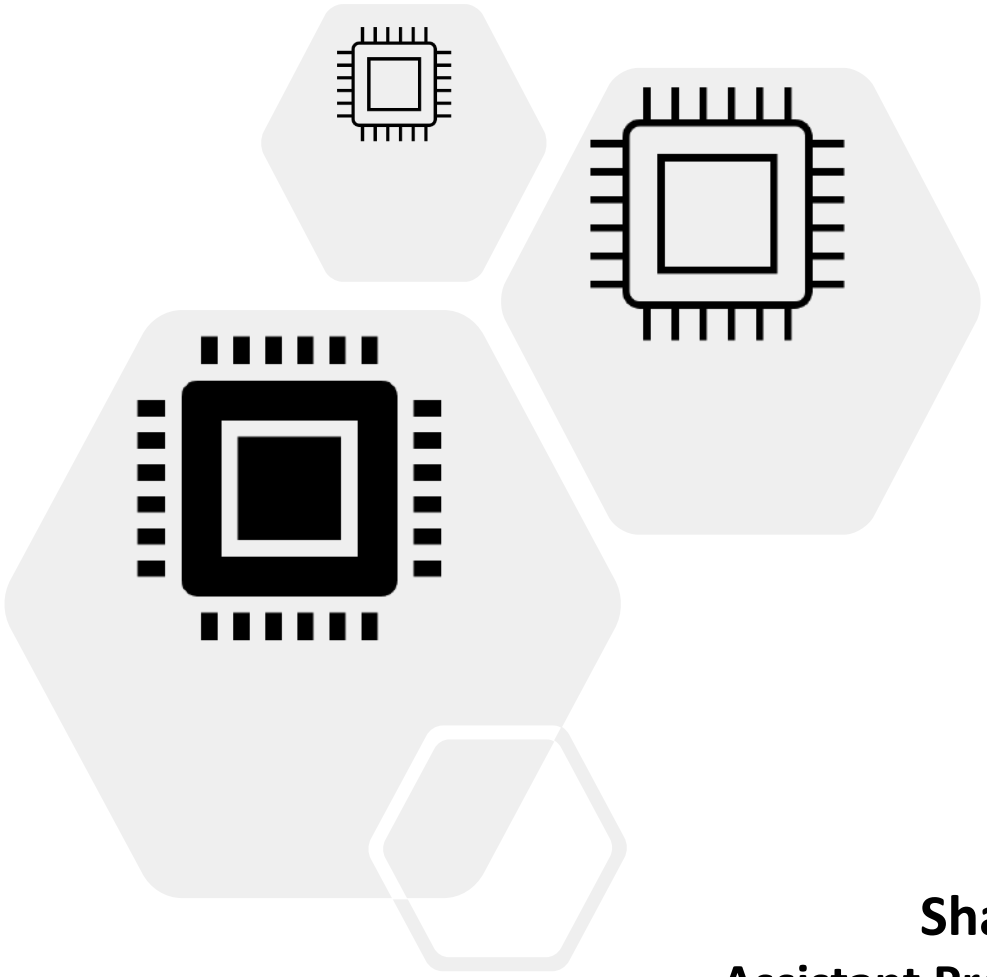
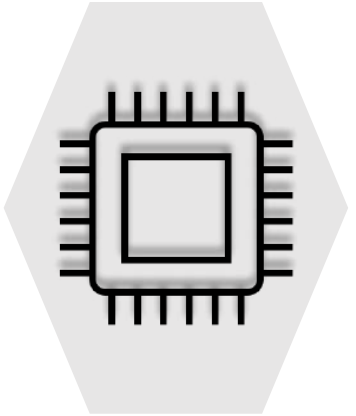
High Performance Computing



Performance Measures for HPC

Shafaque Fatma Syed
Assistant Professor - Dept. of Information Technology
A P Shah Institute of Technology, Mumbai

