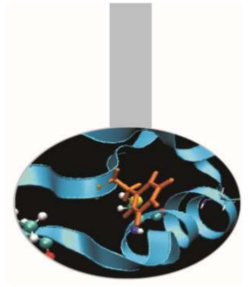
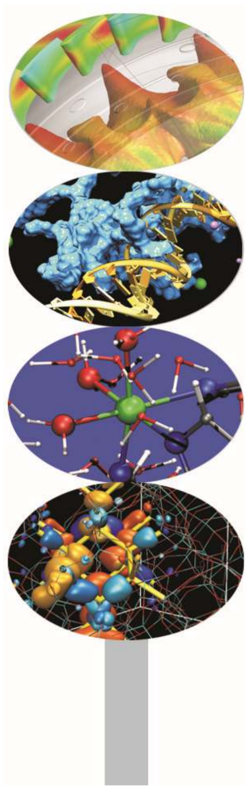
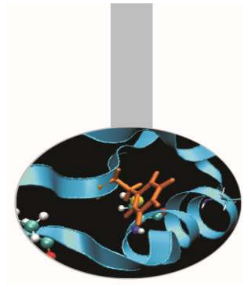


Introduction to parallel computing



P. Ramieri

May 2015

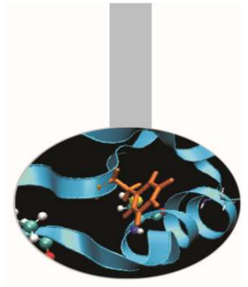


What is Parallel Computing?

Traditionally, software has been written for ***serial*** computation:

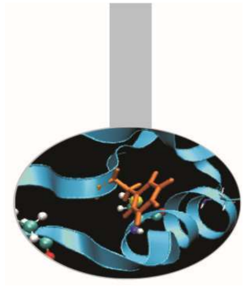
- To be run on a single computer having a single Central Processing Unit (CPU);
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.

What is Parallel Computing?



Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

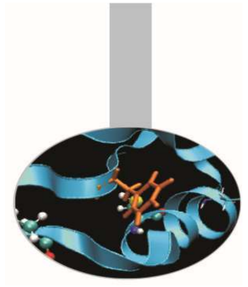
- A problem is broken into discrete parts that can be solved concurrently
- Instructions from each part execute simultaneously on different CPUs



Compute resources

The compute resources might be:

- A single computer with multiple processors;
- An arbitrary number of computers connected by a network;
- A combination of both.



Why Use Parallel Computing?

Save time and/or money:

in theory, more resources we use, shorter the time to finish, with potential cost savings.

Solve larger problems:

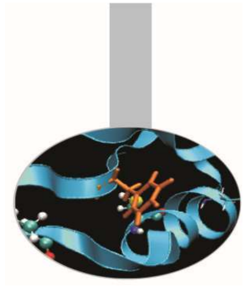
when the problems are so large and complex, it is impossible to solve them on a single computer. For example: the so called "Grand Challenge" problems requiring PetaFLOPS and PetaBytes of computing resources.

(en.wikipedia.org/wiki/Grand_Challenge)

Limits to serial computing: there are physical and practical reasons:

- Transmission speeds
- Limits to miniaturization
- Economic limitations

Why computing power is never enough?

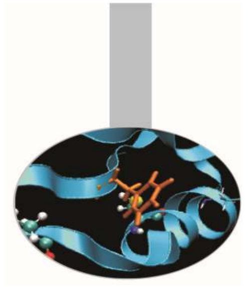


Many scientific problems can be tackled only by increasing processor performances.

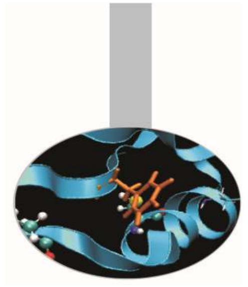
Highly complex or memory greedy problems can be solved only with greater computing capabilities:

- Weather modelling
- Protein analysis
- Medical drugs research
- Energy research
- Huge data amount analysis

TOP500



http://www.top500.org/



Flynn's Taxonomy

There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.

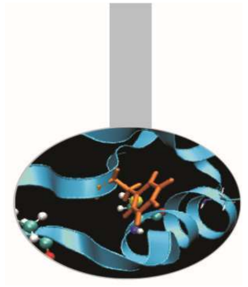
S I S D = Single Instruction, Single Data

S I M D = Single Instruction, Multiple Data

M I S D = Multiple Instruction, Single Data

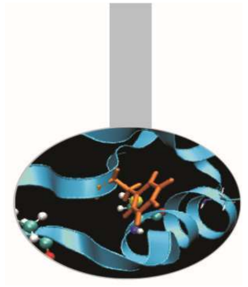
M I M D = Multiple Instruction, Multiple Data

Single Instruction, Single Data (SISD)

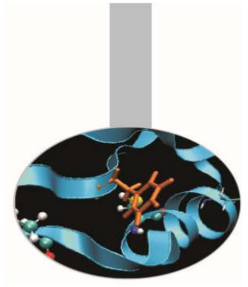


- Classical von Neumann architecture: serial computer
- **Single Instruction:** Only one instruction is executed by the CPU during any one clock cycle
- **Single Data:** Only one data stream is being used as input during any one clock cycle
- This is the oldest and the most common type of computer
- Examples: older generation mainframes and workstations; most modern day PCs.

