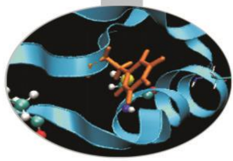
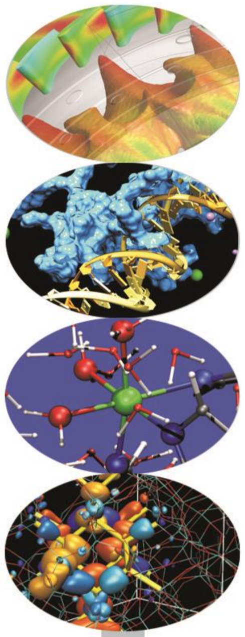
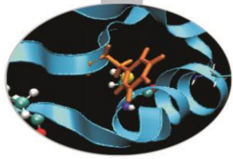
SuperComputing Applications and Innovation



Introduction to HPC



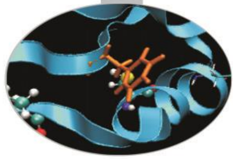
What is Parallel Computing?



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.

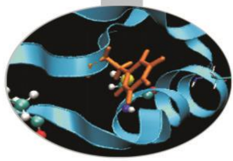
What is Parallel Computing?



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

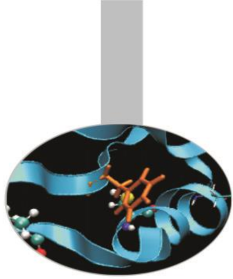


Compute resources

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)

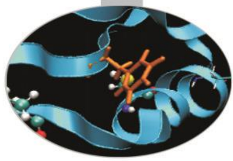
Limits to serial computing: there are physical and practical reasons:

•Transmission speeds

•Limits to miniaturization

•Economic limitations

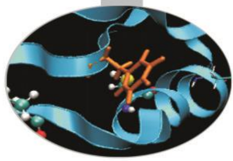
Why computing power is never enough?



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



Challenging research fields

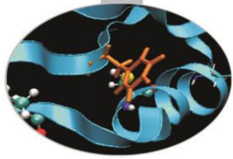
Weather modelling

Global weather forecasts require large datasets and complex computations. Even using the most powerful computers, numerical weather models at now can extend forecasts to about a week only. The accuracy of numerical predictions depends on the quality of observations together with the quality of the numerical models, but anyhow is limited by the chaotic nature of the partial differential equations used for climate modeling.

Protein analysis

Proteins are very complex molecules that have a great importance in life science and disease treatment research. High performance computers are needed to study all possible protein foldings and their interactions with other molecules. These computations may lead to astonishing improvements in treating many diseases including Huntington, Parkinson, Alzheimer

Challenging research fields



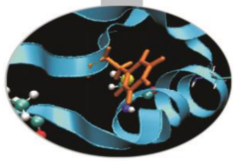
Medical drugs research

Interaction analysis between pharmaceutical molecules and human physiology can lead to discovery of effective medical drugs. As an example gene analysis may be useful to produce more effective personalised medical drugs with less side effects.

Energy research

Design and production of efficient wind turbins, solar cells, electric batteries depend on the availability of high performance computers. This kind of studies are important to increase efficiency of energy production and exploitation.

Challenging research fields



Data analysis

World data storage capability doubles every two years, but many stored informations are never analysed. One of the reasons is that the analysis of such huge amount of data requires enormous computing powers.

For example genome and protein data bases contain a lot of data that can be useful to better understand life evolution or medical drugs behavior.

Huge data amount are generated also by modern particle accelerators during collision events. For example the CERN Large Hadron Collider in Geneva is expected to produce an average of more than 27 TeraBytes of data per day. The analysis of these data could be important in astrophysics, phisics and medicine.

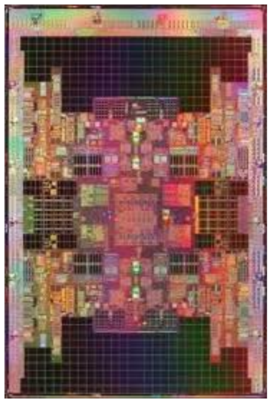
Chip performance improvement

Processor computational capability is growing continuously!

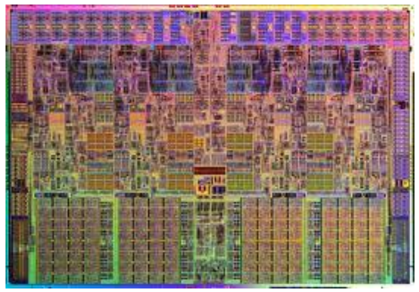
- Smaller transistors => higher processor circuits density
- Higher transistor density => higher computing speed
- Higher computing speed => higher heating and electricity consumption

i486 (1989)
***1,2M** transistors*

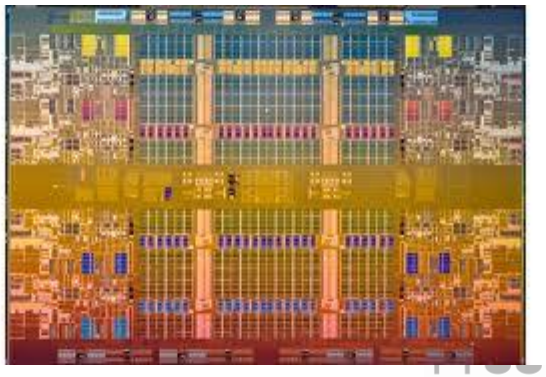
itanium2 (2003)
***220M** transistors*



Xeon nehalem (2007)
***781M** transistors*



Xeon nehalem-ex (2011)
***2300M** transistors*



How to manage transistor density?

Heating of chips in processors grows together with computational speed.

Processor performances decrease and chips may be damaged by too high temperatures.

Processor cooling with air or water is not as efficient as should be.

Even if it would be possible to increase transistor density, there are physical limits in making cheap faster processors.

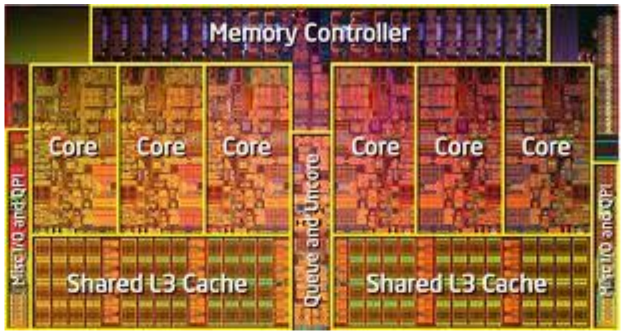
$\longrightarrow$

Multi-core processors

But it is possible to increase computational speed using **parallelism**!

