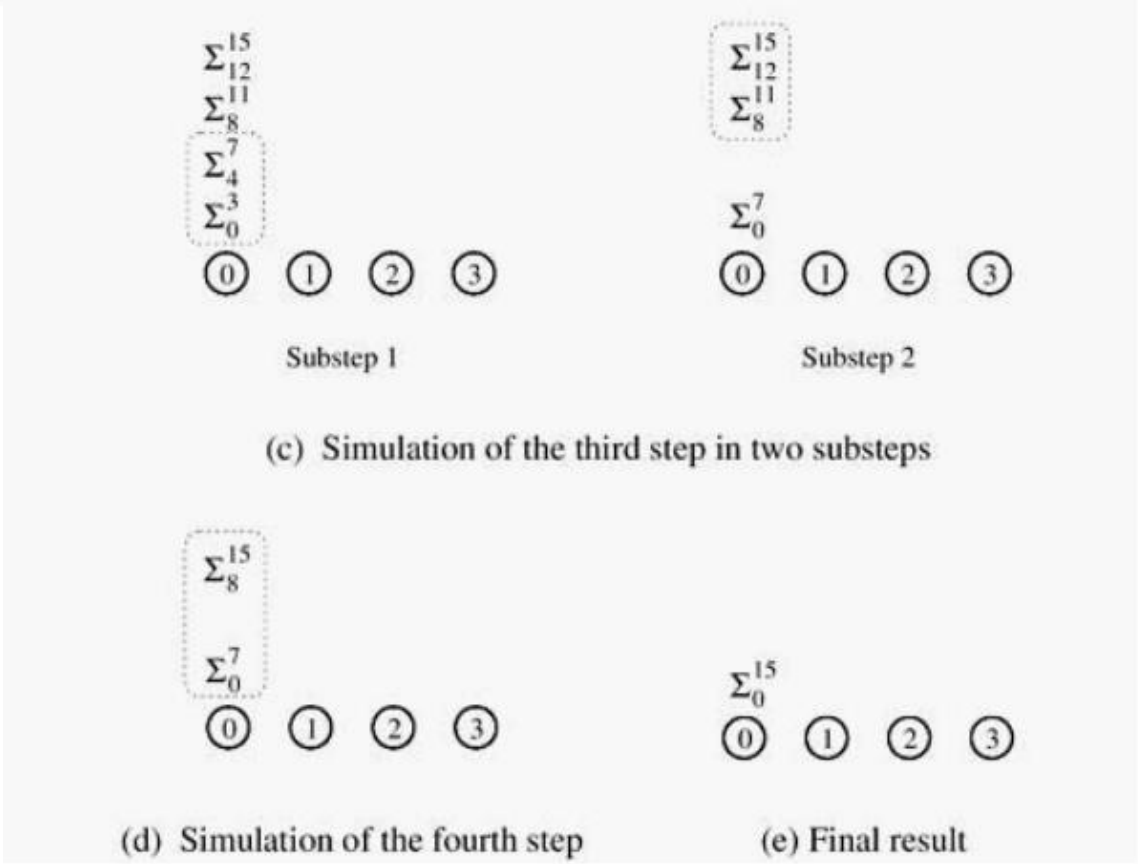


(a) Four processors simulating the first communication step of 16 processors



(b) Four processors simulating the second communication step of 16 processors

Illustrative Example – Adding n Numbers with p processor



p=4
n=16
4 processor & 16 numbers
Inter process communication : 12

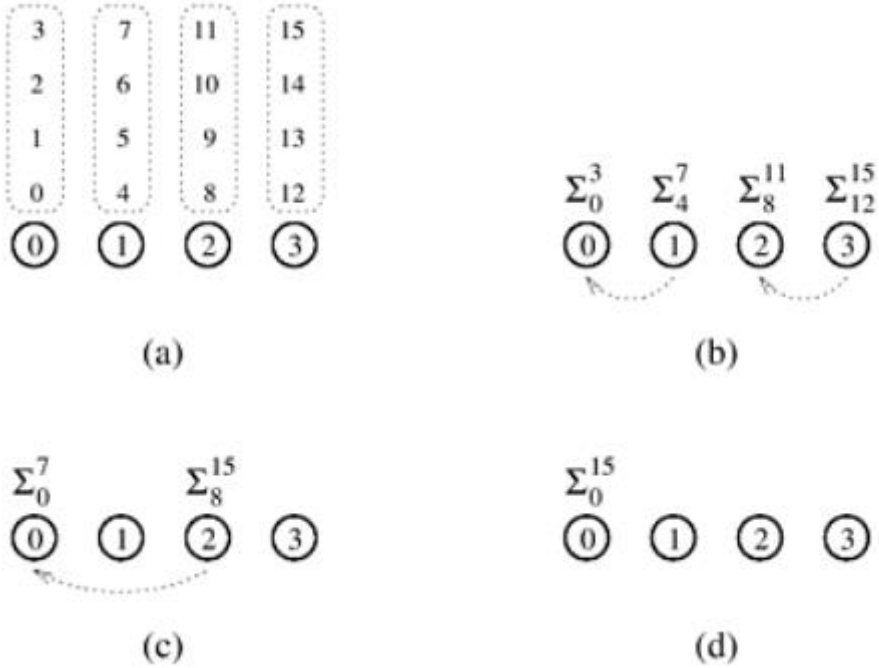
This reduction in communication was not enough to make the algorithm cost-optimal.

Illustrative Example – Adding n numbers cost-optimally

In the first step of this algorithm, each processing element locally **adds its n/p numbers in time Θ(n/p)**. Here it is Θ(4).