Single Instruction, Multiple Data (SIMD)

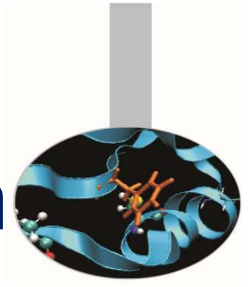


- A type of parallel computer
- **Single Instruction:** All processing units execute the same instruction at any given clock cycle
- **Multiple Data:** Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.



Multiple Instruction, Single Data (MISD)

- A type of parallel computer
- **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams.
- **Single Data:** A single data stream is fed into multiple processing units.
- Few actual examples of this class of parallel computer have ever existed.



Multiple Instruction, Multiple Data (MIMD)

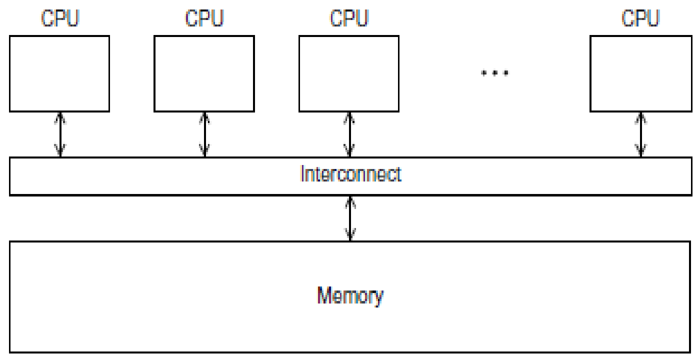
- A type of parallel computer
- **Multiple Instruction:** Every processor may be executing a different instruction stream
- **Multiple Data:** Every processor may be working with a different data stream
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Note: many MIMD architectures also include SIMD execution sub-components

Multiple Instruction, Multiple Data (MIMD)

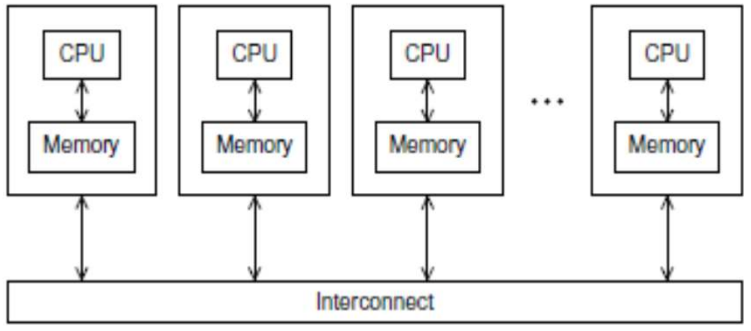
GALILEO Cluster

Concepts and Terminology

- **Shared Memory** = a computer architecture where all processors have direct access to common physical memory. Also, it describes a model where parallel tasks can directly address and access the same logical memory locations.
- **Distributed Memory** = network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

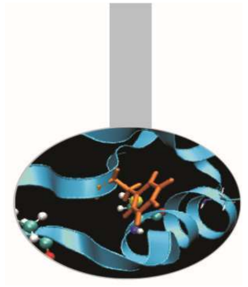


Shared Memory



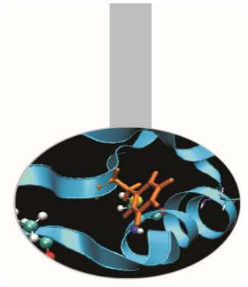
Distributed Memory

Concepts and Terminology



- **Communications** = parallel tasks typically need to exchange data. There are several ways to do that: through a shared memory bus or over a network.

- **Synchronization** = the coordination of parallel tasks in real time, very often associated with communications. Usually implemented by establishing a synchronization point where a task may not proceed further until another task(s) reaches the same or logically equivalent point. Synchronization can cause an increase of the wall clock execution time.



Concepts and Terminology

Speedup

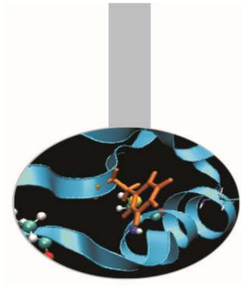
Speedup of a code which has been parallelized, defined as:

Wall-clock time (serial execution) / wall-clock time (parallel execution)

It is used as an indicator for a parallel program's performance.

Parallel Overhead = the amount of time required to coordinate parallel tasks. Parallel overhead can include factors such as:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
- Task termination time



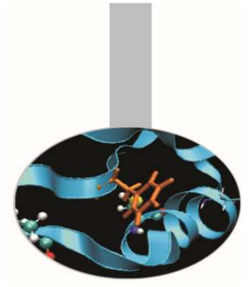
Concepts and Terminology

- **Massively Parallel** = refers to the hardware that comprises a given parallel system - having many processors.

- **Embarrassingly Parallel** = solving many similar, but independent tasks simultaneously; it needs just few coordination between the tasks.