If transistors become smaller it is possible to realize chips with more transistors. Computing power may be increased by duplicating computing circuits rather then raising speed. This leads to more powerful computers with less expensive cooling features.

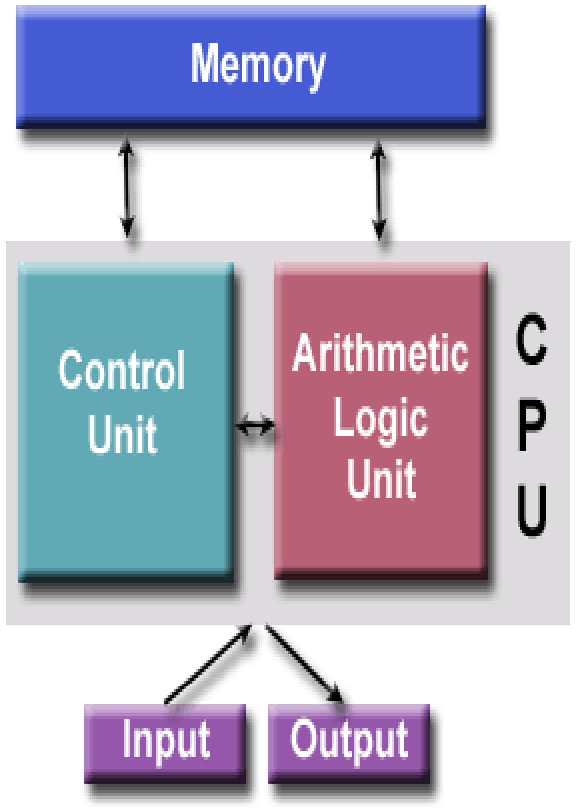
This leads to **multi-core** processors with a multiplicity of computing units



TOP500

http://www.top500.org/

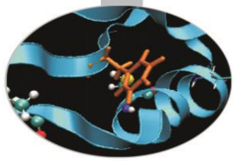
von Neumann Architecture



- RAM is used to store both program instructions and data
- Program instructions are coded data which tell the computer to do something
- Data is simply information to be used by the program
- Control unit fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
- Aritmetic Unit performs basic arithmetic operations
- Input/Output is the interface to the human operator

Parallel computers still follow this basic design, just multiplied in units. The basic, fundamental architecture remains the same.

Flynn's Taxonomy



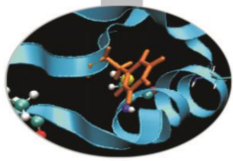
There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.

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

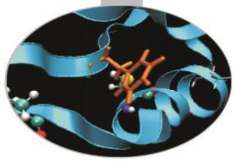
- **Single Data:** Only one data stream is being used as input during any one clock cycle

- This is the oldest and the most common type of computer

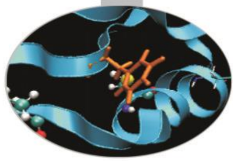
- Examples: older generation mainframes and workstations; most modern day PCs.

SuperComputing Applications and Innovation

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)

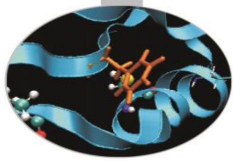
- A type of parallel computer

- **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)



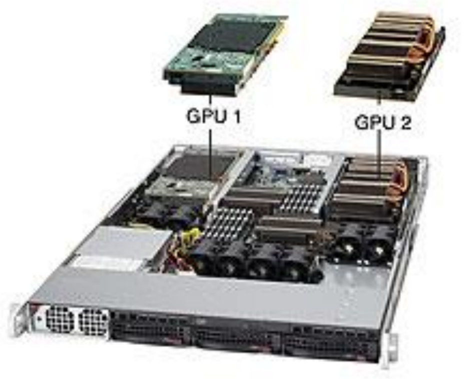
- A type of parallel computer
- **Multiple Instruction:** Every processor may be executing a different instruction stream
- **Multiple Data:** Every processor may be working with a different data stream
- Currently, the most common type of parallel computer - most modern supercomputers fall into this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

Multiple Instruction, Multiple Data (MIMD)

PLX Cluster

CPU-GPU Cluster

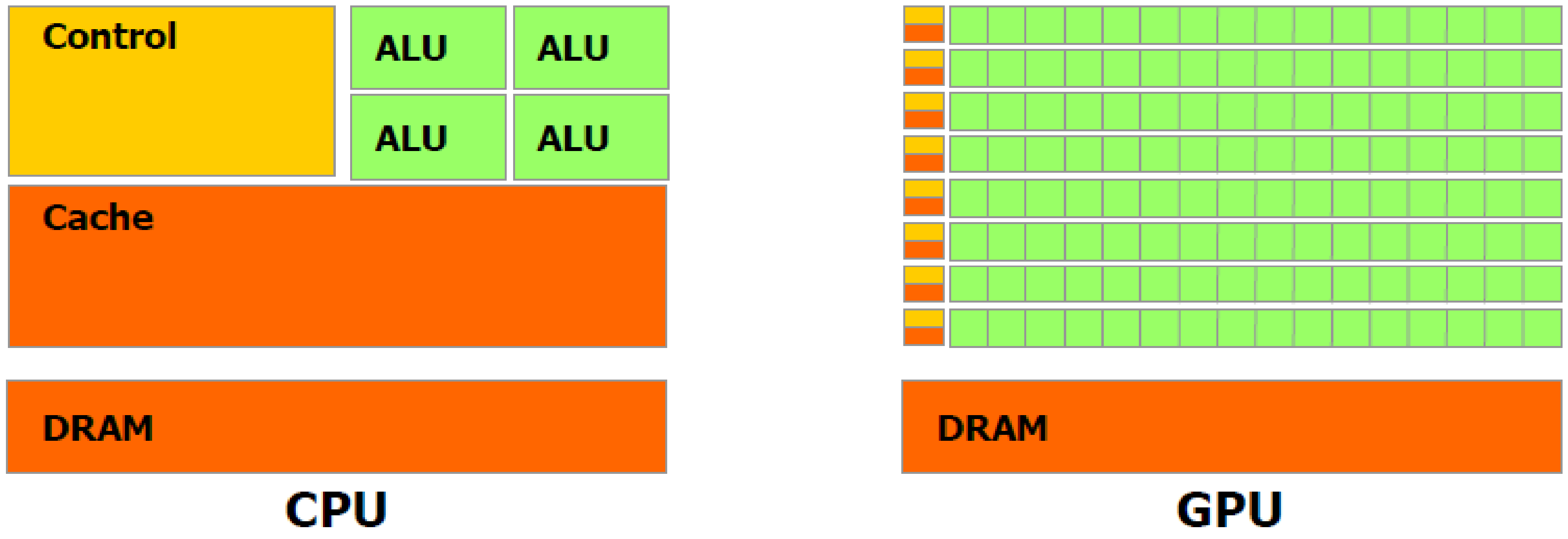
- Hybrid solutions: multicore CPU + manycore GPU
  - Each node has multicore CPU and graphics cards designed for GPU computing
  - considerable theoretical computing power on single node
  - additional memory layer on GPU
  - OpenMP, MPI, CUDA and hybrid solutions with MPI+OpenMP, MPI+CUDA, OpenMP+CUDA, OpenMP+MPI+CUDA