Summing the Partial Sums in Parallel, since each step **halves the number of sums**, the total number of steps required is **$\log_2 p$**, which can be done in time Θ(log p)

Total time is Θ(n/p + log p)



p=4
n=16
4 processor & 16 numbers
Inter process communication : 3

The Effect of Granularity on Performance

- **Cost Analysis**

  Cost is defined as the total work done across all processing elements, i.e.,

  Cost = Parallel Runtime × Number of Processors

  $C = T_p \times p$

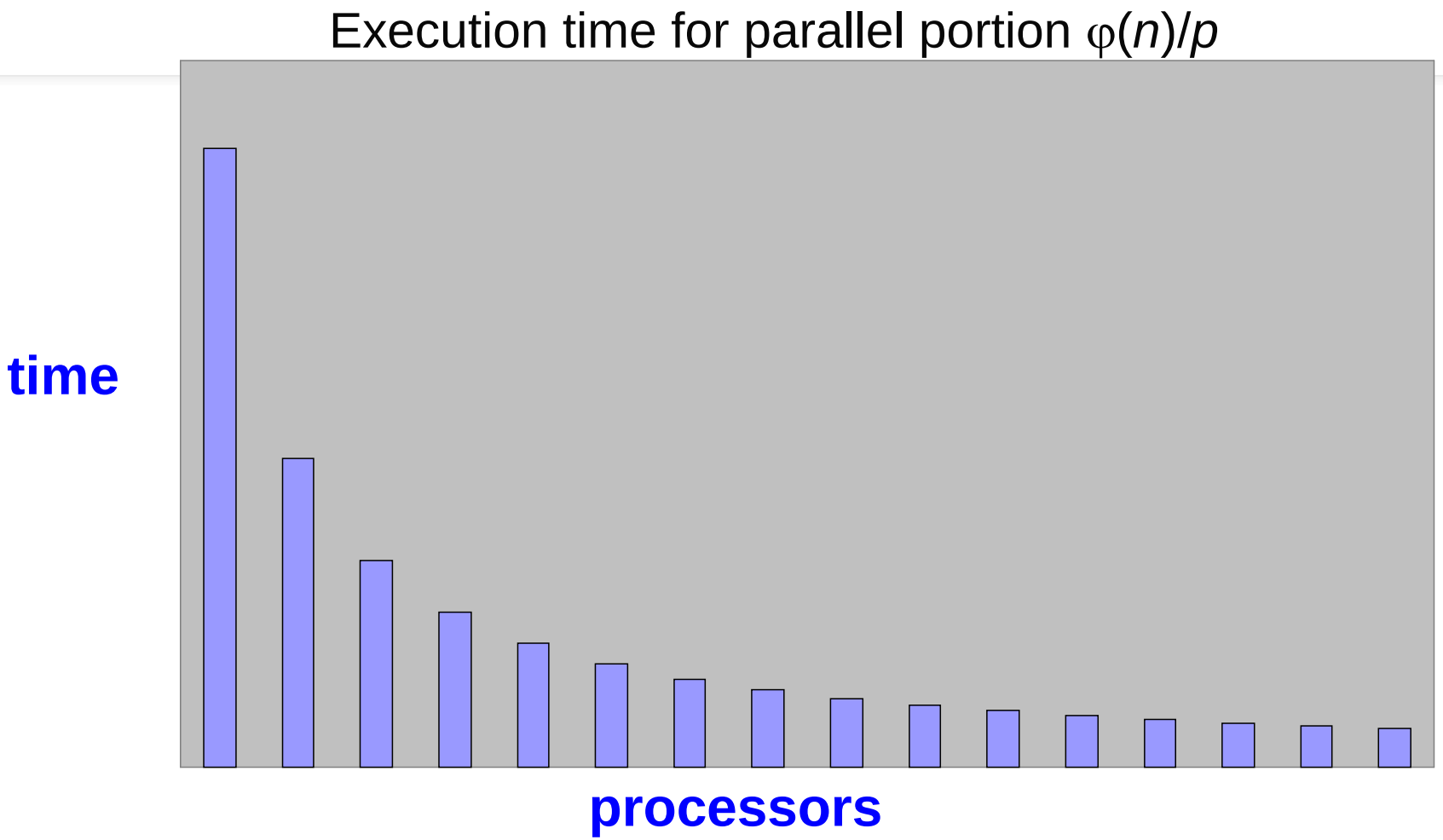
  $C = \Theta(n/p + \log p) \times p = \Theta(n + p \log p)$

- **Cost-Optimality Condition**
- A parallel algorithm is **cost-optimal** if its total cost matches the best-known **serial runtime**, which is **Θ(n) for summation**.
- This holds **if and only if**:

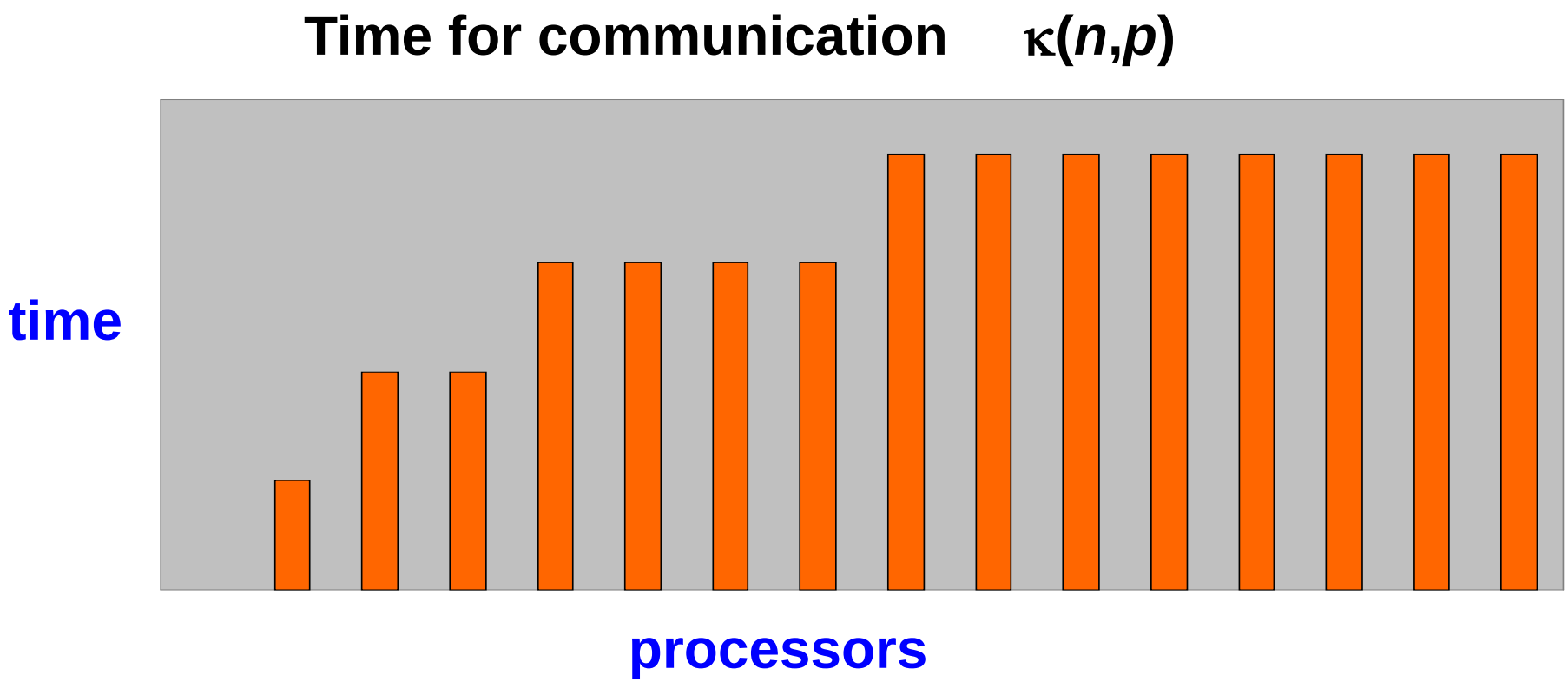
  $n=\Omega(p \log p)$

  This condition ensures that the additional overhead from parallelism (p logp) does not dominate the cost.

