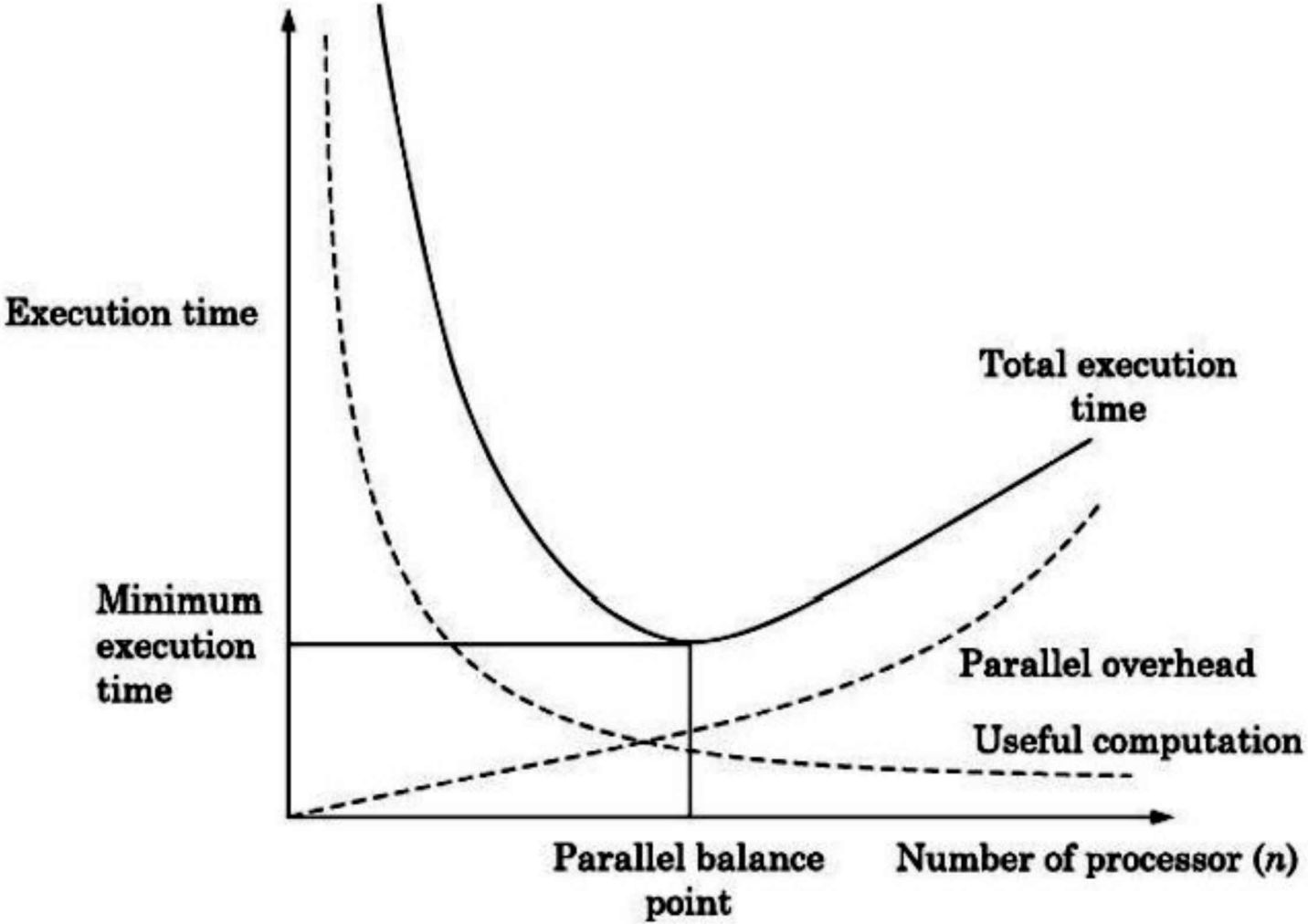

Execution time
Total execution time
Minimum execution time
Parallel overhead
Useful computation
Parallel balance point
Number of processor (n)