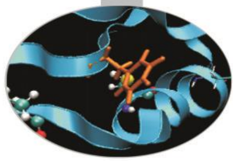


CPU vs GPU

- CPUs are general purpose processor able to solve any kind of algorithm BUT it reach the best performance using just one thread in a computational core.

- GPU are «specialized» processors for problems that can be classified as «intense data-parallel computations»:
  - Lot of small threads working in parallel

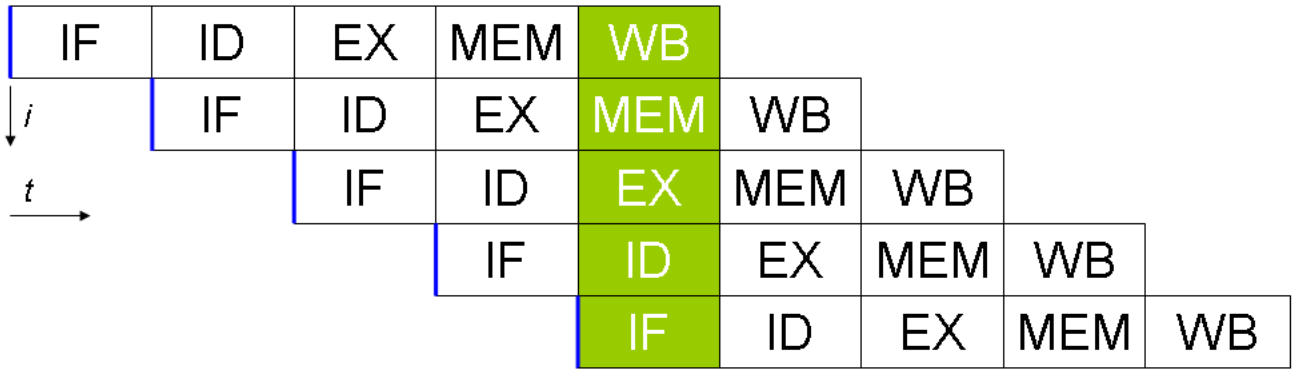




Concepts and Terminology

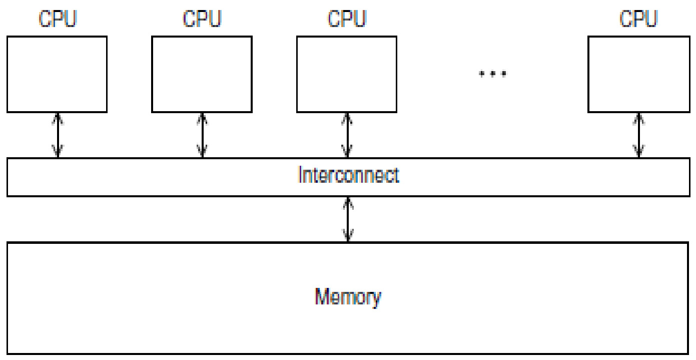
- **Task** = a logically discrete section of computational work. A task is typically a program set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.

- **Pipelining** = breaking a task into steps performed by different processor units; a type of parallel computing.

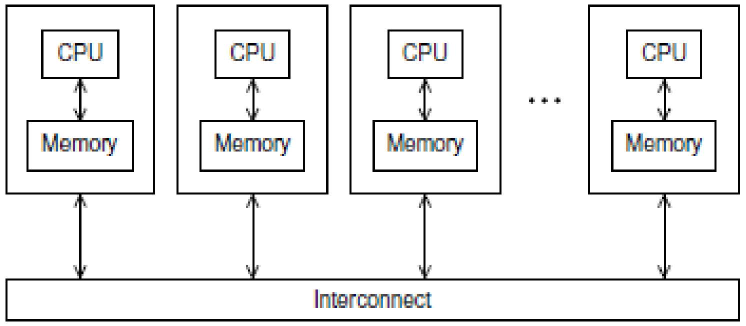


Concepts and Terminology

- **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.

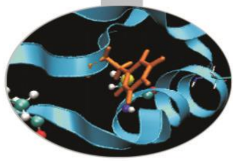


Shared Memory



Distributed Memory

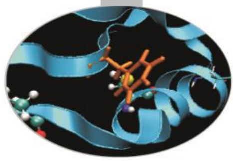
Concepts and Terminology



- **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

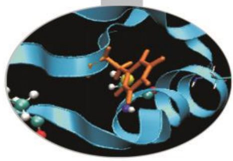
Speedup of a code which has been parallelized, defined as:

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 Computer Memory Architectures

Shared Memory

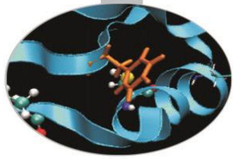
General Characteristics:

•All processorc can access all memory as global address space.

•Multiple processors can operate independently but share the same memory resources.

•Changes in a memory location effected by one processor are visible to all other processors.

•Shared memory machines can be divided into two main classes based upon memory access times: UMA and NUMA.



