ADVANCED ARCHITECTURES

Performance

Annalisa Massini

Lecture 3

2025-2026

SPEED-UP PERFORMANCE AND EXAMPLES

Computer Architecture - A Quantitative Approach
John L. Hennessy, David A. Patterson
Chapter 1 - Fundamentals of Quantitative Design and Analysis
Section 1.8 - Measuring, Reporting, and Summarizing Performance

Multicore and GPU programming
G. Barlas
Chapter 1 - Introduction

Measuring Performance

- When we say one computer is faster than another we can mean different things:
  - *response time* - also referred to as *execution time* - the time between the start and the completion of an event
    - The computer user is interested in reducing the *response time*
  - *throughput* - the total amount of work done in a given time
    - The operator of a warehouse-scale computer may be interested in increasing *throughput*

Measuring Performance

- In comparing design alternatives, we often want to relate the performance of two different computers: X and Y
- When we say ***X is faster than Y*** we mean that the response time or execution time is lower on X than on Y for the given task
- In particular, ***X is n times faster than Y*** will mean:

$$\frac{\text{Execution time } Y}{\text{Execution time } X} = n$$

Measuring Performance

- Note that ***execution time*** is the reciprocal of ***performance***
- Since ***execution time*** *is the reciprocal of performance*, the following relationship holds:

$$n = \frac{\text{Execution time } Y}{\text{Execution time } X} = \frac{\text{Performance } X}{\text{Performance } Y}$$

- The phrase *the **throughput** of X is 1.3 times higher than Y* signifies that the number of tasks completed per unit time on computer X is 1.3 times the number of tasks completed on Y

Measuring Performance

- Unfortunately, **time** is not always the metric quoted in comparing the performance of computers
- But (for Hennessy and Patterson) ***the only consistent and reliable measure of performance is the execution time of real programs***
- All proposed alternatives to time as a metric or to real programs as the items measured have eventually led to misleading claims or even mistakes in computer design

Measuring Performance

- Even execution time can be defined in different ways depending on what we count

- The most straightforward definition of time is called

  wall-clock time, *response time*, or *elapsed time*

  which is the *latency to complete a task*, including *disk accesses, memory accesses, input/output activities, operating system overhead…*

Measuring Performance

- With multiprogramming, the processor works on another program while waiting for I/O and may not necessarily minimize the elapsed time of one program
- Hence, we need a term to consider this activity
- ***CPU time*** recognizes this distinction and means the time the processor is computing, ***not*** including the time waiting for I/O or running other programs
- Clearly, the response time seen by the user is the elapsed time of the program, ***not*** the CPU time

Measuring Performance

- **Benchmarks** can be used to measure performance
- The best choice of benchmarks is *real applications*
- Attempts at running programs much simpler than a real application have led to performance pitfalls
- Examples include:
  - *Kernels*, which are small, key pieces of real applications
  - *Toy programs*, which are 100-line programs (such as quicksort)
  - *Synthetic benchmarks*, which are fake programs invented to try to match the profile and behavior of real applications (as Dhrystone)
- All three are discredited today (compiler writer and architect can conspire to make the computer appear faster than on real applications)

Taking advantage of parallelism

- In the design and analysis of computers, we need
  - Principles and guidelines
  - Observations about design
  - Equations to evaluate alternatives

- Taking advantage of parallelism is one of the most important methods for improving performance
  - Parallelism at the **system level** – scalability
  - Parallelism at the **level of an individual processor** - parallelism among instructions
  - Parallelism at the **level of digital design** - memories and ALUs

Taking advantage of parallelism

- Fundamental observations come from properties of programs
- The most important program property that we regularly exploit is the ***principle of locality***
  - ***Temporal locality*** states that **recently** accessed items are likely to be accessed in the near future
  - ***Spatial locality*** says that items whose **addresses** are near one another tend to be referenced close together in time

Taking advantage of parallelism

- An important and pervasive principle of computer design is to focus on the ***common case***:
  - In making a design trade-off, ***favor the frequent case*** over the infrequent case
- This principle applies when determining how to spend resources, since the impact of the ***improvement is higher*** if the occurrence is ***frequent***
- In applying this simple principle, we have to decide *what the frequent case is* and *how much performance* can be improved by making that case faster

Amdahl's Law

- The performance gain obtained by improving some portion of a computer can be calculated using **Amdahl's law**
- Amdahl's law:
  - states that the ***time/performance improvement is limited*** by the ***fraction of the time*** the faster mode can be used
  - defines the ***speedup*** that can be ***gained by using a particular*** **feature**

$$speeedup =$$

$$= \frac{\textbf{Execution time for entire task without using the enhancement}}{\textbf{Execution time for entire task using the enhancement when possible}}$$

$$= \frac{\textbf{Performance for the entire task using the enhancement when possible}}{\textbf{Performance for the entire task without using the enhancement}}$$