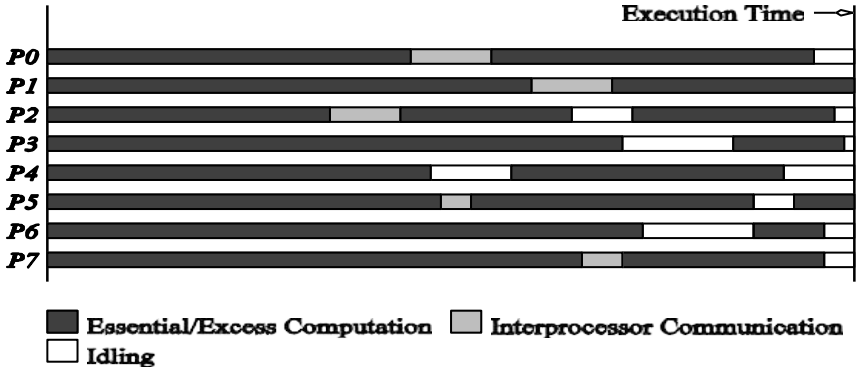
Topics to be discussed



- Sources of Overhead in Parallel Programs (Content Beyond Syllabus)
- Performance Metrics for Parallel Systems
- Effect of Granularity on Performance
- Scalability of Parallel Systems
- Amdahl‘s Law
- Gustafson‘s Law

Sources of Overhead in Parallel Programs

- If I use two processors, shouldn't my program run twice as fast?
- No - a number of overheads, including wasted computation, communication, idling, and contention cause degradation in performance.



The execution profile of a hypothetical parallel program executing on eight processing elements. Profile indicates times spent performing computation (both essential and excess), communication, and idling.

Sources of Overheads in Parallel Programs

- Interprocess interactions: Processors working on any non-trivial parallel problem will need to talk to each other.
- Idling: Processes may idle because of load imbalance, synchronization, or serial components.
- Excess Computation: This is computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication.

Performance Metrics for Parallel Systems: Execution Time

- Serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer.
- The parallel runtime is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution.

- We denote the serial runtime by T_S and the parallel runtime by T_P.

Performance Metrics for Parallel Systems: Total Parallel Overhead

- Let T_{all} be the total time collectively spent by all the processing elements.
- T_S is the serial time.
- Observe that $T_{all} - T_S$ is then the total time spend by all processors combined in non-useful work. This is called the *total overhead*.
- The total time collectively spent by all the processing elements $T_{all} = p\ T_P$ (p is the number of processors).
- The overhead function (T_o) is therefore given by

$$T_o = p\ T_P - T_S \qquad (1)$$

Performance Metrics for Parallel Systems: Speedup

- What is the benefit from parallelism?

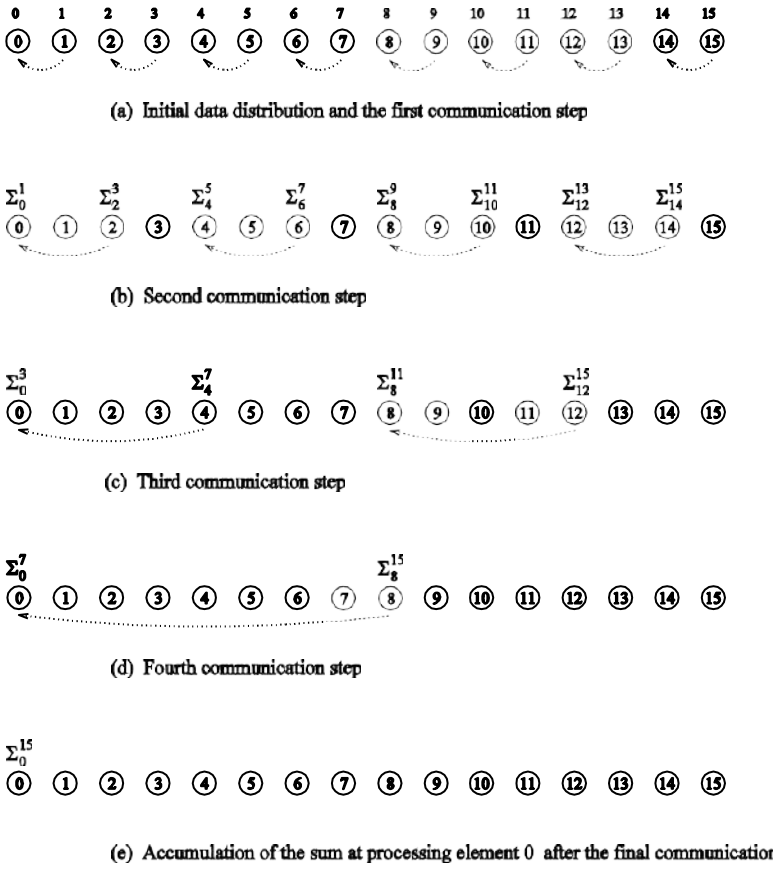
- Speedup (***S***) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with ***p*** identical processing elements.