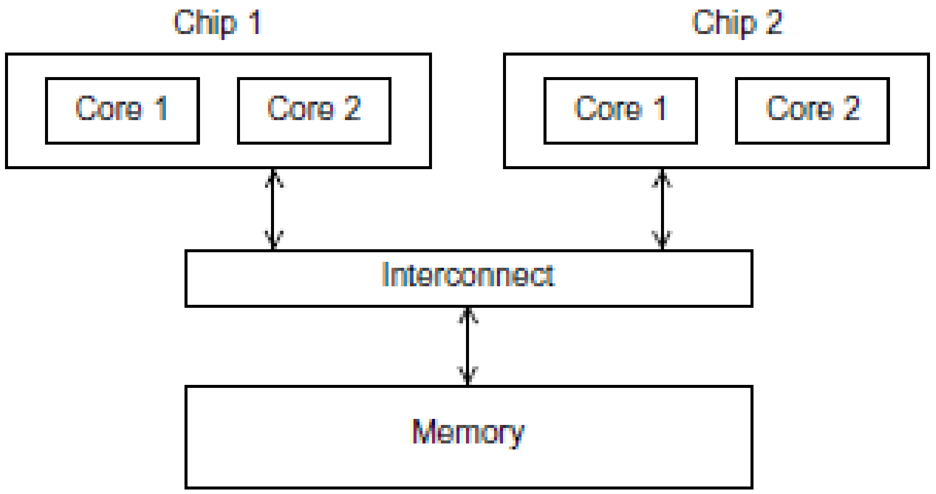
Parallel Computer Memory Architectures

Uniform Memory Access (UMA):

•Represented by Symmetric Multiprocessor (SMP) machines

•Identical processors

•Equal access and access times to memory

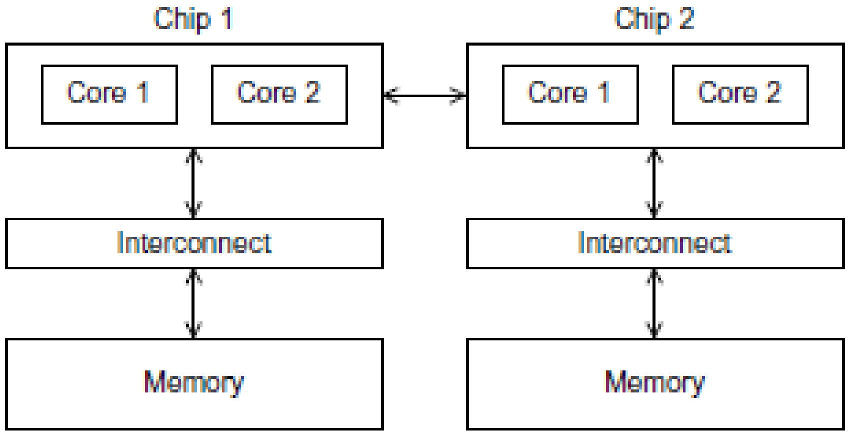


Uniform Memory Access

Parallel Computer Memory Architectures

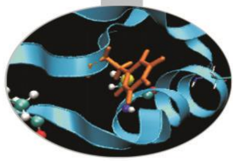
Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower



Non Uniform Memory Access

Parallel Computer Memory Architectures



Distributed Memory

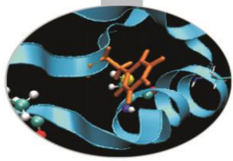
General Characteristics:

•Distributed memory systems require a communication network to connect inter-processor memory.

•Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.

•Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors.

•When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.



Parallel Computer Memory Architectures

Hybrid Distributed-Shared Memory

General Characteristics:

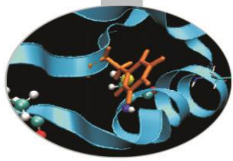
•The largest and fastest computers in the world today employ both shared and distributed memory architectures.

•The shared memory component can be a cache coherent SMP machine and/or graphics processing units (GPU).

•The distributed memory component is the networking of multiple SMP/GPU machines, which know only about their own memory - not the memory on another machine. Therefore, network communications are required to move data from one SMP/GPU to another.

•Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the future.

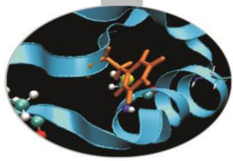
Parallel Programming Models



Overview

There are several parallel programming models in common use:

– Shared Memory
– Threads Model
– Distributed Memory / Message Passing
– Hybrid
– Single Program Multiple Data (SPMD)
– Multiple Program Multiple Data (MPMD)



Parallel Programming Models

Shared Memory

•In this programming model, tasks share a common address space, which they read and write to asynchronously.

•Various mechanisms such as locks / semaphores may be used to control access to the shared memory.

•The notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. Program development can often be simplified.

Implementations:

•Native compilers and/or hardware translate user program variables into memory addresses, which are global.

Parallel Programming Models

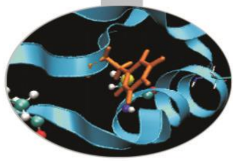
Threads model

•This programming model is a type of shared memory programming.

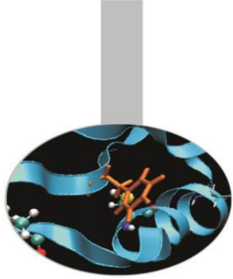
•A single process can have multiple, concurrent execution paths.

- The main program **a.out** is scheduled to run by the native operating system. a.out loads and acquires all of the necessary system and user resources to run.
- a.out performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.
- Each thread has local data, but also, shares the entire resources of a.out: threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.

Implementations: *POSIX Threads* and ***OpenMP***.



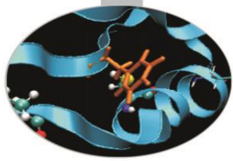
Parallel Programming Models



Distributed Memory / Message Passing Model

- This model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

Implementations: MPI (Message Passing Interface) library



Parallel Programming Models

Hybrid Model

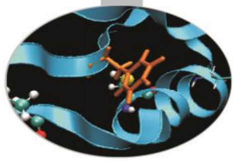
•A hybrid model combines more than one of the previously described programming models.

•Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).

- Threads perform computationally intensive kernels using local, on-node data
- Communications between processes on different nodes occurs over the network using MPI

•Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming.

- GPUs perform computationally intensive kernels using local, on-node data
- Communications between processes on different nodes occurs over the network using MPI



Parallel Programming Models

Single Program Multiple Data (SPMD)

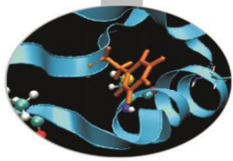
•SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

•SINGLE PROGRAM: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing or hybrid.

•MULTIPLE DATA: All tasks may use different data

•SPMD programs usually have the necessary logic to allow different tasks to conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.

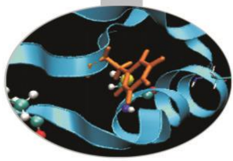
•The SPMD model, using message passing or hybrid programming, is probably the most commonly used parallel programming model for multi-node clusters.