The Effect of Granularity on Performance

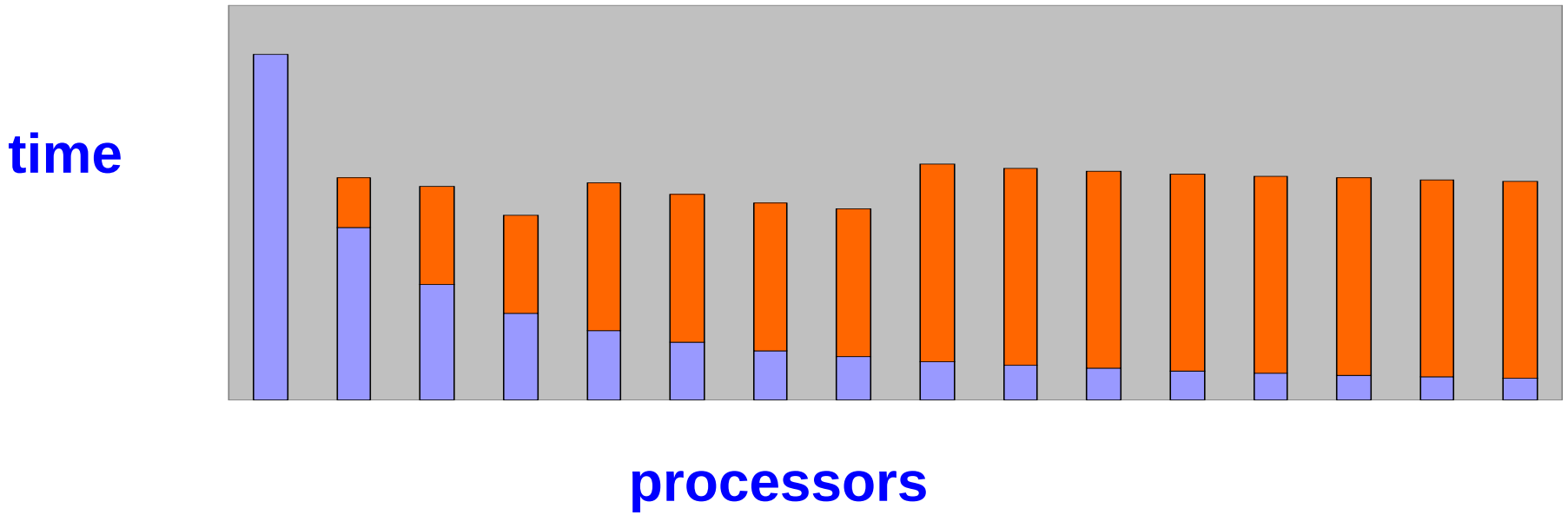


The Effect of Granularity on Performance

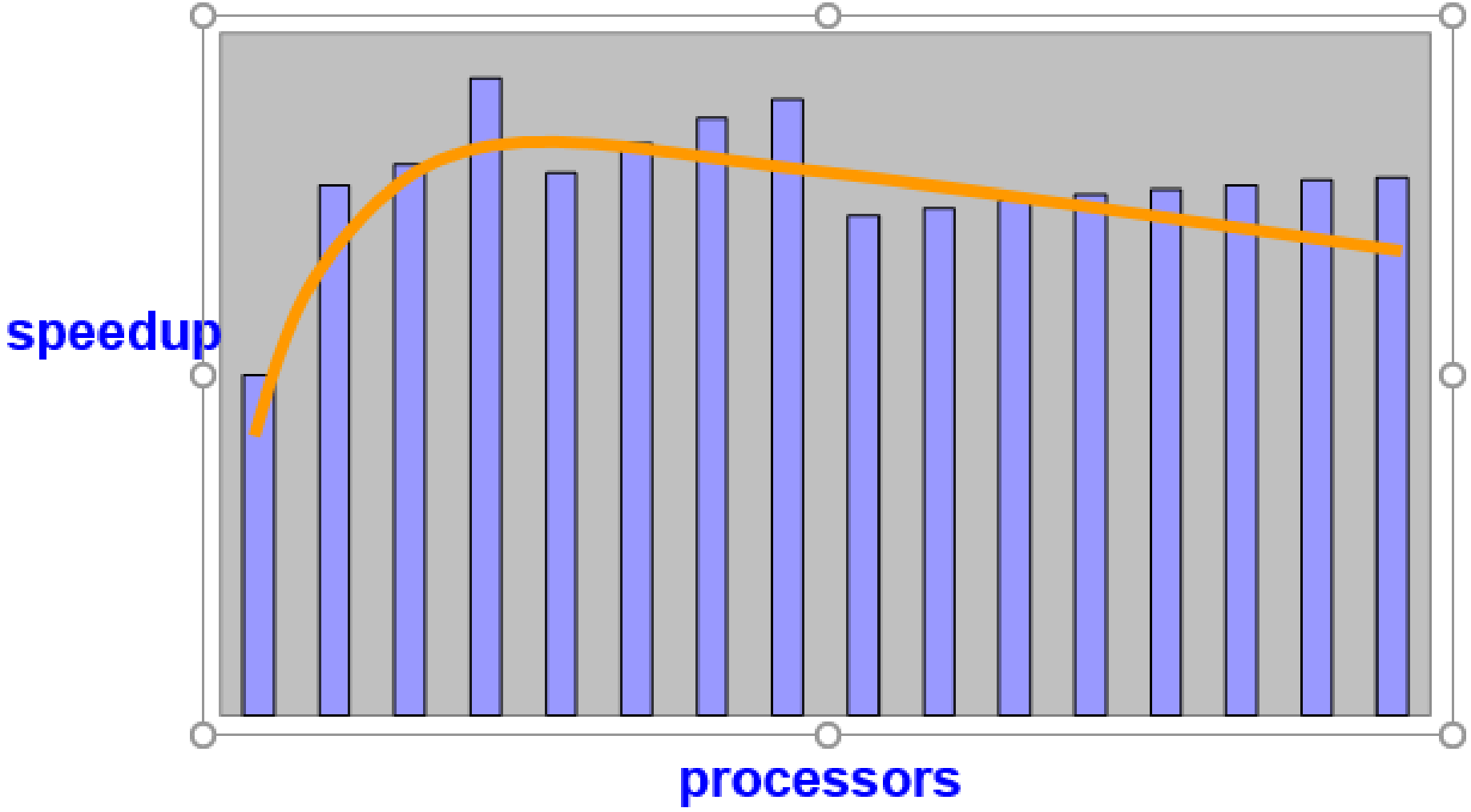


The Effect of Granularity on Performance

Execution Time of parallel portion $\varphi(n)/p + \kappa(n,p)$



Scalability of Parallel Systems



Scalability of Parallel Systems -Amdahl's Law

Let *f* be the fraction of operations in a computation that must be performed sequentially, where $0 \le f \le 1$.

The maximum speedup ψ

achievable by a parallel computer with *n* processors is

$$\psi \equiv S(n) \le \frac{1}{f + (1 - f)/n} \le \frac{1}{f}$$

Usual Argument: If the fraction of the computation that cannot be divided into concurrent tasks is *f*, and no overhead incurs when the computation is divided into concurrent parts, the time to perform the computation with *n* processors is given by $t_p \geq ft_s + [(1 - f)t_s] / n$, as shown below:

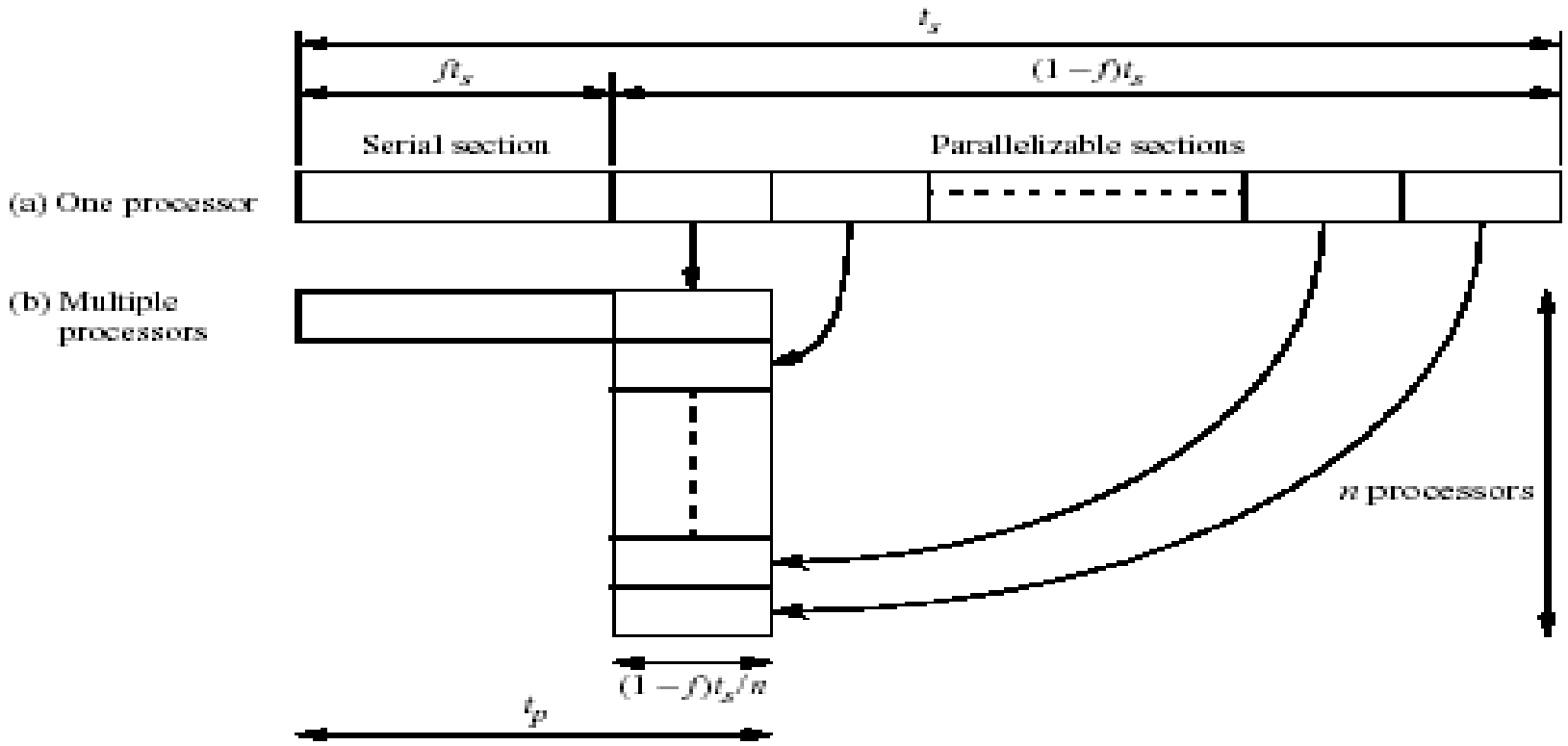


Figure 1.29 Parallelizing sequential problem — Amdahl's law.

Derivation of Amdahl's Law (cont.)

- Using the preceding expression for t_p

$$S(n) = \frac{t_s}{t_p} \le \frac{t_s}{f t_s + \frac{(1-f)t_s}{n}}$$

$$= \frac{1}{f + \frac{(1-f)}{n}}$$

- The last expression is obtained by dividing numerator and denominator by t_S , which establishes Amdahl's law.
- Multiplying numerator & denominator by n produces the following **alternate versions** of this formula:

$$S(n) \le \frac{n}{nf + (1-f)} = \frac{n}{1 + (n-1)f}$$

Potential Benefits, Limits and Costs of Parallel Programming
Amdahl's Law

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

P represents the parallel fraction & fs represents the serial fraction of the workload. Whereas fs+ P =1

Rewriting Amdahl's speedup formula:

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

Substituting $f_s = 1 - P$:

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

For large p, the term P/p becomes very small, and we approximate:

$$S \approx \frac{1}{1-P}$$

- if none of the code can be parallelized, P = 0 and the speedup = 1 (no speedu
- 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.

Potential Benefits, Limits and Costs of Parallel Programming
Amdahl's Law

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

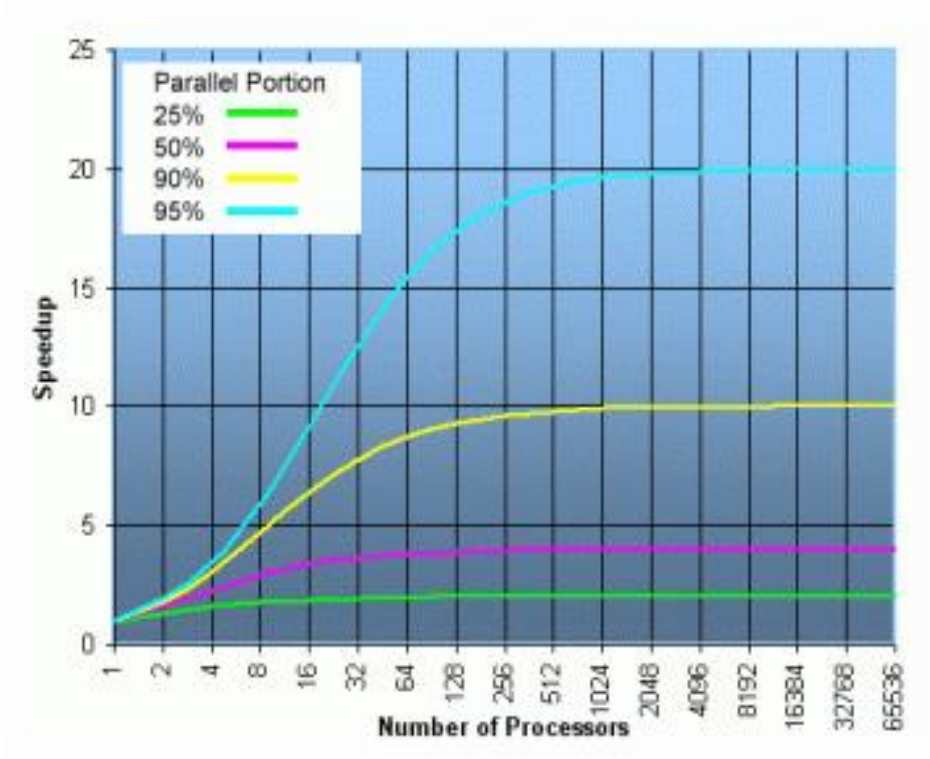
$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

- where
  - P = parallel fraction,
  - N = number of processors and
  - S = serial fraction