Performance Metrics: Example

- Consider the problem of adding ***n*** numbers by using ***n*** processing elements.

- If ***n*** is a power of two, we can perform this operation in **log *n*** steps by propagating partial sums up a logical binary tree of processors.

Performance Metrics: Example



Computing the globalsum of 16 partial sums using 16 processing elements . Σ^j_i denotes the sum of numbers with consecutive labels from ***i*** to ***j***.

Performance Metrics: Example (continued)

- If an addition takes constant time, say, t_c and communication of a single word takes time $t_s + t_w$, we have the parallel time $T_P = \Theta (\log n)$

- We know that $T_S = \Theta (n)$

- Speedup S is given by $S = \Theta (n / \log n)$

Performance Metrics: Speedup

- For a given problem, there might be many serial algorithms available. These algorithms may have different asymptotic runtimes and may be parallelizable to different degrees.

- For the purpose of computing speedup, we always consider the best sequential program as the baseline.

Performance Metrics: Speedup Example

- Consider the problem of parallel bubble sort.
- The serial time for bubble-sort is 150 seconds.
- The parallel time for odd-even sort (efficient parallelization of bubble sort) is 40 seconds.
- The speedup would appear to be 150/40 = 3.75.
- But is this really a fair assessment of the system?
- What if serial quicksort only took 30 seconds? In this case, the speedup is 30/40 = 0.75. This is a more realistic assessment of the system.

Performance Metrics: Speedup Bounds

- Speedup can be as low as 0

- Speedup, in theory, should be upper bounded by ***p*** - after all, we can only expect a ***p***-fold speedup if we use times as many resources.

- A speedup greater than ***p*** is possible only if each processing element spends less than time T_s / p solving the problem.

- In this case, a single processor could be timeslided to achieve a faster serial program, which contradicts our assumption of fastest serial program as basis for speedup.

Performance Metrics: Efficiency

- Efficiency is a measure of the fraction of time for which a processing element is usefully employed
- Mathematically, it is given by

$$E = \frac{S}{p}. \quad (2)$$

- Following the bounds on speedup, efficiency can be as low as 0 and as high as 1.

Performance Metrics: Efficiency Example

- The speedup of adding numbers on processors is given by

$$S = \frac{n}{\log n}$$

- Efficiency is given by

$$E = \frac{\Theta\left(\frac{n}{\log n}\right)}{n}$$

$$= \Theta\left(\frac{1}{\log n}\right)$$

Cost of a Parallel System: Example

- Cost is the product of parallel runtime and the number of processing elements used ($\boldsymbol{p}$ x $\boldsymbol{T_P}$).
- Cost reflects the sum of the time that each processing element spends solving the problem.
- A parallel system is said to be *cost-optimal* if the cost of solving a problem on a parallel computer is asymptotically identical to serial cost.
- Since $\boldsymbol{E} = \boldsymbol{T_S} / \boldsymbol{p}\ \boldsymbol{T_P}$, for cost optimal systems, $\boldsymbol{E} = O(1)$.
- Cost is sometimes referred to as *work* or *processor-time product*.

Cost of a Parallel System: Example

Consider the problem of adding numbers on processors.

- We have, $T_P = \log n$ (for $p = n$).
- The cost of this system is given by $p\,T_P = n \log n$.
- Since the serial runtime of this operation is $\Theta(n)$, the algorithm is not cost optimal.