Parallel Programming Models

Multiple Program Multiple Data (MPMD)

- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MULTIPLE PROGRAM: Tasks may execute different programs simultaneously. The programs can be threads, message passing or hybrid.
- MULTIPLE DATA: All tasks may use different data
- MPMD applications are not as common as SPMD applications, but may be better suited for certain types of problems, particularly those that use an explicit domain decomposition or functional decomposition.



Designing Parallel Program

The first step in developing parallel software is to first understand the problem that you wish to solve in parallel.

Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

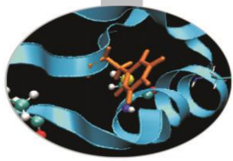
- **Identify the program's hotspots**

  The majority of scientific and technical programs usually accomplish most of their work in a few places: profilers and performance analysis tools can help here. Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

- **Identify bottlenecks in the program**

  For example, I/O is usually something that slows a program down.

  May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

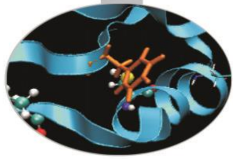


Designing Parallel Program

Overview

In order to design and develop a parallel program, we have to pay attention at several aspects:

- Partitioning
- Communications
- Synchronization
- Data Dependencies
- Load Balancing
- Granularity
- I/O



Designing Parallel Program

Partitioning

•One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.

•There are two basic ways to partition computational work among parallel tasks: ***domain decomposition*** and ***functional decomposition***.

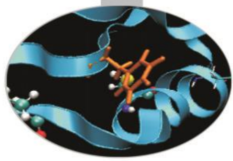
Domain Decomposition:

•In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.

Functional Decomposition:

•In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

Designing Parallel Program



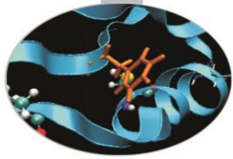
Communications

The need for communications between tasks depends upon your problem:

- **You DON'T need communications**
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data: these types of problems are often called ***embarrassingly parallel***. Very little inter-task communication is required.

- **You DO need communications**
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other. Changes to neighboring data has a direct effect on that task's data.

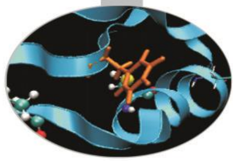
Designing Parallel Program



Communications

There are a number of important factors to consider when designing your program's inter-task communications:

- **Cost of communications**
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
  - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.



Designing Parallel Program

Communications

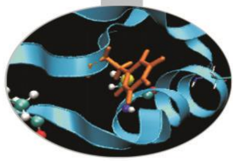
- **Latency vs. Bandwidth**
  - –***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
  - –***bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
  - –Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

Designing Parallel Program

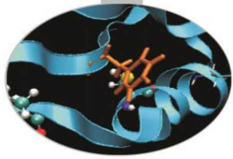
Communications

•Synchronous vs. asynchronous communications

- Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
- Synchronous communications are often referred to as blocking communications since other work must wait until the communications have completed.
- Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
- Asynchronous communications are often referred to as non-blocking communications since other work can be done while the communications are taking place.



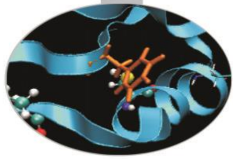
Designing Parallel Program



Communications

- **Scope of communications**
  - Point-to-point: involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - Collective: involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective.

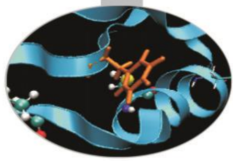
Both of the two scopings described can be implemented synchronously or asynchronously.



Designing Parallel Program

Synchronization

- **Barrier**
  - Usually implies that all tasks are involved: each task performs its work until it reaches the barrier. It then stops, or "blocks". When the last task reaches the barrier, all tasks are synchronized.
- **Lock / semaphore**
  - Can involve any number of tasks: typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code. Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it. Can be blocking or non-blocking.
- **Synchronous communication operations**



Designing Parallel Program

Data Dependencies

Definition:

•A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.

•A ***data dependence*** results from multiple use of the same location(s) in storage by different tasks.

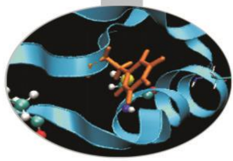
•Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.

How to Handle Data Dependencies:

•Distributed memory architectures: communicate required data at synchronization points.

•Shared memory architectures: synchronize read/write operations between tasks.

Designing Parallel Program



Load Balancing

•Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time.

•Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.

How to Achieve Load Balance:

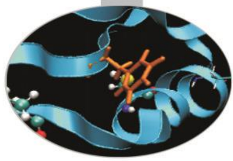
•Equally partition the work each task receives

- For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
- For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.

•Use dynamic work assignment

- It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Designing Parallel Program



Granularity

In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

Periods of computation are typically separated from periods of communication by synchronization events.

Fine-grain Parallelism:

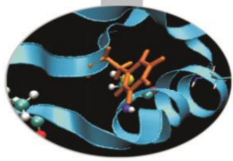
•Relatively small amounts of computational work are done between communication events: if granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

Coarse-grain Parallelism:

•Relatively large amounts of computational work are done between communication/synchronization events but it's harder to load balance efficiently

Designing Parallel Program

I/O



•I/O operations are generally regarded as inhibitors to parallelism

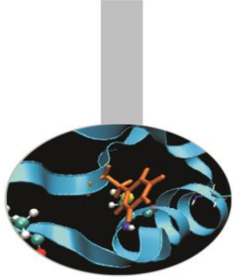
•Parallel I/O systems may be immature or not available for all platforms

•In an environment where all tasks see the same file space, write operations can result in file overwriting

•Read operations can be affected by the file server's ability to handle multiple read requests at the same time

A few advices:

- Reduce overall I/O as much as possible
- Writing large chunks of data rather than small packets is usually significantly more efficient.
- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks.
- Use local, on-node file space for I/O if possible. For example, each node may have /tmp filespace which can used. This is usually much more efficient than performing I/O over the network to one's home directory.



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.