Amdahl's law

- It is named after computer scientist Gene Amdahl (a computer architect from IBM and Amdahl corporation) and was presented at the AFIPS Spring Joint Computer Conference in 1967.
- It is also known as Amdahl's argument.
- It is a formula that gives the theoretical **speedup of the execution of a task at a fixed workload that can be expected of a system whose resources are improved**.
- In other words, it is a formula used to find the maximum improvement possible by just improving a particular part of a system.
- It is often used in parallel computing to predict the theoretical speedup when using multiple processors.
- **Speedup-** Speedup is defined as the ratio of performance for the entire task using the enhancement and performance for the entire task without using the enhancement or speedup can be defined as the ratio of execution time for the entire task without using the enhancement and execution time for the entire task using the enhancement.
    - If **Pe** is the performance for the entire task using the enhancement when possible, **Pw** is the performance for the entire task without using the enhancement, **Ew** is the execution time for the entire task without using the enhancement and **Ee** is the execution time for the entire task using the enhancement when possible then, **Speedup = Pe/Pw** or **Speedup = Ew/Ee**

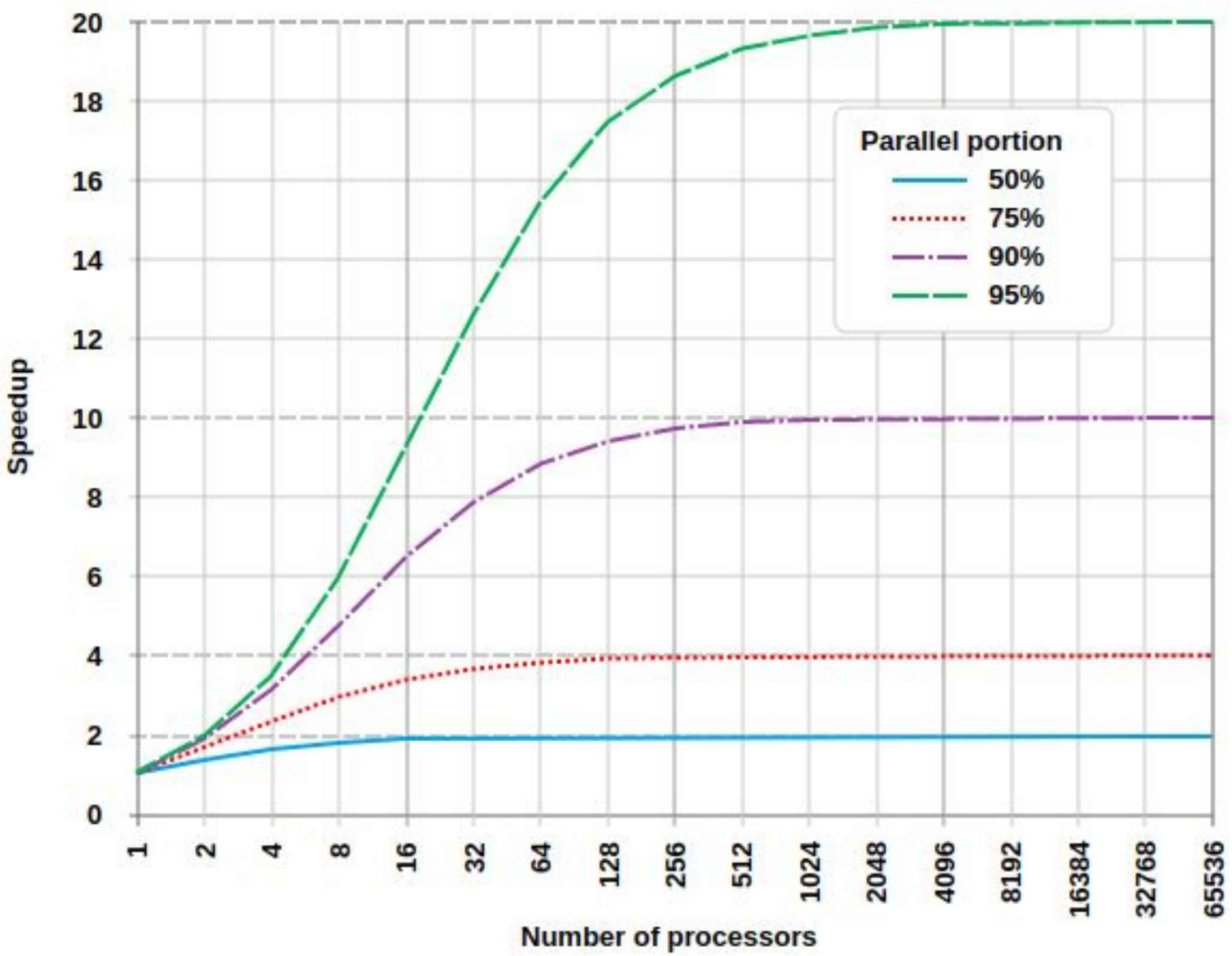
Amdahl's law

- Amdahl's law uses two factors to find speedup from some enhancement:
  - **Fraction enhanced (p)–** The fraction of the computation time in the original computer that can be converted to take advantage of the enhancement. For example- if 10 seconds of the execution time of a program that takes 40 seconds in total can use an enhancement, the fraction is 10/40. This obtained value is Fraction Enhanced. Fraction enhanced is always less than 1.
  - **Speedup enhanced (s)–** The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced mode were used for the entire program. For example – If the enhanced mode takes, say 3 seconds for a portion of the program, while it is 6 seconds in the original mode, the improvement is 6/3. This value is Speedup enhanced. Speedup Enhanced is always greater than 1.