Amdahl’s law

- Amdahl’s law gives us a quick way to find the *speedup* from some enhancement, which depends on two factors:

1) The ***fraction of the computation time*** in the original computer that can be converted to take advantage of the enhancement, that is

   $\mathbf{Fraction_{enhanced}}$ = time with enhancement / *total* time

   Example:

   - A program takes 60 seconds in total
   - 20 seconds of the execution time can use an enhancement
   - The fraction is: 20/60
   - ***This value is always less than or equal to 1***

Amdahl’s law

- Amdahl’s law gives us a quick way to find the speedup from some enhancement, which depends on two factors:

2) 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:

   $\text{Speedup}_{\text{enhanced}}$ = ***original*** mode time / ***enhanced*** mode time

   Example:

   - A portion of the program in the original mode is 5 seconds
   - In the enhanced mode takes 2 seconds
   - The improvement is 5/2
   - ***This value is always greater than 1***

Amdahl's law

- The **execution time** using the original computer with the enhanced mode will be the time spent using the unenhanced portion of the computer plus the time spent using the enhancement:

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

- The overall **speedup** is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

Example

- We want to enhance the processor used for Web serving
- The new processor is **10 times faster** on computation in the Web serving application than the original processor
- Assume that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time
- ***What is the overall speedup gained by incorporating the enhancement?***

Example

- We want to enhance the processor used for Web serving
- The new processor is **10 times faster** on computation in the Web serving application than the original processor
- Assume that the original processor is busy with computation 40% of the time and is waiting for I/O 60% of the time
- ***What is the overall speedup gained by incorporating the enhancement?***

$\text{Fraction}_{\text{enhanced}} = 0.4$ $\text{Speedup}_{\text{enhanced}} = 10$

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}} = \frac{1}{(1 - 0.4) + \dfrac{0.4}{10}} = \frac{1}{0.64} = 1.6$$

Amdahl’s law

- Amdahl’s law can serve as a guide to understand:
  - *how much an enhancement* will improve performance
  - *how to distribute resources* to improve cost-performance

- The goal is to spend resources proportional to where time is spent

- Amdahl’s law is useful
  - to compare ***the overall system performance of two alternatives***
  - to compare **two processor design alternatives**

Example

- A common transformation in graphics processors is ***square root***
- Implementations of floating-point square root (FPSQR) vary significantly in performance among processors for graphics

- Suppose
  - **FPSQR is responsible for 20% of the execution time** of a critical graphics benchmark and
  - **FP instructions are responsible for half of the execution time** for the application

Example

- We want to compare two proposals:
  - To enhance the FPSQR hardware and speed up this operation by a factor of 10
  - To try to make all FP instructions in the graphics processor run faster by a factor of 1.6
- Evaluate and compare these two design alternatives

Example

- We can compare the speedups

- 1) Speedup of 10 with FPSRT hw

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

Example

- We can compare the speedups

- 1) Speedup of 10 with FPSRT hw

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

- 2) FP operations faster of 1,6 factor

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Example

- We can the speedups

- 1) Speedup of 10 with FPSRT hw

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

- 2) FP operations faster of 1,6 factor

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

- Improving the performance of the FP operations overall is *slightly better* because of the *higher frequency*

Amdahl's law

- When we consider a parallel machine with N nodes, the speedup will be:

$$speedup = \frac{t_{seq}}{t_{par}} = \frac{T}{(1-\alpha)T + \frac{\alpha T}{N}} = \frac{1}{(1-\alpha) + \frac{\alpha}{N}}$$

- Note that we are ignoring any partitioning or communication or coordination costs

Processor Performance Equation

- All computers are constructed using a clock running at a constant rate
- Discrete time events are called *ticks, clock ticks, clock periods, clocks, cycles, or clock cycles*
- Computer designers refer to the time of a clock period by its **duration** (e.g., 1 ns) or by its **rate** (e.g., 1 GHz)

- CPU time for a program can then be expressed two ways:
  - **CPU time = CPU clock cycles for a program × Clock cycle time**

or

  - **CPU time = CPU clock cycles for a program / Clock rate**

Processor Performance Equation

- We can also count the number of instructions executed - the ***instruction path length*** or ***instruction count (IC)***
- If we know the **number of clock cycles** (CPU clock cycles) and the **instruction count**, we can calculate the average number of ***clock cycles per instruction (CPI):***