Effect of Granularity on Performance

- If the granularity is too fine, the performance can suffer from the increased communication overhead, whereas if granularity is too coarse the performance can suffer from load imbalance.
- Often, using fewer processors improves performance of parallel systems.
- Using fewer than the maximum possible number of processing elements to execute a parallel algorithm is called *scaling down* a parallel system.
- A naive way of scaling down is to think of each processor in the original case as a virtual processor and to assign virtual processors equally to scaled down processors.
- The communication cost should not increase by this factor since some of the virtual processors assigned to a physical processors might talk to each other. This is the basic reason for the improvement from building granularity.

Building Granularity: Example

- Consider the problem of adding ***n*** numbers on ***p*** processing elements such that ***p*** < ***n*** and both ***n*** and ***p*** are powers of 2.
- Use the parallel algorithm for ***n*** processors, except, in this case, we think of them as virtual processors.
- Each of the ***p*** processors is now assigned ***n*** / ***p*** virtual processors.
- The first **log** ***p*** of the **log** ***n*** steps of the original algorithm are simulated in (***n*** / ***p***) **log** ***p*** steps on ***p*** processing elements.
- Subsequent **log** ***n*** - **log** ***p*** steps do not require any communication.

Building Granularity: Example (continued)

- The overall parallel execution time of this parallel system is $\Theta\left(\left(n / p\right) \log p\right)$.

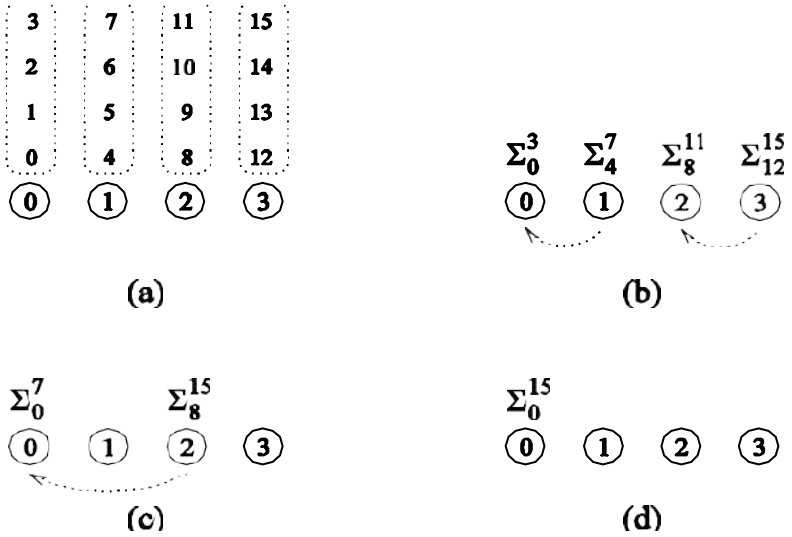
- The cost is $\Theta\left(n \log p\right)$, which is asymptotically higher than the $\Theta\left(n\right)$ cost of adding n numbers sequentially. Therefore, the parallel system is not cost-optimal.

Building Granularity: Example (continued)

Can we build granularity in the example in a cost-optimal fashion?

Each processing element locally adds its n / p numbers in time Θ (n / p).

The p partial sums on p processing elements can be added in time $\Theta(n/p)$.



A cost-optimal way of computing the sum of 16 numbers using four processing elements.