- **Scalability** = the ability of a parallel system to proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
  - Hardware: memory-cpu bandwidths and network communications
  - Application algorithm
  - Parallel overhead

Parallel programs



Generally speaking a program parallelisation implies a subdivision of the problem model.

After subdivision the computing tasks can be distributed among more processes.

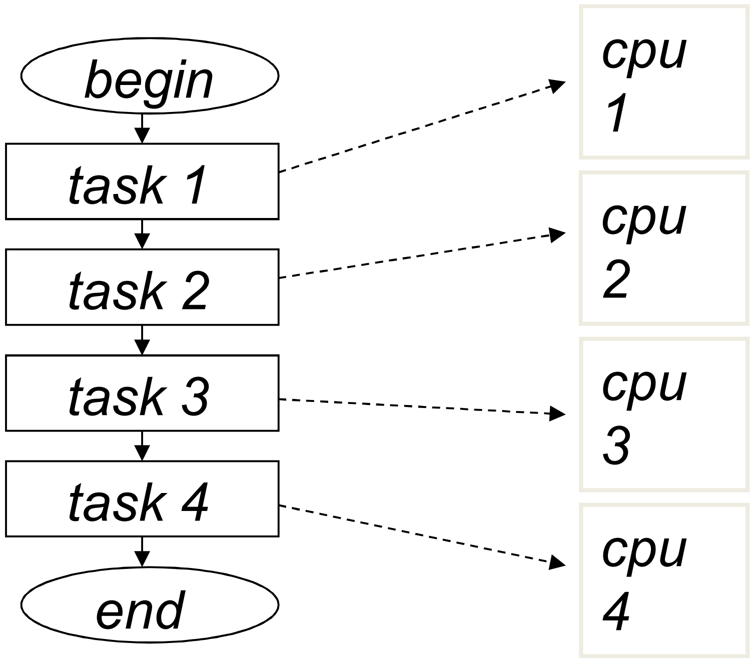
Two main approaches may be distinguished:

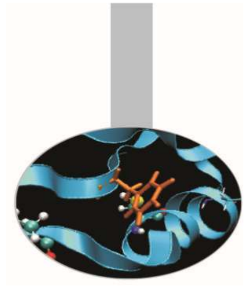
- **Thread** level parallelism
- **Data** level parallelism

Task parallelism

Thread (or task) parallelism is based on parting the operations of the algorithm.

If an algorithm is implemented with series of independent operations these can be spread throughout the processors thus realizing program parallelisation.

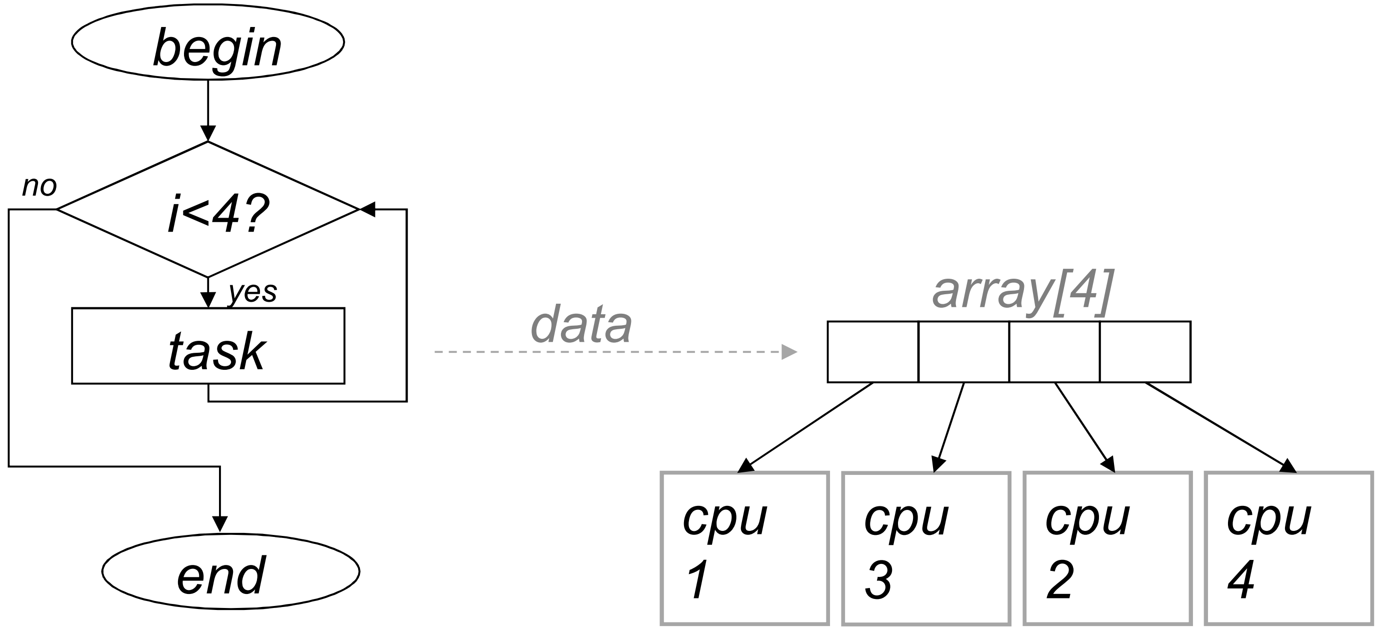




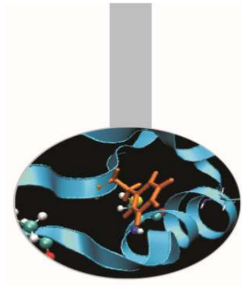
Data parallelism

Data parallelism means spreading data to be computed through the processors.