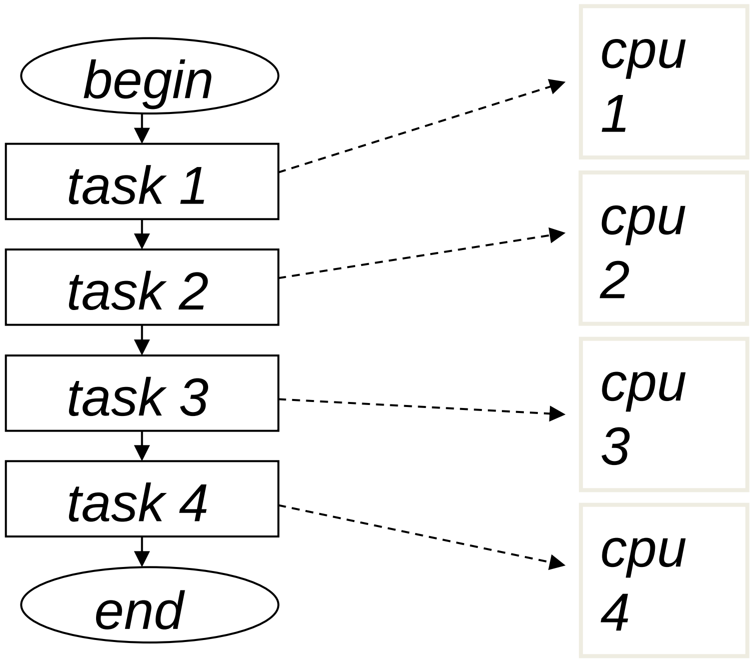
Two main approaches may be distinguished:

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

Task parallelism

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

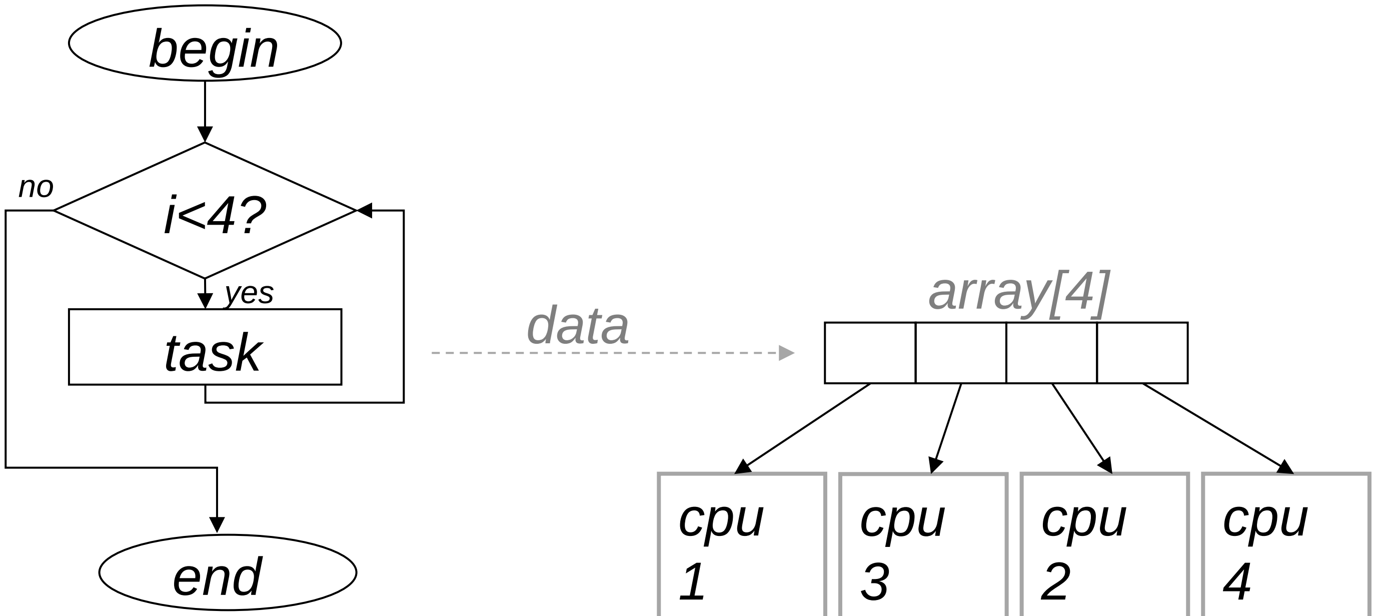
If an algorithm is implemented with series of independent operations these can be spread throughout the processors thus realizing program parallelisation.

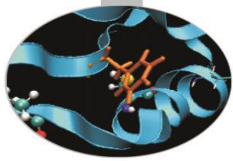


Data parallelism

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?

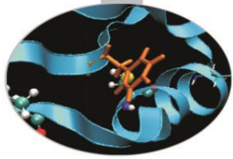
A program is said to be **concurrent** if multiple threads are generated during execution.

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



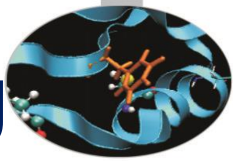
Based on the preceding definitions, parallel and distributed programs are *concurrent* programs, because multiple independent threads are working together to complete computation.

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.

SuperComputing Applications and Innovation

Processes, threads and multitasking

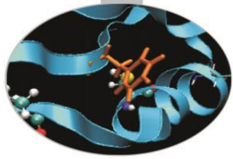


Operating systems are sets of programs that manage software and hardware resources in a computer. Operating systems control the usage of processor time, mass storage, I/O devices and other resources.

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

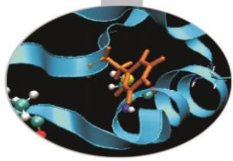
Processes, threads and multitasking



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.

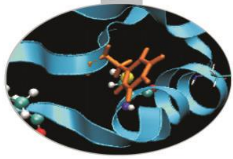
Parallel programs launched on systems where processors share a global memory are often executed as one process containing multiple **threads**, that share the computing resources of the process including process memory and devices.



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.

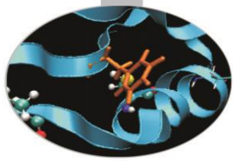
Process interaction leads to synchronisation and hence to a communication if data are transferred.

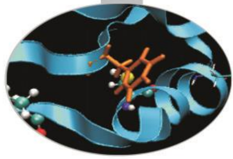
Competition

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).

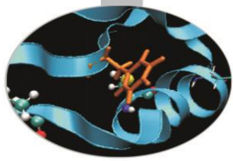




Interference

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.



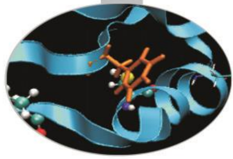
Mutual exclusion

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.

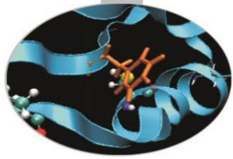
Deadlock



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

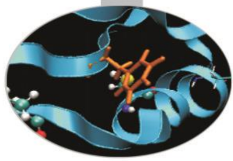


The goal of program parallelisation is to reduce execution elapsed time. This is accomplished by distributing execution tasks across the independent computing units. To measure the goodness of the parallelisation effort the time spent in execution by the sequential version of the program (i.e. the program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

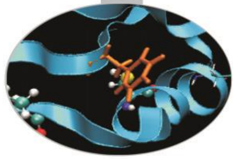
Overheads in a parallel program are introduced by simply dividing the program execution stream. Moreover there is often need of synchronisation and data exchange; furthermore critical sections have to be implemented.