CPI = CPU clock cycles for a program / IC

- From this formula we obtain
  - **CPU clock cycles for a program = CPI x IC**

Processor Performance Equation

- This allows us to use CPI in the execution time formula and obtain the ***performance equation***:
  - **CPU time = IC × CPI × Clock cycle time**
- In fact (using the units of measurement) we have:

$$\mathrm{IC} \times \mathrm{CPI} \times \text{Clock cycle time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycles}} =$$

$$= \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

- Observe that **processor performance** is ***equally dependent*** upon: **clock cycle** (or rate), **clock cycles per instruction**, and ***instruction count***

Processor Performance Equation

- It is useful to calculate the number of total processor clock cycles as

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

- where
  - IC_i is the number of times instruction i is executed in a program
  - CPI_i is the average number of clocks per instruction for instr. i

Processor Performance Equation

- This expression can be used to express CPU time as

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

and the overall CPI as

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

Example

- Suppose we have made the following measurements in the previous example (of Amdahl's Law):
  - Frequency of FP operations = 25%
  - Average CPI of FP operations = 4.0
  - Average CPI of other instructions = 1.33
  - Frequency of FPSQR = 2%
  - CPI of FPSQR = 20
- Assume that the two design alternatives are:
  - To decrease the CPI of FPSQR to 2
  - To decrease the average CPI of all FP operations to 2.5
- Compare these two design alternatives using the processor performance equation

Example

- Observe that *only the CPI changes*
- The *clock rate* and *instruction count* remain identical

- We start by finding the original CPI with no enhancement:

$$\text{CPI}_{\text{original}} = \sum_{i=1}^{n} \text{CPI}_i \times \frac{\text{IC}_i}{\text{Instruction count}} =$$

$$= (4 \times 25\%) + (1.33 \times 75\%) = 2.0$$

Example

- We can compute the CPI for the enhanced FPSQR by subtracting the cycles saved from the original CPI:

$$CPI_{new\,FPSR} = CPI_{original} - 2\% \times (CPI_{old\,FPSR} - CPI_{new\,FPSR\,only}) =$$

$$= 2 - 2\% \times (20 - 2) = 1.64$$

Example

- We can compute the CPI for the enhanced FPSR by subtracting the cycles saved from the original CPI:

$$CPI_{new\,FPSR} = CPI_{original} - 2\% \times (CPI_{old\,FPSR} - CPI_{new\,FPSR\,only}) =$$

$$= 2 - 2\% \times (20 - 2) = 1.64$$

- We can compute the CPI for the enhancement of all FP instructions (the same way or) by summing the FP and non-FP CPIs:

$$CPI_{new\,FP} = (2.5 \times 25\%) + (1.33 \times 75\%) = 1.625$$

- Since the CPI of the overall FP enhancement is slightly lower, its performance will be marginally better

Example

- The speedup for the FPSQR enhancement is

$$\text{Speedup}_{\text{FPSR}} = \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{FPSR}}} = \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{FPSR}}} = \frac{2.0}{1.64} = 1.22$$

- The speedup for the overall FP enhancement is

$$\text{Speedup}_{\text{new FP}} = \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{new FP}}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{original}}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{new FP}}} =$$

$$= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.0}{1.625} = 1.23$$

In summary

- It is often **easier** to use the processor performance equation than Amdahl's law
- In fact,
  - It is often possible to ***measure the constituent parts of the processor performance equation***
  - It may be ***difficult*** to measure things such as the ***fraction of execution time*** for which a set of instructions is responsible
  - In practice, this would probably be computed by summing the product of the instruction count and the CPI for each of the instructions in the set
- Hence, the *starting point is often individual instruction count and CPI measurements* → **performance equation**

Gustafson and Barsis' law

- Two decades after the Amdahl's law was published, Gustafson and Barsis noted that several programs were speeding up exceeding the predicted speedup limits

- They noted that:
  - Problem sizes grow as computer becomes more powerful
  - As the problem size grows, the work required for the parallel part frequently grows much faster than the serial part
  - So the serial part decreases and the speedup improves

Gustafson and Barsis' law

- Gustafson and Barsis managed to examine the problem from a different point of view
- Instead of examining what a parallel program could do relatively to a sequential one, we should examine **how a sequential machine would perform if it were required to solve the same problem that a parallel one can solve**

Gustafson-Barsis's law

Assume:

- We have a parallel application that requires *T* time to execute on *N* CPUs
- The application spend $0 \le \alpha \le 1$ percent of the total time running on all machines
- The remaining $1 - \alpha$ has to be done sequentially