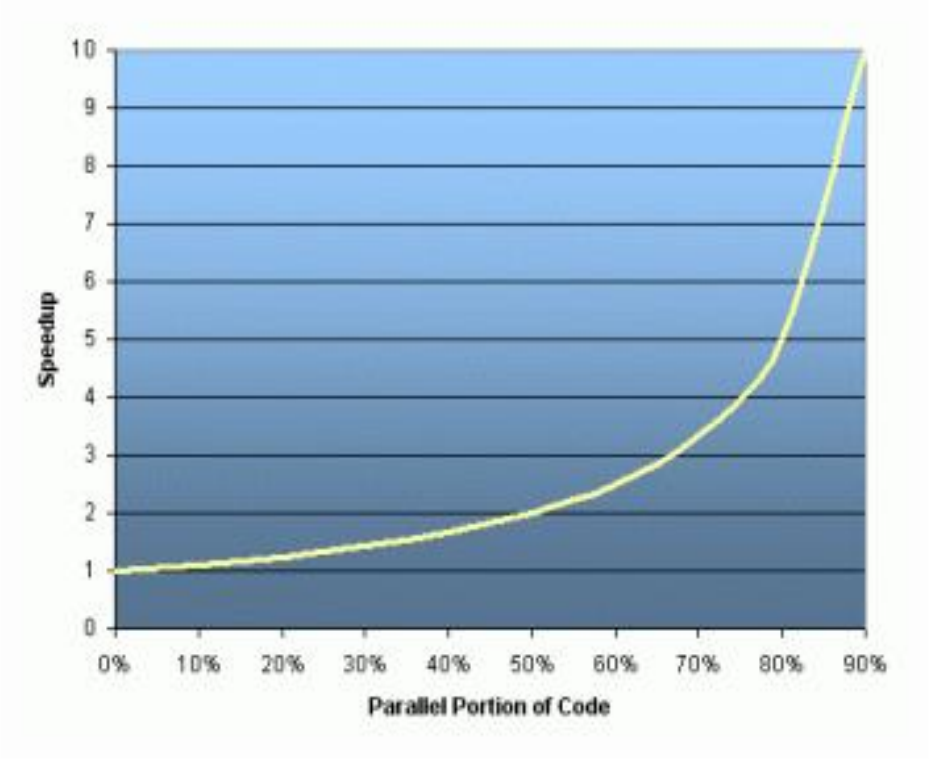
	speedup			
N	P = .50	P = .90	P = .95	P = .99
10	1.82	5.26	6.89	9.17
100	1.98	9.17	16.80	50.25
1,000	1.99	9.91	19.62	90.99
10,000	1.99	9.91	19.96	99.02
100,000	1.99	9.99	19.99	99.90

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

Potential Benefits, Limits and Costs of Parallel Programming
Amdahl's Law



Speedup when introducing more processors



Amdahl's law

Amdahl's law and Strong Scaling

- Provides a way to quantify the theoretical maximum **speedup in latency** that can occur with parallel execution.
- Amdahl's law describes the ratio of the original execution time with the improved execution time, assuming perfect parallelism and no overhead penalty.
- That is, Amdahl's law provides a theoretical limit to how much faster a program can run if it is parallelized.

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

$$S = \frac{1}{(1-p) + \frac{p}{N}} = \frac{N}{N(1-p) + p} = \frac{N}{Nf + 1 - f} = \frac{N}{1 + (N-1)f}$$

$$S = \frac{1}{F + \frac{(1-F)}{N}}$$

Amdahl’s Law

- Suppose a parallel program is executing on **10 processors**, and only **40%** of the time is executing in parallel.
- What is the overall **speedup** gained by incorporating parallelism?
- **Formula for Speedup (Amdahl’s Law):**

$$S = \frac{1}{F + \frac{(1-F)}{P}}$$

$$S = \frac{1}{0.6 + \frac{(1-0.6)}{10}}$$

$$S = \frac{1}{0.6 + 0.04} = \frac{1}{0.64} = 1.56$$

Thus, the program runs **1.56 times faster**, meaning a **56% improvement** in execution time.

Amdahl’s Law Example

- If **90%** of a calculation can be parallelized (**10% is sequential**), then the **maximum speedup** achieved on **5 processors** is:
- **Formula for Speedup (Amdahl’s Law):**

$$S = \frac{1}{F + \frac{(1-F)}{P}}$$

$$S = \frac{1}{0.1 + \frac{(1-0.1)}{5}}$$

$$S = \frac{1}{0.1 + 0.18} = \frac{1}{0.28} = 3.6$$

- Thus, the program runs **3.6 times faster**.
- **Speedup for More Processors**
  - **On 10 processors** → Speedup = **5.3**
  - **On 20 processors** → Speedup = **6.9**
  - **On 1000 processors** → Speedup = **9.9**

  Even with **1000 processors**, the **maximum achievable speedup is only 9.9×** due to the serial portion of the program.

Amdahl's Law Example

- 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

$$\psi \leq \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$

Amdahl's Law Example

- 5% of a parallel program's execution time is spent within inherently sequential code.
- The maximum speedup achievable by this program, regardless of how many PEs are used, is

$$\lim_{p \to \infty} \frac{1}{0.05 + (1 - 0.05) / p} = \frac{1}{0.05} = 20$$

Amdahl's Law Example

- An oceanographer gives you a serial program and asks you how much faster it might run on 8 processors.
- You can only find one function amenable to a parallel solution. Benchmarking on a single processor reveals 80% of the execution time is spent inside this function.
- What is the best speedup a parallel version is likely to achieve on 8 processors?

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

where:

- $f_E = 0.8$ (the fraction of execution time that can be parallelized)
- $1 - f_E = 0.2$ (the sequential portion that **cannot** be parallelized)
- $p = 8$ (number of processors)

$$S = \frac{1}{(1 - 0.8) + \frac{0.8}{8}}$$

$$S = \frac{1}{0.2 + 0.1}$$

$$S = \frac{1}{0.3} = 3.33$$

Amdahl's Law

Let Speedup be denoted by "S", fraction enhanced be denoted by "f_E", and factor of improvement be denoted by "f_I". Then we can write the above equation as

$$S = ((1 - f_E) + (f_E / f_I))^{-1} .$$

The three problem types are as follows:

1. Determine S given f_E and f_I
2. Determine f_I given S and f_E
3. Determine f_E given S and f_I

$$\text{Speedup} = \frac{1}{(1 - \text{fraction enhanced}) + (\text{fraction enhanced}/\text{factor of improvement})}$$

Problem Type 1 – Predict System Speedup

If we know f_E and f_I, then we use the Speedup equation (above) to determine S.