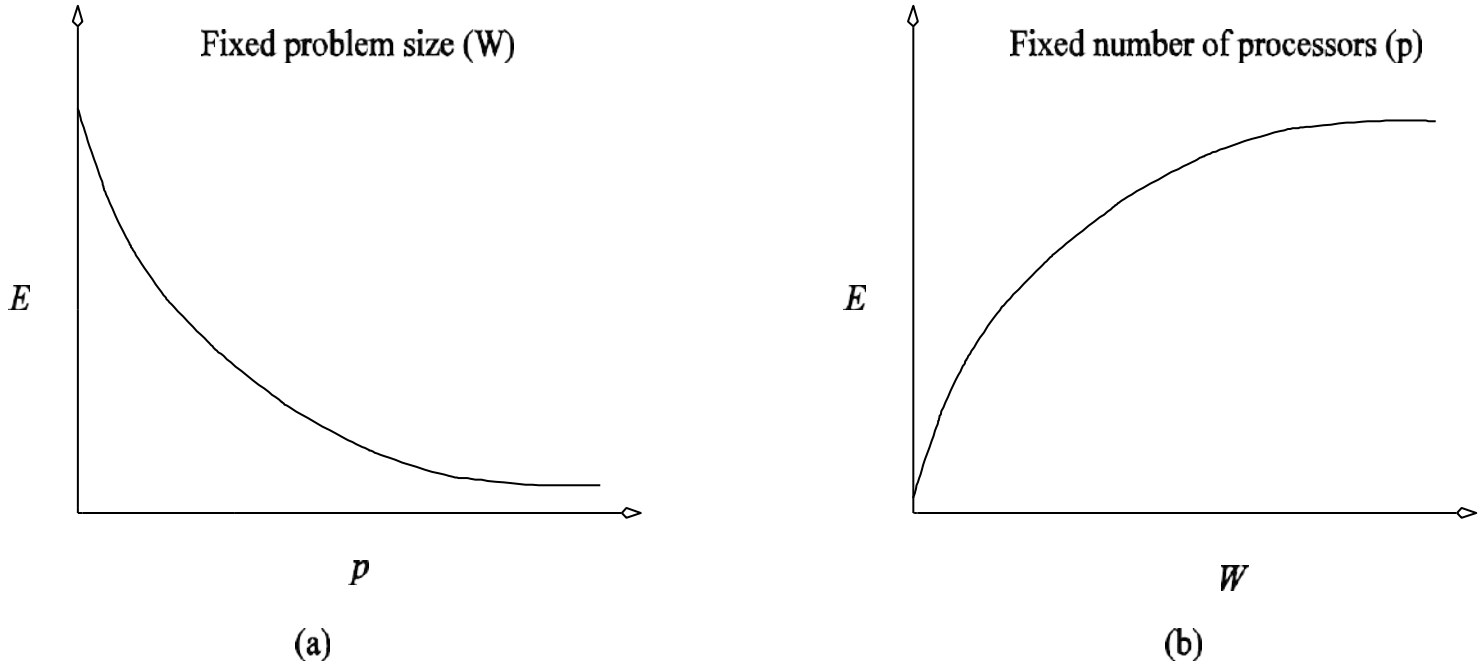
Scalability of Parallel Systems

- Scalability is another important measure of performance for a parallel algorithm.
- Algorithms are said to be scalable if the level of parallelism increases at least linearly with size of problem.
- The architecture to implement an algorithm is scalable if it continues to yield the same performance for processor, even if the problem size increases and also the number of processes increase
- It is seen that the data parallel algorithms are more scalable as compared to the architecture scalable algorithms.
- A parallel computer system is said to be scalable if its efficiency can be fixed by simultaneously increasing the machine size and the problem size. Scalability of a parallel system is the measure of its capacity to increase speed up in proportion to the machine size.
- Increasing the number of processes decreases efficiency. and increasing the amount of computation for processor, increased efficiency.
- To keep the efficiency fixed, both the size of problem and the number of processes must be increased simultaneously

Isoefficiency Metric of Scalability

- The isoefficiency function can be used to measure scalability of parallel computing systems.
- It shows how the size of problem must grow as a function of number of processes and in order to maintain some constant efficiency
- For a given problem size, as we increase the number of processing elements, the overall efficiency of the parallel system goes down for all systems.
- For some systems, the efficiency of a parallel system increases if the problem size is increased while keeping the number of processing elements constant.

Isoefficiency Metric of Scalability



Variation of efficiency: (a) as the number of processing elements is increased for a given problem size; and (b) as the problem size is increased for a given number of processing elements. The phenomenon illustrated in graph (b) is not common to all parallel systems.

Isoefficiency Metric of Scalability

- What is the rate at which the problem size must increase with respect to the number of processing elements to keep the efficiency fixed?