Solving the problem on a sequential machine would require a total time:

$$t_{seq} = (1 - \alpha)T + N\alpha T$$

as the parallel part now have be done sequentially

Gustafson-Barsis's law

- The **speedup** would be:

$$speedup = \frac{t_{seq}}{t_{par}} = \frac{(1-\alpha)T + N\alpha T}{T} = (1-\alpha) + N\alpha$$

- And the corresponding **efficiency**

$$efficiency = \frac{speedup}{N} = \frac{(1-\alpha)}{N} + \alpha$$

- So the $efficiency$ has a lower bound of α as N go to infinity

Anyway, given the total disregard for the communication costs, the results for $speedup$ and $efficiency$ are overestimated

Example

- Assuming a program consists of 50% non-parallelizable code, compute the speedup when using 2 and 4 processors according to: *Gustafson's law* and Amdahl's law

Example

- Assuming a program consists of 50% non-parallelizable code, compute the speed-up when using 2 and 4 processors according to: *Gustafson's law* and *Amdahl's law*
- *Gustafson's law*

$$speedup_2 = (1 - \alpha) + N\alpha = \frac{1}{2} + 2 \cdot \frac{1}{2} = 1{,}5$$

$$speedup_4 = (1 - \alpha) + N\alpha = \frac{1}{2} + 4 \cdot \frac{1}{2} = 2{,}5$$

Example

- Assuming a program consists of 50% non-parallelizable code, compute the speed-up when using 2 and 4 processors according to: *Gustafson's* law and *Amdahl's law*
- *Gustafson's law*

$$speedup_2 = (1-\alpha) + N\alpha = \frac{1}{2} + 2 \cdot \frac{1}{2} = 1{,}5$$

$$speedup_4 = (1-\alpha) + N\alpha = \frac{1}{2} + 4 \cdot \frac{1}{2} = 2{,}5$$

- *Amdahl's law*

$$speedup_2 = \frac{1}{(1-\alpha) + \frac{\alpha}{N}} = \frac{1}{\left(1-\frac{1}{2}\right) + \frac{1}{4}} = \frac{1}{\frac{3}{4}} \cong 1{,}33$$

$$speedup_4 = \frac{1}{(1-\alpha) + \frac{\alpha}{N}} = \frac{1}{\left(1-\frac{1}{2}\right) + \frac{1}{8}} = \frac{1}{\frac{5}{8}} \cong 1{,}6$$

Example

Considerations to understand why speedup results are different

- Gustafson's law assumes that the parallel part of the program increases with the problem size and the sequential part stays fixed

- Amdahl's law sees the percentage of non-parallelizable code as a fixed limit for the speedup, even if we had an infinite amount of processors, according to Amdahl's law, the speedup would never be greater than 2

COMMUNICATION PERFORMANCE

Parallel Computer Architecture: A Hardware/Software Approach

D.E. Culler , J. P. Singh , A. Gupta Morgan Kaufmann, 1998

Chapter 1 - Introduction

Section 1.4 – Fundamental Design Issues

Section 1.4.6 – Performance

Communication Performance

- In evaluating architectural trade-offs, the decision between feasible alternatives rests upon their performance
- Programmers and compiler writers will avoid costly operations where possible
- To have a complete vision of the fundamental issues of parallel computer architecture, we need to understand **performance at many levels of design**

- Fundamentally, there are three performance metrics:
  - ***Latency***: time taken for an operation
  - ***Bandwidth***: rate of performing operations
  - ***Cost***: impact on execution time of program

Communication Performance

- If the processor does one thing at a time these metrics are directly related:
  - **bandwidth** (operation per second) is about
    1/latency (seconds per operation)
  - **cost** is simply ***latency x number of operations***
- But actually it is more complex in modern systems
- Modern computer systems do many different operations at once and the relationship between these performance metrics is much more complex

Communication Performance

- Characteristics apply to overall operations, as well as individual components of a system

- Since the unique property of parallel computer architecture is **communication**, the operations that we are concerned with most often are **data transfers**

Linear Model of Data Transfer Latency

- The time for a **data transfer operation** is generally described by a **linear model**:
  - *Transfer time (n)* $= T_0 + n/B$
    - ***n*** is the amount of data (e.g. number of bytes),
    - ***B*** is the transfer rate of the component moving the data (e.g. bytes per second),
    - the constant term $\boldsymbol{T_0}$ is the start-up cost
- This is a very convenient model, and it is used to describe a diverse collection of operations: messages, memory accesses, bus transactions, and vector operations