$$\text{Overall Speedup} = \frac{\text{Old execution time}}{\text{New execution time}}$$

$$= \frac{1}{\left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)}$$

Amdahl's law


Speedup
Number of processors
Parallel portion
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Proof: Amdahl’s law

- The execution time of the whole task before the improvement of the resources of the system is denoted as **T**. It includes the execution time of the part that would not benefit from the improvement of the resources and the execution time of the one that would benefit from it. The fraction of the execution time of the task that would benefit from the improvement of the resources is denoted by **p**. The one concerning the part that would not benefit from it is therefore **1−p**. Then: **T=(1−p)T+pT**
- It is the execution of the part that benefits from the improvement of the resources that is accelerated by the factor **s** after the improvement of the resources. Consequently, the execution time of the part that does not benefit from it remains the same, while the part that benefits from it becomes: **(p/s)T**
- The theoretical execution time **T(s)** of the whole task after the improvement of the resources is then: **T(s)=(1−p)T+(p/s)T**
- Amdahl's law gives the theoretical speedup in latency of the execution of the whole task at fixed workload W, which yields:
- Overall Speedup = TW/(T(s)W) = T/T(s) = **1 / (1 - p + (p/s))**

Amdahl's law: Another way of writing

- Let T1 denote the computation time on a sequential system. We can split the total time as follows
  - T1 = ts + tp
    - Where
    - ts - computation time needed for the sequential part.
    - tp - computation time needed for the parallel part.
- Clearly, if we parallelize the problem, only tp can be reduced. Assuming ideal parallelization we get
  - Tp = ts + (tp / N)
    - Where
    - N - number of processors.
- Thus we get the speedup: S = T1 / Tp = (ts + tp)/(ts + (tp / N))
- Let f denote the sequential portion of the computation, i.e. f = ts / (ts + tp)
- Thus the speedup formula can be simplified into:
  - S = **1 / (f + (1 - f)/N)** < 1/f
- **Note:** Amdahl assumes the problem size does not change with the number of CPUs.