- This rate determines the scalability of the system. The slower this rate, the better.

- Before we formalize this rate, we define the problem size ***W,*** as the asymptotic number of operations associated with the best serial algorithm to solve the problem.

Isoefficiency Metric of Scalability

- We can write parallel runtime as:

$$T_P = \frac{W + T_o(W,p)}{p}$$

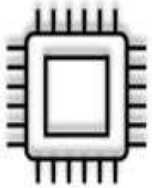
- The resulting expression for speedup is

$$\begin{aligned} S &= \frac{W}{T_P} \\ &= \frac{Wp}{W + T_o(W,p)}. \end{aligned}$$

- Finally, we write the expression for efficiency as

$$\begin{aligned} E &= \frac{S}{p} \\ &= \frac{W}{W + T_o(W,p)} \\ &= \frac{1}{1 + T_o(W,p)/W}. \end{aligned}$$

Speed-Up



If you are using **n** processors, your ***Speedup$_n$*** is:

$$Speedup_n = \frac{T_1}{T_n}$$

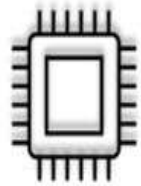
where T_1 is the execution time on **one core** and T_n is the execution time on **n cores**. Note that $Speedup_n$ should be > 1.

And your ***Speedup Efficiency$_n$*** is:

$$Efficiency_n = \frac{Speedup_n}{n}$$

which could be as high as 1., but probably never will be.

Amdahl's law



If you put in *n* processors, you should get *n* times Speedup (and 100% Speedup Efficiency), right? Wrong!

There are always some fraction of the total operation that is inherently sequential and cannot be parallelized no matter what you do. This includes reading data, setting up calculations, control logic, storing results, etc.

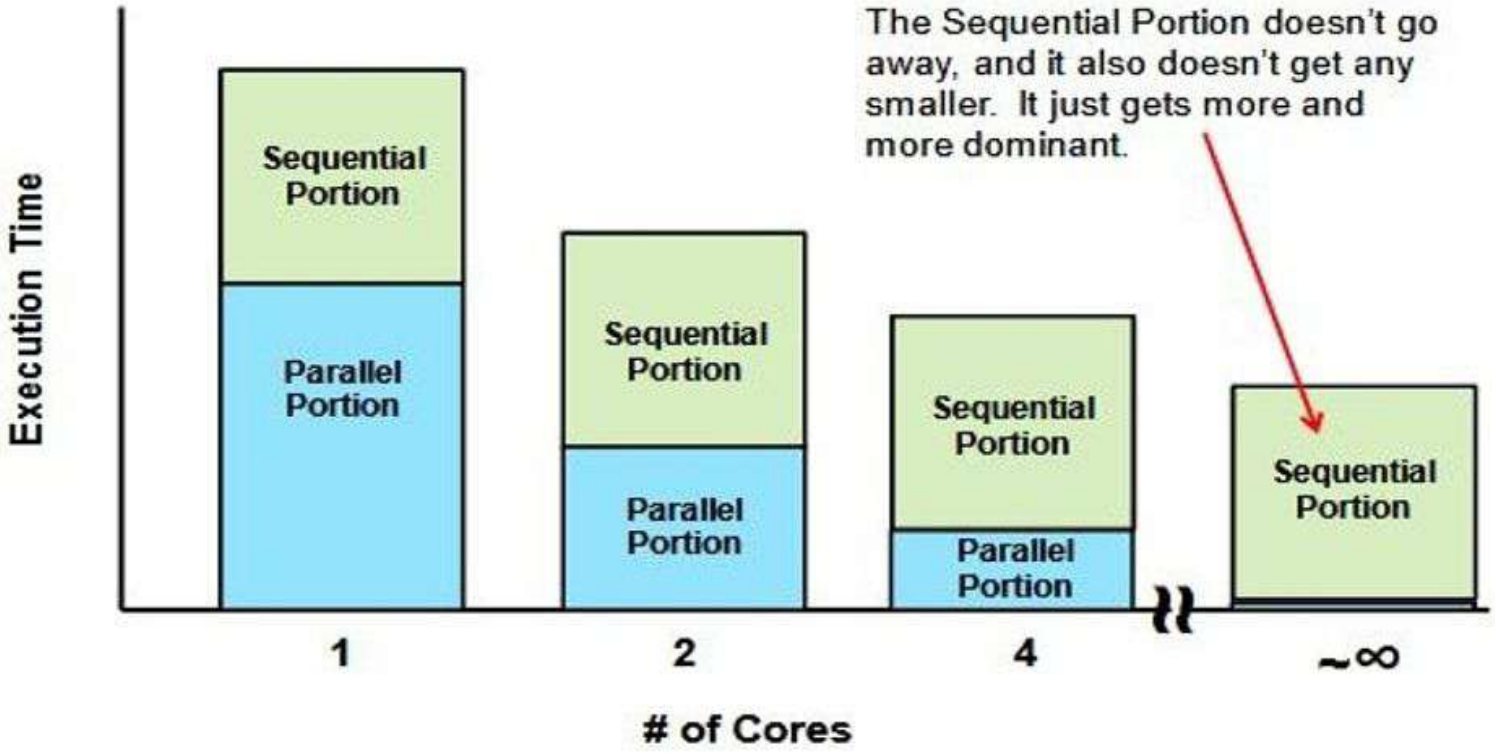
If you think of all the operations that a program needs to do as being divided between a fraction that is parallelizable and a fraction that isn't (i.e., is stuck at being sequential), then **Amdahl's Law** says:

$$Speedup_n = \frac{T_1}{T_n} = \frac{1}{\frac{F_{parallel}}{n} + F_{sequential}} = \frac{1}{\frac{F_{parallel}}{n} + (1 - F_{parallel})}$$

This fraction can be reduced by deploying multiple processors.

This fraction can't.

Shortcomings of Amdahl's Law



Shortcomings of Amdahl's Law

- The work load cannot scale to match the available computing power as the machine size increases. Hence, the fixed load prevents scalability in performance
- The sequential bottleneck is a serious problem

Proof

PROOF OF AMDAHL'S LAW:

For parallel processing,

Let $T(s)$ be the time to execute program on a single processor and $T(p)$ be the time to execute program on N parallel processors.

$\therefore T(s) = T$ (ie. entire executions time in single processor)

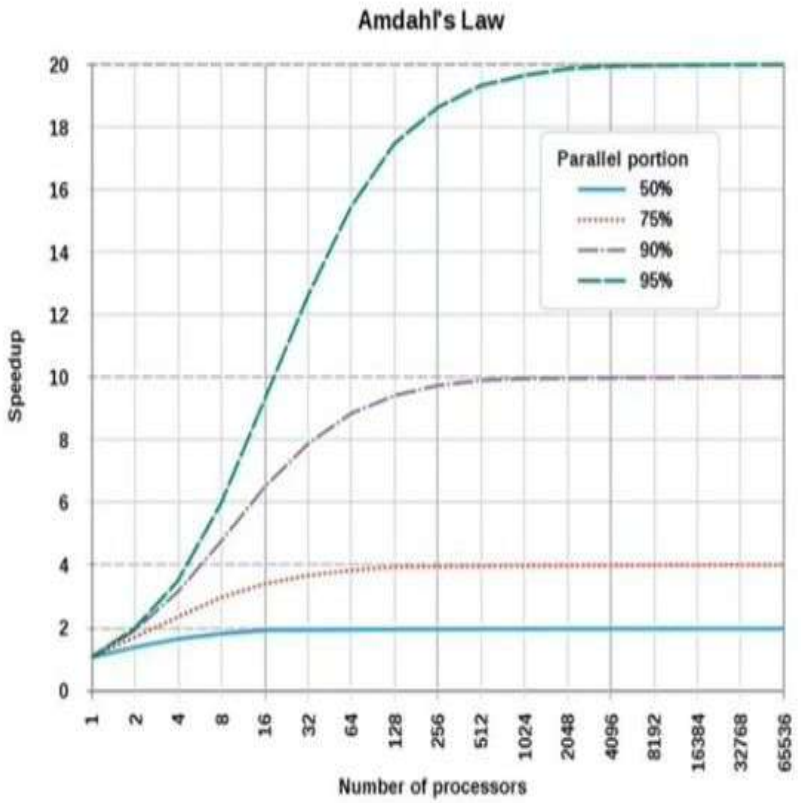
$T(p)$ = Time taken for some serial execution + Time taken for parallel execution.