Linear Model of Data Transfer Latency

- It applies in many aspects of traditional computer architecture, as well

- In such a case, we can observe that the ***transfer time***:
  - For **memory operations**, it is essentially the **access time**
  - For **bus transactions**, it reflects the bus arbitration and command phases
  - For any sort of **pipelined operation**, including pipelined instruction processing or vector operations, it is the **time to fill pipeline**

Linear Model of Data Transfer Latency

- But a linear model is not enough:
  - It does not give any indication when the ***next such operation can be initiated***
  - It does not indicate whether other ***useful work can be performed during the transfer***
- These other factors depend on how the transfer is performed:
  - ***Hence we need to know how transfer is performed***

Communication Cost Model

- The data transfer most interesting in parallel machines is the one that occurs **across the network**
- It is initiated by the processor through the ***communication assist***
- The essential components of this operation can be described by the following simple model:

Communication Time (n) =

= Overhead + Network Delay + Occupancy

Communication Cost Model

- As we know, a generic parallel machine organization comprises a collection of essentially complete computers, each with one or more processors and memory, connected through a scalable communication network

- The ***communications assist*** is some kind of controller or auxiliary processing unit which assists in generating outgoing messages or handling incoming messages
- There is great diversity (and debate) as to what functionality should be provided within the assist and how it interfaces to the processor, memory system, and network

Communication Cost Model

Communication Time (n) =

= Overhead + Network Delay + Occupancy

- The **Overhead** is the time the processor spends initiating the transfer
  - It may be a *fixed cost*, if the processor simply has to tell the communication assist to start
  - It may be linear in *n*, if the processor has to copy the data into the assist

- The **Overhead** represents the time the processor:
  - is busy with the communication event
  - cannot do other useful work or initiate other communication

Communication Cost Model

Communication Time (n) =

= Overhead + Network Delay + Occupancy

- The remaining portions of the communication time is considered the **network latency**
- It is the part that can be hidden by other processor operations

- We can distinguish the two components **Network Delay** and **Occupancy**

Communication Cost Model

Communication Time (n) =

= Overhead + Network Delay + Occupancy

- The **Occupancy** is the time it takes for the data to pass through the slowest component on the communication path:
  - For example, each link that is traversed in the network will be occupied for time *n/B*, where *B* is the bandwidth of the link
  - The data will occupy other resources, including buffers, switches, and the communication assist that is often the bottleneck that determines the occupancy
  - The occupancy limits how frequently communication operations can be initiated
  - The next data transfer will have to wait until the critical resource is no longer occupied before it can use that same resource

Communication Cost Model

Communication Time (n) =

= Overhead + Network Delay + Occupancy

- The remaining communication time is the ***Network Delay***, which includes:
  - The time for a bit to be routed across the actual network
  - And other factors, such as the time to get through the communication assist

- From the processors viewpoint, the specific hardware components contributing to network delay are indistinguishable
- The task of designing the network and its interfaces is very concerned with the specific components and their contribution to the aspects of performance that the processor observes

Communication Cost Model

Communication Time (n) =

= Overhead + Network Delay + Occupancy

- This equation gives a very general model, and can be used to describe data transfers in many situations in computer systems
- For example, consider the time to move a block between cache and memory on a miss:
  - The **overhead** is the time the cache controller spends inspecting the tag to determine that it is not a hit and then starting the transfer
  - The **occupancy** is the block size divided by the bus bandwidth, unless there is some slower component in the system
  - The **delay** includes the normal time to arbitrate and gain access to the bus plus the time spent delivering data into the memory
  - Additional time spent waiting to gain access to the bus or wait for the memory bank cycle to complete is due to contention

Communication Cost Model

- A useful model connecting the program characteristics to the hardware performance is given by

  Communication Cost =
  frequency * (Comm. time - overlap)

- The *frequency of communication*:
  - Is defined as the number of communication operations per unit of work in the program
  - It depends on many programming factors and many hardware design factors

Communication Cost Model

Communication Cost =

= frequency * (Comm. time - overlap)

Note that:

- Hardware may:
  - limit the transfer size and determine the min number of messages
  - replicate data or migrate it to where it is used
- A certain amount of communication is inherent to parallel execution, since data must be shared and processors must coordinate their work
- A machine can support programs with a high communication frequency if the other parts of the communication cost are small: low overhead, low network delay, and small occupancy

Communication Cost Model

Communication Cost =

= frequency * (Comm. time - overlap)

- The **overlap** is the portion of the communication operation which is performed concurrently with other useful work, including computation or other communication
  - This reduction of the effective cost is possible because much of the communication time involves work done by components of the system other than the processor, such as the network interface unit, the bus, the network or the remote processor or memory
  - Overlapping communication with other work is a form of small scale parallelism