The processors execute merely the same operations, but on diverse data sets. This often means distribution of array elements across the computing units.



Parallel, concurrent, distributed



What is the difference between parallel, concurrent and distributed programming?

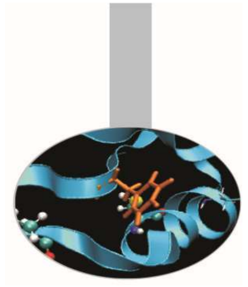
A program is said to be **concurrent** if multiple threads are generated during execution.

A **parallel** program execution is carried on by multiple, tightly cooperating threads.

A program is **distributed** when indipendent processes do cooperate to complete execution.

Anyhow there are not unique definitions and authors may give different versions. The definitions herein cited are those held by P. Pacheco, “An introduction to parallel programming”.

Parallel, concurrent, distributed



Based on the preceding definitions, parallel and distributed programs are *concurrent* programs, because multiple independent threads are working together to complete computation.

Often a program is said to be *parallel* if it is executed on computing units that share the same memory or are elsewhere connected by a high speed network and usually are very closed together.

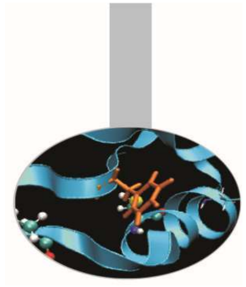
Distributed programs instead are executed on processors physically distributed in a (wide) geographical area and connected by a (not so fast) network. Program processes are therefore considered rather independent each other.

Processes, threads and multitasking

Operating systems are sets of programs that manage software and hardware resources in a computer. Operating systems control the usage of processor time, mass storage, I/O devices and other resources.

When a program execution is started, the operating system generates one or more processes. These are instances of the computer program and contain:

- Executable machine code
- A memory area, often divided in stack, heap and other parts
- A list of computer resources allocated to enable program execution
- Security data to access hardware and software resources
- Informations on the state of the process, i.e. executing, waiting for a resource availability, memory allocation and so on

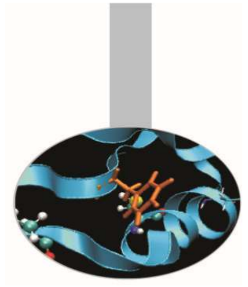


Processes, threads and multitasking

If the operating system is able to manage the execution of multiple processes at one time, it is said to be **multitasking**. On high performance parallel computers multi-tasking is usually of the pre-emptive type, i.e. slices of CPU time are dedicated in turn to each process, unless enough multiple computing units are available.

This means that parallel programs can be executed by concurrent **processes** and the operating system is able to manage their requests. If a computing resource is temporarily unavailable, the requiring process is halted. Anyhow program execution may still be carried on because time slices are granted to the processes that have the availability of the resource.

Parallel programs launched on systems where processors share a global memory are often executed as one process containing multiple **threads**, that share the computing resources of the process including process memory and devices.

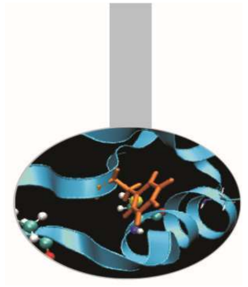


Process interactions

Process interactions may be classified as:

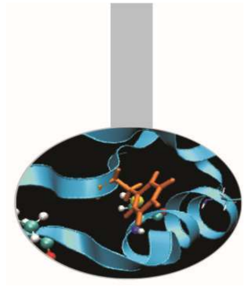
- Cooperation
- Competition
- Interference
- Mutual exclusion
- Deadlock

Cooperation



This kind of interaction is predictable and desirable. Cooperating processes exchange short signals or heavier data transfers.

Process interaction leads to synchronisation and hence to a communication if data are transferred.

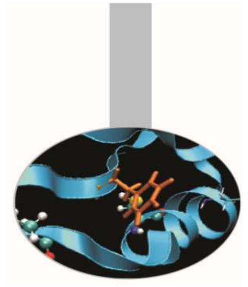


Competition

This kind of interaction is undesirable but nonetheless predictable and unavoidable. It may happen when more processes need to access a common resource that can not be shared (as an example updating a unique counter). Competition may be managed with so called critical sections.

Also contending processes exchange signals and synchronize but in a way different from cooperation.

We can distinguish direct or explicit synchronisation (coming from cooperation) from indirect or implicit synchronisation (caused by competition).

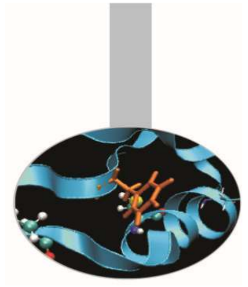


Interference

Interference is an unpredictable and undesirable kind of interaction usually arising from errors in developing a parallel program. Errors could come from interactions not required by the implemented algorithm or from interactions not properly handled.