$$T(p) = (1-f) \cdot T + T \cdot \frac{f}{N}.$$

$$\therefore \text{Speed-Up} = \frac{T(s)}{T(p)}$$

$$= \frac{T}{(1-f) \cdot T + T \cdot \frac{f}{N}}$$

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



Assume 1% of the runtime of a program is not parallelizable. This program is run on 61 cores of a Intel Xeon Phi. Under the assumption that the program runs at the same speed on all of those cores, and there are no additional overheads, what is the parallel speedup?

$$S(61) = 38.125.$$

Finding fraction

Amdahl's law says:

$$S = \frac{T_1}{T_n} = \frac{1}{\frac{F}{n} + (1-F)} \Rightarrow \frac{1}{S} = \frac{F}{n} + (1-F) = 1 + \frac{F - nF}{n} \Rightarrow \frac{1}{S} - 1 = F\frac{(1-n)}{n}$$

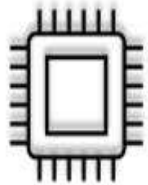
Use this if you know the timing

Use this if you know the speedup

Solving for F:

$$F = \frac{\frac{1}{S} - 1}{\frac{1-n}{n}} = \frac{\frac{T_n}{T_1} - 1}{\frac{1-n}{n}} = \frac{\frac{T_n - T_1}{T_1}}{\frac{1-n}{n}} = \frac{\frac{T_1 - T_n}{T_1}}{\frac{n-1}{n}} = \frac{n(T_1 - T_n)}{T_1(n-1)} = \frac{n}{(n-1)}\frac{T_1 - T_n}{T_1} = \frac{n}{(n-1)}\left(1 - \frac{1}{Speedup}\right)$$

Max. Speedup



Note that these fractions put an upper bound on how much benefit you will get from adding more processors:

$$\max Speedup = \lim_{n\to\infty} Speedup = \frac{1}{F_{sequential}} = \frac{1}{1 - F_{parallel}}$$

Fparallel	max Speedup
0.00	1.00
0.10	1.11
0.20	1.25
0.30	1.43
0.40	1.67
0.50	2.00
0.60	2.50
0.70	3.33
0.80	5.00
0.90	10.00
0.95	20.00
0.99	100.00

Which speed up could be achieved according to Amdahl's Law for infinite number of processes if 5% of a program is sequential and the remaining part is ideally parallel ?

(A) Infinite

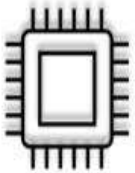
(B) 5

(C) 20

(D) 50

Ans: 20

Gustafson's Law



Starting point for Gustafson's law is the computation on the multiprocessor rather than on the single computer.

In Gustafson's analysis, parallel execution time kept constant, which we assume to be some acceptable time for waiting for the solution.

Parallel computation composed of fraction computed sequentially say f' and fraction that contains parallel parts, $1 - f'$.

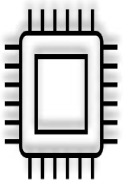
Gustafson's so-called *scaled speedup fraction* given by:

$$S'(p) = \frac{f' t_p + (1 - f')p t_p}{t_p} = p + (1 - p)f'$$

f' is fraction of computation on multiprocessor that cannot be parallelized.

f' is different to f previously, which is fraction of computation on a single computer that cannot be parallelized.

Conclusion drawn from Gustafson's law is almost linear increase in speedup with increasing number of processors, but the fractional part f' needs to remain small.



References Used

- Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar , —Introduction to Parallel Computing,Pearson Education, Second Edition, 2007.
- M. R. Bhujade, —Parallel Computingǁ, 2nd edition, New Age International Publishers, 2009.