Problem Type 1 – Predict System Speedup

- If we know p and s, then we use the Speedup equation to determine overall speedup.
- Example: Let a program have 40 percent of its code enhanced (so p = 0.4) to run 2.3 times faster (so s = 2.3). What is the overall system speedup?
  - 1 / ((1 – 0.4) + (0.4 / 2.3)) = 1.292

Problem Type 2 – Predict Speedup of Fraction Enhanced

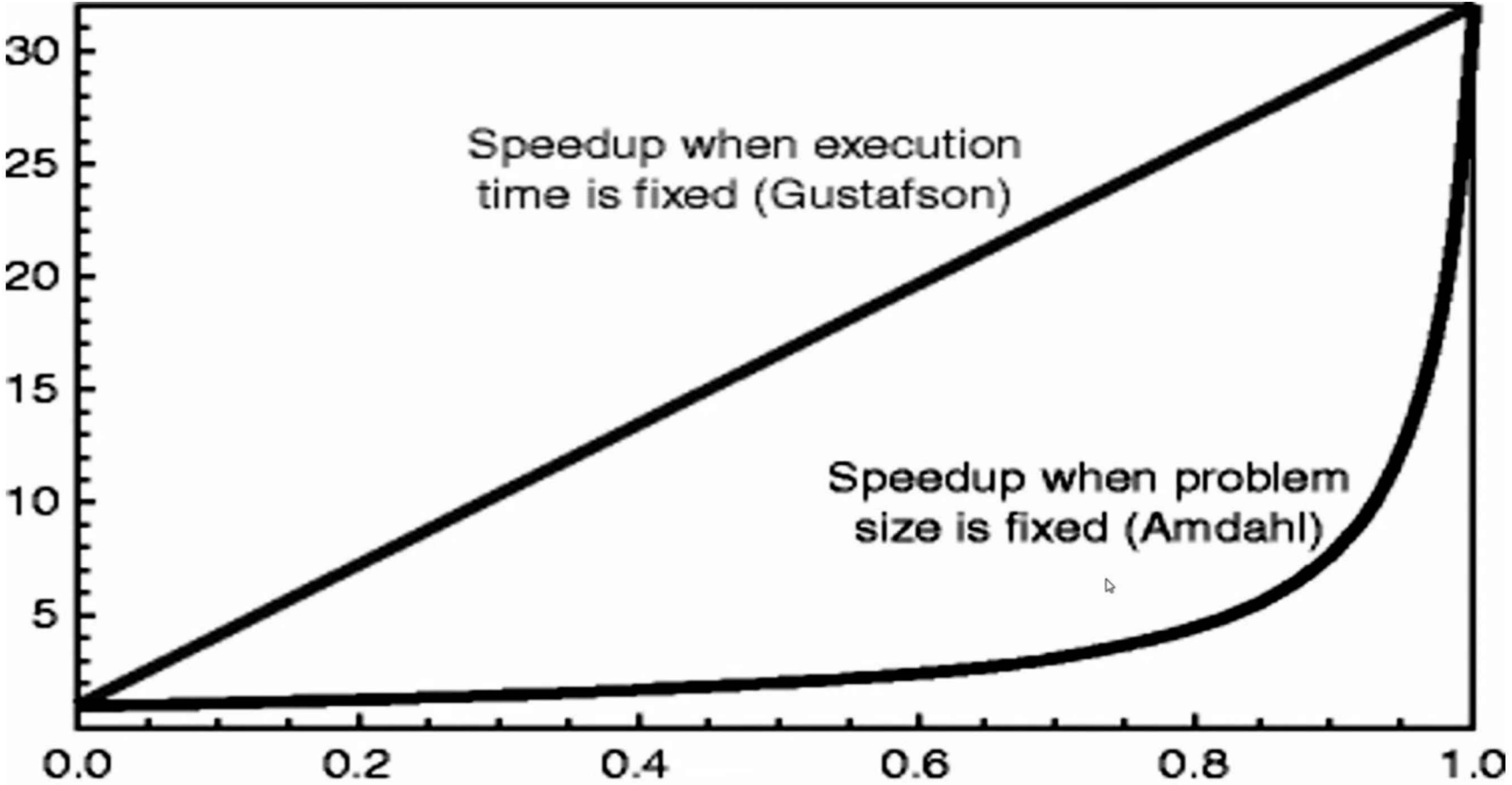
- Example: Let a program have 40 percent of its code enhanced (so p = 0.4) to yield a system speedup 4.3 times faster (so overall speedup = 4.3). What is the factor of improvement s of the portion enhanced?
  - let's determine if by enhancing 40 percent of the system, it is possible to make the system go 4.3 times faster …
  - Step 1: Assume the limit, where s = infinity, so S = 1 / ((1 – p) + (p / s)) = 1 / (1 – p)
  - Step 2: Plug in values & solve S = 1 /((1 – 0.4)) = 1 / 0.6 = 1.67 .
  - Step 3: So S = 1.67 is the maximum possible speedup, and we cannot achieve S = 4.3 !!
- 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 p = infinity, so S = 1/ ((1 – p) + (p / s)) = 1 / (1 – p)
  - Step 2: Plug in values & solve S = 1/ ((1 – 0.4)) = 1 / 0.6 = 1.67 .
  - Step 3: So S = 1.67 is the maximum possible speedup, and we can achieve S = 1.3 !!
  - Step 4: Solve speedup equation for s : 1/S = (1 – p) + (p / s) => s = 2.367
  - Step 5: Check your work: S = 1 / ((1 – p) + (p / s)) = 1 / (0.6 + (0.4/2.367)) = 1.3