This kind of interaction may show up or not depending by process execution flowing.

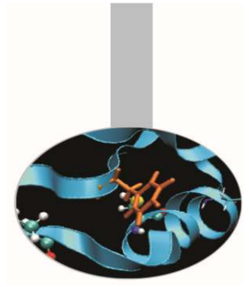
Mutual exclusion



Whenever more processes should not access concurrently a computing resource the problem of realising mutual exclusion has to be managed. This may come up from accessing devices such as writing a disk file or from updating a common memory space.

This kind of problem is often solved using critical sections.

Critical sections do ensure that processes can execute the instructions contained therein but only one at a time.

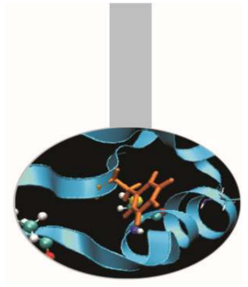


Deadlock

This undesired situation is always due to programming errors and arises when one or more processes are compelled to wait for something that will never happen.

Processes often enter a deadlock state if they encounter a synchronising point while some other process follow a different executing stream. As an example a program could contain two distinct barriers but processes can reach both of them concurrently.

Parallel program performance

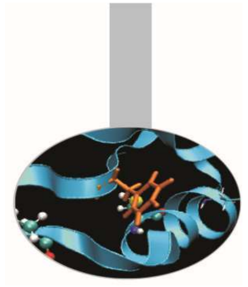


The goal of program parallelisation is to reduce execution elapsed time. This is accomplished by distributing execution tasks across the independent computing units. To measure the goodness of the parallelisation effort the time spent in execution by the sequential version of the program (i.e. the program before parallelisation optimisation) must be compared to the time spent by the parallelised version of the program.

Let us call *Tserial* the execution elapsed time of the sequential version of a program and *Tparallel* the execution elapsed time of the parallel version. In an ideal case if we run the program with *p* computing units (or cores):

$$T_{parallel} = \frac{T_{serial}}{p}$$

If that is true it is said the (parallel) program has a **linear speed-up**.



Speed-up and efficiency

In a real program a linear speed-up is difficult to gain. It has to be considered that the execution flow of the sequential version of the program does not encounter troubles that the parallel version does.

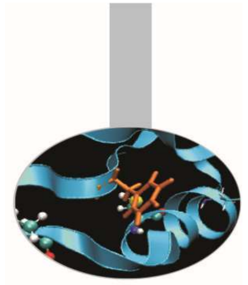
Overheads in a parallel program are introduced by simply dividing the program execution stream. Moreover there is often need of synchronisation and data exchange; furthermore critical sections have to be implemented.

Speed-up is defined as:

$$S = \frac{T_{serial}}{T_{parallel}}$$

The program has a linear speed-up if *S=p*, where *p* is the number of cores used in executing the program.

Speed-up and efficiency



It could be difficult to get a linear speed-up because of the overheads due to synchronisations, communications and often because of an unbalanced distribution of the computing tasks.

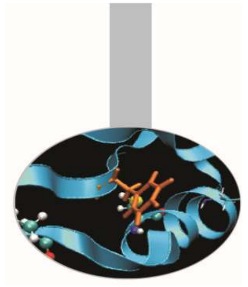
This leads to decreasing speed-up while growing the number of cores, because each core brings added overhead.

Efficiency is said to be the ratio between speedup and number of cores:

$$E = \frac{S}{p} = \left(\frac{\frac{T_{serial}}{T_{parallel}}}{p} \right) = \frac{T_{serial}}{p \cdot T_{parallel}}$$

Usually more cores are added, less efficiency is measured.

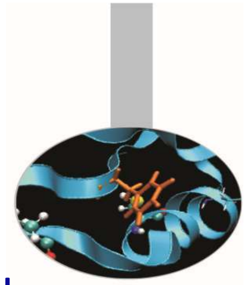
Overhead



Overheads are a significant issue in parallel programs and strongly affect program efficiency.

If overhead delays have to be considered elapsed execution times could be calculated according to:

$$T_{parallel} = \frac{T_{serial}}{p} + T_{overhead}$$



Amdahl’s law

If we can analyze a program and measure the portion of code that must be executed sequentially and the part of code that can be distributed across the cores we are able to forsee the program speed-up.

As an example, if it would be possible to parallelize 90% of a program, the remaining 10% of code runs sequentially; then according to Amdhal law:

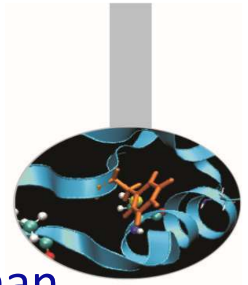
$$Tparallel = (0.9xTserial)/p + 0.1xTserial$$

where p = *number of available cores*

If $Tserial = 20$ *sec* and $p = 6$, then speed-up will be: $S = 20/(18/p + 2) = 4.$

The time spent in the parallel portion of code decreases as the number of cores increases. Eventually this time tends to zero, but the time spent in the sequential part of the code still remains and strongly limits the program speed-up.