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:

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

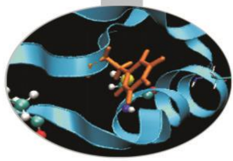
Usually more cores are added, less efficiency is measured.

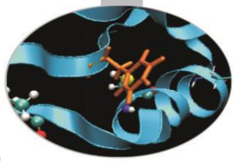
Overhead

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 seq uentially; then according to Amdhal law:

Tparallel = (0.9xTserial)/p + 0.1xTserial = 18/p + 2

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

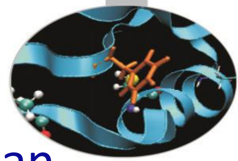
As a consequence Amdahl's law tells that speed-up will always be less than *1/r*, where *r* is the sequential portion of the program.

But let us not worry too much!

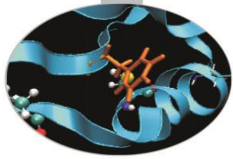
In real parallel computing world we have to take account of many facets and one of the most important is *problem dimension*. If we consider this we can be interested in Gustafson's (or Gustafson-Barsis') law:

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

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

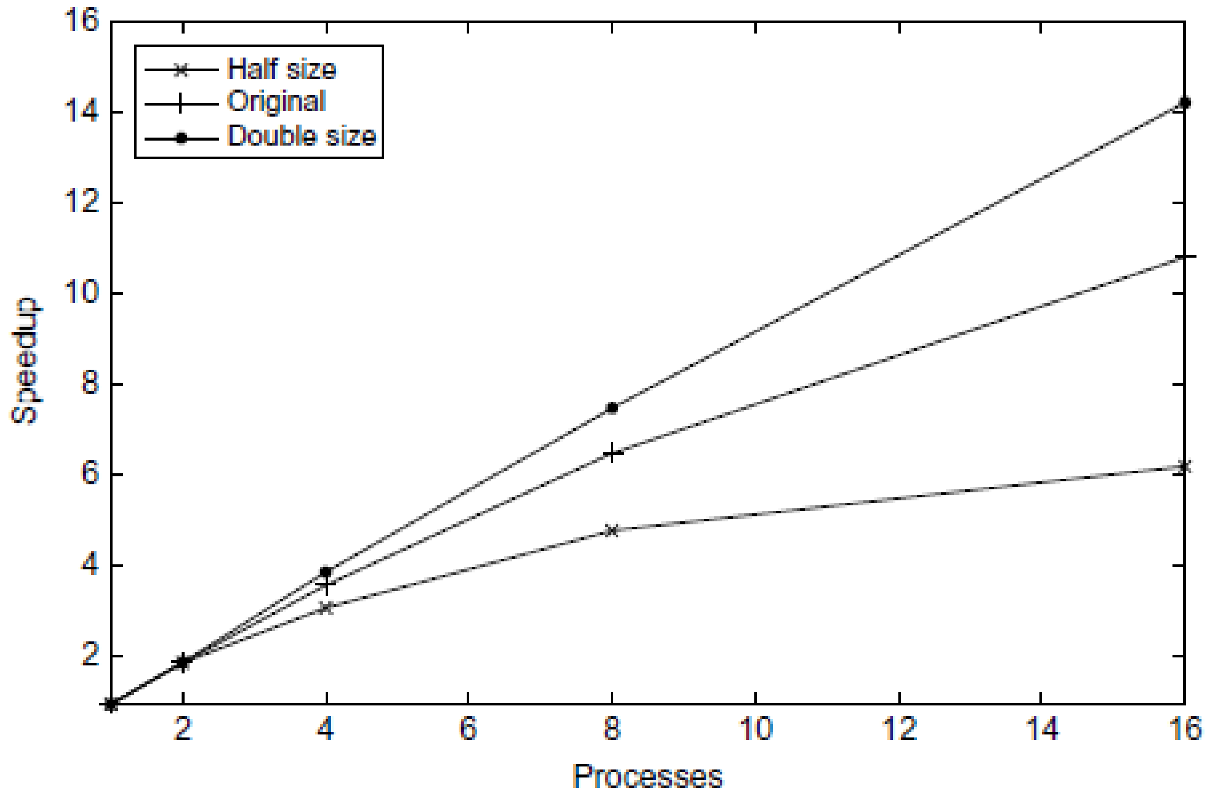


Problem dimension is important because size of data to be computed increases the processors computing time. It is possible to lower global elapsed time by distributing the work across more processors.