Problem Type 3 – Predict Fraction of System to be Enhanced

- Let a program have a portion **p** of its code enhanced to run 4 times faster (so s = 4), to yield a system speedup 3.3 times faster (so S = 3.3). What is the fraction enhanced (p)?
  - Step 1: Can this be done? Assuming s = infinity, S = 3.3 = 1 / ((1 – p))
    - so minimum p = 0.697
    - Yes, this can be done for maximum s, so let's solve the equation to determine actual p.
  - Step-2: Solve speedup equation for p : S = 1 / ((1 – p) + (p / s)) => p = 0.929
  - Step 3: Check your work: S = 1 / ((1 – p) + (p / s)) =1 / (0.071 + (0.929/4)) = 3.3

Gustafson's Law

- John L. Gustafson (born January 19, 1955)
  - American computer scientist and businessman,
  - found out that practital problems show much better speedup than Amdahl predicted.
- Gustafson's law
  - The computation time is constant (instead of the problem size),
  - increasing number of CPUs ⇒ solve bigger problem and get better results in the same time.
- Let Tp denote the computation time on a parallel system. We can split the total time as follows
  - Tp = ts* + tp*
    - Where
    - ts* is computation time needed for the sequential part.
    - tp* is computation time needed for the parallel part.
- On a sequential system we would get
  - T1 = ts* + N·tp*
- Thus the speedup will be
  - S = (ts* + N·tp*)/(ts* + tp*)
- Let f* denote the sequential portion of the computation on the parallel system, i.e. f* = ts* / (ts* + tp*)
- Then S = **f* + N · (1 − f*)**



Observable Parallel fraction of existing workload

Gustafson's Law: Proof

- The bigger the problem, the smaller f - serial part remains usually the same, and f ≠ f*
- Amdahl's says: S = (ts + tp)/(ts + (tp / N)) *(eq. 1)*
- Let now f* denote the sequential portion spent in the parallel computation, i.e. f* = ts / (ts + (tp / N)) and 1 - f* = (tp / N) / (ts + (tp / N))
- Hence, ts = f* · (ts + (tp / N)) and tp = N · (1 − f*) · (ts + (tp / N))
- Using these values in eq. 1, we get:
- S = **f* + N · (1 − f*)**
- what is exactly what Gustafson derived.
- The key is not to mix up the values f and f*
- this caused great confusion that lasted over years!