Amdahl’s law



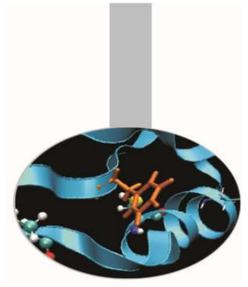
As a consequence Amdahl's law tells that speed-up will always be less than *1/r*, where *r* is the sequential portion of the program.

But let us not worry too much!

In real parallel computing world we have to take account of many facets and one of the most important is *problem dimension*. If we consider this we can be interested in Gustafson's (or Gustafson-Barsis') law:

$$S^G_p = p - a\,(p-1)$$

This formula can be applied to problems for which execution time can be kept constant increasing parallel cores as the problem dimensions increase. This actually applies to many real cases.



Problem dimensions

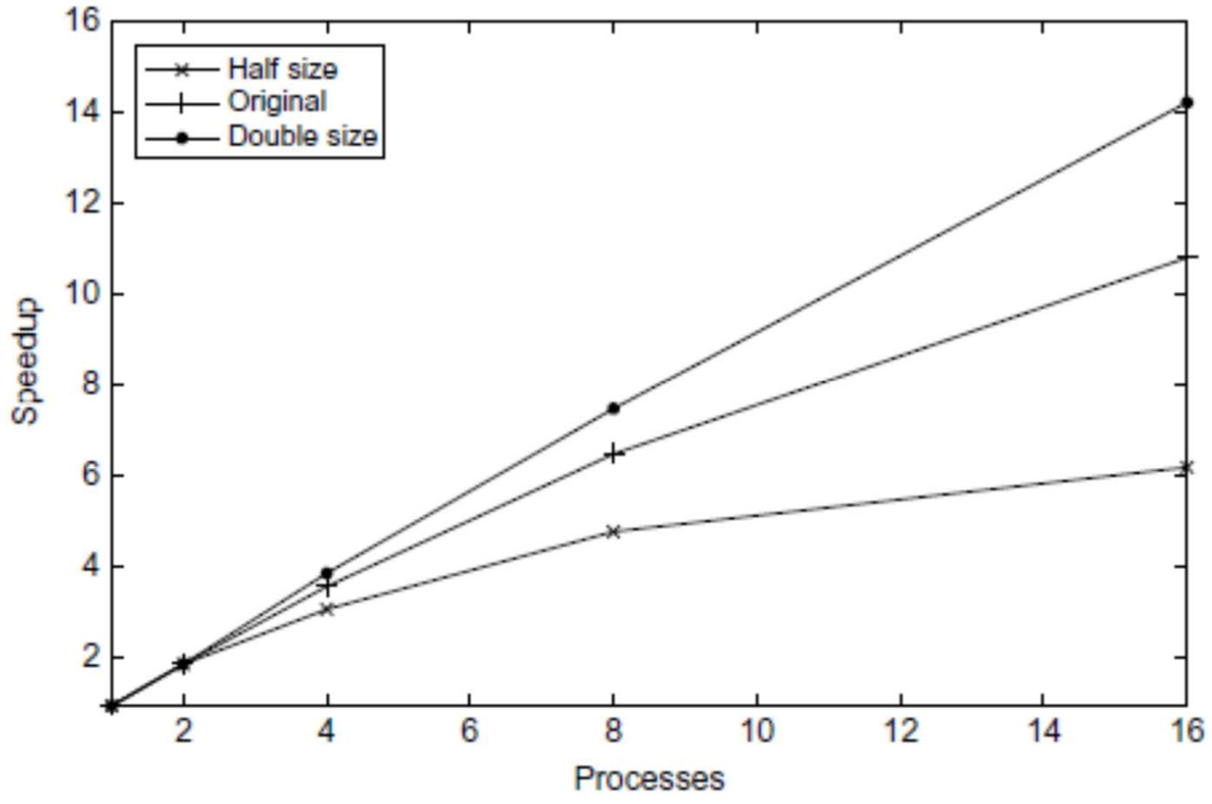
Problem dimension is important because size of data to be computed increases the processors computing time. It is possible to lower global elapsed time by distributing the work across more processors.

But overheads due to parallelisation stuff will not grow as much, hence speed-up is likely to increase.