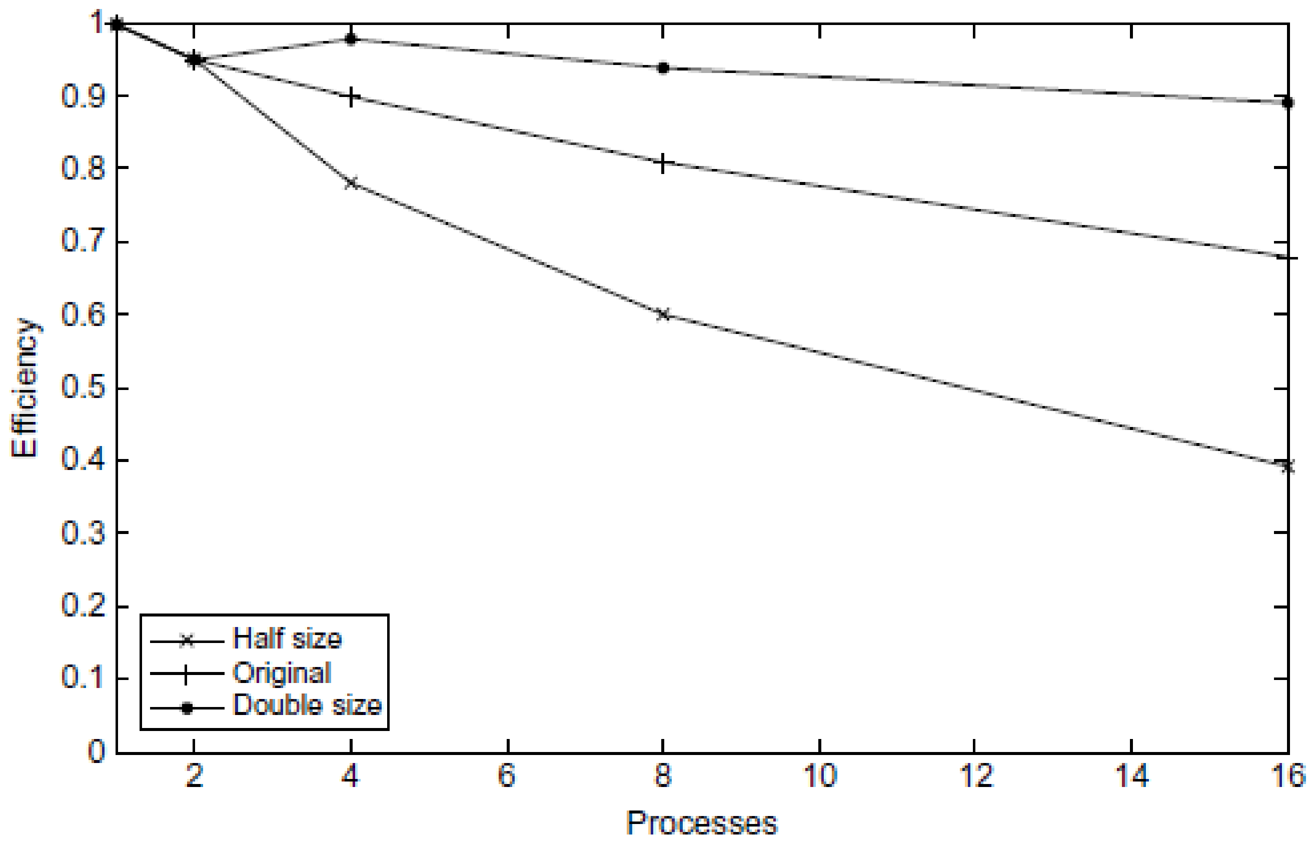
But overheads due to parallelisation stuff will not grow as much, hence speed-up is likely to increase.

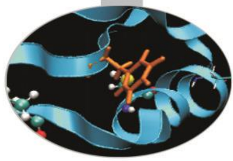
Usually, as the dimension of the problem grows, speed-up will grow as well, if enough parallel processors are added.

Speed-up and problem dimension



Efficiency and problem dimension





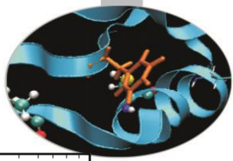
Scalability

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



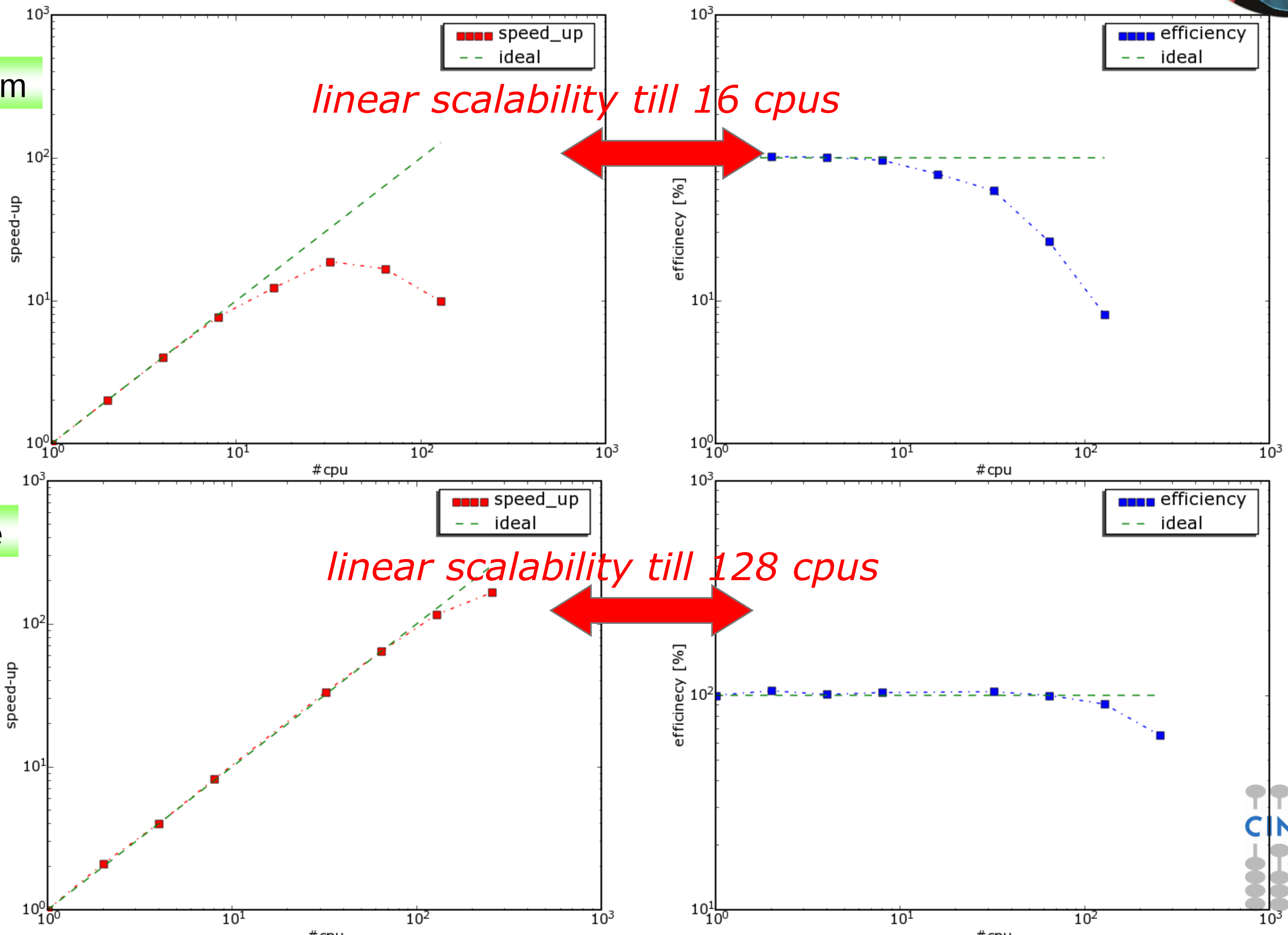
Medium

linear scalability till 16 cpus

Large

L3

linear scalability till 128 cpus


speed_up
ideal
speed-up
#cpu
efficiency
ideal
efficinecy [%]
10^0
10^1
10^2
10^3