If we know f_E and f_I, then we use the Speedup equation (above) to determine S.

Example: Let a program have 40 percent of its code enhanced (so $f_E = 0.4$) to run 2.3 times faster (so $f_I = 2.3$). What is the overall system speedup S?

Step 1: Setup the equation: $S = ((1 - f_E) + (f_E / f_I))^{-1}$

Step 2: Plug in values & solve $S = ((1 - 0.4) + (0.4 / 2.3))^{-1}$

$= (0.6 + 0.174)^{-1} = 1 / 0.774$

$= \mathbf{1.292}$

Problem Type 2 – Predict Speedup of Fraction Enhanced

If we know f_E and S, then we solve the Speedup equation (above) to determine f_I

Example: Let a program have 40 percent of its code enhanced (so $f_E = 0.4$) to yield a system speedup 4.3 times faster (so S = 4.3). What is the factor of improvement f_I of the portion enhanced?

Case #1:

Can we do this? In other words, let's determine if by enhancing 40 percent of the system, it is possible to make the system go 4.3 times faster …

Step 1: Assume the limit, where f_I = infinity, so $S = ((1 - f_E) + (f_E / f_I))^{-1} \rightarrow S = 1 / (1 - f_E)$

Step 2: Plug in values & solve $S = ((1 - 0.4))^{-1} = 1 / 0.6 = \mathbf{1.67}$.

Step 3: So S = 1.67 is the **maximum possible speedup**, and we cannot achieve S = 4.3 !!

Problem Type 2 – Predict Speedup of Fraction Enhanced

If we know f_E and S, then we solve the Speedup equation (above) to determine f_I

Case #2:

A different case: Let's determine if by enhancing 40 percent of the system, it is possible to make the system go 1.3 times faster …

Step 1: Assume the limit, where f_I = infinity, so $S = ((1 - f_E) + (f_E / f_I))^{-1}$ → $S = 1 / (1 - f_E)$

Step 2: Plug in values & solve $S = ((1 - 0.4))^{-1} = 1 / 0.6 =$ **1.67** .

Step 3: So S = 1.67 is the **maximum possible speedup**, and we <u>can</u> achieve S = 1.3 !!

Step 4: Solve speedup equation for f_I :

$1/S = (1 - f_E) + (f_E / f_I)$ [invert both sides]

$1/S - (1 - f_E) = f_E / f_I$ [subtract $(1 - f_E)$]

$(1/S - (1 - f_E))^{-1} = f_I / f_E$ [invert both sides]

$f_E \cdot (1/S - (1 - f_E))^{-1} = f_I$ [multiply by f_E]

Step 5: Plug in values & solve:

$f_I = f_E \cdot (1/S - (1 - f_E))^{-1}$

$= 0.4 \cdot (1/1.3 - (1 - 0.4))^{-1}$

$= 0.4 / (0.769 - 0.6) =$ **2.367**

Step 6: Check your work:

$S = ((1 - f_E) + (f_E / f_I))^{-1} = (0.6 + (0.4/2.367))^{-1} = 1.3$ ✓

Problem Type 3 – Predict Fraction of System to be Enhanced

If we know f_I and S, then we solve the Speedup equation (above) to determine f_E

Example: Let a program have a portion f_E of its code enhanced to run 4 times faster (so $f_I = 4$), to yield a system speedup 3.3 times faster (so S = 3.3). What is the fraction enhanced (f_E)?

Step 1: *Can this be done?* Assuming f_I = infinity, $S = 3.3 = ((1 - f_E))^{-1}$ so minimum $f_E = 0.697$

Yes, this can be done for maximum f_I, so let's solve the equation to determine actual f_E

Step 2: Solve speedup equation for f_E :

$S = ((1 - f_E) + (f_E / f_I))^{-1}$	[state the equation]
$3.3 = ((1 - f_E) + (f_E / 4))^{-1}$	[plug in values]
$(1 - f_E) + f_E/4 = 1/3.3 = 0.303$	[invert both sides]
$1 - 0.75f_E = 0.303$	[regroup]
$0.75f_E = 1 - 0.303 = 0.697$	[commutativity]
$f_E = 0.697 / 0.75 =$ **0.929**	[divide by 0.75]

Step 3: Check your work: $S = ((1 - f_E) + (f_E / f_I))^{-1} = (0.071 + (0.929/4))^{-1} = 3.3$ ✓

Amdahl’s Law

- Amdahl’s Law assumes **a fixed problem size**.
- It focuses on the **speedup limitations** due to the **sequential portion** of a program.
- Even with **infinite processors**, speedup is **limited** by the serial fraction F.
- **Key takeaway**: Parallelism is **bounded** by the serial portion, making **large processor counts inefficient** if F is significant.

Gustafson’s Law

- Gustafson’s Law assumes **a growing problem size**.
- It argues that **as we add more processors, we can handle larger problems**, making parallelism more useful.
- Unlike Amdahl’s Law, **speedup scales better** because **parallel work increases with the number of processors**. Not tied solely to the parallelism.
- It is termed as **Weak Scaling**
- **Key takeaway**: More processors can **increase problem size**, making parallelization more effective.