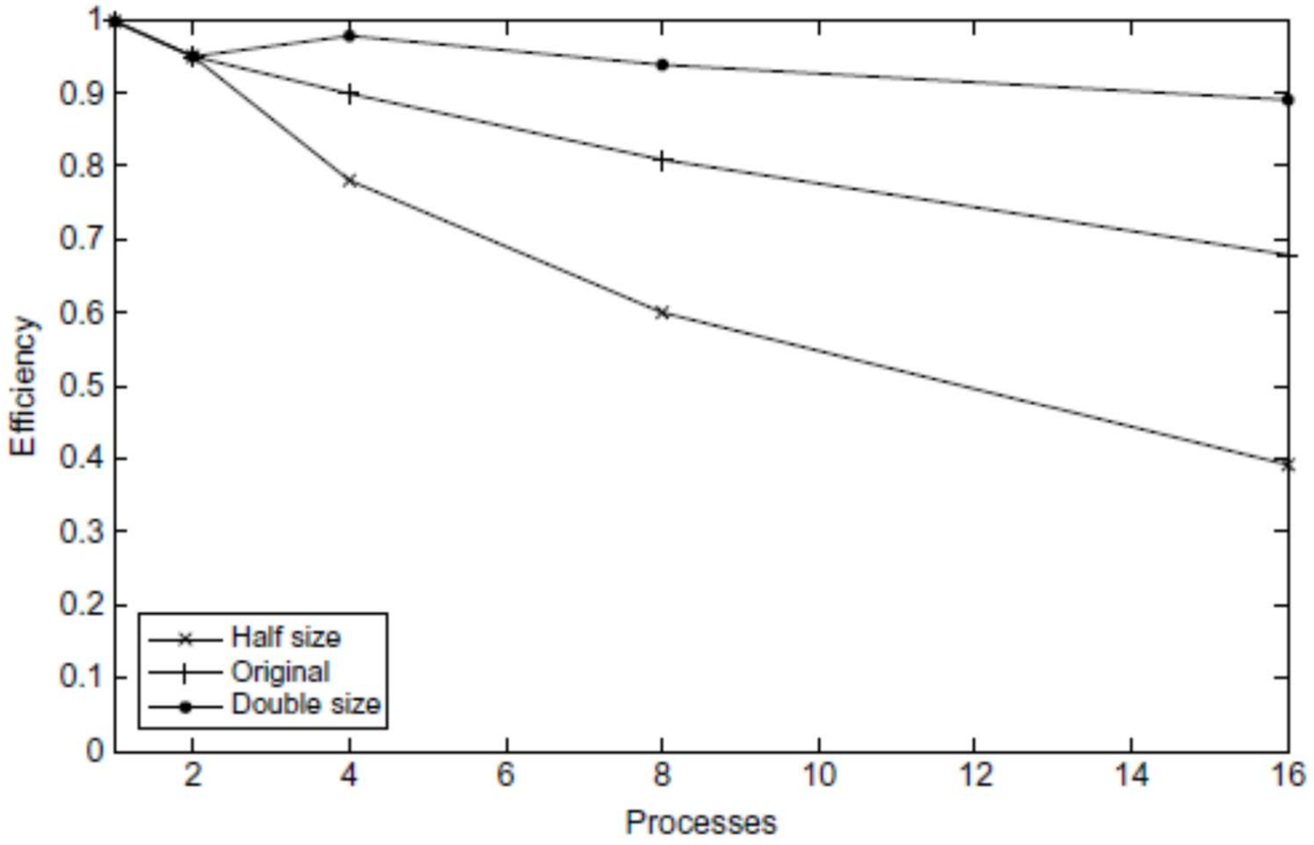
Usually, as the dimension of the problem grows, speed-up will grow as well, if enough parallel processors are added.

SuperComputing Applications and Innovation

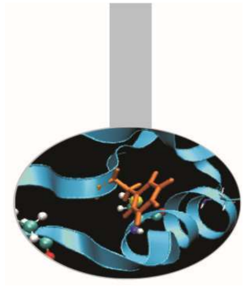
Speed-up and problem dimension



Efficiency and problem dimension



Scalability



In conclusion, there are basically two ways of evaluating scalability of a program.

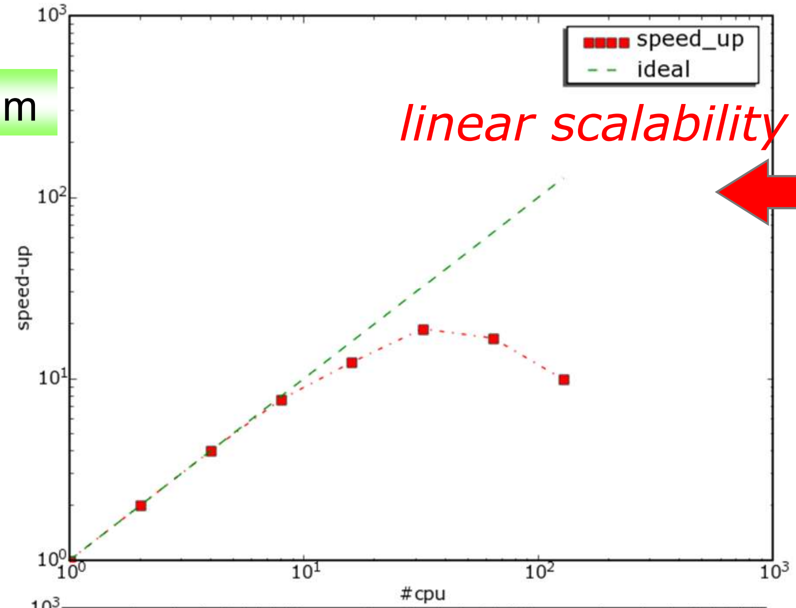
If global problem dimension is fixed and efficiency does not decrease while increasing the number of cores, then it is said that the program is **strongly scalable**.