Sun and Ni's Law

- This one is referred to as a memory bound model.
- It turns out that when the speedup is computed by the problem size limited by the available memory in n-processor system, it leads to a generalization of Amdahl's and Gustafson's law.
- For n-nodes, assume the parallel portion of workload is increased by G(n) times reflecting the increase of memory in n-node system
- Then the speedup factor is given as:
- **$S^* = (f + (1-f)G(n)) / (f + (1-f)G(n)/n)$**

Sun and Ni's Law: as a generalization of Amdahl's and Gustafson's law

- **Case 1: G(n)=1**
  - S* = (f + (1-f)) / (f + (1-f)/n) = 1 / (f + (1-f)/n) => Which is Amdahl's law
- **Case 2: G(n)=n**
  - S* = (f + (1-f)n) / (f + (1-f)n/n) = f + (1-f)n => which is Gustafson's law
- **Case 3: G(n)>n**, Let G(n)=m where m>n. The speedup is then
  - S* = (f + (1-f)m) / (f + (1-f)m/n) = (f + (1-f)m) / (m/n + (1 - (m/n))f)
- This is the case where the workload grows faster than the memory requirement.
- Notice that it produces slightly higher speedup than the case of Gustafson's workload.

Performance Benchmarks

- **MIPS** and **Mflops** can be used to describe the instruction execution rate and floating-point capability of a parallel computer.
- Most computer manufacturers state peak or sustained performance in terms of MIPS or Mfiops.
- These ratings are by no means conclusive.
- The real performance is always prograrlt-dependent or application driven.
- In general, the MIPS rating depends on the instruction set, varies between programs, and even varies inversely with respect to performance

Performance Benchmarks

- **The Dhrynone Result:**
- This is a CPU-intensive benchmark consisting of a mix of about 100 high-level language instructions and data types found in system programming applications where floating-point operations are not used.
- The Dhrystone statements are balanced with respect to statement type, data type, and locality of reference, with no operating system calls and making no use of library functions or subroutines.
- Thus the Dhrystone rating should be a measure of the integer performance of modem processors.
- The unit *KDhrystone/s* is often used in reporting Dhrystone results.

Performance Benchmarks

- **The Whetstone Result:**
- This is a Fortran-based synthetic benchmark assessing the floating-point performance, measured in the number of *KWhetstone/s* that a system can perform.
- The benchmark includes
- both integer and floating-point operations involving array indexing, subroutine calls, parameter passing, conditional branching, and trigonometric/transcendental functions.
- The Whetstone benchmark does not contain any vectorizable code and shows dependence on the system's mathematics library and efficiency of the code generated by a compiler.
- The Whetstone performance is not equivalent to the Mflops performance, although the Whetstone contains a large number of scalar floating-point operations.
- Both the Dhrystone and Whetstone are synthetic benchmarks whose performance results depend heavily on the compilers used.
- Both benchmarks were criticized for being unable to predict the performance of user programs.
- The sensitivity to compilers is a major drawback of these benchmarks.

Performance Benchmarks

- The **TPS** and **KLIPS Rating**:
- The throughput of computers for on-line transaction processing is often measured in transactions per second (TPS).
- Each transaction may involve a database search, query answering, and database update operations.
- Business computers and servers should be designed to deliver a high TPS rate.
- Apart from these, we can also use throughput (in number of requests served and/or amount of data delivered) and average response time.
- In artificial intelligence applications, the measure KLIPS (kilo logic inferences per second) can be used to indicate the reasoning power of an AI machine.

Sources of Parallel Overhead

- Parallel computers in practice do not achieve linear speedup or an efficiency of one because of parallel overhead (the amount of time required to coordinate parallel tasks as opposed to doing useful work).
- The major sources of overhead in a parallel computer are:
  - inter-processor communication,
  - load imbalance,
  - inter-task synchronization, and
  - extra computation.
- Other Overheads: task creation, task scheduling, task termination, processor (shared) memory interconnect latency, cache coherence enforcement and those imposed by parallel languages, libraries, operating system, etc.