Gustafson's Law Formula

$$S = s + p \times N$$

Since $p = 1 - s$, we substitute:

$$S = s + (1 - s) \times N$$

Rearranging:

$$S = N + (1 - N) \times s$$

Where:

- S = Speedup
- s = Serial fraction of the program
- $p = 1 - s$ = Parallel fraction of the program
- N = Number of processors

Gustafson’s Example

Let’s assume:

- **10%** of the program is **serial** → $s = 0.1$
- **90%** is parallel → $p = 1 - s = 0.9$
- **Using $N = 5$ processors**

Using the formula:

$$S = s + (1 - s) \times N$$

$$S = 0.1 + (1 - 0.1) \times 5$$

$$S = 0.1 + 0.9 \times 5$$

$$S = 0.1 + 4.5 = 4.6$$

Thus, with **5 processors**, the program runs **4.6× faster**.

For 1000 Processors (N = 1000)

$$S = 0.1 + (1 - 0.1) \times 1000$$

$$S = 0.1 + 0.9 \times 1000$$

$$S = 0.1 + 900 = 900.1$$

Thus, with **1000 processors**, we achieve a speedup of **900.1×**, showing **near-linear scaling** as we increase processors.

It's also known as "**scaled speedup**".

Gustafson's Example

- As an example, consider a program that can be partially improved with parallel execution.
  - Let us assume that **20% of the program cannot be improved** and some initial empirical results suggest that the **parallel execution portion runs in 1/5th of the time** that it takes sequentially (i.e., an improvement factor of 5).
  - Note that this does not assume anything about how many processors are used, so it can be based on more realistic measurements by running some initial tests.
  - In this case, the speedup would be:

$$S = 0.2 + 5 * 0.8 = 0.2 + 4.0 = 4.2$$

- The proper interpretation of the Gustafson's law notion of speedup is that **this program can achieve 4.2 times as much work in the same amount of time**, which is based on **weak scaling**.
- If the original program could process 10 MB of data in a minute, then the improved version could process 42 MB in the same amount of time.
- With Gustafson's law, the emphasis is on the throughput (amount of work done) rather than a faster time.

Scalability of Parallel Systems

- Increase number of processors → decrease efficiency
- Increase **problem size** → increase efficiency
- Can a parallel system keep efficiency by increasing the number of processors and the problem size simultaneously???
- **Yes**: → **scalable** parallel system
- **No**: → **non-scalable** parallel system

- A **scalable parallel system** can be made **cost-optimal** by **balancing** the number of processors with problem size.

Scalability of Parallel Systems

- *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

It helps determine how the **problem size** must grow to maintain a constant **efficiency** as the number of processors increases

or

Isoefficiency tells us how much we need to increase the problem size as we add processors to keep efficiency constant.

Why is Isoefficiency Important?

- Adding more processors does not always lead to better performance due to **communication overhead** and **sequential bottlenecks** (Amdahl's Law).
- **Efficiency (E)** is defined as:

$$E = \frac{S}{p} = \frac{T_1}{pT_p}$$

- Isoefficiency function W=f(p) determines how the problem size W(total work) must scale with p to maintain a fixed efficiency.
- If the problem size does not grow fast enough, adding processors leads to **poor scalability**.

Deriving the Isoefficiency Function

A parallel program consists of:

1. **Computation time** T_{comp} (grows with problem size W).
2. **Communication overhead** T_{comm} (grows with number of processors p).

Efficiency is given by:

$$E = \frac{T_{\text{comp}}}{T_{\text{comp}} + T_{\text{comm}}}$$

For **constant efficiency**, the ratio $T_{\text{comm}}/T_{\text{comp}}$ must remain the same. This leads to:

$$W = O(f(p))$$

where $f(p)$ **is the isoefficiency function.**

The lower f(p), the better the scalability.

Types of Isoefficiency Growth

- The growth of W with respect to p depends on how communication overhead scales.

- Linear Isoefficiency: $W=O(p)$
  - If **communication overhead is minimal**, the problem size needs to increase **linearly** with processors.
- Logarithmic Isoefficiency: $W=O(plogp)$
  - If **communication grows logarithmically**, the problem size must grow slightly faster than **O(p)**.
- Quadratic Isoefficiency: $W=O(p2)$
  - If **communication overhead grows faster**, problem size must grow **quadratically** to maintain efficiency.

Factors That Limit Parallel Execution

1. Load Imbalance: When tasks are not evenly distributed among processors, some finish early while others are still computing, leading to idle time and reduced efficiency. For example, in parallel sorting, if one processor receives more data, it takes longer to complete, delaying the overall process.

2. Serialization: Some parts of a program must run sequentially, limiting parallel execution and slowing performance. For instance, a critical section in a parallel program that only one processor can access at a time forces others to wait.

3. Communication Overhead: Processors need to exchange data, but excessive communication can slow execution. For example, in a distributed database, frequent synchronization between nodes can reduce overall performance.

4.Bottlenecks: Certain resources, like memory access, can become a limiting factor in performance. For instance, if multiple processors need to access the same memory location, contention can cause delays.

Sample Questions based on topic discussed

- What are various sources of overhead in parallel system? Illustrate with examples.
- Enlist various performance metrics and explain each in short.
- Any 2 performance measures in detail.
- Explain speedup metric with an example.
- D.B speedup and superlinear speedup. Explain superlinear speedup with example.
- Explain speedup, efficiency and parallel time with example.
- What is the effect of granularity on parallel system's performance? Explain with example.
- Illustrate the process of adding n elements with a cost optimal solution.
- Explain Amdahl’s law, Gustavson’s law and performance bottlenecks.
- Compute maximum speedup achievable by the program with 5% of a parallel program’s execution time is spent within inherently sequential code, regardless of number of PEs used.
- An oceanographer gives you a serial program and asks you how much faster it might run on 8 processors. You can only find one function amenable to a parallel solution. Benchmarking on a single processor reveals 80% of the execution time is spent inside this function. What is the best speedup a parallel version is likely to achieve on 8 processors?