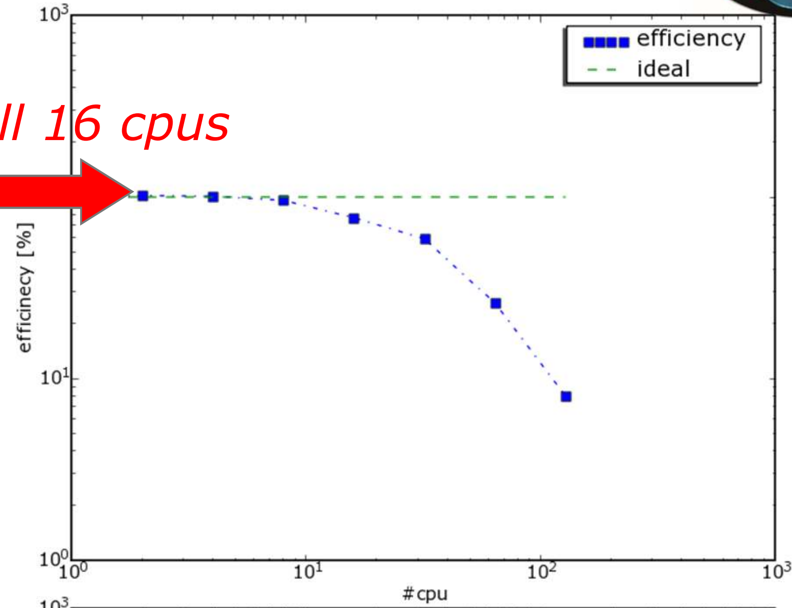
If the efficiency does not decrease when problem dimension per processor (i.e. global dimension has to be augmented as the number of processors increases) is kept almost unchanged, then the program is said to be **weakly scalable**.

Example: ANSYS Fluent benchmarks

Medium

linear scalability till 16 cpus

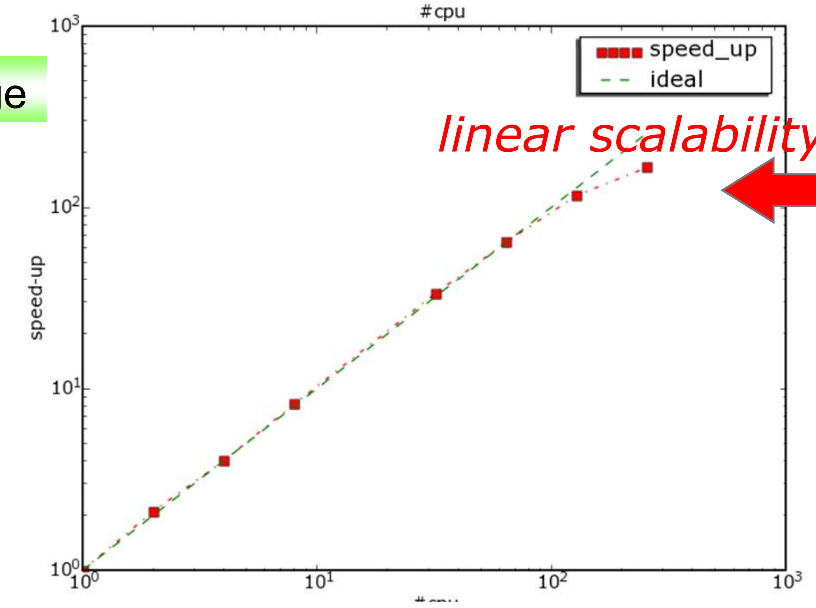


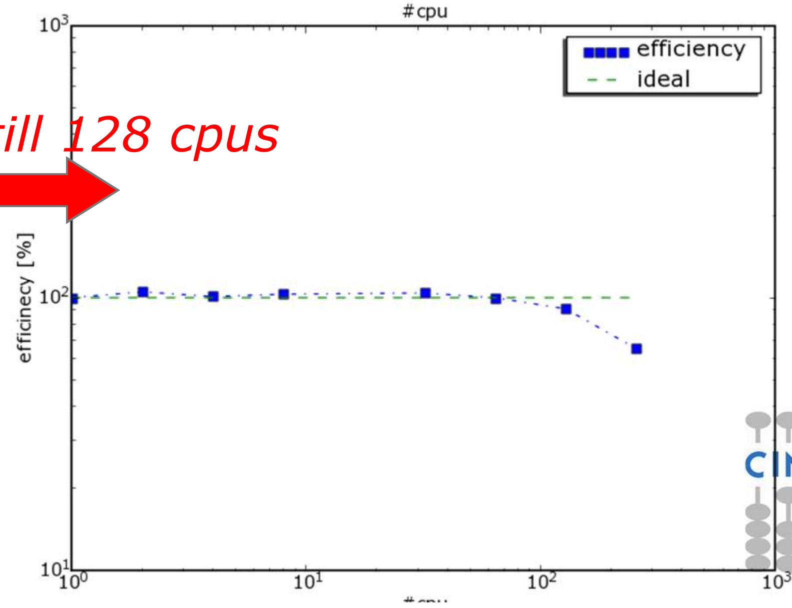


Large

L3

linear scalability till 128 cpus


🟥 speed_up
ideal
speed-up


🟦 efficiency
ideal
efficinecy [%]